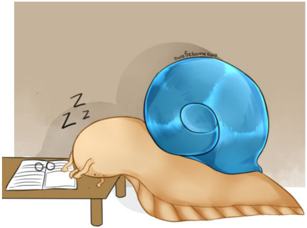
Terms for Efficient Proof Checking & Parsing

Michael Färber

2023-01-17

Introduction

- Automatically generated proofs from ITPs/ATPs tend to be quite large.
- A proof checker can take considerable time checking such proofs.



- We can improve proof checking performance by exploiting parallelism.
- However, it is not easy to do this while achieving:
    - small kernel (for trustworthiness)
    - high single- and multi-threaded performance

Sequential Processing & Parallel Parsing

How to check a sequence of theorems (= statement + proof)?

Sequential Processing & Parallel Parsing

How to check a sequence of theorems (= statement + proof)?

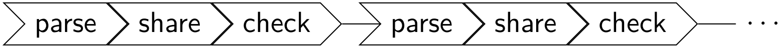


Figure 1: Sequential processing.

Sequential Processing & Parallel Parsing

How to check a sequence of theorems (= statement + proof)?

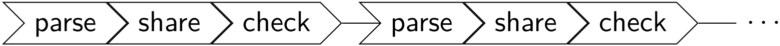


Figure 1: Sequential processing.

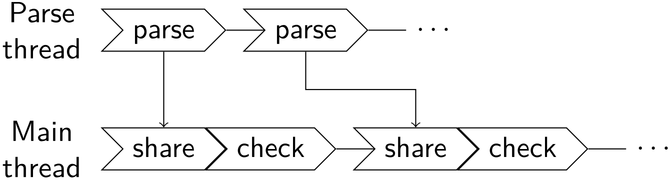


Figure 2: Parallel parsing.

Parallel Checking

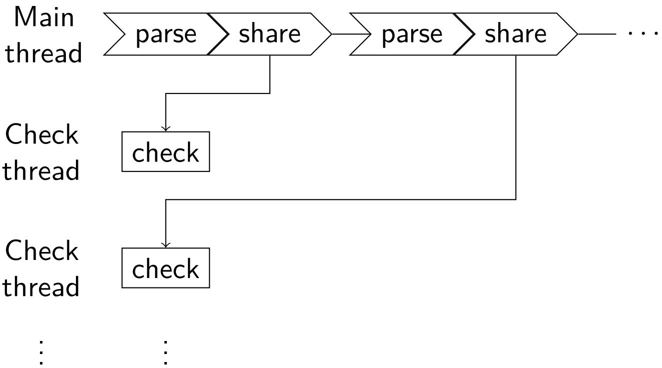


Figure 3: Parallel checking.

Parallel Checking

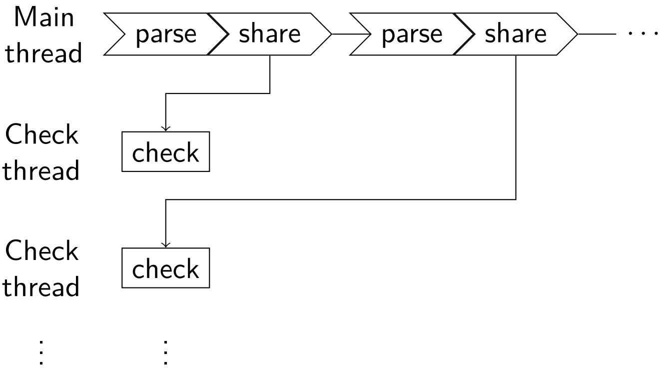


Figure 3: Parallel checking.

Problem

How to efficiently check proofs in different threads?

Previous Work

Previous Work @ CPP'22

- I presented a proof checker called *Kontroli* written in Rust.
- It reimplements large parts of the *Dedukti* proof checker, but supports parallel checking and parsing of proofs.
- It improved the state of the art proof checking performance.

Shortcomings

- It used two different kernels for single- and multi-threaded checking.
- It was far from reaching theoretical optimal parallel performance.

This Work

- Uses *heterogeneous terms* to greatly improve checking performance
- Uses *abstract terms* to improve parsing performance (not covered here)
- Fastest mode is now up to 3.6x as fast as previous fastest mode!

Section 1

Homogeneous Terms

Terms

Terms are *the* central data structure in a proof checker:

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

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

In OCaml, a term type can be specified as:

```
type term =
  | Const of string
  | Var of int
  | Appl of term * term list
  | Abst of term * term
  | Prod of term * term
```

Pointer Types

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

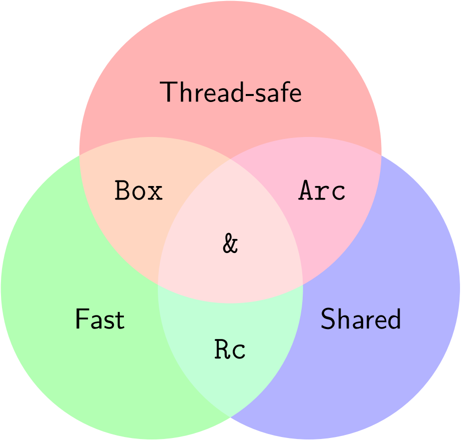
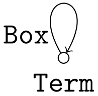


Figure 4: Three commonly used pointer types.

A First Take on Terms in Rust

```rust
enum Term {
    Const(String),
    Var(usize),
    Appl(Box<Term>, Vec<Term>),
    Abst(Box<Term>, Box<Term>),
    Prod(Box<Term>, Box<Term>),
}
```



Problems

- Using Box means that cloning terms takes linear time!
  - This is bad for checking, because checking frequently clones terms.
  - However, this is OK for parsing, because parsing does not clone terms.
- Both Abst and Prod use two Box pointers, but one would suffice.

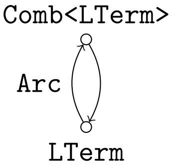
Second Take on Terms

Factoring out the recursive term variants . . .

```
enum Comb<Tm> {
    Appl(Tm, Vec<Tm>),
    Abst(Tm, Tm),
    Prod(Tm, Tm),
}
```

. . . leaves the following term type:

```
enum LTerm {
    Const(&str),
    Var(usize),
    LComb(Arc<Comb<LTerm>>),
}
```



- My CPP'22 paper used this term type.
- Problem: Creating many terms containing `Arc` is slow!

Section 2

Heterogeneous Terms

Heterogeneous Terms

- The global context Γ stores background knowledge.
- The local context Δ stores knowledge for the current checking task.

Property	Terms in Γ	Terms in Δ
Content	Constant types & definitions	Proofs, calculations
Lifetime	Until program exits	Until a proof is checked
Quantity	Few (bounded by input)	Many (unbounded!)
Access	From multiple threads	From single thread

Heterogeneous Terms

- The global context Γ stores background knowledge.
- The local context Δ stores knowledge for the current checking task.

Property	Terms in Γ	Terms in Δ
Content	Constant types & definitions	Proofs, calculations
Lifetime	Until program exits	Until a proof is checked
Quantity	Few (bounded by input)	Many (unbounded!)
Access	From multiple threads	From single thread

Idea

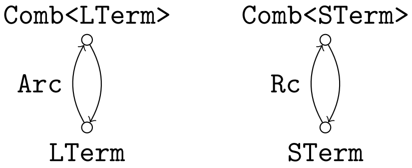
Create separate term types for terms in Γ and Δ!

Naming

- Γ and Δ resemble long & short term memory (thanks to Gilles Dowek).
- I call terms in Γ *long terms* and in Δ *short terms*.

Heterogeneous Terms, First Take

```
enum STerm {
    Const(&str),
    Var(usize),
    SComb(Rc<Comb<STerm>>),
}
```

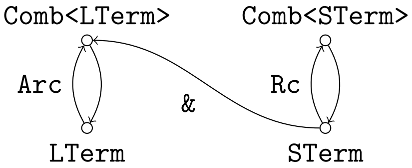


Problem

Converting an `LTerm` to `STerm` takes linear time!

Heterogeneous Terms, Second Take

```
enum STerm {
    Const(&str),
    Var(usize),
    SComb(Rc<Comb<STerm>>),
    LComb(& <Comb<LTerm>>),
}
```



Advantages

- Converting an LTerm to an STerm takes constant time.
- An STerm referencing an LTerm can be created and destroyed very quickly, because it does not involve reference counting.

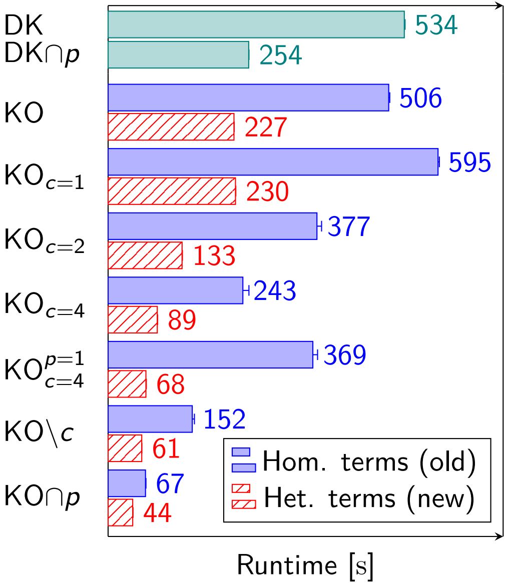
Disadvantage

We cannot "forget" terms in Γ while terms in Δ reference them.

Section 3

Evaluation

Isabelle/HOL Dataset (2.5GB, 1.7M proofs)



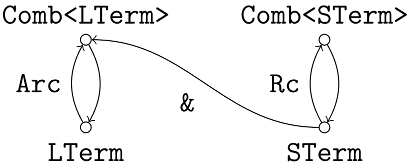
Conf.	Meaning
DK	Dedukti, sequential
DK$\cap p$	DK, parsing only
KO	Kontroli, sequential
KO$_{c=n}$	KO, n check threads
KO$^{p=1}$	KO, parallel parsing
KO$\setminus c$	KO, no checking
KO$\cap p$	KO, parsing only

Section 4

Conclusion

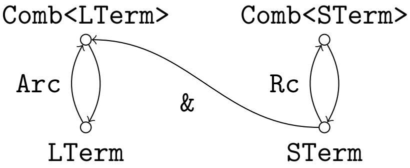
Conclusion

- Homogeneous terms are good for parsing, because they are not shared.
- Heterogeneous terms are good for checking:
  - Fast referencing of Γ-terms in Δ-terms (without reference counting)
  - Single kernel for sequential and parallel checking, without overhead



Conclusion

- Homogeneous terms are good for parsing, because they are not shared.
- Heterogeneous terms are good for checking:
  - Fast referencing of Γ-terms in Δ-terms (without reference counting)
  - Single kernel for sequential and parallel checking, without overhead



Thank you for your attention!