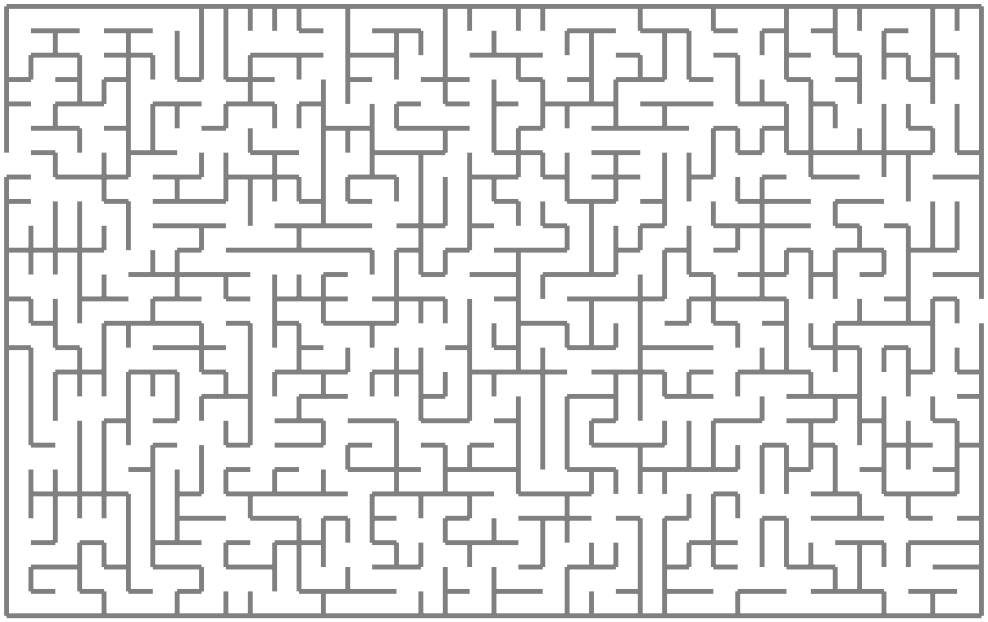
Rethinking Dependent Type Checking, Co-Contextually

Version of April 3, 2026



Gideon Jasper Timo Bot

Rethinking Dependent Type Checking, Co-Contextually

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Gideon Jasper Timo Bot
born in Leiderdorp, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: Random maze.

Rethinking Dependent Type Checking, Co-Contextually

Author: Gideon Jasper Timo Bot
Student id: 5092094
Email: g.j.t.bot@student.tudelft.nl

Abstract

Dependent type systems allow types to depend on values. This is used to encode rich semantic properties directly in types. Currently, dependently typed systems are not widely used for general-purpose programming, but they are commonly used in proof assistants. Agda, Idris, Lean and Rocq all support dependent types.

The expressiveness of the type system comes with a cost: type checking is expensive. This thesis explores co-contextual type checking as an alternative foundation for dependent type systems. Co-contextual type checking inverts the 'traditional' flow of information. This enables type-checking to proceed without immediate access to a complete typing environment.

This work explores a co-contextual formulation of a dependent type system and its accompanying type-checking algorithm. Both incremental and parallel variants of the algorithm are implemented for the dependently-typed lambda calculus *Elara*, based on *LambdaPi*(Löh, McBride, and Swierstra 2010). The research primarily focuses on the feasibility of such a type-checker.

Apart from providing the first co-contextual implementation of a dependent type system, its performance is compared to LambdaPi's contextual bi-directional type-checking algorithm for reference. In the end, it was found that the implemented co-contextual type checker generally performs worse than the contextual reference implementation. Its performance is in the same order of magnitude, with plenty of opportunity for improvement discussed in this work. In particular, co-contextual incremental type-checking has much potential by reusing evaluation, which is not leveraged to the fullest in the conservative approach in this work.

Keywords: Dependent Types, Type Systems, Type Checking, Co-contextual, Incrementalisation, Parallelisation, Performance

Thesis Committee:

Chair:	Dr. J.G.H. Cockx, Faculty EEMCS, TU Delft
Committee Member:	Dr. P. Pawelczak, Faculty EEMCS, TU Delft
Daily Co-Supervisor:	B. Liesnikov, Faculty EEMCS, TU Delft

Preface

This report is the result of my graduate research and is the final part of the MSc Computer Science at Delft University of Technology. The work for this thesis was performed at the Programming Languages (PL) group at the faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS). This project, in its entirety, was done without the use of artificial intelligence (AI), except for a spell checker.

First of all, I would like to thank my thesis advisor, Jesper Cockx, and daily co-supervisor, Bohdan Leisnikov. Our weekly meetings were generally enjoyable and productive. This regular exchange of ideas, has aided me so much throughout the duration of this research. I am also grateful for their advice, recommendations and feedback, as they have done more than I (unfortunately) came to expect after attending this university for so many years. I would also like to thank Ksawery Radziwiłowicz, with whom I have briefly discussed ideas regarding benchmarking type-checkers for dependently typed languages.

Lastly, I would like to conclude with a brief personal note. A few months of this project have not been the most enjoyable, to say the least. During this time, I have lost two family members, my grandmother Marianne Hänninen-Forst (who, coincidentally, worked at this very university a long time ago), her brother Karl Heinrich Wilhelm, and a family friend. I have also lost my cat, Jackey ("Muis"), whom I hold so dear to my heart. These events have demotivated me from working on this project for some time. That is not to say I am not proud of the work I have produced, but I wish to mention them as a testament to their importance in my life. May they not be forgotten. I want to thank my best friend, Boris Kalika (who is currently working on their own Master's Thesis in Berlin; best of luck), and my girlfriend, Angela Aranda Lara, for their continued love and support during this time.

I have generally enjoyed my time at this university, and I look back on it fondly. I have learnt so much in a wide variety of interests. If I had one regret, it would be not taking more classes, but now it is time to start a new chapter in my life.

Gideon Jasper Timo Bot
Delft, the Netherlands
April 3, 2026

Contents

Chapter 1

Introduction

Type systems play a central role in the design and implementation of modern programming languages. At their core, type systems define program structure beyond syntactic correctness. This is used to enforce constraints on how terms can be combined. For example, an argument applied to a function must have the same type as the function's domain in the simply typed lambda calculus. The abstract syntax tree of a term can be traversed algorithmically to verify if an expression is well-typed with respect to the underlying type system. This is the purpose of a type-checking algorithm.

Over time, type systems have evolved from simple base types like those found in early statically-typed languages to expressive systems such as Rust's borrow checker. This thesis focuses on dependent type systems. In these systems, the line between types and terms is blurred; types may depend on values. This expressiveness enables types to encode precise specifications, logical propositions, and invariants that relate directly to program behaviour. A consequence of this expressiveness is the computational cost of type checking. In contrast to simpler type systems, type checking dependent types is no longer a purely structural process. It may require program evaluation to obtain values, testing definitional equality and (higher-order) unification.

Dependently typed languages have some real-world use. They are commonly used to implement proof assistants, enabling formal reasoning through code. This is particularly useful for verified correctness of software and security-critical systems. In the current-day software development cycle, changes are incremental. Programmers rely on responsive tools to give direct feedback to aid their work and boost productivity. Type-checking is no longer an occasional batch process, but an interactive, continuously running service integrated into code editors, build systems, and language servers. A slow type-checker is noticeable, and the added computational complexity for dependently-typed languages means slower type checking.

The two most general approaches for increasing performance for type-checkers are parallel and incremental type checking. The former attempts to make use of as many processors as possible. Modern computers are generally multicore, allowing for part of the type checker to be executed in parallel to obtain a speed-up. The latter attempts to reuse previous computation. If the input to the type-checker only has a few changes compared to a previous run, many expressions need not be rechecked, reducing computation time.

There are also optimisations tailored to a specific type system, like dependent types. Instead of looking into optimisations or variations of existing algorithms, this thesis examines a different approach to type-checking altogether: co-contextual type-checking.

The work is guided by the following research questions:

1. Is a co-contextual type-checking algorithm feasible for dependently typed languages?

2. To what extent can such an algorithm be parallelised?

3. To what extent can such an algorithm be made incremental?

4. How does its performance compare to contextual type-checking algorithms?

The first question explores a co-contextual formulation of a dependent type system. This looks into how the existing theory needs to be adapted to support dependent types, what the constraints generated by such a type system look like, and whether they are practically solvable.

The second and third questions examine parallelisation and incrementalization of a co-contextual type checker for a dependently typed language, respectively. In theory, co-contextual type checking *should* support this, but the complexity of the type system may impose limitations on its implementation or benefits.

The last question ties the other three together. The motivation to consider co-contextual type-checking in the first place is its (potential) performance benefits, as dependent type-checking has proven to be complex. The goal is to compare its performance to a 'traditional' implementation for the same type theory.

The contributions of this thesis are a formulation of a co-contextual dependently-typed type system, and an implementation of said system (including parallelisation and incrementalization) in the language Elara. This type checker is then compared to a contextual, bi-directional, type-checking algorithm sharing the same core language.

The next chapters, chapter 2 and chapter 3, cover a theoretical background and motivation for this research, and related work in this area. The following chapters, chapter 4, chapter 5, and chapter 6, describe the design of a co-contextual type system for dependent types and a practical implementation of said design. Lastly, the performance of this implementation is evaluated in chapter 7.

Chapter 2

Background

This chapter presents the theoretical foundation of this research, as well as the motivation.

2.1 Type Systems

Type systems are a foundational component of modern programming languages. From an academic perspective, type systems offer a mathematical foundation for reasoning about programs. This goes beyond properties such as type safety.

For example, in a simply-typed lambda calculus, there is a notion of values, canonical forms, and small-step reduction. Values tell us what it *means* for a computation to be done, and canonical forms show what a value of a type *looks like*. In combination with its type system, it is possible to reason about the language's behaviour(Pierce et al. 2026). A program that adheres to the typing rules does not get stuck; either the computation has reached a value or a small-step reduction can be applied (*progress* theorem). If a small-step reduction can be applied, the program after reduction still has the same type (*preservation* theorem). Language properties are *provable* and can be generalised to 'real' languages and complex effects. Memory safety through Rust's borrow checker is partially inspired by Vault, an implementation of a type system designed for this purpose(Fahndrich and DeLine 2002).

From the purview of a programmer, such concepts may not be of direct interest, but their consequences certainly are. Type systems improve program reliability and help reduce the possibility of bugs in computer programs. They serve as a guide for program design and enable powerful tooling such as refactoring, code completion and static analysis.

2.2 Type Checking

Type checking is the process of (algorithmically) verifying that a program conforms to the rules of its type system (i.e. is *well-typed*). Looking back at the example in the previous section, properties about a program may be true if and only if the program itself is well-typed. Programmers rely on guarantees made by the type system, hence the need to perform type checking. This is where theory meets practice. The research focuses on *static* type checking; the check is performed at compile time.

With static type checking, the program's source code is analysed. Historically, as long as this process finishes in a reasonable time, it can be used in practice. In the modern software development cycle, however, this needs to be near instant. The approach is iterative; small changes to the source code and direct feedback improve productivity. An Integrated Development Environment (IDE) is used to highlight syntax errors, type errors, and other code smells[1], among other features. If a developer makes a mistake, the IDE will quickly inform

[1] A code smell is any characteristic of source code that hints at a deeper problem

them and allow them to fix the issue. A type system that is elegant on paper is of limited use if its type checker is too slow.

Apart from system-specific optimisations, the most general approaches to improve type-checker performance are incremental and parallel type checking. The former approach focuses on the iterative approach mentioned previously. If the changes are small, it may be possible to reuse previous work done by the type checker, reducing total computation time. This concept can be applied to virtually any type system(Zwaan, Antwerpen, and Visser 2022). The latter attempts to use more of the available computation power. With multi-core processors becoming the norm, computers can execute multiple instructions concurrently. A speed-up can be achieved by distributing the work across multiple cores, enabling parallel computation.

2.3 Dependent Types

Dependent types extend 'traditional' type systems by blurring the lines between types and terms. In the case of the dependent function type (Π-type), the codomain of the function may depend on a value. This is denoted as $\Pi(x : A).B$. Here, $B : A \to *$ is a dependent family of types. Essentially, this is a function that takes a value of type A and produces a type, $*$ denoting the universe of types. Non-dependent function types are functions where the type family B is a constant function.

Type checkers for dependently typed languages are more complex and, as a result, are typically slow. The type-checker intertwines computation, logic, and program equivalence. Often it is required to decide whether two terms are *definitionally equal*. This requires normalising terms that may contain higher-order functions and recursion.

2.4 Co-contextual Type Checking

In traditional *contextual* type checking, typing judgements are made relative to a typing context that records assumptions about the types of variables and other program entities. As the type checker traverses the abstract syntax tree (AST), the context is extended and used to look up these assumptions. This is shown on the left in Figure 2.1. The context is extended with the assumption that the variable x has type α, which in turn is used to infer the type of x.

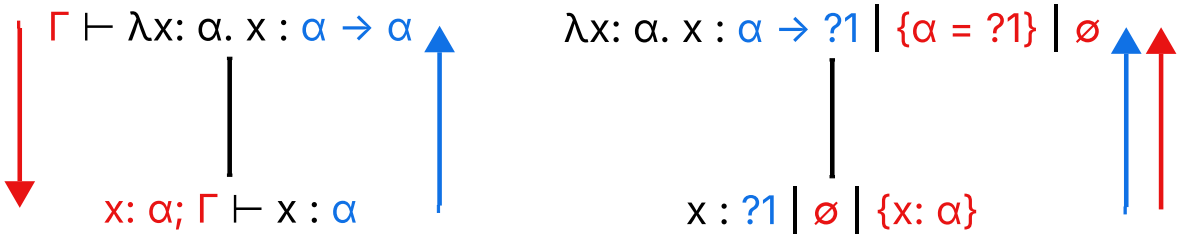


Figure 2.1: Syntax Tree Traversal. Left: contextual type checking. Right: co-contextual type checking.

Co-contextual type judgements form an alternative formulation of a type system. It essentially 'reverses' the flow of the context for a bottom-up approach, as seen on the right in Figure 2.1. This approach is proven to be equivalent to a contextual formulation(Erdweg et al. 2015). In propositional logic, any two equal definitions of a type system are equally

valid, but the type-checking algorithms based on these formulations may have different performance. Co-contextual type checking has been implemented for larger type theories such as Featherweight Java(Igarashi, Pierce, and Wadler 2001), and has been shown to outperform contextual type checkers in some benchmarks(Kuci 2020).

Co-contextual typing judgments are derived from their contextual counterparts through the notion of duality. The correspondence between operations is shown in Table 2.2. Type judgments are of the form $e : T \mid C \mid R$. This means that a term e has a type T if all constraints C are satisfiable and requirements R are met. The requirements are a mapping of variables to their types, similar to a context, but with a different meaning. Informally, a requirement corresponds to what a context *should* contain for a term to be well-typed. Since there is no context, there is no way to determine if a variable is bound, and what its type is. This is how requirements are created, denoting the existence of a binder for said variable and a yet-to-be-determined type (denoted as a placeholder, $?T$). When a binder for a variable is encountered, the identifier can be removed from the requirements as the requirement is satisfied. A consequence of this approach is that two sub-terms may create a requirement for the same identifier. A merge operator solves these conflicts by introducing a constraint that the 'required' types for the same identifier are equal. In its simplest form, all constraints are equalities, but different type systems may introduce more complex constraints.

Contextual	Co-contextual
$\Gamma \vdash e : T$	$e : T \mid C \mid R$
$\Gamma ::= \varnothing \mid x : T;\ \Gamma$	$R \subset x \times T$
$\Gamma(x) = T$	$\{x : ?T\} \in R$
$x : T;\ \Gamma$	$R - \{x\}$
$\Gamma \rightarrow (\Gamma,\ \Gamma)$	$(R,\ C) = merge(R_1, R_2)$
$\Gamma = \varnothing$	$R = \varnothing$

Table 2.2: Correspondence between context and co-context manipulation.

In a contextual setting, a term is well-typed under some context. This means that this 'proof' cannot be reused when the context changes. Conversely, the bottom-up approach does not suffer from this issue as the constraints and requirements remain the same. Instead, all that is required is verification of whether the same constraints and requirements are still satisfiable under the new context. This means the approach is *incremental* in nature; part of the computation is reusable. Similarly, if a definition `foo` uses `bar`, then to (contextually) type-check `foo`, the other definition needs to be checked first, as it needs to be part of the context during the traversal of `foo`. By design, a co-contextual type checker does not require `bar`. Instead, this will leave an open requirement to be filled in later. As a result, these definitions can be processed in parallel, deriving a parallel type checker. In theory, co-contextual type-checking is promising. This thesis explores the viability of such a type-checker for dependently typed languages.

Chapter 3

Related work

This section covers some work regarding dependent type-checking algorithms and performance.

Coquand's Algorithm Thierry Coquand presented a simple type-checker for a dependently typed language, with a formal proof of correctness(Coquand 1996). This algorithm uses normalisation by evaluation (NBE) to obtain normal forms of terms in the language by appealing to their denotational semantics. This has become the de facto standard design of dependently typed elaboration. Two types (terms) are equal if and only if their normal forms are α-equivalent, as to be a normal form is to be *the* canonical representative. This means that when checking if two types are equal, it suffices to normalise both types and check if they are α-equivalent.

SmallTT SmallTT is an implementation of elaboration for a minimal dependently typed language(András Kovács n.d.). It demonstrates a collection of performance optimisation techniques. This is also built on Coquand's algorithm. Apart from algorithmic optimisations, it also includes various low-level optimisations and compiler/runtime system tweaks.

Convertibility Checking Determining whether two lambda-terms are equal up to reductions is a core component of dependently typed languages(Courant and Leroy 2026). This can be achieved by reducing terms to normal form, which potentially requires expensive computations. This paper proposes an algorithm that leverages laziness and concurrency to attempt to avoid fully normalising terms to reach a conclusion.

Heuristics in Unification Unification, in the context of dependent types, is the algorithmic process of solving equality constraints between terms. Dependent types require higher-order unification, which is complex (and undecidable). To improve performance, unification algorithms may use several heuristics to try to point the solver in the right direction. These are often poorly documented or lack any evidence of their usefulness. Furthermore, a large number of heuristics result in unpredictable performance. Ziliani and Sozeau present a new unification algorithm that uses a small number of heuristics that are empirically proven to suffice for many large developments(Ziliani and Sozeau 2017).

Agda Parallel Type Checking At the beginning of 2026, a pull request was merged into Agda that type-checks module imports in parallel[1]. This has proven to drastically reduce the runtime of the type-checker in large, multi-module projects, at the cost of memory usage. This approach for parallelisation is unrelated to dependent type checking as it focuses on module-level parallelisation.

[1] `https://github.com/agda/agda/pull/8332`

Twin Types Danielsson and López Juan have implemented a type checker for *Tog*, a small variant of Agda(López Juan and Danielsson 2020). They reformulate previous work, which introduced twin types and twin variables. The unifier used in this type checker is able to handle some cases that existing type checkers struggle with, while maintaining similar performance.

Chapter 4

Elara

Elara is the programming language at the core of this thesis. The language is a custom implementation of *LambdaPi*(Löh, McBride, and Swierstra 2010). The core language is extended with two inductive data types: natural numbers and vectors.

This chapter covers the language syntax, behaviour, and a formal definition of its type system.

4.1 Syntax

The syntactic structure of a term in Elara is described in Figure 4.1. Definitions may not be mutually dependent or recursive. An example program is given in Listing 4.1.

e : Term	$::=$	e `:` e	type annotation
	$\mid$	`U`	type universe
	$\mid$	$\Pi(x : e).e$	pi type
	$\mid$	x	variable
	$\mid$	$e\ e$	application
	$\mid$	$\lambda x.\ e$	abstraction
	$\mid$	`Nat`	Nat type
	$\mid$	`O`	zero
	$\mid$	`S`	successor
	$\mid$	`natElim`	Nat eliminator
	$\mid$	`Vec`	Vec type constructor
	$\mid$	`Nil`	nil
	$\mid$	`Cons`	cons
	$\mid$	`vecElim`	Vec eliminator

Figure 4.1: Elara syntax.

```
// An identity function
foo :: [a : U] -> a -> a
foo = \a. \x. x

/* foo, specialised to the Nat datatype */
bar :: Nat -> Nat
bar = foo Nat
```

Listing 4.1: Elara Example

4.2 Typing Rules

This section covers the co-contextual type judgements for Elara's type system. These judgements are of the form $e : T \mid C \mid R$. A term e has type T under constraints C and requirements R. The flow of information is clear from the typing judgments shown in this section. The constraints and requirements in the premise are combined accordingly in the conclusion. This is in contrast to *contextual* type judgments; here, the context is expanded upon inside the premise. This difference is illustrated in more detail later.

The constraints in this language are all definitional equalities except for a constraint on function application (explained later). These are used to relate types of different terms. For example, a term e_1 may have some type T_1, but in the term $e_1\ e_2$ (application), a constraint is imposed on T_1: it must be a Π-type.

The requirements are a mapping from variables to their types. Informally, the requirements denote what a *context* must contain for the term to be well-typed. The notation $T\ fresh$ denotes that T is a unique meta-term, which is essentially a placeholder for some real term. Lastly, it is occasionally necessary to merge requirements:

$$
\begin{aligned}
&(R,\ C) = merge(R_1,\ R_2) \\
&\ where \\
&\qquad R = \{x : T \mid (x : T) \in R_1\ \wedge\ x \notin dom(R_2)\} \\
&\qquad C = \{T_1 = T_2 \mid (x : T_1) \in R_1\ \wedge\ (x : T_2) \in R_2\}
\end{aligned}
$$

For every variable with a requirement, one mapping is kept (the first in the sequence of requirements). If multiple requirements contain a mapping for the same variable, a constraint is generated that these types must be equal.

Type Universe The term U represents the universe of types. Concretely, the type of every type is U. This includes the type of U itself. This is static and does not lead to new requirements or constraints.

$$U : U \mid \varnothing \mid \varnothing \quad \text{(CoT-Uni)}$$

Type Annotations The type annotation $e : t$ denotes that a term e has a type t. The inferred type for t must be U, the type of types, as a consequence. This yields the constraint $T_2 = U$. The type inferred for the term e must match the annotated type. This is expressed in the constraint $t = T_1$, where T_1 is the inferred type of e.

$$\frac{e : T_1 \mid C_1 \mid R_1 \qquad t : T_2 \mid C_2 \mid R_2 \qquad (R, C) = merge(R_1,\ R_2)}{(e : t) : t \mid C_1 \cup C_2 \cup C \cup \{t = T_1,\ T_2 = U\} \mid R} \quad \text{(CoT-Ann)}$$

Pi-Types The term $\Pi(x : A).\ B$ represents the type of a (dependent) function with domain A and codomain B. Firstly, both A and B must be types, reflected in the constraints $T_1 = U$ and $T_2 = U$ respectively. Here, T_1 and T_2 are the inferred types of the terms A and B, respectively.

With dependent types, the term B represents a type family. The concrete type (may) depend on x, which is known to be of type A. If B references x, the requirements R_2 will contain x. The first merge, $merge(R_2, \{x : A\})$, adds a constraint that the type B *requires* x to have matches its known type, A. The Π-type binds x, so the requirement for x to exist is satisfied; the requirement for x is safely removed after the merge. The remaining requirements are then merged with the requirements for the type inference of A. Note that the removal of x from the requirements is done before this last merge. The variable x is not in the scope of A, so this avoids creating constraints on a *different* variable x in A in case of name shadowing.

$$\frac{\begin{array}{c} A : T_1 \mid C_1 \mid R_1 \\ B : T_2 \mid C_2 \mid R_2 \qquad (R_3,\ C_3) = merge(R_2, \{x : A\}) \qquad (R_4,\ C_4) = merge(R_1, R_3 - \{x\}) \end{array}}{(\Pi(x : A).\ B) : U \mid C_1 \cup C_2 \cup C_3 \cup C_4 \cup \{T_1 = U,\ T_2 = U\} \mid R_4} \quad \text{(CoT-Pi)}$$

Variables Without context, there is no information available about the type of a variable x. A meta-term is introduced, essentially functioning as a *placeholder* for the actual type. Surrounding terms may introduce constraints on this meta-term. The simplest example is a type annotation; a term $x : U$. x's type is a meta-term, say $?T$. By the type judgment for type annotations, a constraint $?T = U$ is generated, which is a valid substitution for $?T$.

$$\frac{T\ fresh}{x : T \mid \varnothing \mid \{x : T\}} \quad \text{(CoT-Var)}$$

Application For function application $e_1\ e_2$, the function e_1 must have a Π-type. The argument e_2's type must match the domain of the function. Suppose e_1 has type $\Pi(x : A).\ B$, and $e_1\ e_2$ has type T. The conclusion $T = B$ does not follow from this premise. Instead, T represents the type $B(e_2)$; the instance of the type family B where the input is known to be e_2. To represent an *instance* of a Π-type, the notation $\Pi'(e_2 : T_2).\ T$ is used. The constraint containing the Π′-term is not an equality constraint, but a constraint on a Π-type. Concretely, $\Pi(x : A).\ B = \Pi'(e_2 : T_2).\ T \Leftrightarrow A = T_2 \wedge B[x := e_2] = T$. Furthermore, the constraint $T_1 = \Pi'(e_2 : T_2).\ T$ is unsatisfiable if T_1 is not a Π-type.

$$\frac{e_1 : T_1 \mid C_1 \mid R_1 \qquad e_2 : T_2 \mid C_2 \mid R_2 \qquad T\ fresh \qquad (R,\ C) = merge(R_1,\ R_2)}{(e_1\ e_2) : T \mid C_1 \cup C_2 \cup C \cup \{T_1 = \Pi'(e_2 : T_2).\ T\} \mid R} \quad \text{(CoT-App)}$$

Abstraction For abstractions $\lambda x.\ e$, the type of the body e represents the codomain of the function, T_1. There is no information about the domain of the function, so a meta-term T_2 is introduced. If x occurs in the body, the requirements R_1 will contain a mapping for its type. The merge yields a constraint that this type is equal to T_2. Since the requirement for x is satisfied (i.e. the abstraction 'defines' x), it can then be removed from the requirements.

This most clearly demonstrates the information flow. In a contextual setting, the term would be typed under some context Γ, and this context would be expanded in the premise to contain x when typing e. Here, however, the type of e is inferred without requiring the introduction of extra information. The resulting constraints and requirements from the inference are then manipulated appropriately. In general, the constraints and requirements present in the conclusion are *constructed* from the constraints and requirements of the subterms. A context would be passed down to the subterms, potentially with modifications.

$$\frac{e : T_1 \mid C_1 \mid R_1 \qquad T_2\ fresh \qquad (R,\ C) = merge(R_1,\ \{x : T_2\})}{(\lambda x.\ e) : \Pi(x : T_2).\ T_1 \mid C_1 \cup C \mid R - \{x\}} \quad \text{(CoT-Lam)}$$

4.2.1 Inductive Data Types

Elara expands on the core LambdaPi language by introducing two inductive data types: natural numbers and vectors. This expands the core language with terms for the type, constructors and eliminator of these data types. The types of these terms are constant, with no requirements or constraints.

Peano Natural Numbers

The Peano axioms define the arithmetical properties of natural numbers, the set $\mathbb{N}$(Kennedy 1973). The non-logical symbols for the axioms consist of a constant 0 and a unary operator S. Informally, this translates to the constructors O and $S\ n$. The former represents the natural number zero, and the latter the successor of a number n, $n + 1$. Together, these represent the natural numbers. For example, the number 4 can be defined as $S\ (S\ (S\ (S\ O))) :: Nat$. As the language does not support pattern matching or recursion directly, the eliminator for the Nat data type must be used: $natElim$. This is a catamorphism, consuming the inductive data.

Vectors

The vector type is, loosely speaking, a list with its size and element type encoded in the type of the vector. For example, the type $Vec\,T\ n$ is the type of an n-element vector, with elements of type T. Its type and constructors are similar to those of the standard $List$ type as defined in Haskell. Since the size is a value (of type Nat), and not a type, this is a textbook example of a data type that uses dependent types. This type can be used, for example, to write a safe $head$ function to retrieve the first element in the vector. Leveraging the size of the vector in the type, the function can require the input to have at least one element, $head : Vec\,T\ (S\ n) \to T$. In this example, T and n are implicit for clarity. The downside is that the type-checker must now know if the input to any $head$ call is a non-empty vector.

4.3 Type Inference

For a term to be well-typed, all the constraints need to be satisfiable. An expression like $\lambda x.x : U$ would yield an absurd constraint. In this example, the inferred type for the abstraction is a Π-type. This type needs to be equal to the annotated type U. This is visible in the generated constraint $\Pi(x : A).\ B = U$, which is unsatisfiable. Furthermore, the type may not contain any meta variables. By the typing rules, the term $\lambda x.x$ has type $\Pi(x : ?1).?2$ under the constraint $?1 = ?2$. This constraint is satisfiable by defining $?2$ to be the same as $?1$, but the type $\Pi(x : ?1).?1$ is not grounded; it still contains $?1$. An interesting side-effect is that through the constraints, the term $\lambda x.(x : T)$ is well-typed, yielding $\Pi(x : T).T$. This term's type is not inferable using LambdaPi's bidirectional type checker, as the lambda is a *checked* term.

Algorithmically, to infer the type of a term, the typing rules are applied recursively to the child nodes in the syntax tree. This yields the inferred type, constraints and requirements. These constraints are then processed. If any constraint is not satisfiable, or the requirements contain definitions that are not found in the context (i.e. other definitions inside a file), the term is not well-typed.

As a consequence of this approach, type *checking* is essentially the same as type inference. With type inference, the algorithm *generates* the type of a term, and with type *checking*, the algorithm verifies if a term has a given type. In a co-contextual setting, type checking is achieved merely by adding a constraint on the inferred type.

In Elara, these type judgements are expanded upon. Firstly, a distinction is made between local and global variables. The motivation is that eagerly using type signatures and definitions of other (global) variables improves performance, as constraints may be solved earlier during traversal. This does mean that Elara requires a context to look this information up from, making the actual type system a hybrid between co-contextual and contextual type systems. This does not mean that definitions need to be checked in a topological order. This is further expanded upon in chapter 6.

Secondly, the type-checking algorithm attempts to eagerly solve constraints. This is done using a function $solve : C \to \sigma \times C^- \times C^?$. This attempts to solve all constraints C, splitting them up in unsatisfiable constraints C^- and 'stuck' constraints $C^?$ that cannot be progressed at this time. It also returns a substitution for all 'solved' meta terms σ. By the nature of co-contextual type systems, these substitutions can be applied directly. Note that with this approach, unsatisfiable constraints are accumulated during traversal instead of a fail-fast approach.

These changes are reflected in the full type judgments in Appendix A. The judgments take the form $e : T \mid C^- \mid C^? \mid R$.

4.4 Constraints

Constraints in this type system are term equalities. A majority of these constraints can be solved through normalisation by evaluation. The cases discussed here are the Π' terms denoting instances of Π-types, constraints containing meta terms, and special terms like constructors and eliminators for inductive data types.

Π' Constraints A constraint of the form $e_l = \Pi'(e_r : T_1).\ T_2$ is only satisfiable if e_l is a Π-type. Suppose it is of type $\Pi(x : A).B$. The constraint is only satisfiable if and only if $A = T_1 \ \wedge \ T_2 = B[x := e_r]$. Note that since Π' does not represent a 'real' term, it may not be used as a substitution (e.g. if the constraint is of the form $?T = \Pi'(e : A).\ B$.

Meta Terms In general, a constraint of the form $?T = e$ yields a substitution for the meta term $?T$. The substitution leaves a trivial constraint, $e = e$, which holds by reflexivity. Under some circumstances, such a constraint may be stuck until other constraints are solved:

- e contains $?T$. A circular definition is not allowed as a substitution.
- e is a Π' term.

Type Constructors, Constructors and Eliminators The special terms `S`, `natElim`, `Vec`, `Nil`, `Cons`, and `vecElim` are all injective by definition. This can be leveraged: any constraint of the form $f\ x = f\ y$, with f being one of these terms, can be reduced to the constraint $x = y$. This is generalised: $Vec\ x_1\ x_2 = Vec\ y_1\ y_2 \Leftrightarrow x_1 = y_1 \wedge x_2 = y_2$.

4.5 Implementation

To implement the type checker, the following scaffold is used:

1. $context : \Gamma$: A static context of *global* definitions. This is used to access their signatures and bodies.
2. $fresh : Stream(\mathbb{N}) \to \mathbb{N} \times Stream(\mathbb{N})$: A function to get the next symbol in a symbol stream. This is used for generating unique meta terms. The state management of the stream is omitted here for simplicity.
3. $infer : e \to T \times C^- \times C^? \times R$: A function to infer the type of any term in the language. It also returns the set of unsatisfiable constraints, 'stuck' constraints and the requirements for this inference.
4. $solve : C \to \sigma \times C^- \times C^?$: A function that attempts to solve a set of constraints. A constraint is either solved (removed), unsatisfiable (included in C^-), 'stuck' (included

in $C^?$) or a solution to a meta term (included in the substitution σ). In the case a constraint is propagated yielding further constraints, the original constraint is removed, and the new constraints are processed.

5. $finalize : C^- \times C^? \to \sigma \times C^-$: A function that attempts to solve all remaining constraints. This function is similar to $solve$, except that all 'stuck' constraints are marked as unsatisfiable. This is used as a last attempt to solve constraints. This is included because it is possible some constraints get 'unstuck' if, for example, some meta term gets resolved.

6. $merge : R \times R \to R \times C$: The function to merge two requirement mappings, as described earlier.

The general layout to infer the type of a term is described in Algorithm 1. Recursively, every child node is traversed. Next, meta terms are generated. This is only needed for some terms, like local variables. The requirements of all the child nodes are merged. For some terms, such as abstractions, additional requirements are introduced. Some requirements may also be removed if they are satisfied, and the merge order can be important. This is explained in more detail for the Π-type type judgement. The constraints generated from requirement merging, as long as any additional constraints (depending on the term), are then attempted to be solved. Finally, everything can be put together to return the inferred type, constraints, and requirements. The substitution yielded from solving the constraints is applied directly here too.

Algorithm 1 Incomplete Type Inference

function INFER(e)
  for each child node e_i of e **do** ▷ Recurse over the child nodes
    $T_i,\ C_i^-,\ C_i^?,\ R_i \leftarrow$ INFER(e_i)
  end for
  $U \leftarrow$ FRESH ▷ Generate meta terms if needed
  $R,\ C \leftarrow$ MERGE($R_0, \ldots, R_n$) ▷ Merge requirements
  $\sigma,\ C^-,\ C^? \leftarrow$ SOLVE(C) ▷ Solve new constraints
  $T \leftarrow \ldots$ ▷ Create return type
  return $\sigma(T), \sigma(C^- \cup \bigcup C_i^-), \sigma(C^? \cup \bigcup C_i^?), \sigma(R)$
end function

This function is incomplete; it is missing the $finalize$ step, and some validation. The result is *partial*. The complete version is given in Algorithm 2. There should be no unsatisfiable constraints, and no unsatisfied requirements.

Algorithm 2 Complete Type Inference

function INFER-COMPLETE(e)
  $T,\ C^-,\ C^?,\ R \leftarrow$ INFER(e)
  $\sigma,\ C \leftarrow$ FINALIZE($C^-,\ C^?$)
  if $C \neq \varnothing$ **or** $R \neq \varnothing$ **then**
    return $error$
  else
    return $\sigma(T)$
  end if
end function

4.5.1 Constraint Solving

To solve a constraint, the terms in the equality are first reduced to some notion of a weak-head normal form to try to avoid unnecessary computation. In this step, type annotations (in the head position) are stripped, and function application (including eliminators for the inductive data types) is performed. Depending on whether incremental type checking is enabled, global definitions are unfolded as well.

Trivial reflexive constraints, like $U = U$, are solved directly. Other constraints are solved gradually. For example, the constraint $\Pi(x : A_1).\ B_1 = \Pi(y : A_2).\ B_2$ propagates the constraints $A_1 = A_2$ and $B_1 =_\alpha B_2$.

The 'special' cases are handled as described in section 4.4. Any constraint that cannot be progressed (but may still be feasible) is marked as stuck. An example of such a constraint is $x_g = y_g$, an equality of two global definitions.

Any other constraint is marked unsatisfiable. This includes constraints like $U = Nat$.

Chapter 5

Parallelization of Elara's Type Checker

Parallel computing is a type of computation in which many calculations or processes are carried out simultaneously(Almasi and Gottlieb 1989). If a large problem can be split into smaller, independent tasks, these tasks can be solved at the same time. Instead of solving these tasks sequentially, solving them at the same time by utilising multiple Central Processing Unit (CPU) cores decreases the total runtime. Multicore architectures are widespread as of today, so making more efficient use of the extra computing resources can increase performance.

This chapter covers the approach for the parallelisation of Elara's type-checker. Elara's type checker has two parts: an algorithm that handles multiple definitions in a module, and an algorithm that handles a single definition. The parallelisation of these parts is discussed in the next sections. Their efficiency will be measured in terms of a speed-up(Hennessy and Patterson 2007).

5.1 Module-level type-checking

Parsing a file yields a list of definitions (barring I/O or parse errors). These definitions may reference each other and may be declared in any order. No definitions may have the same name, and recursion is not supported. This also means cyclical definitions are not allowed.

Type-checking a single definition requires the other definitions it depends on (and transitive dependencies) in a contextual setting. Typically, definitions are processed and added to the context in topological order with respect to the other definitions a definition may use to guarantee the context contains the required entries. This allows for *some* parallel type checking, but does enforce a strict partial order.

When inferring the type of a variable in a co-contextual setting, no context is required. This means that all definitions in a file may be checked concurrently, regardless of their interdependence. Because the language does not support recursion, the dependency graph must still be checked for cycles.

Parallelisation of module-level type-checking (or type-checking multiple files) is not new. There are many approaches, not limited to co-contextual type-checkers or dependent types. A more sophisticated implementation, like using scope states (Antwerpen and Visser 2021), is outside of the scope of this project.

5.2 Definition-level type-checking

When traversing a syntax tree to type-check expressions, it becomes clear why (in theory) co-contextual type-checking algorithms are well-suited for parallelisation.

Consider the following typing judgment from *LambdaPi*:

$$\frac{\Gamma \vdash e_1 :\uparrow \Pi(x : A).B \qquad \Gamma \vdash e_2 :\downarrow A \qquad B[x := e_2] \Downarrow B'}{\Gamma \vdash e_1 e_2 :\uparrow B'} \quad \text{(T-App)}$$

Note that since its type-checking algorithm is bidirectional, the inferred and checked types are differentiated by $\uparrow$ and $\downarrow$ respectively. $\Downarrow$ is the evaluation relation.

Inspecting this rule, it states applying expression e_2 to the expression e_1 returns a type B' under context Γ. This only holds if and only if the e_1 is a function type, and the argument's type must match the function's domain (A).

In the implementation, first the type of e_1 is inferred. This yields the domain A, which is then used to check the type of e_2. Clearly, e_2 cannot be type-checked by this algorithm before e_1 is. In the co-contextual setting, their types can be (partially) inferred completely independently. Then, a constraint is added to show the relation between their types. This is not just for function application, but a general rule for co-contextual type checkers: a term's children can all have their types inferred independently. This is very similar to the divide-and-conquer algorithm design paradigm, where each child in the syntax tree is an independent sub-problem. Analogously, the constraints that get added after inferring these types are 'combining the partial solutions to provide a solution to the original problem'.

To obtain a parallel version of this 'divide-and-conquer' algorithm, the fork-join model is used(Lea 2000). Essentially, if a problem is sufficiently large, the task is divided into sub-problems. These are then solved in parallel while the original process is suspended, waiting for all the spawned tasks to finish. Once this is the case, the results can be combined to yield the solution to the original problem. This same process is applied to the sub-problems, dividing the task if it is large enough, et cetera. In the case of Elara, subprocesses are only created when inferring the type of expressions containing multiple sub-expressions. These expressions are type annotations, Π-types and function application, and they all have exactly two sub-expressions.

There are some things to keep in mind with this approach in parallelisation:

1. The size of expressions is computed in the preprocess step as discussed in section B.1 to make finding the size of a term $\mathcal{O}(1)$. Traversing the syntax tree to compute this when needed introduces computational overhead.
2. Only a few types of terms can benefit from this parallelisation, as mentioned previously.
3. The balance of the syntax tree greatly affects the speed-up.
4. The generated constraints are still processed sequentially. Two tasks may process their respective set of constraints in parallel, but a more sophisticated implementation may parallelise this processing step too.
5. Multiple definitions may be type-checked in parallel. This potentially skews the performance of type-checking a single definition as it consumes resources.

5.3 Implementation

To allow for parallel type checking, two primitives are introduced. One to create a parallel task, `async`, and another to block until a given parallel task is completed and get its result, `wait`. During traversal of the syntax tree, these primitives are used to process child nodes in parallel.

On top of this, a semaphore is used to keep track of the number of running threads, bounding the parallelism. Large terms may try to spawn a vast number of threads if uncapped; this is bounded to avoid overwhelming the task scheduler. The semaphore is initialised with the number of capabilities of the Haskell run-time system. If no resources are

available before a call to `async`, the action is computed sequentially instead of waiting for resources to become available. Care is taken to release the resources back to the semaphore in case of an error. Bounding the parallelism is mainly intended for parallelisation at the definition level. For completeness, this is extended to the module-level type checker, so that if multiple definitions are type-checked in parallel, they still share resources. The module-level type checker for LambdaPi does not use semaphores for its parallelisation.

In contrast to the standard fork-join model, to avoid spawning an extra process while the caller is suspended, instead of suspending the caller, it infers the type of one of the sub-expressions itself. Furthermore, spawning additional processes only happens when the sub-expressions are sufficiently large. Even though Haskell's concurrency uses green threads (Marlow 2011) and the API is also purposely lightweight (Peyton Jones, Gordon, and Finne 1996; Li et al. 2007), it still introduces overhead. Consider the following expression: $\lambda f.\ \lambda x.\ f\ x$. The inference application $f\ x$ can technically be split into two tasks: inferring the type of f and inferring the type of g. By the inference rules, this only requires generating a fresh meta variable for both their types, and there are no constraints to solve. This computation is so trivial that it is not desirable to parallelise it. To decide if the sub-expressions are large enough, a rudimentary metric is used: the size of the syntax tree. A complex heuristic to guess if an expression is computationally hard to type-check is a topic in and of itself. In the implementation, both sub-expressions must contain at least 32 nodes to be considered 'split-worthy'. This value was picked by trial-and-error on a system with 16 logical cores.

A third primitive is used for module-level parallelisation, spawning a thread for every task in a sequence and waiting for all of them to finish. Using this, every definition is allotted its own thread that can run concurrently. Through the use of a Haskell `MVar`[1], threads can wait on and signal each other. This is used to suspend the spawned threads until all the dependencies are type-checked.

Caution is taken to ensure these processes do not deadlock. Circular definitions would end up waiting indefinitely on each other to finish type-checking. Cycle checking is done beforehand to allow raising an error before the tasks are created.

These primitives enable parallelisation with minimal changes to the base algorithm. In the implementation, `async` spawns an additional thread every time it is used. While care has been taken in Haskell to reduce overhead when spawning additional threads, the overhead itself can still be significant compared to the cost of type-checking. An approach that reuses the same threads, a thread pool for example, allows reusing the threads. Another issue is that resources are distributed on a 'first-come, first-served' basis. This is unfair and potentially forces the tasks that benefit most from concurrent computation to run sequentially.

[1] Analogous to a single-element blocking queue

Chapter 6

Incrementalisation of Elara's Type Checker

In theory, a co-contextual type checker's results may be reused as long as the original expression remains the same. This also means that two different expressions containing the same term can share the result of inferring the type of said term. This fine-grained reuse of computation is not implemented in Elara. Instead, the type checker considers changes in top-level definitions only.

Consider a module containing two independent definitions: `foo` and `bar`. If changes are made to `foo`, `bar` does not need to be checked again. This notion of incremental type checking also works for contextual type checkers. Now consider the case where `bar` uses `foo`. In this case, the context under which `bar` is type checked is changed, so even though `bar` does not change, it needs to be rechecked. In a co-contextual setting, this would not be the case.

Recall that Elara's type system is a hybrid, however, requiring a context of global definitions. Does this mean that `bar` needs to be rechecked, just like a contextual type checker would? Consider the (Church-encoded) proof that $0+n = n$ in Listing 6.1. The type checker uses the type of `plus` to check if `plus 0 n` is well-typed, not the definition (every definition is required to have a type signature). On the other hand, the constraint $plus\ O\ n = n$ requires unfolding the definition of `plus` and does not use the type signature. Dependent type systems require both the types and the definitions for type checking.

```
CId :: [a: U] -> a -> a -> U
CId = \a. \x. \y. ([p: a -> U] -> p x -> p y)

crefl :: [a: U] -> [x: a] -> CId a x x
crefl = \a. \x. \p. id (p x)

plus :: Nat -> Nat -> Nat
plus = natElim (\p. (Nat -> Nat)) (\q. q) (\x. \f. \y. S (f y))

zeroPlus :: [n: Nat] -> CId Nat (plus O n) n
zeroPlus = crefl Nat
```

Listing 6.1: Proof for $0 + n = n$.

The incrementalisation of Elara builds on the assumption that type signatures change less frequently than the definitions themselves. The type-checker will look up signatures of global definitions, but avoid unfolding these definitions to solve constraints. The unfolding is delayed until the final call to the solver. This way, this intermediate result may be reused between iterations (given that the signatures of used functions do not change).

Going back to the `foo`/`bar` example where `bar` uses `foo`. Elara's type-checker will store the intermediate result. If `foo`'s type signature and/or definition is changed, the type-checker

must reprocess it. On the other hand, `bar` can reuse the intermediate result from the previous iteration if only `foo`'s definition changed. Changes in `bar` do not affect `foo`, so in this case the intermediate results for `foo` can be reused.

This approach has some issues:

1. Performance hinges on the signatures not being changed. This approach assumes this is less likely to happen than changes in the definition. Furthermore, this 'trick' works as the language requires an explicit type to be provided through a signature. If instead the signature was inferred from the definition, this does not work.

2. Delaying unfolding potentially makes type-checking slower. In a sequential setting, this is irrelevant, as this would merely change the order of computation. Combined with parallelisation, however, delayed unfolding means a constraint may not be solved concurrently.

3. A large part of the computation consists of evaluating parts of the program. This may be locked behind unfolding definitions like the $plus\ O\ n = n$ example. The result of such computation is not stored in the intermediate result; future iterations will still need to perform this computation. This can be somewhat mitigated by reusing computation in the case that the definition also does not change between iterations, but this adds significant complexity.

6.1 Implementation

As described, the type-checker reuses the constraints and requirements generated by previous iterations if and only if the signatures of the global definitions used are unchanged. Note that this requires the definitions not to be unfolded when processing constraints, as the goal is to be able to reuse these constraints even if the definition changes. This does not necessarily mean constraints referencing global variables are stuck, e.g. $x_g = x_g$ is still solvable. The constraints that require unfolding are marked as stuck.

Algorithm 3 Incremental Type Check

```
1:  procedure Infer(cache, definition)
2:      if cache contains defintion then
3:          partial ← Lookup definition in cache
4:          if partial is not valid then
5:              partial ← Compute partial result for definition
6:              Put (definition, partial) in cache
7:          end if
8:      else
9:          partial ← Compute partial result for definition
10:         Put (definition, partial) in cache
11:     end if
12:     result ← finalize partial
13:     return result
14: end procedure
```

The procedure is described in Algorithm 3. When type-checking a definition, a partial result is fetched from a cache. If a partial result is found, it verifies two properties:

1. The partial result is for an identical definition (same type signature and body). In the implementation, this equality must be exact; α-equality is not considered.

2. None of the dependencies have a different signature.

If these conditions are not met, the partial result represents an invalid solver state (w.r.t. the new input) and thus needs to be recomputed. If a new partial result is computed, it is stored in the cache for a future iteration. In addition to the inferred type, constraints and requirements, this partial result also contains the solver state and exact definition. The state is required to continue the computation, and the definition is needed for the validity checking.

The actual type-checking algorithm is almost identical to its non-incremental counterpart, with a 'breakpoint' before `finalize`. The key difference is that the initial type inference (before `finalize`) does not unfold global definitions and only considers their signatures. The partial result is essentially a snapshot of the state at this point.

When module-level parallelisation is enabled, these partial results may be computed concurrently (if needed), and are not required to be processed in a topological order. Then, in topological order, the remaining constraints are solved for each definition. This ensures behavioural consistency with the non-incremental version. If a dependency is not successfully type checked, this definition is skipped. Note that even in this case, the partial result is stored in the cache to be reused.

Chapter 7

Evaluation

This chapter covers the performance comparison between algorithms. The approach and the results are listed, alongside an analysis of the results.

7.1 Benchmarks

Finding appropriate benchmarks has proven difficult. The provided benchmark programs provide definitions of various complexities using all the terms provided by the language. No claims are made regarding how well these programs represent real-world code.

Many different factors may influence the benchmarks. The frequency of different terms in the language, the size of a definition, and the computational complexity of the terms all play a role. Writing benchmarks that accurately represent real-world scenarios is non-trivial. In particular, estimating how 'difficult' a term is to type-check is an open problem. Alternatively, porting existing dependently-typed code to Elara would provide a solid foundation. Transpilation[1] is out of the scope of this project, however, due to the challenges that come with it. Elara is not 'batteries included'; many features seen in common dependently-typed programming languages are not supported by Elara.

The benchmarks were performed on a system running Windows 11, with an AMD Ryzen 7 7800X3D CPU and 128GB DDR5-6000 CL30 RAM. The benchmark times are the average completion time across a thousand consecutive runs unless specified otherwise. This is done to reduce variance. While running the benchmarks, no other programs were running (excluding background processes) to be able to access most of the available computing resources.

The files used for benchmarking can be found in the `elara` directory inside the root directory of the source code. Next is a description of their contents.

church.elara This file contains various Church-encoded data types(Church 1941). In particular, booleans, natural numbers, empty, unit, identity, product, and sum types. Alongside their definitions are some constructors and operators on these data types. The file concludes by 'proving' false using the inconsistency introduced by type-in-type semantics, via Hurken's paradox(Hurkens 1995).

nats.elara This file uses the built-in `Nat` data type, and uses its eliminator to write proofs by induction and recursive definitions. Several common operations are implemented, such as successor/predecessor operators, addition, subtraction, and multiplication. Furthermore, it proves associativity and commutativity of addition using a Church-encoded identity type, and some auxiliary lemmas.

[1] Source-to-source compilation

vecs.elara This file uses the built-in `Vec` data type. It defines some operations on this data type, such as appending an element, concatenating vectors, applying a function on the elements of a vector, and some folds over the elements.

stlc.elara This is an Elara port of the simply-typed lambda calculus benchmark included in SmallTT. This provides a Church-encoded simply-typed lambda calculus with the following types: natural numbers, top and bottom, products and sums. Strictly speaking, such a lambda calculus should only have one type constructor, the function type. Note that although the expressions are identical, the results are not comparable to SmallTT's benchmarks. The main reason for this is that SmallTT uses implicit arguments (which Elara does not support), adding additional complexity.

asymptotics.elara This is another Elara port of a benchmark included in SmallTT. Similarly to `stlc.elara`, the results are also not comparable to SmallTT's benchmarks. Its purpose in SmallTT is to measure elaboration speed asymptotics. Here, on the other hand, it is used to measure the type-check time of 'large' terms. This particular benchmark requires significantly more time type-check. Instead of a thousand iterations, these are run ten times only.

7.2 Results

Performance is analysed in four categories. The baseline performance represents the base implementation of the algorithm. No additional flags are enabled. Next, the two different approaches to parallelisation are compared. Lastly, incremental performance is discussed.

7.2.1 Baseline Performance

This section covers performance measurements of the type-checker without any additional features. The results are shown in Table 7.1. In general, Elara's type checker performs worse than LambdaPi's. However, the performance remains in the same order of magnitude. The exception is `asymptotics.elara`, which Elara really struggles with.

File	LambdaPi	Elara	$T_{Elara}/T_{LambdaPi}$
church.elara	0.41ms	0.72ms	1.84
nats.elara	0.33ms	0.99ms	3.05
vecs.elara	0.10ms	0.42ms	4.20
stlc.elara	16.90ms	22.34ms	1.32
asymptotics.elara	3821.87ms	145461.78ms	38.06

Table 7.1: Baseline performance metrics.

7.2.2 Module-Level Parallelisation

In Figure 7.2, the (individually) normalised performance across all the benchmarks is shown. The full results are available in Appendix C. The approach to this parallelisation is identical across both type-checkers. This is clearly demonstrated, as the curves follow the same trend. Elara's type-checker gains a greater speed-up from this approach, but does not outperform LambdaPi's type-checker.

Looking at the graph, it is also clear that the performance gain bottoms out. The limited number of definitions per file, combined with the added constraints of the dependency

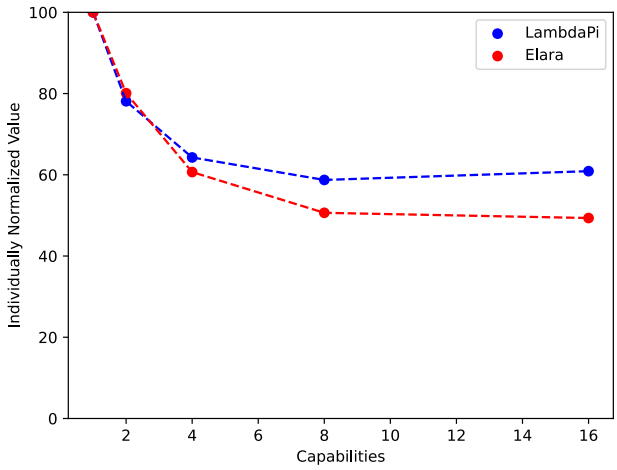


Figure 7.2: Individually-normalised results for module-level benchmarks.

graph, means that only a few cores are used. By, for example, copy-pasting the same definitions multiple times to artificially increase the number of definitions that can be checked concurrently, all cores can be used.

Note that in theory, the performance for $N = 1$ should be similar to the sequential (baseline) performance. In practice, there is a small discrepancy; the baseline performance is slightly better. Most likely, this is caused by the concurrency approach. Many threads are created, but only one is run at a time, whereas in the sequential setting, no threads are created.

7.2.3 Definition-Level Parallelisation

This section covers performance measurements of definition-level parallelisation. In these runs, module-level parallelisation is disabled, and timings are reported per definition rather than per file. The performance is not compared to LambdaPi as it does not have an equivalent form of parallelisation.

This analysis focuses on the definition `mul` from `stlc.elara`. This definition has both a large syntax tree and is relatively expensive to type-check. The definitions in `asymptotics.elara` are not considered here as these are less realistic. The other definition, `bench1` from `bench1.elara`, is purposely designed to exploit this approach to parallelisation. This term is a perfectly balanced binary tree of height 10, generated recursively from $e = e \cdot e$. The 2^{10} leaves of the tree are a constant, 42. The numbers and multiplication are Church-encoded. In Table 7.3, the timing and speed-up of these single definitions are given, alongside the number of capabilities N.

For `mul`, the performance stays virtually the same. Profiling shows that it *does* parallelise part of the traversal. The parallelisation clearly works for `bench1`, with a maximum speed-up of 2.19, although still far from the theoretical optimum.

7.2.4 Incremental Performance

To check incremental type-check performance, four cases are considered. The benchmarks are solely run on `stlc.elara`, as this file contains the most definitions. The benchmarks are executed with $N = 1$ and $N = 16$. First, the base case. On a first run, the cache is empty, so no partial results are available for reuse. When incremental type-checking is enabled, global

File	Definition	N	Time	Speed-up
stlc.elara	mul	1	2.67ms	N/A
stlc.elara	mul	2	2.70ms	0.99
stlc.elara	mul	4	2.84ms	0.94
stlc.elara	mul	8	2.77ms	0.96
stlc.elara	mul	16	2.53ms	1.06
bench1.elara	bench1	1	1.35ms	N/A
bench1.elara	bench1	2	1.07ms	1.26
bench1.elara	bench1	4	0.83ms	1.62
bench1.elara	bench1	8	0.62ms	2.19
bench1.elara	bench1	16	0.73ms	1.86

Table 7.3: Results for definition-level parallelisation.

definitions are not directly unfolded. Reflexive constraints of the form $x_g = x_g$ may still be solved. The other cases are run *after* the initial type-check, with access to the partial results.

1. No changes: the same source code is fed to the type-checker. This case serves two purposes: verifying if the incremental approach works, and as a baseline to compare the other cases to.

2. A change is made in the *definition* of a node in the dependency graph with a large degree. This means that many other definitions depend on this definition. A traditional contextual approach would have to recheck these definitions.

3. A change is made in the *definition* of a node in the dependency graph with degree zero (i.e. a leaf node). No definitions depend on this definition.

File	Description	N	Time	$T/T_{first\ run}$
stlc.elara	First run	1	9.19ms	N/A
stlc.elara	First run	16	2.90ms	N/A
stlc.elara	No changes	1	7.06ms	0.77
stlc.elara	No changes	16	2.48ms	0.86
stlc.elara	root_change.patch	1	7.14ms	0.78
stlc.elara	root_change.patch	16	2.54ms	0.88
stlc.elara	leaf_change.patch	1	7.11ms	0.77
stlc.elara	leaf_change.patch	16	2.61ms	0.90
bench1.elara	First run	1	7.9384991ms	N/A
bench1.elara	No changes	1	0.0703861ms	0.01

Table 7.4: Results for incremental type-checking.

In Table 7.4, the results of the benchmark is shown. An excerpt of the applied changes can be found in Appendix C. The base cases are the slowest, as is expected. Strangely, it greatly outperforms the non-incremental performance, when similar or slightly worse is expected. The only difference is the lack of unfolding of definitions. It is clear that the implementation is too eager with unfolding, introducing a lot of additional work.

The incremental cases do show a performance increase, but less than one may expect. A large part of the computation time is dedicated to evaluating parts of the program. With this approach to incremental type-checking, much of this computation is 'locked' behind the unfolding of definitions. This means the partial results to generate the constraints and

requirements are inexpensive in comparison. This is demonstrated by reusing the same synthetic benchmark `bench1.elara` used for the definition-level parallelisation benchmark. This contains large terms that do not use dependent types. As such, little to no unfolding of definitions (or evaluation) is required. When rechecked without changes, the performance increase is clear, less than 1% of the first run.

The performance increase itself is consistent across the different cases. This is as expected. Co-contextual type-checkers do not care about the relative order of the definitions in the dependency graph, which the results support.

Lastly, this approach gains a greater speed-up as N increases than the non-incremental definition-level parallelism. This is because all definitions are traversed concurrently, rather than in a topological order.

7.3 Discussion

This section analyses the results and answers the research questions listed in chapter 1.

Is a co-contextual type-checking algorithm feasible for dependently typed languages? Co-contextual type checking has proven to work for dependent types. Unfortunately, at a base level, Elara performs worse than LambdaPi. Elara especially struggles with `asymptotics.elara`. Profiling the type-checker shows no significant increase in the number of constraints with respect to the size of the terms. The constraints themselves are more complex to solve. On further inspection, Elara outperforms LambdaPi on the `test1` benchmark, but performs significantly worse on `test2` and `test3`. The latter one in particular performs poorly, taking up over 80% of the total time. Interestingly enough, this definition performs significantly better than the other ones for LambdaPi.

To what extent can such an algorithm be parallelised? In this work, two different approaches were implemented for parallelisation.

Module-level Parallelisation The module-level parallelisation works as expected, but this approach is also applicable for contextual type-checking. As such, the relative performance improvements are very similar. Interestingly enough, the co-contextual type checker gains a greater speed-up. This is likely due to its overall poorer performance.

Definition-level Parallelisation The timing remains relatively stable, even though in theory there should be a speed-up. The discrepancies can be chalked up to variance or overhead, as profiling shows parallel computation does occur. The number of opportunities can be increased by decreasing the size cut-off, enabling parallel computation of smaller terms. This has proven to worsen the performance, most likely due to the overhead of creating and scheduling more threads for smaller computations.

This does not mean this approach is a dead end. As shown, it *can* work. Furthermore, the syntax tree for Elara is at best a binary tree. Different languages may have nodes with a larger degree[2]. Such languages would offer more opportunities for this granular parallelisation.

Elara's implementation can also be improved. Firstly, the constraints need not be solved sequentially, but can also be parallelised, too. Secondly, submitting work to a service distributing the load over a thread pool reduces thread creation overhead.

The main problem remains, however: when is it optimal to traverse part of the syntax tree in parallel? In the implementation, the number of nodes in the syntax tree is used as

[2]For a given node, its number of children

a heuristic for the computation cost. This, combined with bounding the parallelism and a minimum size requirement, make for a primitive estimation.

Lastly, the potential benefits may be minor in comparison to per-definition or per-file parallelisation. Instead of allocating more computing resources for a singular definition, larger projects may see a greater benefit by allocating them elsewhere.

To what extent can such an algorithm be made incremental? It is possible to create a co-contextual incremental type-checker. Its main theoretical benefit, namely that dependencies between definitions are irrelevant to the incremental performance, is supported by the results. Unfortunately, the approach does not yield as large a benefit as expected. There is room for improvement, however. A more sophisticated data structure for the partial results may, for example, track both the base constraints, and their progressed versions where definitions are unfolded. Then, if a definition that occurs in a constraint remains unchanged, the progressed constraints may be used rather than the base version. This way, more computation can be reused.

How does its performance compare to contextual type-checking algorithms? In most tested metrics, the co-contextual type checker performs worse than LambdaPi's bidirectional contextual type checking algorithm. The performance difference is generally in the same order of magnitude. This, combined with the numerous ways to improve the implementation of Elara, means that this approach *does* have potential, and is worth looking further into.

Chapter 8

Future Work

Elara demonstrates a co-contextual approach to type-checking dependently typed programs. While it correctly type-checks some examples, there is no strong guarantee of correctness. A strong mathematical foundation is a requirement for theorem provers, a common use case of dependent types.

Furthermore, the language is a minimal working example of a co-contextual type system supporting dependent types. Some features, like universe hierarchy and Σ-types are left out. Other practical language features are also missing. Implicit arguments, user-defined types, (dependent) pattern matching, and recursion are desirable in production languages. Lastly, better error messaging and typed holes are nice to have.

Extending the base language with such features will undoubtedly increase complexity. The feasibility and performance analysis of such additions is left for future work. Ultimately, the aim is to retrofit a co-contextual type checker in existing dependently-typed programming languages like Agda if a feature-complete version is proven to outperform existing contextual implementations.

The performance of Elara's type-checker itself has been shown to be comparable to LambdaPi's original type-checking algorithm. This research has listed various design issues, which, if addressed, potentially allow it to outperform the original algorithm. An alternative route to increase performance is finding some balance between contextual and co-contextual type inference, as these different approaches have different strengths and weaknesses. Further research could uncover better ways to type-check.

The main bottleneck of the incremental version of Elara's type-checker is its conservative approach. Currently, constraints are reused between iterations, meaning that no additional traversal of the terms is required. It was shown, however, that the majority of the computation time is used in unfolding definitions and normalising constraints after these constraints are generated. Even if the definitions themselves did not change, this computation needed to be redone. If a definition is known to not change, this computation can be reused. A more sophisticated data structure that can yield further processed constraints based on what definitions changed between iterations could be a solution.

Chapter 9

Conclusion

Type systems supporting dependent types are the de facto standard for proof assistants. This is their main real-world use case. The expressiveness of these type systems comes at a cost: type checking carries significant computational complexity.

This work explored an alternative formulation of a type system supporting dependent types: a co-contextual type system. It introduces the first co-contextual dependent type system and accompanying type-checking algorithm. The implemented algorithm was adapted to support both parallel type-checking and incremental type-checking. Lastly, the performance of the type-checker was compared to a more traditional contextual algorithm. Specifically, the reference implementation of LambdaPi's contextual bi-directional type-checker.

In the end, neither approach performed significantly worse than the other, but the contextual reference implementation came out on top. The co-contextual algorithm has further potential that has not been leveraged in this research. It is plausible that when addressing the issues found with this work's implementation, a co-contextual type-checker can outperform existing algorithms. Furthermore, with a more sophisticated incremental type-checking approach, it is possible to reuse computation to a larger extent than is possible with contextual approaches.

Bibliography

Almasi, G.S. and A. Gottlieb (1989). *Highly Parallel Computing*. Benjamin/Cummings series in computer science and engineering. Benjamin/Cummings. ISBN: 9780805301779. URL: https://books.google.nl/books?id=TcgmAAAAMAAJ.

András Kovács (n.d.). *smalltt*. Version 0.2.0.2. URL: https://github.com/AndrasKovacs/smalltt.

Antwerpen, Hendrik van and Eelco Visser (2021). "Scope States: Guarding Safety of Name Resolution in Parallel Type Checkers". In: *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Ed. by Anders Møller and Manu Sridharan. Vol. 194. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 1:1–1:29. ISBN: 978-3-95977-190-0. DOI: 10.4230/LIPIcs.ECOOP.2021.1. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2021.1.

Augustsson, Lennart, Mikael Rittri, and Dan Synek (1994). "Functional Pearl: On generating unique names". In: *Journal of Functional Programming* 4.1, pp. 117–123. DOI: 10.1017/S0956796800000988.

Church, Alonzo (1941). *The Calculi of Lambda Conversion.* (*AM-6*). Princeton University Press. ISBN: 9780691083940. URL: http://www.jstor.org/stable/j.ctt1b9x12d (visited on 02/23/2026).

Coquand, Thierry (1996). "An algorithm for type-checking dependent types". In: *Science of Computer Programming* 26.1, pp. 167–177. ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(95)00021-6. URL: https://www.sciencedirect.com/science/article/pii/0167642395000216.

Courant, Nathanaëlle and Xavier Leroy (Jan. 2026). "A Lazy, Concurrent Convertibility Checker". In: *Proc. ACM Program. Lang.* 10.POPL. DOI: 10.1145/3776695. URL: https://doi.org/10.1145/3776695.

de Bruijn, N.G (1972). "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)* 75.5, pp. 381–392. ISSN: 1385-7258. DOI: https://doi.org/10.1016/1385-7258(72)90034-0. URL: https://www.sciencedirect.com/science/article/pii/1385725872900340.

Erdweg, Sebastian et al. (2015). "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*. OOPSLA 2015. Pittsburgh, PA, USA: Association for Computing Machinery, pp. 880–897. ISBN: 9781450336895. DOI: 10.1145/2814270.2814277. URL: https://doi.org/10.1145/2814270.2814277.

Fahndrich, Manuel and Robert DeLine (2002). "Adoption and focus: practical linear types for imperative programming". In: *Proceedings of the ACM SIGPLAN 2002 Conference on*

Programming Language Design and Implementation. PLDI '02. Berlin, Germany: Association for Computing Machinery, pp. 13–24. ISBN: 1581134630. DOI: 10.1145/512529.512532. URL: https://doi.org/10.1145/512529.512532.

Hennessy, John L. and David A. Patterson (2007). *Computer Architecture - A Quantitative Approach*. Fourth. Morgan Kaufmann.

Hurkens, Antonius J. C. (1995). "A simplification of Girard's paradox". In: *Typed Lambda Calculi and Applications*. Ed. by Mariangiola Dezani-Ciancaglini and Gordon Plotkin. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 266–278. ISBN: 978-3-540-49178-1.

Igarashi, Atsushi, Benjamin C. Pierce, and Philip Wadler (May 2001). "Featherweight Java: a minimal core calculus for Java and GJ". In: *ACM Trans. Program. Lang. Syst.* 23.3, pp. 396–450. ISSN: 0164-0925. DOI: 10.1145/503502.503505. URL: https://doi.org/10.1145/503502.503505.

Kennedy, HubertT C. (1973). "The principles of arithmetic, presented by a new method (1889)". In: *Selected Works of Giuseppe Peano*. University of Toronto Press, pp. 101–134. ISBN: 9781487592240. URL: http://www.jstor.org/stable/10.3138/j.ctt1vxmd8x.13 (visited on 02/24/2026).

King, David J. and John Launchbury (1995). "Structuring depth-first search algorithms in Haskell". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: Association for Computing Machinery, pp. 344–354. ISBN: 0897916921. DOI: 10.1145/199448.199530. URL: https://doi.org/10.1145/199448.199530.

Kuci, Edlira (2020). "Co-Contextual Type Systems: Contextless Deductive Reasoning for Correct Incremental Type Checking". PhD thesis. Technical University Darmstadt. DOI: https://doi.org/10.25534/tuprints-00011419. URL: https://tuprints.ulb.tu-darmstadt.de/handle/tuda/4935.

Lea, Doug (2000). "A Java fork/join framework". In: *Proceedings of the ACM 2000 Conference on Java Grande*. JAVA '00. San Francisco, California, USA: Association for Computing Machinery, pp. 36–43. ISBN: 1581132883. DOI: 10.1145/337449.337465. URL: https://doi.org/10.1145/337449.337465.

Li, Peng et al. (2007). "Lightweight concurrency primitives for GHC". In: *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*. Haskell '07. Freiburg, Germany: Association for Computing Machinery, pp. 107–118. ISBN: 9781595936745. DOI: 10.1145/1291201.1291217. URL: https://doi.org/10.1145/1291201.1291217.

Löh, Andres, Conor McBride, and Wouter Swierstra (Apr. 2010). "A Tutorial Implementation of a Dependently Typed Lambda Calculus". In: *Fundam. Inf.* 102.2, pp. 177–207. ISSN: 0169-2968.

López Juan, Víctor and Nils Anders Danielsson (2020). "Practical dependent type checking using twin types". In: *Proceedings of the 5th ACM SIGPLAN International Workshop on Type-Driven Development*. TyDe 2020. Virtual Event, USA: Association for Computing Machinery, pp. 11–23. ISBN: 9781450380515. DOI: 10.1145/3406089.3409030. URL: https://doi.org/10.1145/3406089.3409030.

Marlow, Simon (2011). "Parallel and concurrent programming in Haskell". In: *Proceedings of the 4th Summer School Conference on Central European Functional Programming School*. CEFP'11. Budapest, Hungary: Springer-Verlag, pp. 339–401. ISBN: 9783642320958. DOI: 10.1007/978-3-642-32096-5_7. URL: https://doi.org/10.1007/978-3-642-32096-5_7.

Peyton Jones, Simon, Andrew Gordon, and Sigbjorn Finne (1996). "Concurrent Haskell". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. St. Petersburg Beach, Florida, USA: Association for Computing Machinery, pp. 295–308. ISBN: 0897917693. DOI: 10.1145/237721.237794. URL: https://doi.org/10.1145/237721.237794.

Pierce, Benjamin C. et al. (2026). *Programming Language Foundations*. Ed. by Benjamin C. Pierce. Vol. 2. Software Foundations. Version 7.0. Electronic textbook.

Ziliani, Beta and Matthieu Sozeau (2017). "A comprehensible guide to a new unifier for CIC including universe polymorphism and overloading". In: *Journal of Functional Programming* 27, e10. DOI: 10.1017/S0956796817000028.

Zwaan, Aron, Hendrik van Antwerpen, and Eelco Visser (Oct. 2022). "Incremental type-checking for free: using scope graphs to derive incremental type-checkers". In: *Proc. ACM Program. Lang.* 6.OOPSLA2. DOI: 10.1145/3563303. URL: https://doi.org/10.1145/3563303.

Appendix A

Typing Judgments

This appendix covers the full typing judgments for Elara's type system.

$$\frac{\begin{array}{c} e : T_1 \mid C_1^- \mid C_1^? \mid R_1 \qquad t : T_2 \mid C_2^- \mid C_2^? \mid R_2 \\ (R, C) = merge(R_1,\ R_2) \qquad (\sigma, C_3^-, C_3^?) = solve(C \cup \{t = T_1,\ T_2 = U\}) \end{array}}{(e : t) : t \mid \sigma(C_1^- \cup C_2^- \cup C_3^-) \mid \sigma(C_1^? \cup C_2^? \cup C_3^?) \mid \sigma(R)} \quad \text{(CoT-Ann)}$$

$$U : U \mid \varnothing \mid \varnothing \mid \varnothing \quad \text{(CoT-Uni)}$$

$$\frac{\begin{array}{c} A : T_1 \mid C_1^- \mid C_1^? \mid R_1 \qquad B : T_2 \mid C_2^- \mid C_2^? \mid R_2 \qquad (R_3, C_3) = merge(R_2, \{x : A\}) \\ (R_4, C_4) = merge(R_1,\ R_3 - \{x\}) \qquad (\sigma, C_3^-, C_3^?) = solve(C_3 \cup C_4 \cup \{T_1 = U,\ T_2 = U\}) \end{array}}{(\Pi(x : A).\ B) : U \mid \sigma(C_1^- \cup C_2^- \cup C_3^-) \mid \sigma(C_1^? \cup C_2^? \cup C_3^?) \mid \sigma(R_4)} \quad \text{(CoT-Pi)}$$

$$\frac{T\ fresh}{x_l : T \mid \varnothing \mid \varnothing \mid \{x : T\}} \quad \text{(CoT-Var-Local)}$$

$$\frac{T\ fresh}{x_g : \Gamma(x_g) \mid \varnothing \mid \varnothing \mid \varnothing} \quad \text{(CoT-Var-Global)}$$

$$\frac{\begin{array}{c} e_1 : T_1 \mid C_1^- \mid C_1^? \mid R_1 \qquad e_2 : T_2 \mid C_2^- \mid C_2^? \mid R_2 \\ T\ fresh \qquad (R,\ C) = merge(R_1,\ R_2) \qquad (\sigma, C_3^-, C_3^?) = solve(C \cup \{T_1 = \Pi'(e_2 : T_2).\ T\}) \end{array}}{(e_1\ e_2) : \sigma(T) \mid \sigma(C_1^- \cup C_2^- \cup C_3^-) \mid \sigma(C_1^? \cup C_2^? \cup C_3^?) \mid \sigma(R)} \quad \text{(CoT-App)}$$

$$\frac{\begin{array}{c} e : T_1 \mid C_1^- \mid C_1^? \mid R_1 \\ T_2\ fresh \qquad (R,\ C) = merge(R_1,\ \{x : T_2\}) \qquad (\sigma, C_2^-, C_2^?) = solve(C) \end{array}}{(\lambda x.\ e) : \sigma(\Pi(x : T_2).\ T_1) \mid \sigma(C_1^- \cup C_2^-) \mid \sigma(C_1^? \cup C_2^?) \mid \sigma(R - \{x\})} \quad \text{(CoT-Lam)}$$

$$Nat : U \mid \varnothing \mid \varnothing \mid \varnothing \quad \text{(CoT-Nat)}$$

$$O : Nat \mid \varnothing \mid \varnothing \mid \varnothing \quad \text{(CoT-Zero)}$$

$$S : Nat \to Nat \mid \varnothing \mid \varnothing \mid \varnothing \quad \text{(CoT-Succ)}$$

$$[P : Nat \to U] \to P\ O \to ([n : Nat] \to P\ n \to P\ (S\ n)) \to [n : Nat] \to P\ n \qquad \text{(nat-ind)}$$
$$natElim : nat\text{-}ind \mid \varnothing \mid \varnothing \mid \varnothing \quad \text{(CoT-NatElim)}$$

$$Vec : U \to Nat \to U \mid \varnothing \mid \varnothing \mid \varnothing \quad \text{(CoT-Vec)}$$

$$Nil : [a : U] \to Vec\ a\ O \mid \varnothing \mid \varnothing \mid \varnothing \quad \text{(CoT-Nil)}$$

$$Cons : [a : U] \to [n : Nat] \to a \to Vec\ a\ n \to Vec\ a\ (S\ n) \mid \varnothing \mid \varnothing \mid \varnothing \quad \text{(CoT-Cons)}$$

$$\begin{aligned}&[a : U] \to [P : [n : Nat] \to Vec\ a\ n \to U] \to P\ O\ (Nil\ a) \to \\ &([n : Nat] \to [x : a] \to [xs : Vec\ a\ n] \to P\ n\ xs \to P\ (S\ n)\ (Cons\ a\ n\ x\ xs)) \to \\ &[n : Nat] \to [xs : Vec\ a\ n] \to P\ n\ xs \qquad \text{(vec-ind)}\end{aligned}$$
$$vecElim : vec\text{-}ind \mid \varnothing \mid \varnothing \mid \varnothing \quad \text{(CoT-VecElim)}$$

Appendix B

Supporting Implementation Details

This chapter describes some implementation details and design decisions not directly related to the type-checking algorithm. These low-level program details are included for completeness. The project is written in Haskell[1] and built using the Cabal package manager[2]. The benchmarking code makes use of some primitives supplied by the Glasgow Haskell Compiler, making it non-portable. The source code can be found on GitHub[3].

B.1 Input Preprocessing

The terms obtained from parsing have to be pre-processed before being passed to the type checker. This step performs two tasks.

Firstly, the language syntax does not differentiate between local and global variables. The type-checker requires this, so the pre-processor splits variables into two terms: one for local and one for global variables.

Secondly, for parallelisation, the size of a term (number of nodes in the syntax tree) needs to be known. The pre-processor pre-computes this, so the size can be retrieved in $\mathcal{O}(1)$ time during type-checking.

The time complexity of the preprocessing step is $O(n \cdot log(n))$ where n is the number of nodes in the syntax tree.

At the file level, a check is done for duplicate definitions and (mutual) recursive definitions. The former requires an $O(n \cdot log(n))$ traversal of the definitions, where n is the number of definitions found in the file. Checking for cycles is $O((n + E) \cdot log(n))$ where E is the number of dependents between definitions. This uses an algorithm for finding strongly connected components in a directed graph(King and Launchbury 1995), which also yields a topological order.

B.2 Variable representation

Elara uses named representation for its variables, distinguishing between local and global variables. Name shadowing is allowed in the source language, and the type-checking algorithm accommodates it. As long as variable substitution is capture-avoiding, name shadowing will not cause issues.

[1] GHC 9.12.2, base 4.21.0.0

[2] Cabal 3.14.2.0

[3] `https://github.com/DragonHunt3r/elara` - Release 0.1.0.0

B.2.1 Requirements and Constraints

For all the advantages of the commonly-used De Bruijn indices(de Bruijn 1972) (locally nameless) over a named representation, it does have its disadvantages for co-contextual type checking. When traversing up the syntax tree, if a binder is encountered, all the requirements generated from the child nodes need to have their local variables shifted if De Bruijn indices were to be used.

Consider the K-combinator using De Bruijn indices $\lambda\ \lambda\ 2$. Inferring the type of the body of the inner lambda will yield a requirement $\{2 : T\}$. When inferring the type of the inner lambda itself, this requirement needs to be shifted to make it refer to the correct binder, leaving $\{1 : T\}$.

B.2.2 Constraint Solving

When solving constraints, α-equivalence comes into play. Suppose there is a constraint $\lambda x.x = \lambda y.y$. These two terms are α-equivalent. Using De Bruijn indices, α-equivalence is the same as syntactic equivalence, making for a simple comparison. For Elara, since a system to generate fresh symbols is already in place, a symbol is generated and applied to both functions. This is equivalent to the following proposition: $f = g \iff \forall x.f\ x = g\ x$ (i.e. functional extensionality). Informally, the generated symbol represents an 'arbitrary' term. If the proposition $f\ x = g\ x$ holds for some opaque x (the generated symbol), then it holds for any x.

In practice, this means evaluating the function application and comparing the results. This covers α-equivalence, but is more powerful since more complex constraints can be solved this way as well. The constraint $\lambda x.(\lambda y.y)\ x = \lambda y.y$ is not α-equivalent but can be solved this way.

B.2.3 Substitution

With named variable representation, care is taken to avoid variable capture during substitution. For example, consider the term $\lambda x.\ y$. If the variable y is substituted naïvely for some term where x occurs freely, the behaviour changes. This is undesirable. To avoid this, binders may need to be renamed, and the substitute term needs to be traversed to find its free variables. This is generally expensive.

Another issue is that the same term may be traversed multiple times for different substitutions. A faster implementation would simply store the substitutions as a lookup table instead of eagerly substituting. Haskell's laziness does not help here; computations may not be done immediately, but have to be done fully eventually.

B.3 Symbol Generation

The type-checking algorithm needs to generate unique terms. A use for this is assigning a placeholder type to an expression.

For simplicity, this uses an internal counter that gets fetched and incremented every time a new symbol is requested. The access to this counter is atomic to prevent race conditions in the parallel case. The actual number produced is irrelevant; the equivalence of two fresh symbols is the only use for them. As such, the computation can be deferred lazily using, meaning the atomic operation is only performed when the value is demanded.

This approach likely degrades performance when parallelisation comes into play. This single-counter approach may result in multiple threads waiting to generate a fresh term, as the same reference must be accessed sequentially. Other approaches that do not suffer this

issue exist(Augustsson, Rittri, and Synek 1994), but would make the implementation more convoluted.

B.4 Benchmarking

The main performance statistic of interest is the time it takes to type check. To do this, the computation is run, and a timestamp is taken right before and after the computation. The elapsed time between these time stamps measures the computation time. Unfortunately, a single execution is not a good measure as it may be biased. Furthermore, very small executions may not be accurately measurable due to the precision of the clock used. Instead, a single computation is repeated multiple times, and the average time is computed, reducing bias. The primitives to execute an action multiple times are seen in Listing B.1. They are adapted from the package `criterion-measurement`. This loop is heavily optimised to minimise overhead. The return value is also fully evaluated, as Haskell's laziness otherwise would prevent the full computation from being performed. The inputs to these functions are fully evaluated as well before the benchmarks are run, so the benchmark does not include evaluating thunks that are otherwise unrelated to the computation being measured.

```
nf' :: (b -> ()) -> (a -> b) -> a -> Natural -> IO b
nfIO' :: (b -> ()) -> (a -> IO b) -> a -> Natural -> IO b
```

Listing B.1: Benchmarking primitives.

The command-line interface allows the user to set the number of iterations performed by the benchmark and to swap between timing module-level and definition-level performance. The default number of iterations is 100.

When benchmarking, only the actual type-checking is measured. The parsing and preprocessing are not measured.

B.5 Profiling

The type checker supports some profiling of what the type checker is actually doing under the hood. This keeps track of various statistics like the degree of parallelisation and solver data, such as how many constraints are solved, progressed or stuck. The accumulated data is separated by the definition being type-checked and the depth in the syntax tree. The resulting data is stored in a `CSV` file. The column format is described in the project's `README.md`. This data may be used to reason about the algorithm's performance. Note that this data accumulation introduces overhead, so profiling should not be enabled while benchmarking.

B.6 LambdaPi Compatibility

LambdaPi's core language is the same as Elara's, by design. Conversion is straightforward, but there are some minor differences between the languages:

- LambdaPi uses a locally nameless variable representation, whereas Elara does not. This conversion is trivial.
- LambdaPi does not support partial applications of the type constructors, value constructors and eliminators of the `Nat` and `Vec` data types. For example, $f = S :: Nat \to Nat$, where S is the successor constructor of the `Nat` data type, is not allowed in LambdaPi, whereas Elara allows this. During conversion, such terms are η-expanded until enough arguments are applied. In this example, the converted result would be $f = \backslash x.\ S\ x$ (barring conversion to locally nameless representation). Note that a term

$Vec\,T :: Nat \to U$ is expanded to $\backslash x.\, Vec\,T\,x$ and not $(\backslash x.\, \backslash y.\, Vec\,x\,y)\,T$. As this introduces additional terms, type-checking comparison of terms that require expansion for LambdaPi is unfair, but the overhead is insignificant..

- LambdaPi uses a bidirectional type-checking algorithm, differentiating between inferrable and checkable terms. This requires the left-hand side of a function application to be an inferable term. The top-level term also needs to be inferable, but this is no issue, as Elara requires type annotations for top-level terms, making it inferable.

Appendix C

Benchmark Data

This appendix contains various plots for the benchmarks.

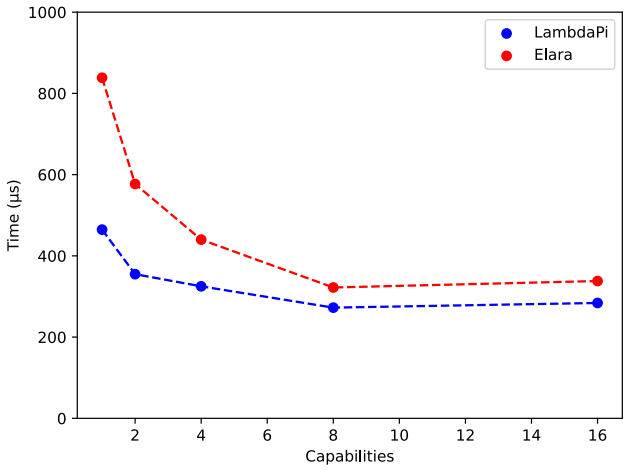


Figure C.1: Module-level parallelisation benchmarks for `church.elara`.

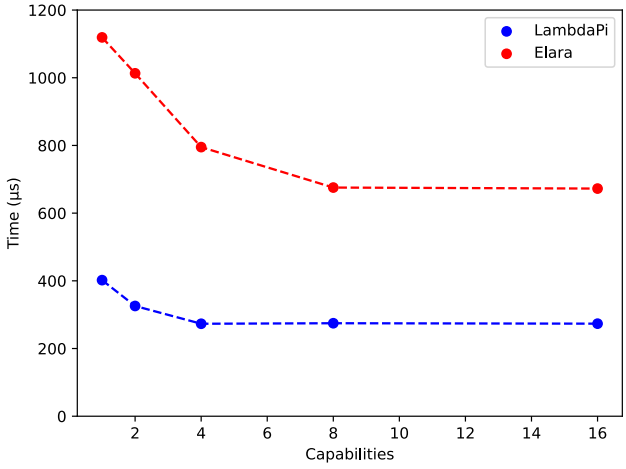


Figure C.2: Module-level parallelisation benchmarks for `nats.elara`.

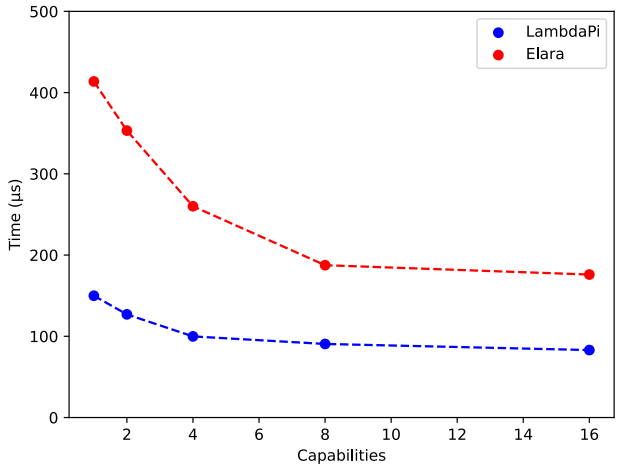


Figure C.3: Module-level parallelisation benchmarks for `vecs.elara`.

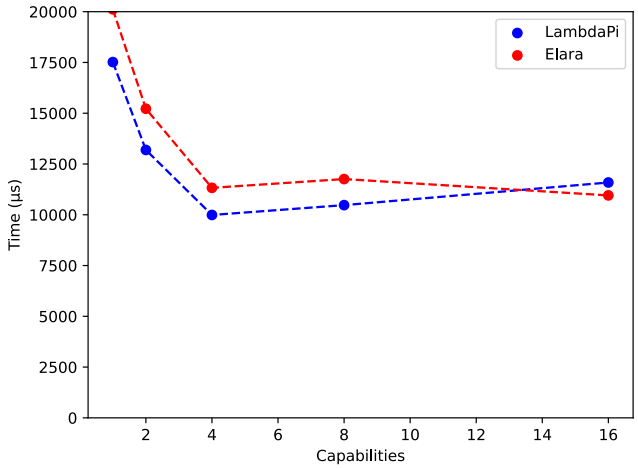


Figure C.4: Module-level parallelisation benchmarks for `stlc.elara`.

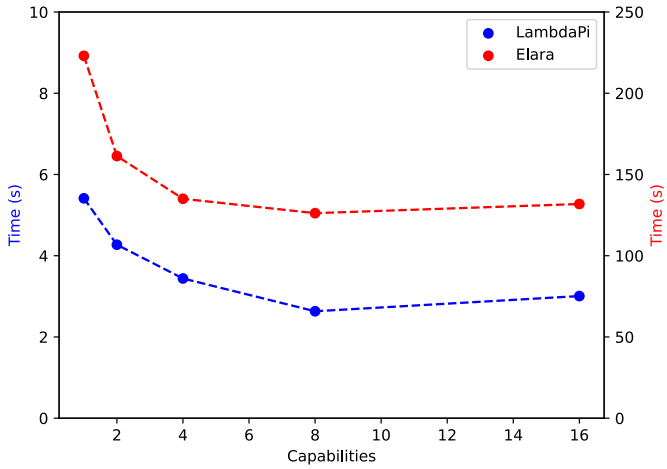


Figure C.5: Module-level parallelisation benchmarks for `asymptotics.elara`.

```
-Ty = [Ty: U] -> [nat: Ty] -> [top: Ty] -> [bot: Ty] -> [arr: Ty -> Ty -> Ty]
-    -> [prod: Ty -> Ty -> Ty] -> [sum: Ty -> Ty -> Ty] -> Ty
+Ty = [Ty: U] -> [nat: Ty] -> [top: Ty] -> [bot: Ty] -> [arr: Ty -> Ty -> Ty]
+    -> [prod: Ty -> Ty -> Ty] -> [sum: Ty -> Ty -> Ty] -> ((\x. x) Ty)
```

Listing C.1: Root definition change.

```
-tfalse = \ctx. right ctx top top (tt ctx)
+tfalse = \ctx. right ctx top ((\x. x) top) (tt ctx)
```

Listing C.2: Leaf definition change.