Safe, Fast, Concurrent Proof Checking for the lambda-Pi Calculus Modulo Rewriting
Michael Färber

To cite this version:
Michael Färber. Safe, Fast, Concurrent Proof Checking for the lambda-Pi Calculus Modulo Rewriting, 11th ACM SIGPLAN International Conference on Certified Programs and Proofs (CCP’22), Jan 2022, Philadelphia, PA, United States. 10.1145/3497775.3503683. hal-03143359v3

HAL Id: hal-03143359
https://inria.hal.science/hal-03143359v3
Submitted on 2 Mar 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License
Safe, Fast, Concurrent Proof Checking for the lambda-Pi Calculus Modulo Rewriting

Michael Färber
University of Innsbruck
Innsbruck, Austria
michael.faerber@uibk.ac.at

Abstract
Several proof assistants, such as Isabelle or Coq, can concurrently check multiple proofs. In contrast, the vast majority of today’s small proof checkers either does not support concurrency at all or only limited forms thereof, restricting the efficiency of proof checking on multi-core processors. This work shows the design of a small, memory- and thread-safe kernel that efficiently checks proofs both concurrently and sequentially. This design is implemented in a new proof checker called Kontroli for the lambda-Pi calculus modulo rewriting, which is an established framework to uniformly express a multitude of logical systems. Kontroli is faster than the reference proof checker for this calculus, Dedukti, on all of five evaluated datasets obtained from proof assistants and interactive theorem provers. Furthermore, Kontroli reduces the time of the most time-consuming part of proof checking using eight threads by up to 6.6x.

CCS Concepts: • Theory of computation → Logic and verification; Equational logic and rewriting; Interactive proof systems; Automated reasoning; Higher order logic.

Keywords: concurrency, performance, sharing, rewriting, reduction, verification, type checking, Dedukti, Rust

1 Introduction
Proof assistants are tools that provide a syntax to rigorously specify mathematical statements and their proofs, in order to mechanically verify them. A strong motivation to use proof assistants is to increase the trust in the correctness of mathematical results, such as the Kepler conjecture [18], which has been verified using the proof assistants HOL Light [21] and Isabelle [35], and the Four-Colour Theorem [17], which has been verified using Coq [8]. However, why should we believe that a proof is indeed correct when a proof assistant says so? We might trust such a statement if we were certain that the proof assistant was correct, i.e. that the proof assistant only accepts valid proofs. To verify the correctness of the proof assistant, we can either inspect it by hand or verify it with another proof assistant in whose correctness we trust. However, many proof assistants are too complex and change too often to make such an endeavour worthwhile. Still, even if we ignore the correctness of a proof assistant, we may trust its statements, provided that the proof assistant justifies all statements in such a way that we can comprehend the justifications and write a program to verify them. A proof assistant “satisfying the possibility of independent checking by a small program is said to satisfy the de Bruijn criterion” [5]. We call such small programs proof checkers.

The logical framework Dedukti has been suggested as a universal proof checker for many different proof assistants [4]. Its underlying calculus, the lambda-Pi calculus modulo rewriting [13], is sufficiently powerful to efficiently express a variety of logics, such as those underlying the proof assistants HOL and Matita [3], PVS [16], and the B method [19].

The Dedukti theories generated by proof assistants and automated theorem provers can be in the order of gigabytes and take considerable amounts of time to verify. The current architecture of Dedukti, which is written in OCaml, allows only for a limited form of concurrent proof checking, restricting the efficiency of proof checking on multi-core processors. Like Dedukti, most other existing small proof checkers do not (fully) exploit multiple cores.

Rust is a functional systems programming language that aims to combine safety, performance, and concurrency. These properties make Rust an interesting candidate to implement proof checkers in. This article evaluates the effectiveness of Rust as implementation language for proof checkers by
reimplementing a fragment of Dedukti in Rust, and uses the opportunity to explore meaningful uses of concurrency.

The major difficulty when porting Dedukti to Rust is sharing of values. Functional programming languages such as OCaml and Haskell use a garbage collector, which allows them to implicitly share values. In contrast, systems programming languages such as Rust or C do not use a garbage collector, and thus do not share values implicitly. In return, such languages offer fine-grained sharing; for example, data can be marked to never be shared, shared within a single thread, or shared between multiple threads. Using just the right amount of sharing enables higher performance, in particular when introducing concurrency. However, due to implicit sharing, it is difficult to establish where sharing is actually used in functional programs, including proof checkers such as Dedukti.

This paper deals with the following research questions: Where and which kinds of sharing are necessary in a proof checker? Which constraints does concurrency impose on sharing? How to implement a proof checker that uses the appropriate amount of sharing for both concurrent and sequential use, while keeping the virtues of being small, memory- and thread-safe, and fast? How much performance can be gained by using such a proof checker?

I make the following contributions in this paper: I present a generic term data type that can be instantiated to vary the sharing behaviour for constants and terms, yielding a family of term types for efficient concurrent and sequential parsing and verification. I refine this term type by reducing the number of pointers, improving performance especially of concurrent verification (section 3). I study reduction of terms and show that concurrent reduction implies significant overhead, making it slower than sequential reduction (section 4). I study verification of theories and show that it can be parallelised neatly by breaking it into two parts, where the more time-intensive part can be delayed and executed in parallel. This is the most successful use of concurrency explored in this work (section 5). I show that parsing of theories can be accelerated by an efficient representation of constants, and that concurrent parsing incurs such a large overhead that it is slower than sequential parsing (section 6). I implement all the presented techniques in a new proof checker called Kontroli, supporting a fragment of Dedukti that is sufficient to verify HOL-based theories. Kontroli is written in Rust, which combines the safety of functional programming languages with the fine-grained control over sharing of system programming languages. This is crucial in assuring that Kontroli is small, memory- and thread-safe, and fast (section 7). I evaluate Kontroli and Dedukti on five different datasets stemming from interactive and automated theorem provers. On all datasets, the sequential version of Kontroli is consistently faster than both concurrent and sequential versions of Dedukti. When concurrently checking theories, Kontroli speeds up the most time-consuming part of proof checking by up to 6.6x when using eight threads (section 8).

## 2 Background

### 2.1 The ΛΠ-Calculus Modulo Rewriting

Let $C$ denote a set of constants. A term has the shape

$$t := c \mid s \mid tu \mid x \mid \lambda x.t : u \mid \Pi x.t : u,$$

where $c \in C$ is a constant, $s := Type \mid Kind$ is a sort, $t$ and $u$ are terms, and $x$ is a bound variable. If $x$ does not occur freely in $u$, we may write $t \rightarrow u$ for $\Pi x.t : u$.

A rewrite pattern has the shape $p := x \mid cp_1 \ldots p_n$, where $x$ is a variable, $c \in C$ is a constant, and $p_1 \ldots p_n$ is a potentially empty sequence of rewrite patterns applied to $c$.

A rewrite rule has the shape $r := cp_1 \ldots p_n \rightarrow t$, where we call $cp_1 \ldots p_n$ the left-hand side, $t$ the right-hand side, and $c$ the head symbol of $r$. The free variables of the right-hand side are required to be a subset of the free variables of the left-hand side, i.e. $\bigcup_i \mathcal{FV}(p_i) \supseteq \mathcal{FV}(t)$.

A global context $\Gamma$ contains statements of the form $c : A$ and $cp_1 \ldots p_n \rightarrow t$. A local context $\Delta$ contains statements of the form $x : A$.

We beta-reduce terms via $(\lambda x.t)u \rightarrow^\beta t[u/x]$, where $t[u/x]$ denotes the substitution of $x$ in $t$ by $u$. Additionally, we reduce $t \rightarrow^\tau u$ iff there exists a term rewrite rule $(t' \leftarrow u') \in \Gamma$ and a substitution $\sigma$, so that $t' \sigma = t$ and $u' \sigma = u$.

Let $\rightarrow^\tau = \rightarrow^\beta \cup \rightarrow^\tau_{gt}$ be our reduction relation.\footnote{To simplify the presentation, I only introduce first-order rewriting. Note that Dedukti uses higher-order rewriting [25].} We say that two terms $t, u$ are $\Gamma$-convertible, i.e. $t \rightarrow^\tau u$, when there exists a term $v$ such that $t \rightarrow^\tau_\Gamma v$ and $u \rightarrow^\tau_{gt} v$.

We write $\Gamma \vdash t : A$ and say that the term $t$ has the type $A$ in the global context $\Gamma$ if we can find a derivation of $\Gamma, \Delta \vdash t : A$ using the rules in Figure 1 [adapted from 28, Figure 2.4].

![Figure 1. Inference rules.](image)

1\footnote{To simplify the presentation, I only introduce first-order rewriting. Note that Dedukti uses higher-order rewriting [25].}

2\footnote{The implementations of the calculus optionally eta-reduce terms via $(\lambda x.tx) \rightarrow^\eta t$.}
where $\Delta$ is an empty local context. Type inference determines a unique type $A$ for a term $t$ and a global context $\Gamma$ such that $\Gamma \vdash t : A$. Type checking verifies for terms $t$ and $A$ and a global context $\Gamma$ whether $\Gamma \vdash t : A$. If the reduction relation $\rightarrow^*$ is type-preserving, terminating, and confluent, then type inference and type checking terminate [28, Theorem 6.3.1].

A command introduces either a new constant $c : A$ or a rewrite rule $cp_1 \ldots p_n \leftarrow t$. A theory is a sequence of commands.

We check a theory as follows: We start with an empty set of constants $C = \emptyset$ and an empty global context $\Gamma = \emptyset$. For every command in the theory, we distinguish: If the command introduces a constant $c : A$, we verify that $c \notin C$ and that $\Gamma \vdash A : A'$ for some $A'$, then we add $c$ to $C$ and extend the global context such that $(c : A) \in \Gamma$. If the command introduces a rewrite rule $cp_1 \ldots p_n \leftarrow t$, we verify the existence of a local context $\Delta$ and a type $A$ such that $\Gamma, \Delta \vdash cp_1 \ldots p_n : A$ and $\Gamma, \Delta \vdash t : A$, then we extend the global context such that $(cp_1 \ldots p_n \leftarrow t) \in \Gamma$.

**Example 2.1.** Consider the following theory:

```
prop : Type
imp : prop → prop → prop
prf : prop → Type
prf (imp x y) ← prf x → prf y
imp_refl : Πx:prop. prf (imp x x)
imp_refl ← λx:prop. λp:prf x. p
```

This theory first defines types of propositions, implications, and proofs. Next, (4) introduces a rewrite rule that interprets proofs of implications. (5) asserts that implication is reflexive, and (6) proves it via a rewrite rule.

### 2.2 Concurrent Verification

Concurrent verification designates the simultaneous verification of different parts of a theory. Following Wenzel’s terminology [34], concurrency can happen at different levels of granularity. I distinguish concurrent verification on the level of theories (granularity 0) and on the level of commands/proofs (granularity 1). This work focuses on command-concurrent verification. I will evaluate the two approaches in section 8.

#### 2.2.1 Theory-Concurrent Verification

A theory can be divided into smaller theories, as long as the theory dependencies form a directed acyclic graph. To verify a theory, all of its (transitive) dependencies must be verified before. Theory-concurrent verification exploits that theories that do not transitively depend on each other can be checked concurrently.

---

Wenzel gives yet another level of granularity, namely sub-proofs. However, there is no concept of sub-proofs in Dedukti.

---

An example of a theory dependency graph is shown in Figure 2 for a formalisation of Fermat’s little theorem in Matita. The “breadth” of the graph determines the maximum amount of theories that can be concurrently verified; for example, for Figure 2 we can verify at most six theories concurrently, namely $\text{exp}$, $\text{bigops}$, $\text{gcd}$, $\text{cong}$, $\text{fact}$, and $\text{permutation}$.

Theory-concurrent verification can be implemented by launching a verification process for every theory, producing for every theory a global context that contains the commands in that theory. To verify a theory, it is necessary to load the global contexts of the theory’s dependencies. As loading of global contexts comes with some overhead, dividing a theory into smaller theories increases the number of theories that can be verified concurrently, at the cost of the individual theories taking longer to verify.

#### 2.2.2 Command-Concurrent Verification

The verification of a command can be broken into multiple tasks. Where theory-concurrent verification exploits that independent theories can be checked concurrently, command-concurrent verification exploits that independent tasks to verify a command can be performed concurrently.

This is illustrated in Figure 3. We consider a proof checker that performs four tasks for every command of a theory, namely parsing, sharing, (type) inference, and (type) checking, which will be further explained in the remainder of this paper. Sequential processing (Figure 3a) checks a command only once all tasks have been performed for preceding
parallel parsing (Figure 3b) moves parsing to a different thread, and parallel checking (Figure 3c) distributes checking among an arbitrary number of threads. For both parallel parsing and checking, multiple operations for different commands are executed at the same time; for example, the second command may be parsed while the first command is still being checked, or the first and second command may be checked while the third command is being shared and the fourth command is parsed. Theoretically, the combination of parallel parsing and checking could reduce wall-clock time to check a theory by the time taken for parsing and checking. In practice, however, the overhead of concurrency often leads to much smaller gains, as I will show in section 8.

Command-concurrent verification allows for the concurrent verification of commands regardless of the theory graph. Where the maximum number of concurrently verifiable theories is bounded by the graph breadth, the maximum number of concurrently verifiable commands is bounded by the total number of commands to verify. Where theory-concurrent verification lends itself well to processes, command-concurrent verification lends itself well to threads, because threads allow for the sharing of the global context between concurrent verifications and thus to omit the I/O overhead of loading global contexts, which would become noticeable if done for every command. However, this comes at the cost of using thread-safe data structures for the global context, as I will discuss in section 5.

2.3 Sharing and Concurrency
Sharing enables multiple references to the same memory region. We call such references physically equal. Sharing and physical equality are exploited in Dedukti; for example, we immediately know that physically equal terms are convertible. In many garbage-collected programming languages, such as Haskell and OCaml, sharing is implicit, i.e. members of any type may be shared, whereas in many programming languages without garbage collector, such as C++ and Rust, sharing is explicit, i.e. only members of special types are shared. Such special types include C++’s shared_ptr and Rust’s Rc. To check for physical equality in Rust, we need to explicitly wrap objects with a type such as Rc (Listing 2), whereas in OCaml, such wrapping is implicit (Listing 1).

Reference counting is a technique that is commonly used in languages without garbage collection to manage memory of shared objects: A reference-counted object keeps a counter to register how often it is referenced. Whenever a reference to an object is created, its counter is increased, and whenever a reference to an object goes out of scope, its counter is decreased. Finally, when an object’s counter turns zero, the object is freed.
let a = Some(0) in
let b = a in
let c = Some(0) in
assert (a == b);
assert (b == c);
assert (not (b == c));

Listing 1. Structural and physical equality in OCaml.

let a = Rc::new(Some(0));
let b = a.clone();
let c = Rc::new(Some(0));
assert!(a == b);
assert!(b == c);
assert!(Rc::ptr_eq(&a, &b));
assert!(!Rc::ptr_eq(&b, &c));

Listing 2. Structural and physical equality in Rust.

We call data structures that can be safely shared between threads thread-safe. When a reference-counted object is shared between multiple threads, its counter has to be modified atomically, to ensure that multiple concurrent modifications to the counter do not interfere. Non-atomic modifications can result in memory corruption (a counter turning 0 despite the object still being referenced) and memory leaks (a counter remaining greater than 0 despite the object not being referenced). However, atomic modifications imply a significant runtime overhead. This means that thread-safe reference counting comes with significant overhead.

Languages that do not share values implicitly allow us to minimise concurrency overhead by choosing appropriate types for sharing. In Rust, wrapping objects with different smart pointer types marks them as either shareable only within one thread (Rc, i.e. reference-counted), shareable between multiple threads (Arc, i.e. atomically reference-counted), or not shareable at all (Box). Any of these smart pointer types has two out of three properties: thread-safety (Box, Arc), sharing (Rc, Arc), and performance (Box, Rc), see Figure 4. In addition, we have a non-smart pointer type, namely references (&), which has all three desiderata mentioned above, but requires us to prove that it points to a valid object.\footnote{Rust is a memory-safe language, so unlike e.g. C/C++, the compiler throws an error if we attempt to use a reference pointing to an invalid object. This protects against a large class of memory-related bugs.}

In summary, for concurrent type checking, we need to carefully choose our pointer types, as this choice has a direct impact on performance.

3 Terms

The central data structure of our proof checker are terms. Let us have a closer look at how they are defined. See subsection 2.3 for an explanation of the pointer types used here. Listing 3 shows the definition of Dedukti terms in OCaml, and Listing 4 shows its direct translation to Rust. I call the constructors Kind, Type, Const, and Var atomic, and the constructors App, Lam, and Pi non-atomic. Unlike the OCaml terms, the Rust terms are generic over the type of constants C and the type of term references Tm. We will see in section 6 how the choice of C is useful. Based on the non-inductive Term type, the Rust version defines three inductive term
types, namely \(\text{BTerm}, \text{RTerm}, \text{and ATerm}\). In \(\text{BTerm}\), term references are unshared, whereas in \(\text{RTerm}\) and \(\text{ATerm}\), term references are shared, using non-atomic and atomic reference counting, respectively. As discussed in subsection 2.3, the term types satisfy the following properties (under the assumption that the constant type \(\mathcal{C}\) is thread-safe and can be copied and compared in constant time):

- Unlike \(\text{RTerm}\), both \(\text{BTerm}\) and \(\text{ATerm}\) can be used across threads.
- Unlike \(\text{BTerm}\), both \(\text{RTerm}\) and \(\text{ATerm}\) can be compared for physical equality, taking constant time.
- Copying a \(\text{BTerm}\) deep clones the term, whereas copying \(\text{RTerm}\) and \(\text{ATerm}\) modifies their reference counter, which is faster for \(\text{RTerm}\) than for \(\text{ATerm}\).
- \(\text{BTerm}, \text{RTerm}, \text{and ATerm}\) are increasingly slow to create.

Using \& as pointer type at the place of \text{Box} etc., it is possible to create a term datatype that is thread-safe, shareable, and fast. This is particularly interesting for concurrent verification of commands. However, such a term type requires us to specify at compile-time the lifetime of each term. Because we cannot precisely predict how long each term is going to be used, we have to over-approximate its lifetime to be as long as the verification of a command. That means that throughout the verification of a command, we have to keep in memory every term that is created. Compared to using reference-counted terms, this significantly increases memory usage, because verification may create a large number of intermediate terms. Therefore, I did not further pursue using \& as pointer type for terms.

The three inductive term types require us to wrap every term constructor with a pointer type (Box, Rc, or Arc). However, the atomic constructors Kind, Type, Const, and Var do not contain any terms and can be cloned in constant time, therefore having to wrap them with a pointer type is pointless. For this reason, I give a refined term type in Listing 5, in which the non-atomic constructors have moved to the \text{TermC} type. The refined \(\text{RTerm}\) and \(\text{ATerm}\) can be defined analogously to the original \(\text{RTerm}\) and \(\text{ATerm}\). Using the refined \text{Term}, we do not need to wrap atomic constructors with a pointer type, but in exchange, we need to wrap non-atomic constructors in a \text{Comb}.

**Example 3.1.** Figure 5 and Figure 6 show the encoding of the term \(\text{prop} \to \text{Type}\) in the original and refined \text{BTerm},

\[
\begin{align*}
\text{enum Term}<\mathcal{C}, \mathcal{Tm}> & \{ \\
& \text{Kind, Type,} \\
& \text{Const(\(\mathcal{C}\)), Var(usize),} \\
& \text{Comb(\(\mathcal{Tm}\)),} \\
\}
\end{align*}
\]

\[
\begin{align*}
\text{enum TermC}<\mathcal{Tm}> & \{ \\
& \text{App(\(\mathcal{Tm}\), Vec<\(\mathcal{Tm}\)>),} \\
& \text{Lam(Option<\(\mathcal{Tm}\)>, \(\mathcal{Tm}\)),} \\
& \text{Pi(\(\mathcal{Tm}\), \(\mathcal{Tm}\)),} \\
\}
\end{align*}
\]

\[
\begin{align*}
\text{struct BTermC}<\mathcal{C}>(\text{Box<TermC<BTerm<C>>>>));
\end{align*}
\]

\[
\begin{align*}
\text{type BTerm}<\mathcal{C}> = \text{Term}<\mathcal{C}, \text{BTermC}<\mathcal{C}>>;
\end{align*}
\]

**Listing 5.** Refined Rust terms.

![Figure 5. Original BTerm encoding of prop → Type.](image1)

![Figure 6. Refined BTerm encoding of prop → Type.](image2)

where prop is a user-defined constant. In these graphical representations, a box is shown by a rectangle with rounded corners. We can see that the original BTerm uses three constructors and three boxes, whereas the refined BTerm uses four constructors and one box.

Using fewer pointers (boxes) benefits performance most when using Arc and less when using Rc, because Arc has the largest overhead of the pointer types. On one dataset, using the refined term types reduced total proof checking time by 20% when using RTerm and by 29% when using ATerm.

**4 Reduction**

Asperti et al. [2] have introduced abstract machines to efficiently reduce terms to WHNF. Of all Dedukti components, reimplementing abstract machines in Rust was the most complicated, because they involve sharing, mutability, and lazy evaluation. This section studies the feasibility of concurrent reduction.
struct State<C> {
  ctx: Vec<LazyTerm<C>>,
  term: RTerm<C>,
  stack: Vec<StatePtr<C>>,
}

type StatePtr<C> = Rc<RefCell<State<C>>>;

Listing 7. Abstract machine state in Rust.

An abstract machine encodes a term \( u \) via a substitution \( \sigma \) called context, a term \( t \), and a stack \([t_1, \ldots, t_n]\), such that \( u = (t_\sigma) t_1 \ldots t_n \). Listing 6 shows the definition of an abstract machine state in Dedukti. The context is a list of lazy terms, and the stack of arguments is a list of mutable references to states.

Example 4.1. Consider an abstract machine \( m \) consisting of an empty context, a term \( t \) and a stack \([t_1, t_2]\). Suppose that \( t = \text{add} \) and \( t_1 \) and \( t_2 \) are states encoding the terms fib 5 and fib 6, respectively. Then the machine \( m \) encodes the term add\((\text{fib} 5)(\text{fib} 6)\). Now suppose that we try to match \( m \) with the left-hand side of a rewrite rule \( \text{add} 0 \leftrightarrow n \). This will evaluate the state \( t_2 \) corresponding to fib 5 to some new state \( t_2' \) and replace \( t_1 \) with \( t_1' \). Because the stack is implemented as a list of mutable references, all copies of the original machine \( m \) will also contain \( t_1' \). This avoids recomputing fib 5 in copies of \( m \).

Listing 7 shows the corresponding definition of abstract machines in Rust. The counterpart to Dedukti’s state ref is an Rc-shared mutable reference \((\text{RefCell})\) to a state, and the counterpart to Dedukti’s term Lazy.t is an Rc-shared Thunk from a state pointer to a term. A Thunk<\( T, U \) is a delayed one-time transformation from a type \( T \) to \( U \). Here, evaluating a lazy term transforms a pointer to a state \( (\sigma, t, [t_1, \ldots, t_n]) \) to the WNF of a term \( (t_\sigma)t_1 \ldots t_n \).

Two operations performed during reduction can be trivially parallelised:

- Substitution: We can substitute \( (t_1 \ldots t_n) \sigma \) by substituting \( t_i \sigma \) for multiple \( i \) in parallel. Here, \( t_i \) is a shared lazy term (LazyTerm).
- Matching: We can match a term \( c t_1 \ldots t_n \) with a pattern \( c p_1 \ldots p_m \) by matching \( t_i \) with \( p_i \) for multiple \( i \) in parallel. Here, \( t_i \) is a shared mutable reference to an abstract machine state (StatePtr).

Both substitution and matching involve evaluation of abstract machines. Therefore, parallelising either of these operations requires thread-safe abstract machines. However, the shown definition of abstract machine states uses several thread-unsafe types, namely RTerm, Rc, RefCell, and Thunk. We can obtain thread-safe abstract machines by replacing these types with thread-safe types, such as RTerm with ATerm and Rc with Arc, and RefCell with Mutex. However, each of these types adds some overhead. My experiments showed that this overhead is so large that even with concurrent substitution and matching, the proof checker is significantly slower. Therefore, I did not further pursue concurrent reduction.

5 Verification

This section describes how to verify a sequence of commands, and how to parallelise it. The resulting approach will perform command-concurrent verification as introduced in subsection 2.2.

Let us revisit the verification procedure outlined in subsection 2.1: We start with an empty global context \( \Gamma \) and perform the following for every command: If the command introduces a constant \( c : A \), we infer the type \( A' \) such that \( \Gamma \vdash A : A' \). If the command introduces a rewrite rule \( ! \leftrightarrow r \) in a local context \( \Delta \), we infer the type \( A \) such that \( \Gamma, \Delta \vdash r : A \) and check that \( \Gamma, \Delta \vdash ! : A \). Finally, we add the command to the global context \( \Gamma \).

Proof checking usually spends the largest portion of time checking that \( \Gamma, \Delta \vdash r : A \). Using this observation, we can parallelise verification by deferring these checks and performing them in parallel in a thread pool. This puts certain constraints on the used data types: Because we are sending type checking tasks \( \Gamma, \Delta \vdash r : A \) across threads, the global and local contexts \( \Gamma \) and \( \Delta \) as well as the terms \( r \) and \( A \) need to be thread-safe. However, type checking uses thread-unsafe shared terms (RTerm). I am going to discuss two approaches to resolve this dilemma.

The first approach is to use thread-safe shared terms (ATerm) in \( \Gamma, \Delta \) as well as for \( r \) and \( A \). This implies that type checking and all algorithms performed as part of it (such as reduction and substitution) should operate on ATerm. However, if all these algorithms accept ATerm, then sequential proof checking would also be forced to use ATerm, which would result in an unnecessary overhead compared to using RTerm. This can be circumvented by creating a sequential and a parallel version of the kernel; the only difference between these is that the parallel version uses ATerm whereas the sequential version uses RTerm. This allows us to use the same kernel code for both overhead-free sequential as well as for parallel verification. One downside to this approach is that concurrent access of multiple check threads to the same shared term has to be synchronised. This is why it is important to reduce the amount of sharing in terms, as done with the optimised term type in section 3. But even with this optimisation, multiple check threads accessing the same shared term simultaneously can become a bottleneck.

The second approach is to use unshared terms (BTerm) in \( \Gamma, \Delta \) as well as for \( r \) and \( A \). As will be explained in section 7,
only atomic terms contained in $\Gamma$, $\Delta$ are shared. Because $\text{BTerm}$ preserves the sharing of atomic terms, using $\text{BTerm}$ for the terms in $\Gamma$, $\Delta$ preserves the sharing of $\text{ATerm}$ or $\text{RTerm}$. However, during type checking, we also want to share non-atomic terms, which we cannot do with $\text{BTerm}$. Therefore this approach requires us to convert the unshared terms in $\Gamma$, $\Delta$ to shared terms before we can use them for type checking. Unlike the $\text{ATerm}$ approach, this approach allows us to keep using $\text{RTerm}$ (as opposed to $\text{ATerm}$) for type checking, because the converted terms remain in the same thread. That means that in the $\text{ATerm}$ approach, we have some continual overhead from using $\text{ATerm}$, whereas in the $\text{BTerm}$ approach, we have overhead whenever we convert a term from $\Gamma$, $\Delta$ to $\text{BTerm}$, but once it is converted, we have less continual overhead from using $\text{RTerm}$ (compared to $\text{ATerm}$). I evaluated the following strategy: Whenever type checking requests a term from $\Gamma$, $\Delta$, it converts it from a $\text{BTerm}$ to an $\text{RTerm}$. Using this strategy, type checking is much slower than using the $\text{ATerm}$ approach, because conversions from $\text{BTerm}$ to $\text{RTerm}$ happen very frequently. An alternative strategy is to cache the converted terms, such that multiple requests to the same term result in only a single conversion. The cache could be persistent for each check task or even across check tasks. To limit memory consumption, such a cache could be limited to contain only e.g. the $n$ most frequently or recently requested terms. All of these strategies, however, are significantly more complex to implement than the $\text{ATerm}$ approach, and make it more challenging to create a kernel that can be also used without overhead for sequential verification. For that reason, I did not further investigate the $\text{BTerm}$ approach and use the $\text{ATerm}$ approach instead.

### 6 Parsing

Parsing of theories is a surprisingly expensive operation that can take up to half the time of proof checking, as will be shown in section 8. This section presents the design of a theory parser that can be used both sequentially and concurrently.

The parser takes a reference to an input string ($\&\text{str}$) and lazily yields a stream of commands. The type of terms contained in a command is $\text{BTerm}<\&\text{str}>$, where $\&\text{str}$ is the type of constants (see section 3). Using $\&\text{str}$ as constant type allows us to copy constants in constant time and to store constants as slices of the original input string. This is significantly more efficient than using $\text{String}$, which copies constants in linear time and allocates new memory for every constant in the term. For example, parsing the HOL Light dataset (which will be introduced in the evaluation in section 6) takes 21.3 seconds using $\text{BTerm}<\&\text{str}>$ (this corresponds to $\text{KO}'\rho$ in the evaluation) and 28.4 seconds using $\text{BTerm}<\text{String}>$.

We can parallelise parsing as follows: In the original thread, we parse commands and send them via a channel to a new thread, in which we perform all subsequent operations such as sharing (which will be explained in section 7 and checking (see section 5). However, we still need to address one issue: We cannot send references such as $\&\text{str}$ through the channel, because we cannot prove that the references remain valid, so we cannot send the parsed commands, which contain $\text{BTerm}<\&\text{str}>$. One solution to this dilemma is the following: When parsing in a separate thread, convert $\text{BTerm}<\&\text{str}>$ to $\text{BTerm}<\text{String}>$ by duplicating the parts of the input string that refer to constants. Allowing for this is the main motivation for the $\text{Term}$ type being generic over the constant type.

Parallel parsing comes with considerable overhead, in particular from sending commands through the channel. To recall, parsing the HOL Light dataset to commands containing $\text{BTerm}<\text{String}>$ takes 28.4 seconds. This increases to 96.4 seconds when additionally sending every command through a channel. Not all of this overhead shows up in the runtime of the proof checker, because parsing and sending is performed in a separate thread. Still, the evaluation shows that the proof checker with parallel parsing is slower than with sequential parsing.

### 7 Implementation

I implemented the techniques described in the previous sections in a proof checker called Kontroli.

#### 7.1 The Virtues of Rust

Kontroli is implemented in the functional system programming language Rust. Rust combines the memory safety of functional programming languages with the fine-grained sharing of system programming languages.

The safety of concurrency is verified by virtue of Rust’s type system. For example, the Rust compiler signals an error if we parallelise reduction without replacing all thread-unsafe types used in the underlying abstract machines (section 4), if we parallelise verification using the kernel version with thread-unsafe terms (section 5), or if we parallelise a function that mutates shared state without synchronisation, such as the inference operation which mutates the global context. These safety checks rule out a large class of bugs that other system programming languages, such as C and C++, do not protect against. This is extremely useful when experimenting with concurrency.

The kernel of Kontroli does not perform I/O; it is pure. This is verified by the Rust compiler (using the #!no_std keyword) and allows the kernel to be used in restricted computing environments, such as web browsers.

#### 7.2 Details

The parser that is outlined in section 6 is implemented using a lexer that is automatically generated by the Logos library from an annotated algebraic data type. All intermediate data
structures generated during parsing, such as lexemes, are free of reference-counted sharing, which contributes to the performance. Before this approach, I implemented Dedukti parsers with parser combinators (using the Nom library in Rust and the attoparsec library in Haskell). The parser in this work is significantly faster than these approaches as well as the parser implemented in Dedukti using ocamllex and Menhir, as I will show in section 5.

All constants in terms yielded by the parser are physically unequal to each other, because they all point to different regions in the input string. For example, the parser transforms the input string \texttt{id : A} \rightarrow \texttt{A} into a command that introduces the constant \texttt{id} with the type \texttt{A} \rightarrow \texttt{A}, where \texttt{id}, \texttt{A}, and the second \texttt{A} all point to different parts of the input string. However, it is desirable that equivalent constants are represented by physically equal string references, because this allows us to compare and hash constants (operations that are frequently performed during checking) using only their pointer addresses, which takes constant time.

This constant normalisation is fulfilled by the sharer: The sharer maps every constant contained in a term of a command to an equivalent canonical constant. Because the sharer is generic over the used constant type, it works regardless of whether &str or String are used as constant types and can thus be used on terms yielded by both sequential and parallel parsing. Furthermore, because type inference and checking operate on shared terms (RTerm or ATerm), the sharer converts from BTerm to RTerm or ATerm, respectively. Finally, when a command introduces a new constant \(c\), the sharer introduces \(c\) into the set of canonical constants such that future references to this constant will be all mapped to the same &str.

The sharer maps only equivalent atomic terms to physically equal terms; it does not map equivalent non-atomic terms to physically equal terms. For example, for constants \(f\) and \(c\), the sharer maps occurrences of the non-atomic term \(fc\) in different parts of a term to physically unequal terms, even though it maps \(f\) and \(c\) to physically equal terms. Reduction preserves sharing, but does not introduce new sharing. For example, if the substitution \(ts\) is equivalent to \(t\), then \(ts\) is physically equal to \(t\). On the other hand, if two non-equivalent terms \(t \neq u\) reduce to two equivalent non-atomic terms \(t' = u'\), then \(t'\) is not physically equal to \(u'\). This approach to sharing is also implemented in Dedukti.

Implementing parsing and sharing as separate steps enables a compact and modular implementation that achieves high performance. On the other hand, when parsing to terms that contain references to the input string (as done during sequential parsing), the separation of parsing and sharing forces us to read the whole input file before we can start parsing and to keep the whole input file in memory until parsing and sharing is finished. These restrictions could be overcome by integrating the sharing step into the parser, at the cost of a more complicated and less modular implementation.

Checking tasks of the shape \(\Gamma, \Delta \vdash r : A\), as introduced in section 5, are distributed among a thread pool using the Rayon library. This involves creating a copy of the global context \(\Gamma\) for every checking task. The global context is implemented as hash map that maps every constant \(c\) to the type of \(c\) and the rewrite rules having \(c\) as head symbol. The hash map type in Rust’s standard library takes \(O(n)\) to copy, making it unsuitable as hash map for the global context, because the global context may grow quickly and need frequent copying. Therefore, I use an immutable hash map type from the im library, which takes \(O(1)\) to copy.

7.3 Kernel Size & Supported Features

The Kontroli kernel consists of 663 lines of code, whereas the Dedukti kernel consists of 3470 lines. The size of several other proof checkers is given in section 9.

To obtain such a small kernel, I omitted in Kontroli certain features of Dedukti such as higher-order rewriting [25], matching modulo AC [12], and type inference of variables in rewrite rules. While there is no particular obstacle to implementing these features, they neither offer a challenge for the concurrency of proof checking, nor significantly increase the number of datasets that can be evaluated. On the other hand, these features increase the kernel size, making experiments with alternative data structures more time-consuming.

I also omitted several optimisations present in Dedukti, the most prominent one being decision trees: Decision trees accelerate the matching of terms in the presence of many rewrite rules [22]. However, for the theories I evaluate in section 8, decision trees are not strictly necessary for performance.

8 Evaluation

I evaluate the performance of Dedukti and Kontroli on five datasets derived from theorem provers.

A dataset is a set of theories whose dependencies form a directed acyclic graph, as illustrated in subsection 2.2. Every evaluated dataset consists of two parts, namely a human-written encoding of its underlying logic and propositions and proofs automatically generated from a theorem prover. Compared to the second part, the first part is insignificantly small and takes insignificant time to check.

I evaluate Kontroli and Dedukti on two kinds of datasets: problems from automated theorem provers (ATPs) and interactive theorem provers (ITPs) [15]. ATP datasets consist of theory files that can be checked independently, whereas ITP datasets consist of theory files that depend on each other. Among the ATP datasets, I evaluate proofs of TPTP problems generated by iProver Modulo and proofs of theorems from

---

1Dedukti was obtained from https://github.com/Deducteam/Dedukti, rev. 38e0c57. Kontroli was obtained from https://github.com/01mf02/kontroli-rs, rev. e980688. Lines of code include neither comments nor blank lines. I used Tokei 11.0.0 to count the lines for Kontroli by tokei src/kernel and for Dedukti by tokei kernel.
Therefore, given a graph for a dataset, the 𝑦-sequential verification subsection 2.2. First, I evaluate that correspond to the types of verification introduced in dataset is larger than the largest command of the HOL Light approximately 10 command of the HOL Light dataset, and that each of the dataset is several orders of magnitude larger than the largest command of the Isabelle/HOL dataset, and the its leftmost point is the size of the largest command in the distribution of the sizes of the commands for the ITP datasets is higher-order rewriting.

Theorem proved in Matita [31]. An evaluation of Coq datasets with OCaml 4.08.1 and Rust 1.54. I evaluate all datasets ten times and obtain their average running time as well as the standard deviation. Therefore, to evaluate the ATP datasets, I use theory-concurrent verification for both Dedukti and Kontroli, limiting the number of simultaneously verified theories to 24.

The evaluation system features 32 Intel Broadwell CPUs à 2.2 GHz and 32 GB RAM. Dedukti and Kontroli are compiled with OCaml 4.08.1 and Rust 1.54. I evaluate all datasets ten times and obtain their average running time as well as the standard deviation.

I now discuss the results for the ITP datasets shown in Figure 8. The sequential Kontroli configuration KO is always faster than both sequential and concurrent Dedukti configurations DK and DK新冠病毒. Furthermore, the parser of Kontroli (KO新冠病毒) is significantly faster than the parser of Dedukti (DK新冠病毒); on the Isabelle/HOL dataset, it is 4.5x as fast. Parallel parsing (KO新冠病毒), however, increases runtime on all datasets. Like KO, KO新冠病毒 processes only one command at a time; however, KO新冠病毒 uses ATerm where KO uses RTerm, so it serves to measure the overhead incurred by ATerm. For the HOL Light dataset, for example, we see that it increases runtime by 28.2%. Despite this overhead, already the configuration that uses two threads for type checking (KO新冠病毒) is faster than the single-threaded KO configuration on all datasets. To measure how well type checking parallelises, we compare the type checking times of two configurations. Type checking parallelises moderately on the Matita and HOL Light datasets; using \( n = 2 \) threads, KO新冠病毒 reduces type checking time compared to KO by 1.4x on HOL Light and

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>Theories</th>
<th>Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matita</td>
<td>2.0MB</td>
<td>19</td>
<td>478</td>
</tr>
<tr>
<td>HOL Light</td>
<td>2.0GB</td>
<td>25</td>
<td>1776535</td>
</tr>
<tr>
<td>Isabelle/HOL</td>
<td>2.5GB</td>
<td>1</td>
<td>116927</td>
</tr>
<tr>
<td>iProver</td>
<td>431.4MB</td>
<td>6613</td>
<td>2549602</td>
</tr>
<tr>
<td>Zenon</td>
<td>15.4GB</td>
<td>10330</td>
<td>5032442</td>
</tr>
</tbody>
</table>

Figure 7. Size of commands for evaluated ITP datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>Theories</th>
<th>Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matita</td>
<td>2.0MB</td>
<td>19</td>
<td>478</td>
</tr>
<tr>
<td>HOL Light</td>
<td>2.0GB</td>
<td>25</td>
<td>1776535</td>
</tr>
<tr>
<td>Isabelle/HOL</td>
<td>2.5GB</td>
<td>1</td>
<td>116927</td>
</tr>
<tr>
<td>iProver</td>
<td>431.4MB</td>
<td>6613</td>
<td>2549602</td>
</tr>
<tr>
<td>Zenon</td>
<td>15.4GB</td>
<td>10330</td>
<td>5032442</td>
</tr>
</tbody>
</table>

Table 1. Statistics for evaluated datasets.

B method set theory generated by Zenon Modulo [9]. For the ITP datasets, I evaluate parts of the standard libraries from HOL Light (up to finite Cartesian products) and Isabelle/HOL (up to Hol.List), as well as Fermat’s little theorem proved in Matita [31]. An evaluation of Coq datasets is unfortunately not possible because its encoding relies on higher-order rewriting.

Statistics for the datasets are given in Table 1. The distribution of the sizes of the commands for the ITP datasets is shown in Figure 7. A data point \((x, y)\) on the figure means that the \(x\)th largest command in a dataset is \(y\) bytes large. Therefore, given a graph for a dataset, the \(y\)-coordinate of its leftmost point is the size of the largest command in the dataset, and the \(x\)-coordinate of its rightmost point is the total number of commands in the dataset. The figure shows us for example that the largest command of the Isabelle/HOL dataset is several orders of magnitude larger than the largest command of the HOL Light dataset, and that each of the approximately \(10^3\) largest commands of the Isabelle/HOL dataset is larger than the largest command of the HOL Light dataset.

I evaluate different configurations of Dedukti and Kontroli that correspond to the types of verification introduced in subsection 2.2. First, I evaluate sequential verification; that is, processing always at most one command from a single theory. The configurations of Dedukti and Kontroli that perform sequential verification are called DK and KO. The configurations DK新冠病毒 and KO新冠病毒 perform only the parsing step of DK and KO. This serves to measure the impact of parsing on overall performance. Similarly, KO新冠病毒 omission the (type) checking step of KO. This serves as a lower bound for parallel checking, as will be explained below. Next, I evaluate concurrent verification. The configuration DK新冠病毒 performs theory-concurrent verification; that is, processing at most \(n\) theories concurrently, but processing at most one command from every theory at the same time. When \(n = \infty\), an unlimited amount of theories is processed concurrently. The remaining configurations perform command-concurrent verification; that is, processing at most one theory at the same time, but processing several commands from this theory concurrently. KO新冠病毒 performs parallel parsing using a single separate parse thread. KO新冠病毒 performs parallel (type) checking of at most \(n\) commands simultaneously, using Arc-shared terms with one checking thread per command. As mentioned above, the runtime of KO新冠病毒 (KO without type checking) serves as lower bound for the runtime of KO新冠病毒. The type checking time of a Kontroli configuration is the difference between the runtime of the configuration and the runtime of KO新冠病毒.
1.5x on Matita, and there is no statistically significant improvement between \( n = 4 \) and \( n = 8 \) threads. Type checking parallelises best on Isabelle/HOL, where \( KO_{c=n} \) reduces type checking time compared to KO by 1.6x for \( n = 2 \) threads, 3.3x for \( n = 4 \) threads, and 6.6x for \( n = 8 \) threads.\(^6\)

The peak memory consumption of a few configurations is shown in Figure 9. On the Matita dataset, all Kontroli configurations consume less memory than Dedukti, and memory usage slightly increases when increasing the number of checking threads. On the HOL Light dataset, we have the interesting case that all configurations consume roughly the same amount of memory, regardless of concurrency. This can be explained by the relatively small size of the commands in that dataset. On the Isabelle/HOL dataset, we note two peculiarities: First, KO uses significantly more memory than DK. As explained in section 7, this is because Kontroli’s parser keeps the whole input file in memory until the theory is checked, whereas Dedukti’s parser loads the input file as needed and discards the parts that were parsed. If we subtract the size of the Isabelle/HOL dataset (a single theory of 2.5 GB) from KO’s memory consumption, we arrive at a memory consumption close to DK. Second, with increasing number of checking threads, the memory consumption of KO rises drastically. This can be explained as follows: Figure 7 shows that the Isabelle/HOL dataset features larger commands than the HOL Light dataset. Larger commands tend to take more space and time to check than smaller commands. The total memory consumption is composed of the memory consumption of all checking threads. Therefore, when increasing the number of checking threads, in a dataset with larger commands such as Isabelle/HOL, a high peak memory consumption is likelier to occur than in a dataset with smaller commands such as HOL Light.

For the ATP datasets shown in Figure 10, we have that Kontroli is faster than Dedukti. Kontroli checks the Zenon dataset in 62.1% of the time taken by Dedukti. In conclusion, on the evaluated datasets, Kontroli consistently improves performance over Dedukti, both in sequential and in concurrent settings.

9 Related Work

The related work can be divided by two criteria, namely size and concurrency. Work related to small size is mostly about proof checkers, and work related to concurrency is about proof assistants. To the best of my knowledge, this work is the first that combines the two aspects by creating a proof checker that is both concurrent and small.

9.1 Proof Checkers & Size

The type-theoretic logical framework LF is closely related to Dedukti, being based on the lambda-Pi calculus by Harper et al. [20]. Appel et al. have created a proof checker for LF that is similar to this work due to their pursuit of small size [1]. Their proof checker consists of 803 LOC, where the kernel (dealing with type checking, term equality, DAG creation and manipulation) consists of only 278 LOC and the prekernel (dealing with parsing) consists of 428 LOC. The small size of the proof checker is remarkable considering that it is written in C and does not rely on external libraries.

LFSC is a logical framework that extends LF with side conditions. It is used for the verification of SMT proofs, where LFSC acts as a meta-logic for different SMT provers, similarly to Dedukti acting as meta-logic for different proof assistants [29]. Stump et al. have created a proof checker generator for
Concurrent proof checking is nowadays mostly found in interactive theorem provers. Early work includes the Distributed Larch Prover [32] and the MP refiner [26].

The Paral-ITP project improved parallelism in provers that were initially designed to be sequentially executed, such as Coq and Isabelle [6]. Among others, as part of the Paral-ITP project, Barras et al. introduced parallel proof checking in Coq that resembles this work in the sense that it delegates checking of opaque proofs [7]. However, unlike this work, Coq checks the opaque proofs using processes instead of threads, requiring marshalling of data between the prover and the checker processes.

Isabelle features concurrency on multiple levels: Aside from concurrently checking both theories and top-level proofs (similar to Dedukti and Kontroli), it also concurrently checks sub-proofs. Furthermore, it executes some tactics in parallel, Python. Furthermore, Metamath allows to import OpenTheory proofs and thus to verify proofs from HOL Light, HOL4, and Isabelle [10].

The aut program is a proof checker for the Automath system developed by Wiedijk [36]. It is written in C and consists of 3048 LOC. It can verify the formalisation of Landau’s “Grundlagen der Analysis”.

HOL Light is a proof assistant whose small kernel (396 LOC of OCaml) qualifies it as a proof checker [21]. However, the code in HOL Light that extends the syntax of its host language OCaml is comparatively large (2753 LOC). Among others, HOL Light has been used to certify SMT [29] as well as tableaux proofs [14, 23]. Checking external proofs in a proof assistant also benefits its users, who can use external tools as automation for their own work and have their proofs certified.

9.2 Proof Assistants & Concurrency

Concurrent proof checking is nowadays mostly found in interactive theorem provers. Early work includes the Distributed Larch Prover [32] and the MP refiner [26].

The Paral-ITP project improved parallelism in provers that were initially designed to be sequentially executed, such as Coq and Isabelle [6]. Among others, as part of the Paral-ITP project, Barras et al. introduced parallel proof checking in Coq that resembles this work in the sense that it delegates checking of opaque proofs [7]. However, unlike this work, Coq checks the opaque proofs using processes instead of threads, requiring marshalling of data between the prover and the checker processes.

Isabelle features concurrency on multiple levels: Aside from concurrently checking both theories and top-level proofs (similar to Dedukti and Kontroli), it also concurrently checks sub-proofs. Furthermore, it executes some tactics in parallel, Python. Furthermore, Metamath allows to import OpenTheory proofs and thus to verify proofs from HOL Light, HOL4, and Isabelle [10].

The aut program is a proof checker for the Automath system developed by Wiedijk [36]. It is written in C and consists of 3048 LOC. It can verify the formalisation of Landau’s “Grundlagen der Analysis”.

HOL Light is a proof assistant whose small kernel (396 LOC of OCaml) qualifies it as a proof checker [21]. However, the code in HOL Light that extends the syntax of its host language OCaml is comparatively large (2753 LOC). Among others, HOL Light has been used to certify SMT [29] as well as tableaux proofs [14, 23]. Checking external proofs in a proof assistant also benefits its users, who can use external tools as automation for their own work and have their proofs certified.
for example the simplification of independent subgoals [33, 34].

Like Isabelle, ACL2 checks theories and toplevel proofs in parallel, but differs from Isabelle by automatically generating subgoals that are verified in parallel [27]. In both Isabelle and ACL2, threads are used to handle concurrent verification.

10 Conclusion

In this work, I presented several techniques to parallelise proof checking. I introduced a term type that abstracts over the type of constants and term references, allowing it to be used both sequentially and concurrently in parsing and checking. I further refined the term type by reducing the number of pointers, especially improving concurrent performance. I showed that parallelising reduction using abstract machines involves replacing several thread-unsafe data types by thread-safe ones, adding up too much overhead to reduce checking time in practice. I showed that command-concurrent verification can be achieved by breaking verification into an inference and a checking operation, where multiple checking operations can be executed concurrently. This necessitates thread-safe global contexts and terms. To allow for both overhead-free sequential and concurrent verification, I created two versions of the kernel that differ only by the used term type. I showed that parsing can be parallelised by moving it to a separate thread, from which the parsed commands are sent to the main thread via a channel. The overhead of sending commands through a channel unfortunately is so high that parallel parsing does not improve performance.

I implemented these techniques in a new proof checker called Kontroli. Kontroli is written in the programming language Rust, which played a crucial role to ensure memory- and thread-safety, while allowing for a small kernel that can be used for efficient sequential and parallel checking. The evaluation shows that on all datasets, sequential Kontroli is faster than sequential and theory-concurrent Dedukti. On the Isabelle/HOL dataset, the command-concurrent Kontroli speeds up type checking by 6.6x when using eight threads.

Acknowledgments

I would like to thank François Thiré for the inspiration to write this article. Furthermore, I would like to thank Gaspard Ferey, Guillaume Genestier and Gabriel Hondet for explaining to me the inner workings of Dedukti, and Emilie Grienenberger for providing me with the Dedukti export of the HOL Light standard library. Finally, I would like to thank the anonymous CPP reviewers, David Cerna, Thibault Gauchier, Guillaume Genestier, Emilie Grienenberger, Gabriel Hondet, Fabian Mitterwallner, and François Thiré for their helpful comments on drafts of this article. This research was funded in part by the Austrian Science Fund (FWF) [J 4386].

References


