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

Problem

How to efficiently check proofs in different threads?

Figure 3: Parallel checking.
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 are the central data structure in a proof checker:

\[
\begin{align*}
\text{application} & \quad \text{abstraction} \\
t & := c \mid x \mid tu \mid \lambda x : t. u \mid \Pi x : t. u,
\end{align*}
\]

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

In OCaml, a term type can be specified as:

```ocaml
type term =
    | Const of string
    | Var of int
    | Appl of term * term list
    | Abst of term * term
    | Prod of term * term
```
Rust requires use of pointers to obtain inductive types (such as terms).

Figure 4: Three commonly used pointer types.
```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.
Factoring out the recursive term variants …

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

… leaves the following term type:

```cpp
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 $\Gamma$ stores background knowledge.
- The local context $\Delta$ stores knowledge for the current checking task.

<table>
<thead>
<tr>
<th>Property</th>
<th>Terms in $\Gamma$</th>
<th>Terms in $\Delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
<td>Constant types &amp; definitions</td>
<td>Proofs, calculations</td>
</tr>
<tr>
<td>Lifetime</td>
<td>Until program exits</td>
<td>Until a proof is checked</td>
</tr>
<tr>
<td>Quantity</td>
<td>Few (bounded by input)</td>
<td>Many (unbounded!)</td>
</tr>
<tr>
<td>Access</td>
<td>From multiple threads</td>
<td>From single thread</td>
</tr>
</tbody>
</table>
Heterogeneous Terms

- The global context $\Gamma$ stores background knowledge.
- The local context $\Delta$ stores knowledge for the current checking task.

<table>
<thead>
<tr>
<th>Property</th>
<th>Terms in $\Gamma$</th>
<th>Terms in $\Delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
<td>Constant types &amp; definitions</td>
<td>Proofs, calculations</td>
</tr>
<tr>
<td>Lifetime</td>
<td>Until program exits</td>
<td>Until a proof is checked</td>
</tr>
<tr>
<td>Quantity</td>
<td>Few (bounded by input)</td>
<td>Many (unbounded!)</td>
</tr>
<tr>
<td>Access</td>
<td>From multiple threads</td>
<td>From single thread</td>
</tr>
</tbody>
</table>

Idea

Create separate term types for terms in $\Gamma$ and $\Delta$!

Naming

- $\Gamma$ and $\Delta$ resemble long & short term memory (thanks to Gilles Dowek).
- I call terms in $\Gamma$ *long terms* and in $\Delta$ *short terms*. 
enum STerm {
    Const(&str),
    Var(usize),
    SComb(Rc<Comb<STerm>>),
}

Problem
Converting an LTerm to STerm takes linear time!
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 $\Gamma$ while terms in $\Delta$ reference them.
Section 3

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

<table>
<thead>
<tr>
<th></th>
<th>DK</th>
<th>DK ∩ p</th>
<th>KO</th>
<th>KO_{c=1}</th>
<th>KO_{c=2}</th>
<th>KO_{c=4}</th>
<th>KO_{p=1}</th>
<th>KO \ c</th>
<th>KO ∩ p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hom. terms (old)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Het. terms (new)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Runtime [s]</td>
<td>534</td>
<td>254</td>
<td>506</td>
<td>595</td>
<td>377</td>
<td>243</td>
<td>369</td>
<td>152</td>
<td>61</td>
</tr>
<tr>
<td>Runtime [s]</td>
<td>67</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Conf. | Meaning
--- | ---
DK | Dedukti, sequential
DK ∩ p | DK, parsing only
KO | Kontroli, sequential
KO_{c=n} | KO, n check threads
KO_{p=1} | KO, parallel parsing
KO \ c | KO, no checking
KO ∩ 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 $\Gamma$-terms in $\Delta$-terms (without reference counting)
  - Single kernel for sequential and parallel checking, without overhead
Homogeneous terms are good for parsing, because they are not shared. Heterogeneous terms are good for checking:
- Fast referencing of $\Gamma$-terms in $\Delta$-terms (without reference counting)
- Single kernel for sequential and parallel checking, without overhead

Thank you for your attention!