# Safe, Fast, Concurrent Proof Checking

for the lambda-Pi Calculus Modulo Rewriting

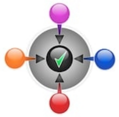
Michael Färber

2022-01-18

Section 1

## Dedukti

## Dedukti



- Dedukti is a proof checker based on the $\lambda\Pi$-calculus modulo rewriting.
- It checks proofs from systems such as Coq, HOL Light, Isabelle, . . .
- Proofs can become quite large and take long to check.

**Question**

How can we check Dedukti proofs faster, while keeping a small kernel?

## Dedukti: Theories

### Concepts

- Theory: a sequence of commands
- Command: introduces a constant or adds a rewrite rule

### A Theory About Implication

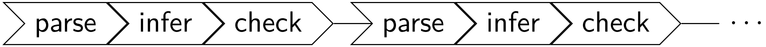
$$\mathrm{prop} : \mathrm{Type} \tag{1}$$

$$\mathrm{imp} : \mathrm{prop} \to \mathrm{prop} \to \mathrm{prop} \tag{2}$$

$$\mathrm{prf} : \mathrm{prop} \to \mathrm{Type} \tag{3}$$

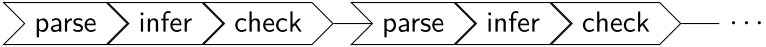
$$\mathrm{prf}\,(\mathrm{imp}\,x\,y) \hookrightarrow \mathrm{prf}\,x \to \mathrm{prf}\,y \tag{4}$$

## Dedukti: Checking



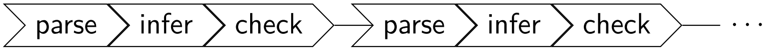
1. Parsing:
   `[x, y] prf (imp x y) --> prf x -> prf y` becomes
   $\text{prf}\,(\text{imp}\,x\,y) \hookrightarrow \text{prf}\,x \to \text{prf}\,y$.
2. Type Inference: $\text{prf}\,(\text{imp}\,x\,y) : A$
3. Type Checking: $\text{prf}\,x \to \text{prf}\,y : A$?

## Dedukti: Checking



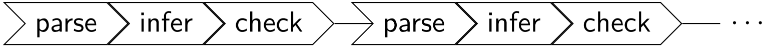
1. Parsing:
   `[x, y] prf (imp x y) --> prf x -> prf y` becomes
   $\mathsf{prf}\ (\mathsf{imp}\ x\ y) \hookrightarrow \mathsf{prf}\ x \to \mathsf{prf}\ y$.
2. Type Inference: $\mathsf{prf}\ (\mathsf{imp}\ x\ y) : A$
3. Type Checking: $\mathsf{prf}\ x \to \mathsf{prf}\ y : A$?

## Dedukti: Checking



1. Parsing:
   `[x, y] prf (imp x y) --> prf x -> prf y` becomes
   $\mathrm{prf}\ (\mathrm{imp}\ x\ y) \hookrightarrow \mathrm{prf}\ x \to \mathrm{prf}\ y$.
2. Type Inference: $\mathrm{prf}\ (\mathrm{imp}\ x\ y) : \mathrm{Type}$
3. Type Checking: $\mathrm{prf}\ x \to \mathrm{prf}\ y : \mathrm{Type}$?

## Dedukti: Checking



1. Parsing:
   `[x, y] prf (imp x y) --> prf x -> prf y` becomes
   $\mathsf{prf}\,(\mathsf{imp}\,x\,y) \hookrightarrow \mathsf{prf}\,x \to \mathsf{prf}\,y$.
2. Type Inference: $\mathsf{prf}\,(\mathsf{imp}\,x\,y)$ : Type
3. Type Checking: $\mathsf{prf}\,x \to \mathsf{prf}\,y$ : Type? ✓

## Dedukti: Concurrency

- Dedukti checks multiple *theories* concurrently (one process per theory).
- For each theory, it processes only one command at a time.
- Can we somehow process multiple *commands* concurrently?

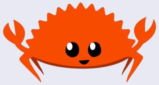
## Programming Languages

### OCaml

- Dedukti is implemented in OCaml
- Multicore support not (yet) available

### Rust

- Functional systems programming language
- Memory- and thread-safe (unlike C)
- Focus on performance and concurrency



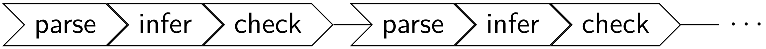
### Goal

- Reimplement core of Dedukti in Rust
- Process multiple *commands* concurrently, using threads

Section 2

## Concurrent Proof Checking

## Sequential Proof Checking



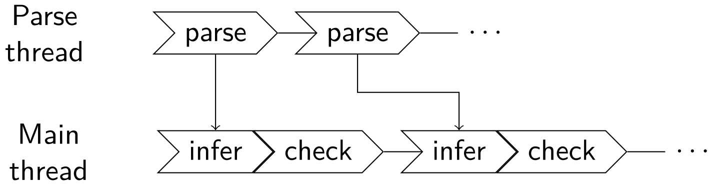
Most time is spent in parsing and type checking
(69% for HOL Light and 85% for Isabelle/HOL corpora)

### Concurrency

- Delegate parsing to an own thread
- Delegate type checking to multiple threads

# Concurrent Parsing

Parse commands in a thread and send them via a channel to main thread:

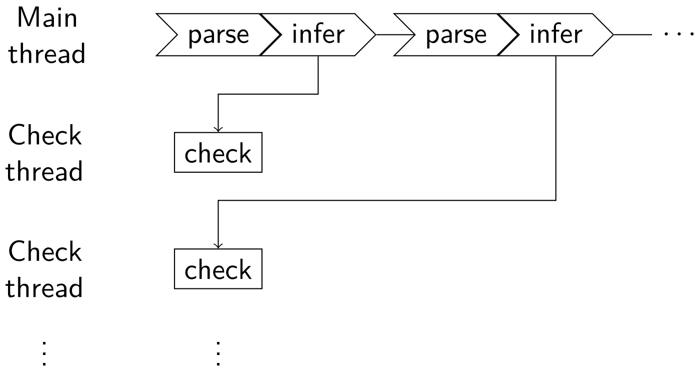


Best-case improvement: Reduce proof checking time by parsing time

In practice: channel overhead too large to make it pay off

## Concurrent Type Checking

Launch a thread for every type checking task:



Best-case improvement: Reduce proof checking time by type checking time

Section 3

## Terms

## Terms

Terms are *the* central data structure in Dedukti:

$$t \coloneqq c \mid x \mid \overbrace{t\, u}^{\text{application}} \mid \overbrace{\lambda x : t.\, u \mid \Pi x : t.\, u}^{\text{abstraction}},$$

where $t$ and $u$ are terms, $c$ is a constant, $x$ is a variable

## Pointer Types

Rust requires use of pointers to obtain inductive types (such as terms).

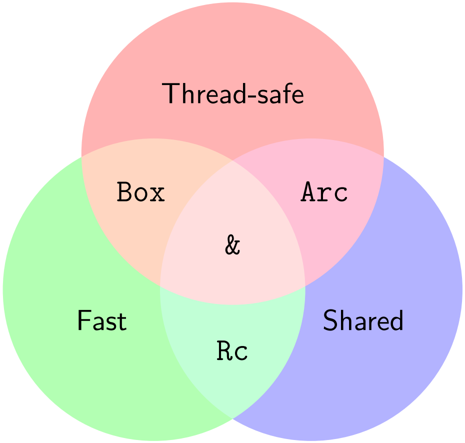


Figure 1: Three commonly used pointer types.

## Three Types of Terms

Terms using different pointer types have different downsides:

- `Box`-terms take linear time to duplicate.
- `Rc`-terms cannot be used across threads.
- `Arc`-terms are slow.

| Task | Mode | Term pointer |
|---|---|---|
| Parsing | Any | `Box` |
| Type checking | Sequential | `Rc` |
| Type checking | Concurrent | `Arc` |

## Increasing Term Performance: Unboxing

- Omit pointers around constants and variables (do not have subterms)
- Reduces runtime by **20%** when using `Rc`-terms and **29%** when using `Arc`-terms.

Section 4

## Implementation

## Kontroli

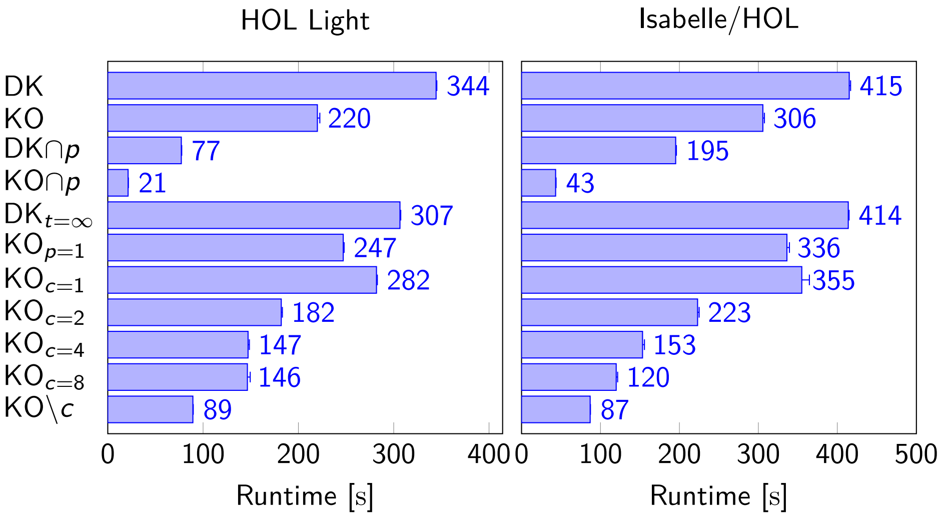
Kontroli is a minimal concurrent proof checker for the $\lambda\Pi$-calculus modulo.

https://github.com/01mf02/kontroli-rs

| Program | Kernel |
|---|---:|
| Dedukti | 3470 LOC |
| Kontroli | 663 LOC |

- Kontroli supports only a subset of Dedukti's features
- Large enough to verify HOL-based theories

# Evaluation



HOL Light
Isabelle/HOL
DK
KO
DK∩$p$
KO∩$p$
$\mathrm{DK}_{t=\infty}$
$\mathrm{KO}_{p=1}$
$\mathrm{KO}_{c=1}$
$\mathrm{KO}_{c=2}$
$\mathrm{KO}_{c=4}$
$\mathrm{KO}_{c=8}$
KO\\$c$
344
220
77
21
307
247
282
182
147
146
89
415
306
195
43
414
336
355
223
153
120
87
0
100
200
300
400
500
Runtime [s]
Runtime [s]

Section 5

## Conclusion

## Conclusion

- Terms using `Box`, `Rc`, and `Arc` nicely fit parsing, sequential type checking, and parallel type checking.
- Fewer pointers in the term type greatly benefit performance.
- Parsing is one of the largest bottlenecks in Dedukti.
- Concurrent parsing increases runtime, due to channel overhead.
- Concurrent type checking significantly reduces runtime (up to 6.6x for 8 threads).

A small & safe proof checker with fast concurrency is possible!

## Conclusion

- Terms using `Box`, `Rc`, and `Arc` nicely fit parsing, sequential type checking, and parallel type checking.
- Fewer pointers in the term type greatly benefit performance.
- Parsing is one of the largest bottlenecks in Dedukti.
- Concurrent parsing increases runtime, due to channel overhead.
- Concurrent type checking significantly reduces runtime (up to 6.6x for 8 threads).

A small & safe proof checker with fast concurrency is possible!

Thank you for your attention!