Small, Fast, Concurrent Proof Checking for the lambda-Pi Calculus Modulo Rewriting

Michael Färber

Small, Fast, Concurrent Proof Checking for the lambda-Pi Calculus Modulo Rewriting

Michael Färber
Université Paris-Saclay, ENS Paris-Saclay, CNRS, Inria, LSV, 91190, Gif-sur-Yvette, France
michael.faerber@gedenkt.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 presents a small proof checker with support for concurrent proof checking, achieving state-of-the-art performance in both concurrent and non-concurrent settings. The proof checker implements the lambda-Pi calculus modulo rewriting, which is an established framework to uniformly express a multitude of logical systems. The proof checker 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.

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

1 Introduction

> *Perfection is attained*
> *not when there is nothing more to add,*
> *but when there is nothing more to remove.*
>
> — Antoine de Saint-Exupéry,
> *Wind, Sand and Stars*

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 [17], which has been verified using the proof assistants HOL Light [21] and Isabelle [38], and the Four-Colour Theorem [16], 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 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 [15], and the B method [18].

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 allows only for a restricted 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 exploit multiple cores. This article deals with the following question: how to implement a small and efficient concurrent proof checker, and how much time can be saved using such a checker?

This work sets out to answer the research question by reimplementing a small, yet expressive subset of the Dedukti kernel in the programming language Rust. The resulting kernel allows for concurrent proof checking, compromising neither conciseness of the code nor performance. The proof checker built around this kernel is faster than Dedukti on all of five evaluated datasets. To the best of my knowledge, this is also the first proof

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

checker that is both small in the sense of the de Bruijn criterion as well as concurrent.

The contributions of this work are the following: the presentation of an architecture allowing for concurrent proof checking, the implementation of a proof checking kernel for aforementioned architecture, and an evaluation of its performance with benchmarks derived from automated and interactive theorem provers.

2 Background

2.1 The lambda-Pi Calculus Modulo Rewriting

I introduce here the lambda-Pi calculus modulo rewriting. Saillard's PhD thesis presents it in more detail [30].

A term $t \in \mathcal{T}$ has the shape

$$t \coloneqq c \mid s \mid tu \mid x \mid \lambda x{:}t.\, u \mid \Pi x{:}t.\, u,$$

where $c \in \mathcal{C}$ is a constant, $s \coloneqq \text{Type} \mid \text{Kind}$ is a sort, t and u are terms, and x is a bound variable. We denote the substitution of x in t by u as tux.

A rewrite pattern p has the shape $p \coloneqq x \mid cp_1 \dots p_n$, where x is a bound variable, $c \in \mathcal{C}$ is a constant, and $p_1 \dots p_n$ is a potentially empty sequence of rewrite patterns applied to c. A rewrite rule $r \in \mathcal{R}$ has the shape $r \coloneqq cp_1 \dots p_n \hookrightarrow t$, where we call $cp_1 \dots p_n$ 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. ${}_i\,\mathcal{FV}\text{ar}p_i \supseteq \mathcal{FV}\text{ar}t$.[1]

A *signature* for a set of constants $\mathcal{C}$ is a pair $T, R \in \mathcal{C} \to \mathcal{T} \times \mathcal{C} \nrightarrow 2^{\mathcal{R}}$, where Tc returns the type of the constant c and Rc returns the set of rewrite rules having c as head symbol. Here, R is a partial function with the property that $c \in \text{dom}R$ iff we want to allow the addition of rewrite rules having c as head symbol. Restricting the rewritability of a constant is useful to assure that it is injective.

A *context* Γ is a function from variables to terms.

For a signature $\Sigma = T, R$, we say that $\Sigma \vdash c : A$ if $Tc = A$ and $\Sigma \vdash t \hookrightarrow u$ if $t \hookrightarrow u \in Rc$, where c is the head symbol of t. Furthermore, we say that for a context Γ, $\Gamma \vdash x : A$ if $\Gamma x = A$. We say that the term t has the type A under the signature Σ if we can find a derivation of $\Sigma, \Gamma \vdash t : A$ using the rules in Figure 1 [adapted from 30, Figure 2.4], where Γ is a context with an empty domain.

We can beta-reduce terms via $\lambda x.tu \to_\beta tux$. Additionally, we have a reduction rule $t \to_{\gamma\Sigma} u$ which is admissible when there is a term rewrite rule $\Sigma \vdash t' \hookrightarrow u'$ and a substitution σ, so that $\sigma t' = t$ and $\sigma u' = u$.

[1] To simplify the presentation, I only introduce first-order rewriting. Note that Dedukti uses higher-order rewriting [26].

$$\text{Type}\quad \frac{}{\Sigma, \Gamma \vdash \text{Type} : \text{Kind}}$$

$$\text{Application}\quad \frac{\Sigma, \Gamma \vdash t : \Pi x{:}A.\, B \quad \Sigma, \Gamma \vdash u : A}{\Sigma, \Gamma \vdash t\, u : Bux}$$

$$\text{Abstraction}\quad \frac{\Sigma, \Gamma\, \{x \to A\} \vdash t : B \quad \Sigma, \Gamma \vdash \Pi x{:}A.\, B : s}{\Sigma, \Gamma \vdash \lambda x{:}A.\, t : \Pi x{:}A.\, B}$$

$$\text{Product}\quad \frac{\Sigma, \Gamma \vdash A : \text{Type} \quad \Sigma, \Gamma, x : A \vdash t : s}{\Sigma, \Gamma \vdash \Pi x{:}A.\, t : s}$$

$$\text{Conversion}\quad \frac{\Sigma, \Gamma \vdash t : A \quad \Sigma, \Gamma \vdash B : s \quad A \sim_\Sigma B}{\Sigma, \Gamma \vdash t : B}$$

Figure 1. Inference rules.

Let $\to_\Sigma = \to_\beta \cup \to_{\gamma\Sigma}$ be our reduction relation.[2] We say that two terms t, u are Σ-convertible, i.e. $t \sim_\Sigma u$, when there exists a term v such that $t \to_\Sigma^* v$ and $u \to_\Sigma^* v$.

Type inference determines a unique term A for a term t and a signature Σ such that $\Sigma \vdash t : A$. Type checking verifies for terms t and A and a signature Σ whether $\Sigma \vdash t : A$. If the reduction relation $\to_\Sigma^*$ is type-preserving, terminating, and confluent, then type inference and type checking terminate [30, Theorem 6.3.1].

2.2 Verification

In mathematics, a *theory* can be seen as a directed graph of definitions, axioms, and theorems. In contrast to this, I consider a theory to be a sequence of commands, which I will introduce in this section. In the following, I explain how to verify a theory.

To verify a theory, it suffices to treat the sequence of its commands one by one. The set of constants $\mathcal{C}$ and the signature Σ for $\mathcal{C}$ capture the state of the theory verification, i.e. the essence of all previously treated commands. We start from an empty set of constants $\mathcal{C} = \emptyset$ and an initial signature T, R with $Tc = \bot$ and $\text{dom}R = \emptyset$. For every command, we update the constants $\mathcal{C}$ and the signature T, R. The theory is verified if this process does not fail at any step. I will now define commands, before showing how to update the current state with a command.

A *command* either introduces a new constant or adds a rewrite rule. A new constant c can be introduced in three different ways: (a) A *declaration* $c : A$ introduces an injective constant c of type A. (b) A *definition* introduces a constant c with either a type A ($c : A$), a defining term t ($c \coloneqq t$), or both ($c : A \coloneqq t$, which should be read as $c : A \coloneqq t$). Only constants introduced by definition may occur as head symbol of the left-hand side of rewrite

[2] The implementations of the calculus optionally eta-reduce terms via $\lambda x.tx \to_\eta t$.

rules. If t is given, then occurrences of c in subsequent commands are replaced by t, and if A is additionally given, then the type of t must be convertible with A. (c) A *theorem* $c : A \coloneqq t$ introduces a constant c of type A with a proof term t, where the type of t must be convertible with A and occurrences of c in subsequent commands are *not* replaced by t; that is, t is an opaque proof.

Example 2.1. Consider the following theory about implication: The first two commands *declare* prop : Type and imp : prop → prop → prop, where $t \to u$ is notation for $\Pi x{:}t.\, u$. The next command *defines* prf : prop → Type. Having defined (rather than declared) prf allows the next command to add a rewrite rule for prf, namely $\mathrm{prf}\,\mathrm{imp}\,x\,y \hookrightarrow \mathrm{prf}\,x \to \mathrm{prf}\,y$. The last command then introduces a *theorem* imp_refl : $\Pi x : \mathrm{prop}\,.\ \mathrm{prf}\,\mathrm{imp}\,x\,x \coloneqq \lambda x : \mathrm{prop}\,.\,\lambda p : \mathrm{prf}\,x.\,p$.

I now introduce a new concept to abstract from the several ways by which constants can be introduced.

A *typing* is a triple $c, A, t \in \mathcal{C} \times \mathcal{T} \times \mathcal{T} \cup \{\square\}$, asserting $c : A$ when $t = \square$, otherwise $c : A \coloneqq t$. We *infer* a typing from declarations, definitions, and theorems with a signature Σ as follows: (a) A declaration or definition $c : A$ becomes a typing $c, A, \square$ if $\Sigma \vdash A : \mathrm{Kind}$ or $\Sigma \vdash A : \mathrm{Type}$. (b) A definition or theorem $c : A \coloneqq t$ becomes a typing c, A, t if $\Sigma \vdash A : A'$ for some A' and $A \neq \mathrm{Kind}$. (c) A definition $c \coloneqq t$ becomes a typing c, A, t if $\Sigma \vdash t : A$ and $A \neq \mathrm{Kind}$. We *check* a typing with Σ by verifying that either $t = \square$ or $\Sigma \vdash t : A$. Note that checking a typing can only fail when the typing was inferred from a definition or theorem $c : A \coloneqq t$.

Now we can finally state how to update the current state with a new command. That is, given some command, how to update a set of constants $\mathcal{C}$ and a signature $\Sigma = T, R$ for $\mathcal{C}$ to a set of constants $\mathcal{C}'$ and a signature $\Sigma' = T', R'$ for $\mathcal{C}'$? For this, we distinguish the two kinds of commands.

If the command introduces a new constant c, then we fail if $c \in \mathcal{C}$, otherwise we infer and check the typing $c : A \coloneqq t$ with Σ and set $\mathcal{C}' = \mathcal{C} \cup \{c\}$ as well as $T' = T\{c \to A\}$. If the command does not introduce c by a definition, then $R' = R$, otherwise we initialise the set of rewrite rules for c by $R' = R\{c \to \emptyset\}$ if $t = \square$ and $R' = R\{c \to \{c \hookrightarrow t\}\}$ otherwise.

If the command adds a rewrite rule $r \in \mathcal{R}$ with a head symbol c, then we keep $\mathcal{C}' = \mathcal{C}$ and $T' = T$. Furthermore, we add the new rewrite rule r to the existing rules for c by $R' = R\{c \to Rc \cup \{r\}\}$. Note that this fails if $c \notin \mathrm{dom} R$; that is, if c was not introduced via a definition.[3]

[3]To assure termination of type inference and checking, it is necessary to assure subject reduction, termination, and confluence of rewrite rules. However, even a theory verifier that does not verify these properties can be useful, because many theories add only a fixed number of rewrite rules. This makes it feasible to establish aforementioned properties for such theories by other means.

Listing 1. Structural and physical equality in OCaml.

```
let a = Some(0) in
let b = a in
let c = Some(0) in
assert (a =  b);
assert (b =  c);
assert (a == b);
assert (not (b == c));
```

Listing 2. Structural and physical equality in Rust.

```
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));
```

2.3 Sharing and Concurrency

There are two closely related technical subjects relevant to this work: 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 consider physically equal terms to be 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).

One common technique to manage memory of shared objects is *reference counting*: A reference-counted object keeps a counter to register how 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.

While reference counting is efficient in single-threaded scenarios, it can pose performance problems in multi-threaded scenarios: 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

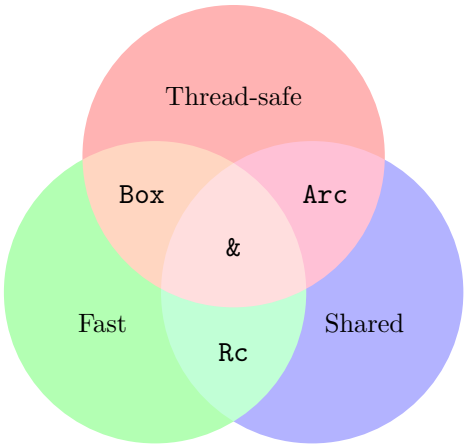


Figure 2. Venn diagram of common Rust pointer types and their properties.

turning 0 despite references still pending) and memory leaks (a counter remaining greater than 0 despite no references left). However, atomic modifications imply a significant runtime overhead. We call data structures that can be safely shared between threads *thread-safe*.

Unlike implicitly sharing languages, explicitly sharing languages 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 2. 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.[4]

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

3 Concurrent Verification

Concurrent verification designates the simultaneous verification of different parts of a theory. Following Wenzel's terminology [37], 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).[5]

[4]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.

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

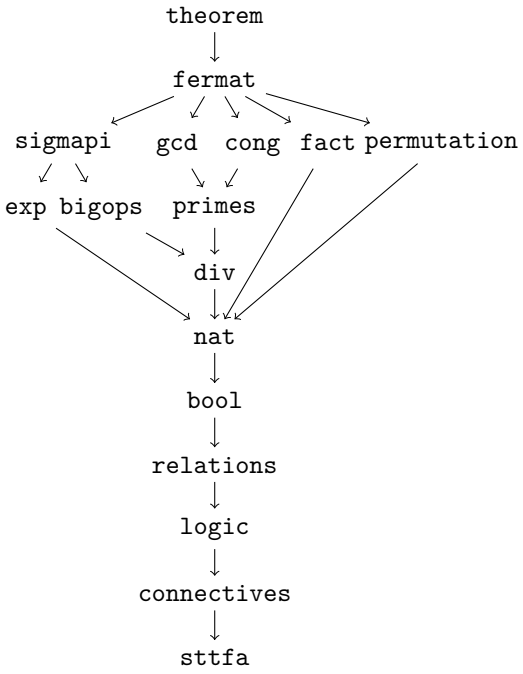


Figure 3. Theory dependency graph of Fermat's little theorem in Matita, encoded in STTfa.

3.1 Theory-Concurrent Verification

Theory-concurrent verification exploits the fact that we can divide a theory 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. Theories that do not transitively depend on each other can be checked concurrently.

An example of a theory dependency graph is shown in Figure 3 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 3 we can verify at most six theories concurrently, namely `exp`, `bigops`, `gcd`, `cong`, `fact`, and `permutation`.

Theory-concurrent verification can be implemented by launching a verification process for every theory, producing for every theory a signature that contains the typings and rewrite rules contained in that theory. To verify a theory, it is necessary to load the signatures of the theory's dependencies. As loading of signatures 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.

3.2 Command-Concurrent Verification

Command-concurrent verification allows for the concurrent verification of commands regardless of the theory

Listing 3. Definition of a term type for parsing.

```
enum PTerm {
    Constant(String),
    Application(Box<PTerm>, Box<PTerm>),
}
```

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 signature between concurrent verifications and thus to omit the I/O overhead of loading signatures, which would become noticeable if done for every command. However, this comes at the cost of using shareable data structures for the signature, as I will discuss in section 4.

I will focus on command-concurrent verification in section 4, as it is the model that I have implemented for this work, see section 5. I will evaluate the two approaches in section 6.

4 Command-Concurrent Verification

In this section, I show how to concretely implement the abstract theory verification procedure introduced in subsection 2.2 such that commands can be verified concurrently.

4.1 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.

There are several term transformations during proof checking. It turns out to be beneficial to define multiple term types corresponding to these transformations.

The first of these term types, called `PTerm`, is produced by the parser. As parsing does not perform any sharing, it makes sense to make `PTerm` unshared, which will allow it to be efficiently sent across threads. A simplified version of this type, restricted to constants and binary application, is shown in Listing 3. We can see that `PTerm` represents constants by `String`s, which are unshared. Furthermore, in the application, we see that `PTerm` is wrapped in the pointer type `Box`, which is also unshared. As a result, `PTerm` as a whole is unshared.

It turns out that all term types we need can be defined by abstracting over the type of constants `C` (`String` in `PTerm`) and the type of term pointers `T` (`Box<PTerm>` in `PTerm`). This yields an abstract term type shown in Listing 4, which can be instantiated to define `PTerm` as well as two other types, `STerm` and `TTerm`.

Listing 4. Definition of term types for parsing, scoping, and typing, using an abstract term type.

```
enum Term<C, T> {
    Constant(C),
    Application(T, T),
}

struct PTerm(Term<String, Box<PTerm>>);
struct STerm<'c>(Term<&'c str, Box<STerm<'c>>>);
struct TTerm<'c>(Term<&'c str, Ptr<TTerm<'c>>>);
```

`STerm` (for "scoped term") is a term type that uses `&str`, i.e. string references, instead of `String` to represent constants. String references are a natural choice here because comparing and hashing `&str` (which are frequent operations during typing) takes constant time and dereferencing `&str` is always guaranteed to yield a string.[6] Furthermore, using `&str` instead of e.g. `Rc<String>` ensures that freeing a term does not involve modifying any reference counters for its constants. On the other hand, using `&str` requires us to ensure that the referenced strings remain in memory during checking, which can be achieved e.g. by putting every newly introduced constant name into a designated region [19].

`TTerm` is a term type that is used during typing, i.e. type inference and type checking. As these operations reduce terms, it is beneficial to allow for the sharing of `TTerm`s. However, we would like to use variants of `TTerm` that use the efficient, thread-unsafe pointer type `Rc` when typing using a single thread and slow, thread-safe pointer type `Arc` when typing using multiple threads. This can be achieved by generating two versions of the kernel, where in one, `Ptr` is substituted by `Rc` and in the other, `Ptr` is substituted by `Arc`. This generates one kernel version for sequential and one for concurrent scenarios.[7]

Having defined the basic term types, we can define other data structures that contain terms. In particular, we define `PCommand` and `SCommand` to be commands that contain `PTerm`s and `STerm`s, respectively.

[6] If we would map constants to, say, integers instead of string references, which have similar performance characteristics, we would need to manually maintain the invariant that every integer represents a valid constant. This can be avoided by using string references, at the price of needing to assure their lifetime.

[7] A more elegant and flexible solution might be enabled by the implementation of generic associated types (similar to higher-kinded types) in Rust, allowing for the definition of an abstract type `Term<C, Ptr>`.

4.2 The Procedure

I will now explain how a theory file is handled in practice. To this end, I introduce a procedure building on the one in subsection 2.2. The repeated execution of this procedure on a file until its consumption corresponds to verifying the file.

This procedure is illustrated in Figure 4, where edge labels (e.g. `PCommand`) represent temporary data relevant to the current run of the procedure, boxed nodes (the current "File", the set of constants $\mathcal{C}$, and the signature Σ) represent data persisting between runs of the procedure, and unboxed nodes (e.g. "parse") represent functions.

The procedure performs the following: First, we *parse* a command from the current position of the file and update the current position. Parsing yields a `PCommand`. We then *scope* the `PCommand`, which assures that all unbound symbols occurring in the `PCommand`'s terms or rewrite rules were previously introduced into the set of constants. Scoping yields an `SCommand`, in which identical constants are mapped to physically equal objects. After scoping, we perform the procedure outlined in subsection 2.2, by distinguishing the type of the `SCommand`: If the command adds a rewrite rule (the `Rule` case), we add the rewrite rule to the signature Σ. If the command introduces a new constant c (the `Intro` case), we add c to the set of constants $\mathcal{C}$. We then convert any `STerm` inside the command to a `TTerm`, in order to infer and check a typing using Σ. Finally, we add the typing to Σ.

We are now going to illustrate how the procedure in Figure 4 deals with a sequence of commands. To simplify the resulting image, let us assume that we only deal with commands that introduce constants. This results in a lattice-like graph shown in Figure 5. This lattice shows the flow of the persistent data structures ("File", $\mathcal{C}$, Σ) from top to bottom and the flow of temporary data structures (`PCommand`, `SCommand`, `Typing`) from left to right. To obtain an item at an outgoing arrow, all incoming arrows have to be (transitively) satisfied first; for example, to obtain the second `SCommand`, it is necessary to first have obtained the second `PCommand` and the constants resulting from the first `PCommand`. However, the second `SCommand` requires neither the first `SCommand` nor the first `Typing`.

A crucial detail is the special handling of infer and check: The signature update is handled uniquely by infer and does not depend on check, which makes it possible to *defer* checking. In the remainder of this section, I will show how to exploit this to parallelise checking.

4.3 Execution Orders

The lattice in Figure 5 specifies dependencies between data; however, it does not impose a particular order of operations. For this, I have selected three orders in Figure 6.

The first order in Figure 6a corresponds to single-threaded execution, which is what is implemented in Dedukti. The leftmost parse block has to be supplied with a file, a set of constants, and a signature. It passes these three components between its succeeding scope/infer/check operations, which have to succeed before a new command is treated by the next connected block, which is fed with the updated file, constants, and signature.

The second order in Figure 6b corresponds to two-threaded execution, where parsing is separated from the rest. Parsing is a natural candidate because it is the largest operation that can be separated from the others without requiring slow thread-safe data structures. This is due to the fact that the output of parsing, namely `PCommand`, can be transferred safely between threads due to being unshared, whereas other data structures, such as `Typing`, cannot be transferred between threads without overhead due to being shared. Figure 6b shows that parsing is performed at the same time as scope/infer/check, and the outputs of the parse thread (`PCommand`s) are passed to the main thread. Here, communication across threads is indicated by lines ending in arrows.

The third order in Figure 6c corresponds to multi-threaded execution scaling to an arbitrary number of threads. This cuts the main thread shown in Figure 6b into two parts, namely scope/infer and check. It is safe to do this because the main thread does not depend on the result of the check thread, other than reporting errors from it. Usually, the check function is more expensive than the scope and infer functions, so it makes sense to move it to an own thread. However, as discussed before, this comes at a cost: Because we move a shared data structure (`Typing`) between threads, we have to use thread-safe terms in *all* data structures used by the kernel, including the signature.

4.4 Towards Overhead-Free Concurrency

Using thread-safely shared data structures implies a considerable overhead compared to thread-unsafely shared ones, as I will show in section 6. I will now discuss a few ideas to avoid the usage of thread-safe sharing in the future.

The most critical point for concurrency is between scoping and typing. Typing operations operate on shared data structures. If we store shared data structures in the main thread, then they have to be thread-safely shared in order to enable concurrency. This, as shown in section 6, brings a considerable overhead. As the signature itself usually contains mostly entries whose size is linear in the size of the input (except for types that are inferred from a definition), it seems natural to use unshared structures for the signature. However, we then have to

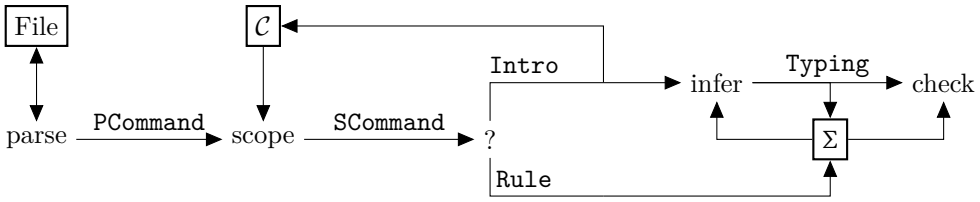


Figure 4. The command processing flow.

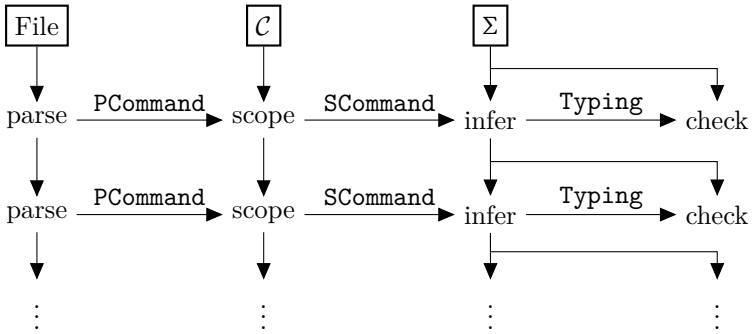
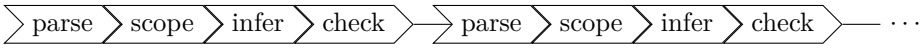
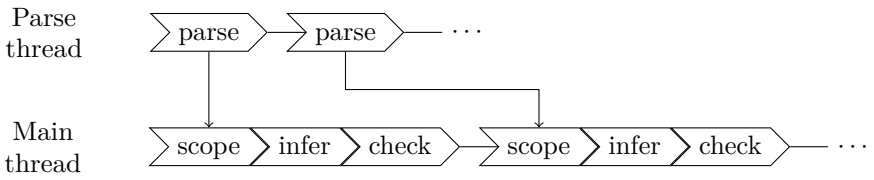


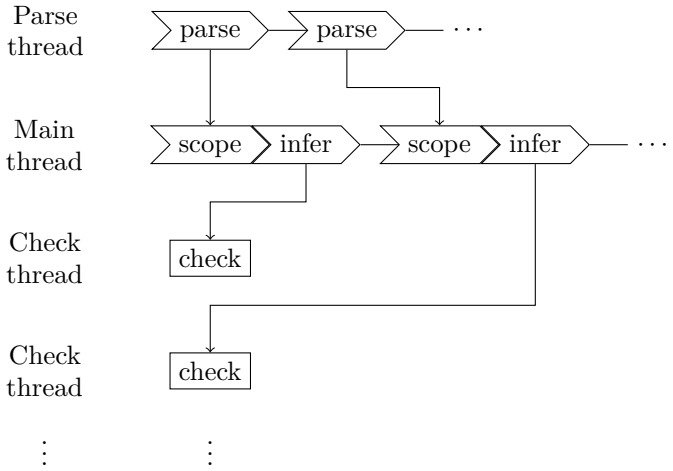
Figure 5. The introduction command processing lattice.



(a) Sequential processing.



(b) Parallel processing with two threads.



(c) Parallel processing with more than two threads.

Figure 6. Execution strategies.

convert between unshared and shared structures when typing. Doing this whenever an entry from the signature is retrieved is quite costly and does not compensate the omitted overhead implied by thread-safe sharing.

An alternative is to cache the signature entries for constants we have seen during typing of one command, and to reuse the cached shared data structures afterwards. In my experiments, this turned out to be also more costly than using thread-safe sharing. One future idea is to cache shared signature entries persistently *across* commands. However, this can significantly increase memory consumption, because in the worst case, all check threads keep the whole signature in memory. This could be remedied by smarter schemes, such as a cache with fixed size, removing typing info for constants not seen for a certain time. Also, this complicates the structure of the proof checker, because it requires to keep a per-thread data structure, which requires a more complicated program structure than the currently used map-reduce style processing.

One might be tempted to use only a single, global signature and to pass it to all check threads. However, at this point it becomes important to recall that the simplification we used so far does not consider rewrite rules. Rewrite rules can change typing behaviour, so if we were to use a single global signature, then adding a rewrite rule might influence checking of commands given before the rewrite rule. To prevent such situations, we duplicate the signature whenever we pass it to a check thread. This is an inexpensive operation, as we use for the signature a functional hash map that can be copied with minimal overhead.

4.5 Parallel Checking

Could we go further, by dividing between multiple threads the verification of a single command? To explore this question, I parallelised substitution and pattern matching, two of the most frequently used operations during type checking. In particular, due to $\sigma f t_1, \ldots, t_n = f \sigma t_1, \ldots, \sigma t_n$, we can calculate multiple σt_i in parallel and then merge the results. The problem here is that some σ functions we use are thread-unsafe. In particular, the evaluation of terms uses a σ that maps variables to shared lazy terms that are evaluated from a shared mutable state. Because this σ is normally not shared between threads (even when multiple check operations are executed in parallel), the data type of σ contains four instances of thread-unsafe types (for sharing, lazy evaluation, again sharing, and mutation). To parallelise this σ, it is thus necessary to replace each of these types by a slower thread-safe version. To parallelise matching, i.e. to check for multiple rewrite patterns in parallel whether they match a shared lazy term, the same type replacement as for parallelising substitution is required. As a result, the type checking performance with either parallelised substitution or parallelised matching is far below the performance of single-threaded execution.

5 Implementation

I implemented a small, parallel verifier named *Kontroli*[8], supporting all execution strategies shown in section 4. It is implemented in Rust, a functional programming language that forgoes a garbage collector in favour of RAII [23].

Kontroli tries to satisfy the following properties:

- **Minimality**: The size of a proof checker is relevant for our trust in its correctness, as a smaller code base can be more easily evaluated and understood than a larger one. Furthermore, as can be observed from other systems with small kernels, such as the proof assistant HOL Light [21] or the theorem prover leanCoP [28], such systems lend themselves well to modifications.
- **Concurrency**: The kernel of Kontroli is concurrency-agnostic, that is, it does not presuppose its usage in either concurrent or non-concurrent settings. This allows the kernel to perform without overhead in a non-concurrent context, while allowing at the same time to be used concurrently.
- **Performance**: The theories exported from proof assistants and automated theorem provers can be large, with several gigabytes not being an exception. It is therefore important to process data efficiently in order to reduce the waiting time for users.
- **Safety**: Imperative programming languages such as C/C++ often offer performance at the expense of (memory) safety, whereas functional programming languages such as Haskell or OCaml often offer safety at the expense of performance. However, for a proof checker operating on large amounts of data, both properties are highly desirable. The Rust programming language is a hybrid between imperative and functional programming, offering both performance and memory safety.
- **Compatibility**: In order to use the many datasets available for Dedukti, Kontroli aims to be compatible with Dedukti.

We will now look at the structure of Kontroli, see Figure 7. Kontroli is divided into two parts: a library and a command-line program, where the library is divided into a prekernel and a kernel. The division between these parts obeys the following principles: The library is the largest possible part of the type checker that contains only I/O-free functions, and the prekernel is the largest possible part of the library that is free of shareable smart (i.e. reference-counted) pointers. Consequentially,

[8] Esperanto for "to check", "to verify".

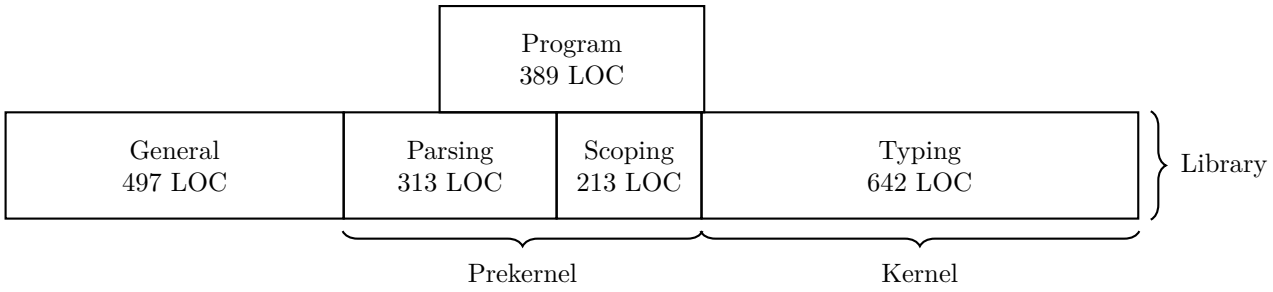


Figure 7. The structure of Kontroli.

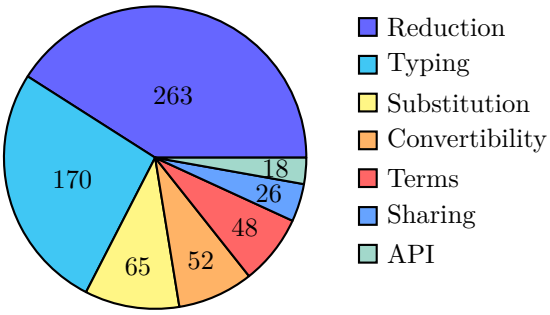


Figure 8. Lines of code of different Kontroli components.

the kernel is the smallest possible part of the library dealing with shareable smart pointers, and the program is the smallest possible part of the type checker dealing with I/O. The I/O-freeness of the library is verified by the Rust compiler (using the `#[no_std]` keyword). This allows the library to be used in restricted environments, such as web applications. The fact that the prekernel uses no shareable smart pointers can be syntactically established. This means that Rust allows us to come up with and to enforce clear boundaries between the different parts of the type checker.

What functionalities does the library provide? The prekernel implements parsing and scoping of terms and commands. The kernel implements type inference & checking, lazy evaluation via an abstract machine [2], substitution, convertibility checking, and first-order rewriting, whose respective sizes are shown in Figure 8. As described in section 4, the kernel is realised as a functor-like module that has a hole for a shared pointer type `Ptr`, allowing us to instantiate the kernel with both thread-safe and thread-unsafe pointer types. The Kontroli kernel consists of 642 lines of code, whereas the Dedukti kernel consists of 3470 lines. The Kontroli prekernel consists of 526 lines of code, whereas the Dedukti prekernel consists of 780 lines.[9]

To achieve its small size, Kontroli omits several features present in Dedukti:

- **Higher-order rewriting**, that is, rewrite patterns of the shape $\lambda x.p$ [26].
- **Matching modulo AC** [12]
- **Rewrite rule verification**: Several properties of the calculus demand that rewrite rules added as part of a theory preserve confluence, termination, and types. While these properties can be difficult to verify, there are many interesting theories for which the rewrite rules are fixed and few, compared to the total size of the theory. For this reason, it makes sense to omit the verification of rewrite rule properties from the type checker kernel, while leaving open the possibility to extend the type checker by appropriate checks outside of the kernel.
- **Evaluation & Assertion**: Dedukti has built-in commands to output the weak-head and strong normal form of terms, as well as to check the convertibility of terms. These functionalities are implemented in the kernel, although they are not strictly necessary for proof checking. We can emulate the convertibility check command in Kontroli: To verify that two terms $t, u : A$ are convertible, it suffices to declare $\mathrm{Eq} : x : A \to \mathrm{Type}$ and $\mathrm{eq} : x : A \to \mathrm{Eq}\, x$, before checking that $\mathrm{eq}\, t$ has the type $\mathrm{Eq}\, u$.

Furthermore, Kontroli does not implement several optimisations present in Dedukti, one of them being decision trees: Decision trees accelerate the matching of terms in the presence of many rewrite rules [22]. On the other hand, for the theories we consider in this article, decision trees are not strictly necessary for performance.

6 Evaluation

I am now going to evaluate the performance of Dedukti and several execution orders of Kontroli. The evaluation

[9]Dedukti was obtained from https://github.com/Deducteam/Dedukti, rev. 38e0c57. Kontroli was obtained from https://github.com/01mf02/kontroli-rs, rev. 5fb49d8. 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 `tokei src/pre`, and for Dedukti by `tokei kernel` and `tokei parsing && sed '/^\s*$/d' parsing/*.ml{l,y} | wc -l` (summing the results for the last command).

was performed on a cluster with 56 2.2GHz Intel Haswell CPUs, 120GB RAM and a 50GB HDD.

I evaluate Kontroli and Dedukti on two kinds of datasets: ATP and ITP problems. 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 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 [34]. An evaluation of Coq datasets is unfortunately not possible due to its current encoding relying on higher-order rewriting.

I evaluate different configurations of Dedukti and Kontroli. For Dedukti, I evaluate a configuration running at most one instance at a time (DK) and one configuration running as many instances as possible (DKj), using theory-level concurrency, see section 3. For Kontroli, I evaluate a configuration running at most one thread at a time (KO), one configuration where we parse at the same time as the rest (KOp), and several configurations using `Arc`-shared data structures with a maximum number of n simultaneous check and scope threads (KOn). For comparison, I also evaluate a version with simultaneous parsing and scope/infer, but without checking (KOi). This version serves as a lower bound for the runtime of KOn.

For the ATP datasets, proof checking is an embarrassingly trivial problem because we can check any amount of problems at the same time. Therefore, I use for both Kontroli and Dedukti a configuration that resembles DKj, but I limit the number of maximal running instances to 32.

All datasets were evaluated ten times and their average running time as well as the standard deviation were obtained.

I now discuss the results for the ITP datasets shown in Figure 9. The single-instance Dedukti configuration, DK, takes the most time across all datasets. Running several instances of Dedukti (DKj) significantly improves performance for those datasets that are divided into multiple theories, i.e. Matita and HOL Light. Still, on all datasets, the single-threaded Kontroli configuration (KO) is faster than running a maximum number of Dedukti instances in parallel. This can be explained by the theory graphs of the ITP libraries not being very "broad", which restricts the effectiveness of Dedukti's theory-level concurrency. This might be related to Wenzel's observation that the standard library of Isabelle was among the Isabelle libraries where parallelism yielded the least gain [36]. Parsing and scoping/checking in parallel (KOp) can improve performance, but as it comes with some overhead, it may also have the adverse effect, as we can see from the HOL Light dataset. One reason for this is the per-command overhead stemming from using channels, which becomes dominant when we have a large number of small commands, as is the case for the HOL Light dataset.

I now discuss the concurrent configurations. KO1 is always the slowest of all Kontroli configurations. That is because this configuration suffers from the `Arc`-induced overhead, but does not actually run scoping and typing concurrently. KO1 differs from KOp only in that it uses `Arc` instead of `Rc`. In that sense, KO1 is just a reference configuration that is there to show the overhead of `Arc` compared to `Rc` and to serve as a baseline for concurrent configurations. Starting from $n = 2$, all KOn configurations are faster than KOp, and starting from $n = 4$, all KOn configurations are the fastest overall. Furthermore, for large n, the time needed for Kontroli to check a theory converges towards the time needed for Kontroli to only parse and scope the theory. For example, parsing and scoping the Isabelle dataset without checking requires 406 seconds (KOi), which increases only by two seconds to 408 seconds (+0.45%) when checking with eight threads in parallel (KO8). In comparison, checking with a single thread with simultaneous parsing (KOp) increases total time to 454 seconds (+11.8%), and without simultaneous parsing (KO), time further rises to 475 seconds (+17.0%).

For the ATP datasets shown in Figure 10, we also have that Kontroli is faster than Dedukti.

In conclusion, on the evaluated datasets, Kontroli consistently improves performance over Dedukti, both in sequential and in concurrent settings.

7 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.

7.1 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

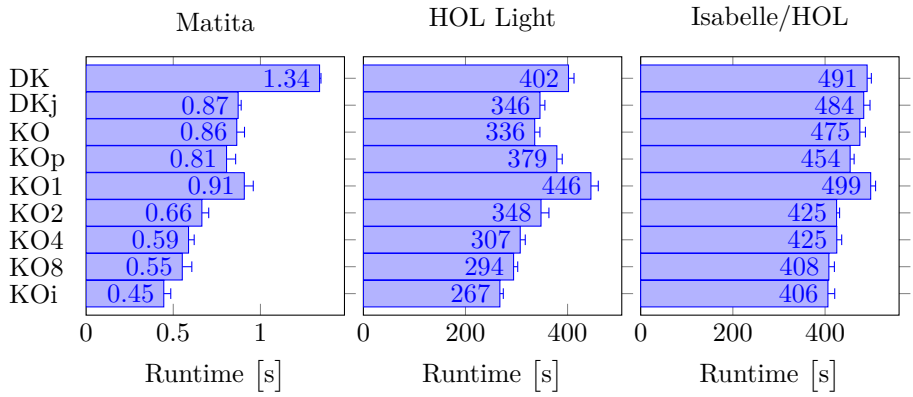


Figure 9. ITP dataset evaluation.

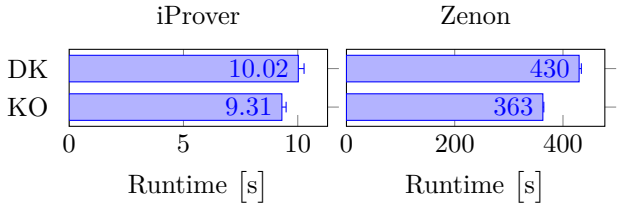


Figure 10. ATP dataset evaluation.

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 by 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 [31]. Stump et al. have created a proof checker *generator* for LFSC that creates a proof checker from a signature of proof rules [32]. The size of the generator is 5912 LOC of C++, and the kernel of a proof checker generated for SAT problems is 600 LOC of C++.[10]

Checkers is a proof checker based on foundational proof certificates (FPCs) developed by Chihani et al. [11]. Unlike Dedukti, which requires a *translation* of proofs into its calculus, FPCs allow for the *interpretation* of the proofs in the original proof calculus (modulo syntactic transformations), given an interpretation for the original calculus. The proof checker is implemented in λProlog and is the smallest work evaluated in this section, consisting of only 98 LOC[11]. Where LFSC generates a proof checker from a signature, Checkers generates a problem checker from a signature and a proof certificate, due to relying on λProlog for parsing signatures and proof certificates. Chihani et al. evaluated Checkers on a set of proofs generated by E-Prover, which unfortunately does not permit a comparison with neither Dedukti nor Kontroli due to currently not supporting E-Prover proofs.

Metamath is a language for formalising mathematics based on set theory [25]. There exist several proof verifiers for Metamath, one of the smallest being written in 308 LOC of Python.[12] 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 [39]. 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).[13] Among others, HOL Light has been used to certify SMT [31] as well as tableaux proofs [14, 24]. 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.

[10] Obtained from https://github.com/CVC4/LFSC, rev. 11fefc6. Measured with `tokei src/ -e CMake*` and `lfscc --compile-scc sat.plf && tokei scccode.*`.

[11] Obtained from https://github.com/proofcert/checkers, rev. 241b3c8. Measured with `sed -e '/^$/d' -e '/^%/d' lkf-kernel.mod | wc -l`.

[12] Obtained from https://github.com/david-a-wheeler/mmverify.py, rev. fb2e141. Measured with `tokei mmverify.py`.

[13] Obtained from https://github.com/jrh13/hol-light, rev. 4c324a2. Measured with `tokei fusion.ml` and `tokei pa_j_4.xx_7.xx.ml`.

7.2 Proof Assistants & Concurrency

Concurrent proof checking is nowadays mostly found in interactive theorem provers. Early work includes the Distributed Larch Prover [35] and the MP refiner [27].

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 toplevel proofs (similar to Dedukti and Kontroli), it also concurrently checks sub-proofs. Furthermore, it executes some tactics in parallel, for example the simplification of independent subgoals [36, 37].

Like Isabelle, ACL2 checks theories and toplevel proofs in parallel, but differs from Isabelle by automatically generating subgoals that are verified in parallel [29]. In both Isabelle and ACL2, threads are used to handle concurrent verification.

8 Conclusion

I have presented Kontroli, a small proof checker for the lambda-Pi calculus modulo rewriting. Despite its small size, Kontroli allows to verify proofs concurrently, without incurring any concurrency overhead when verifying proofs sequentially. I achieved this by abstracting over the shared pointer type in the kernel using a functor-like technique, which allows to parametrise the kernel both with slow thread-safe and fast thread-unsafe pointer types. Already in sequential mode, Kontroli is faster on all evaluated datasets than Dedukti, and in concurrent mode, Kontroli further significantly reduces checking time.

The current implementation of Kontroli omits several features present in Dedukti, such as detailed error reporting and higher-order rewriting. Despite this, Kontroli can be used to verify a considerable set of theories, including proof exports from HOL Light, Isabelle, Matita, iProver Modulo, and Zenon Modulo. The limited detail of errors reported by Kontroli makes it a candidate for automated testing where errors are the exception. One particular application of Kontroli could be in continuous integration: For example, whenever the kernel of a proof assistant or of an automated theorem prover is changed, one could export the proofs generated by the proof assistant or the theorem prover on some corpus and verify it with Kontroli. Kontroli could also be used to verify proofs generated during competitions like CASC [33]. In such scenarios, errors detected by Kontroli can then further be diagnosed by Dedukti.

This work can serve as a case study for implementing automated reasoning tools in Rust, whose combination of absent garbage collection and compiler-verified memory/thread safety currently makes it a unique programming language. It is encouraging to see that the resulting proof checker is not only performant, but also concise and safe. Having a proof checker in Rust simplifies future concurrency-related experiments, such as overhead-free concurrent verification.

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 Thibault Gauthier, Guillaume Genestier, Emilie Grienenberger, Gabriel Hondet, and François Thiré for their helpful comments on drafts of this article.

References

[1] Andrew W. Appel, Neophytos G. Michael, Aaron Stump, and Roberto Virga. 2003. A Trustworthy Proof Checker. *J. Autom. Reasoning* 31, 3-4 (2003), 231–260. https://doi.org/10.1023/B:JARS.0000021013.61329.58

[2] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. 2009. A compact kernel for the calculus of inductive constructions. *Sadhana* 34 (2009), 71–144. https://doi.org/10.1007/s12046-009-0003-3

[3] Ali Assaf. 2015. *A framework for defining computational higher-order logics. (Un cadre de définition de logiques calculatoires d'ordre supérieur)*. Ph.D. Dissertation. École Polytechnique, Palaiseau, France. https://tel.archives-ouvertes.fr/tel-01235303

[4] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. [n.d.]. Dedukti: a Logical Framework based on the λΠ-Calculus Modulo Theory. ([n. d.]). http://www.lsv.ens-cachan.fr/~dowek/Publi/expressing.pdf

[5] Henk Barendregt and Freek Wiedijk. 2005. The challenge of computer mathematics. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 363, 1835 (2005), 2351–2375. https://doi.org/10.1098/rsta.2005.1650

[6] Bruno Barras, Lourdes Del Carmen González-Huesca, Hugo Herbelin, Yann Régis-Gianas, Enrico Tassi, Makarius Wenzel, and Burkhart Wolff. 2013. Pervasive Parallelism in Highly-Trustable Interactive Theorem Proving Systems. In *Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK, July 8-12, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7961)*, Jacques Carette, David Aspinall, Christoph Lange, Petr Sojka, and Wolfgang Windsteiger

(Eds.). Springer, 359–363. https://doi.org/10.1007/978-3-642-39320-4_29

[7] Bruno Barras, Carst Tankink, and Enrico Tassi. 2015. Asynchronous Processing of Coq Documents: From the Kernel up to the User Interface. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9236)*, Christian Urban and Xingyuan Zhang (Eds.). Springer, 51–66. https://doi.org/10.1007/978-3-319-22102-1_4

[8] Yves Bertot. 2008. A Short Presentation of Coq. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5170)*, Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar (Eds.). Springer, 12–16. https://doi.org/10.1007/978-3-540-71067-7_3

[9] Guillaume Burel, Guillaume Bury, Raphaël Cauderlier, David Delahaye, Pierre Halmagrand, and Olivier Hermant. 2020. First-Order Automated Reasoning with Theories: When Deduction Modulo Theory Meets Practice. *J. Autom. Reasoning* 64, 6 (2020), 1001–1050. https://doi.org/10.1007/s10817-019-09533-z

[10] Mario M. Carneiro. 2016. Conversion of HOL Light proofs into Metamath. *J. Formalized Reasoning* 9, 1 (2016), 187–200. https://doi.org/10.6092/issn.1972-5787/4596

[11] Zakaria Chihani, Tomer Libal, and Giselle Reis. 2015. The Proof Certifier Checkers. In *Automated Reasoning with Analytic Tableaux and Related Methods - 24th International Conference, TABLEAUX 2015, Wroclaw, Poland, September 21-24, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9323)*, Hans de Nivelle (Ed.). Springer, 201–210. https://doi.org/10.1007/978-3-319-24312-2_14

[12] Evelyne Contejean. 2004. A Certified AC Matching Algorithm. In *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3091)*, Vincent van Oostrom (Ed.). Springer, 70–84. https://doi.org/10.1007/978-3-540-25979-4_5

[13] Denis Cousineau and Gilles Dowek. 2007. Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo. In *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4583)*, Simona Ronchi Della Rocca (Ed.). Springer, 102–117. https://doi.org/10.1007/978-3-540-73228-0_9

[14] Michael Färber and Cezary Kaliszyk. 2019. Certification of Nonclausal Connection Tableaux Proofs. In *Automated Reasoning with Analytic Tableaux and Related Methods - 28th International Conference, TABLEAUX 2019, London, UK, September 3-5, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11714)*, Serenella Cerrito and Andrei Popescu (Eds.). Springer, 21–38. https://doi.org/10.1007/978-3-030-29026-9_2

[15] Frédéric Gilbert. 2018. *Extending higher-order logic with predicate subtyping: Application to PVS. (Extension de la logique d'ordre supérieur avec le sous-typage par prédicats)*. Ph.D. Dissertation. Sorbonne Paris Cité, France. https://tel.archives-ouvertes.fr/hal-01673518

[16] Georges Gonthier. 2008. Formal Proof—The Four-Color Theorem. *Notices of the American Mathematical Society* 55 (2008), 1382–1393. Issue 11. http://www.ams.org/notices/200811/tx081101382p.pdf

[17] Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. 2017. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi* 5 (2017). https://doi.org/10.1017/fmp.2017.1

[18] Pierre Halmagrand. 2016. *Automated Deduction and Proof Certification for the B Method. (Déduction Automatique et Certification de Preuve pour la Méthode B)*. Ph.D. Dissertation. Conservatoire national des arts et métiers, Paris, France. https://tel.archives-ouvertes.fr/tel-01420460

[19] David R. Hanson. 1990. Fast Allocation and Deallocation of Memory Based on Object Lifetimes. *Softw. Pract. Exp.* 20, 1 (1990), 5–12. https://doi.org/10.1002/spe.4380200104

[20] Robert Harper, Furio Honsell, and Gordon D. Plotkin. 1993. A Framework for Defining Logics. *J. ACM* 40, 1 (1993), 143–184. https://doi.org/10.1145/138027.138060

[21] John Harrison. 2009. HOL Light: An Overview. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 60–66. https://doi.org/10.1007/978-3-642-03359-9_4

[22] Gabriel Hondet and Frédéric Blanqui. 2020. The New Rewriting Engine of Dedukti (System Description). In *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference) (LIPIcs, Vol. 167)*, Zena M. Ariola (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 35:1–35:16. https://doi.org/10.4230/LIPIcs.FSCD.2020.35

[23] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

[24] Cezary Kaliszyk, Josef Urban, and Jiří Vyskocil. 2015. Certified Connection Tableaux Proofs for HOL Light and TPTP. In *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, Xavier Leroy and Alwen Tiu (Eds.). ACM, 59–66. https://doi.org/10.1145/2676724.2693176

[25] Norman D. Megill and David A. Wheeler. 2019. *Metamath: A Computer Language for Mathematical Proofs*. Lulu Press, Morrisville, North Carolina. http://us.metamath.org/downloads/metamath.pdf

[26] Dale Miller. 1991. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *J. Log. Comput.* 1, 4 (1991), 497–536. https://doi.org/10.1093/logcom/1.4.497

[27] Roderick Moten. 1998. Exploiting Parallelism in Interactive Theorem Provers. In *Theorem Proving in Higher Order Logics, 11th International Conference, TPHOLs'98, Canberra, Australia, September 27 - October 1, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1479)*, Jim Grundy and Malcolm C. Newey (Eds.). Springer, 315–330. https://doi.org/10.1007/BFb0055144

[28] Jens Otten. 2017. Non-clausal Connection Calculi for Non-classical Logics. In *Automated Reasoning with Analytic Tableaux and Related Methods - 26th International Conference, TABLEAUX 2017, Brasília, Brazil, September 25-28, 2017, Proceedings (Lecture Notes in Computer Science,*

Vol. 10501), Renate A. Schmidt and Cláudia Nalon (Eds.). Springer, 209–227. https://doi.org/10.1007/978-3-319-66902-1_13

[29] David L. Rager, Warren A. Hunt Jr., and Matt Kaufmann. 2013. A Parallelized Theorem Prover for a Logic with Parallel Execution. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7998)*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.). Springer, 435–450. https://doi.org/10.1007/978-3-642-39634-2_31

[30] Ronan Saillard. 2015. *Typechecking in the lambda-Pi-Calculus Modulo : Theory and Practice. (Vérification de typage pour le lambda-Pi-Calcul Modulo : théorie et pratique)*. Ph.D. Dissertation. Mines ParisTech, France. https://tel.archives-ouvertes.fr/tel-01299180

[31] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. 2013. SMT proof checking using a logical framework. *Formal Methods Syst. Des.* 42, 1 (2013), 91–118. https://doi.org/10.1007/s10703-012-0163-3

[32] Aaron Stump, Andrew Reynolds, Cesare Tinelli, Austin Laugesen, Harley Eades, Corey Oliver, and Ruoyu Zhang. 2012. LFSC for SMT Proofs: Work in Progress. In *Second International Workshop on Proof Exchange for Theorem Proving, PxTP 2012, Manchester, UK, June 30, 2012. Proceedings (CEUR Workshop Proceedings, Vol. 878)*, David Pichardie and Tjark Weber (Eds.). CEUR-WS.org, 21–27. http://ceur-ws.org/Vol-878/paper1.pdf

[33] Geoff Sutcliffe. 2016. The CADE ATP System Competition - CASC. *AI Magazine* 37, 2 (2016), 99–101. https://doi.org/10.1609/aimag.v37i2.2620

[34] François Thiré. 2018. Sharing a Library between Proof Assistants: Reaching out to the HOL Family. In *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP@FSCD 2018, Oxford, UK, 7th July 2018 (EPTCS, Vol. 274)*, Frédéric Blanqui and Giselle Reis (Eds.). 57–71. https://doi.org/10.4204/EPTCS.274.5

[35] Mark T. Vandevoorde and Deepak Kapur. 1996. Distributed Larch Prover (DLP): An Experiment in Parallelizing a Rewrite-Rule Based Prover. In *Rewriting Techniques and Applications, 7th International Conference, RTA-96, New Brunswick, NJ, USA, July 27-30, 1996, Proceedings (Lecture Notes in Computer Science, Vol. 1103)*, Harald Ganzinger (Ed.). Springer, 420–423. https://doi.org/10.1007/3-540-61464-8_71

[36] Makarius Wenzel. 2009. Parallel Proof Checking in Isabelle/Isar. In *The ACM SIGSAM 2009 International Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS). Munich, August 2009*, Gabriel Dos Reis and Laurent Théry (Eds.). ACM Digital library.

[37] Makarius Wenzel. 2013. Shared-Memory Multiprocessing for Interactive Theorem Proving. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7998)*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.). Springer, 418–434. https://doi.org/10.1007/978-3-642-39634-2_30

[38] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. 2008. The Isabelle Framework. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5170)*, Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar (Eds.). Springer, 33–38. https://doi.org/10.1007/978-3-540-71067-7_7

[39] Freek Wiedijk. 2002. A New Implementation of Automath. *J. Autom. Reasoning* 29, 3-4 (2002), 365–387. https://doi.org/10.1023/A:1021983302516