How I Learned to Stop Worrying and Implement Dedukti Myself[1]

Michael Färber

June 25, 2022

[1]This research was funded in part by the Austrian Science Fund (FWF) [J 4386].

Section 1

Introduction

My Background

- 2019–2020: member of DEDUCTEAM
- Used Dedukti (DK) to verify proofs exported from Isabelle
- How efficient is DK? Reimplement it to find out!

My Background

- 2019–2020: member of DEDUCTEAM
- Used Dedukti (DK) to verify proofs exported from Isabelle
- How efficient is DK? Reimplement it to find out!

Figure 1: Beware of Hofstadter's Law.

Wish List for a Proof Checker

- Nice syntax
- Helpful error messages
- Small & simple kernel
- Take little time & memory
- ...

Wish List for a Proof Checker

- Nice syntax
- Helpful error messages
- Small & simple kernel
- Take little time & memory
- ...

Figure 2: Vouloir le beurre et l'argent du beurre.

Kontroli

Kontroli (KO) is an alternative implementation of Dedukti.

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

Wish List

- Minimal: leave out the 80% of DK needed for only 20% of the theories
- Small, simple kernel: 658 LOC vs. 3475 LOC for DK
- Fast: test-bed for multi-threaded theory checking
- Compatible: understand DK syntax, stay true to the DK spirit ;)
- Provide a second opinion to DK's output (*kontroli* = to verify)

Goal of this Talk

I wish to share with you what I learned about DK while implementing KO.

This should lower the barrier for you to tackle fun projects such as:

Processing DK Theories

- Transform DK theories to a proof blockchain
- Learn theorem proving from DK proofs
- Compress DK proofs (big data!)

Using / Modifying DK

- Integrate DK into a proof assistant as alternative backend
- Implement some cool feature into DK
- Reimplement DK (again)

Menu du jour

- Parsing
- Theory Checking
- Reduction
- Sharing

Section 2

Preliminaries

Terms

Table 1: Definition of terms t, u.

$t, u :=$	description	examples
$\mid s$	sort	$\texttt{Type}$, $\texttt{Kind}$ (the type of $\texttt{Type}$)
$\mid c$	constant	vec, nat
$\mid v$	variable	x
$\mid t\ u$	application	$vec\, x$
$\mid t \to u$	product	$nat \to nat$
$\mid \lambda x : t.\, u$	abstraction	$\lambda x : nat.\, x$
$\mid \Pi x : t.\, u$	dep. product	$\Pi x : nat.\, vec\, x \to vec\, x$

Terms

Table 1: Definition of terms t, u.

$t, u :=$	description	examples
$\mid s$	sort	$\texttt{Type}$, $\texttt{Kind}$ (the type of $\texttt{Type}$)
$\mid c$	constant	*vec*, *nat*
$\mid v$	variable	x
$\mid t\ u$	application	$\mathit{vec}\,x$
$\mid t \to u$	product	$\mathit{nat} \to \mathit{nat}$
$\mid \lambda x : t.\, u$	abstraction	$\lambda x : \mathit{nat}.\, x$
$\mid \Pi x : t.\, u$	dep. product	$\Pi x : \mathit{nat}.\, \mathit{vec}\,x \to \mathit{vec}\,x$

The encoding of terms has an *enormous* impact on performance!

Commands

Table 2: Definition of introduction commands *cmd*.

$cmd :=$	introduces	examples
$\mid c : t$	constant	$nat : \mathtt{Type}$, $vec : nat \to Type$
$\mid l \hookrightarrow r$	rewrite rule	$rev\ nil \hookrightarrow nil$

A *theory* is a sequence of commands.

Section 3

Parsing

Parsing

To process DK theories, often a parser is all you need.

Challenges

- Theories can be very large (>1GB)
- Terms (mostly proofs) can be very large (>100MB)

Off-the-shelf parsing tools might struggle with this.

Performance

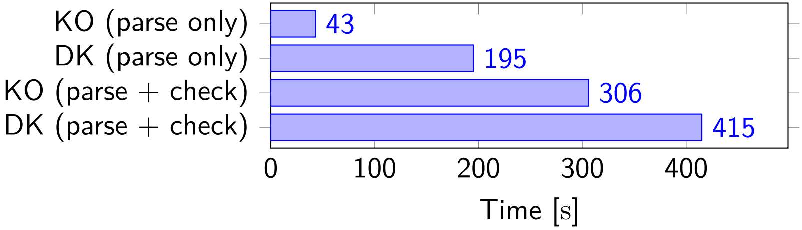

Figure 3: Processing the Isabelle/HOL dataset with Kontroli & Dedukti.

Parsing can take up to half the total proof checking time.

Existing parsers

OCaml: the parser in DK

- Automatically generated
- Good error reporting
- Supports full DK syntax (by definition)

Rust: the parser in KO (`dedukti_parse`)

- Hand-written by yours truly :)
- Lazy and strict parsing with and without scoping
- Optimised for performance (up to ∼4x faster)
- Easy-to-use API
- Abysmal error reporting
- Supports a large subset of DK syntax, but not everything

My tip

Use an existing parser, if you can!

Strict vs. Lazy

Strict Parsing

- Parse file only once it has been read completely into memory
- Lower total runtime
- Easier to implement

Lazy Parsing

- Parse file line by line
- Lower latency: parsing starts once a single line is read
- Lower memory consumption: only one line in memory instead of file

Scoping

$\lambda f\, x. f\, x$ becomes $\lambda f\, x. \overline{1}\, \overline{0}$

$\overline{1}$ and $\overline{0}$ are de Bruijn variables, which encode bound variables in $\mathbb{N}$.

- Saves memory (if we parse lazily)
- Takes more time (because we keep track of bound variables)
- Often required anyway for proof checking

Example: Pretty-Printing with dedukti_parse

```rust
fn main() {
    // read stdin line-by-line
    use std::io::{stdin, BufRead};
    let lines = stdin().lock().lines().map(|l| l.unwrap());

    // parse the commands in stdin, without scoping
    use dedukti_parse::{Lazy, Symb};
    let cmds = Lazy::<_, Symb<String>, String>::new(lines);

    // print every command
    for cmd in cmds {
        println!("{}.", cmd.unwrap());
    }
}
```

Demo

Prerequisites: a Rust toolchain, e.g. from https://rustup.rs/

```
$ cargo new kofmt
$ cd kofmt
$ echo 'dedukti-parse = "0.3"' >> Cargo.toml # add dependency
$ # paste source code from previous slide into `src/main.rs`
$ cargo run --
a: Type.
b:
Type.
[] a --> b. [] b --> a.
```

Demo: Lazy Parsing in DK / KO

```
$ for i in `seq 0 100`; do echo a$i: Type.; done
$ 1001cmds() {
      for i in `seq 0 1000000`; do echo a$i: Type.; done
  }
$ time 1001cmds > /dev/null
$ 1001cmds | dkcheck --stdin mod -v
$ 1001cmds | LOG=info kocheck -
```

Section 4

Theory Checking

Theory Checking

Processing a theory

For every command in the theory:

1. If it introduces $c : t$, check that c is new and the type of t is a sort.
2. If it introduces $l \hookrightarrow r$, check that the rewrite rule preserves types.
3. Add it to *global context* Γ (initially empty).

Theory Checking

Processing a theory

For every command in the theory:

1. If it introduces $c : t$, check that c is new and the type of t is a sort.
2. If it introduces $l \hookrightarrow r$, check that the rewrite rule preserves types.
3. Add it to *global context* Γ (initially empty).

What we need

- How to infer the type of a term (find A such that $\Gamma \vdash t : A$)?

Theory Checking

Processing a theory

For every command in the theory:

1. If it introduces $c : t$, check that c is new and the type of t is a sort.
2. If it introduces $l \hookrightarrow r$, check that the rewrite rule preserves types.
3. Add it to *global context* Γ (initially empty).

What we need

- How to infer the type of a term (find A such that $\Gamma \vdash t : A$)?
- How to check that a rewrite rule preserves types (subject reduction)?

Type Checking & Inference

Type checking & inference consists of applying rules such as the following, where Δ is a *local context* (contains statements of shape $x : A$):

$$\frac{}{\Gamma, \Delta \vdash \texttt{Type} : \texttt{Kind}}\ \text{Type}$$

$$\frac{\Gamma, \Delta \vdash A : \texttt{Type} \qquad \Gamma, \Delta, x : A \vdash t : s}{\Gamma, \Delta \vdash (\Pi x{:}A.\, t) : s}\ \text{Prod}$$

Convertibility: the *modulo* in "$\lambda\Pi$-calculus modulo"

$$\frac{\Gamma, \Delta \vdash t : A \quad \Gamma, \Delta \vdash B : s \quad \Gamma \vdash A \equiv_{\beta\mathcal{R}} B}{\Gamma, \Delta \vdash t : B}\,\text{Conv}$$

($\Gamma \vdash l \equiv_{\beta\mathcal{R}} r$ means that l and r are convertible.)

- The convertibility rule "Conv" leaves it up to us to choose B.
- Dedukti cannot guess B, so it does *not* implement this Conv rule.
- Instead, it modifies all other rules to account for convertibility.

Subject Reduction

How to check that a rewrite rule preserves types?

With Type Annotations (KO)

- Example: `[X: nat] square X --> mult X X.`
- Put variable bindings (`X: nat`) into local context Δ.
- Find A, B such that $\Gamma, \Delta \vdash \texttt{square X} : A$ and $\Gamma, \Delta \vdash \texttt{mult X X} : B$.
- Verify that A and B are convertible.

Subject Reduction

How to check that a rewrite rule preserves types?

With Type Annotations (KO)

- Example: `[X: nat] square X --> mult X X.`
- Put variable bindings (`X: nat`) into local context Δ.
- Find A, B such that $\Gamma, \Delta \vdash \texttt{square X} : A$ and $\Gamma, \Delta \vdash \texttt{mult X X} : B$.
- Verify that A and B are convertible.

Without Type Annotations (DK)

- Example: `[X] square X --> mult X X.`
- We do not know the type of `X`, so we cannot put it into Δ!
- DK uses bidirectional type checking to check subject reduction
- Highly complex (I do not really understand how it works)

Convertibility Check

To check whether l and r are convertible ($l \sim r$):

1. If $l = r$, return true.
2. Reduce l and r to weak-head normal form (WHNF).
3. If l and r match any case in table 3, check all constraints.
4. Else return false.

Table 3: Constraints.

l	r	constraints
sort, const., or var.	l	
$\lambda x : A.t$	$\lambda y : B.u$	$t \sim u$
$\Pi x : A.t$	$\Pi y : B.u$	$t \sim u$, $A \sim B$
$t_1\, t_2 \dots t_n$	$u_1\, u_2 \dots u_n$	$t_1 \sim u_1, \dots, t_n \sim u_n$

Convertibility Check

To check whether l and r are convertible ($l \sim r$):

1. If $l = r$, return true.
2. Reduce l and r to weak-head normal form (WHNF).
3. If l and r match any case in table 3, check all constraints.
4. Else return false.

Table 3: Constraints.

l	r	constraints
sort, const., or var.	l	
$\lambda x : A.t$	$\lambda y : B.u$	$t \sim u$
$\Pi x : A.t$	$\Pi y : B.u$	$t \sim u$, $A \sim B$
$t_1\, t_2 \ldots t_n$	$u_1\, u_2 \ldots u_n$	$t_1 \sim u_1, \ldots, t_n \sim u_n$

Section 5

Reduction

Reduction

- How to get the WHNF of a term in the presence of rewrite rules?
- This part is about 40% of the Kontroli kernel!

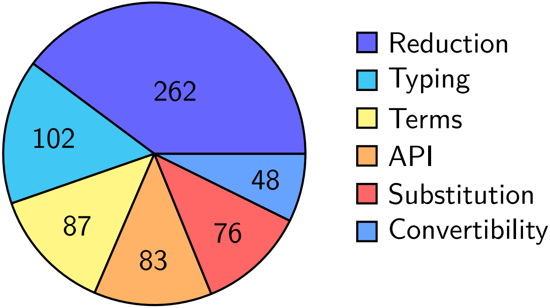

Figure 4: Lines of code of all parts of the Kontroli kernel.

Challenges

- Laziness: $\mathit{ite}\,\top\;T\;F \hookrightarrow T$, $\mathit{ite}\,\bot\;T\;F \hookrightarrow F$
  (evaluates only one of T and F)
- Sharing: $\mathit{double}\,X \hookrightarrow \mathit{add}\,X\,X$
  (evaluating the first argument of *add* also evaluates the second)
- Equality constraints: $\mathit{eq}\,X\,X \hookrightarrow \top$
  (checks whether first and second argument of *eq* are convertible)

Abstract Machines

DK (and KO) encode terms during reduction as *abstract machines*:

```
type state = {
  ctx : term Lazy.t list; (* substitution applied to term *)
  term : term;
  stack : state ref list; (* arguments applied to term *)
}
```

Abstract Machines

DK (and KO) encode terms during reduction as *abstract machines*:

```
type state = {
  ctx : term Lazy.t list; (* substitution applied to term *)
  term : term;
  stack : state ref list; (* arguments applied to term *)
}
```

Memoization: Matching term $ite\,(eq\ 0\ 1)\,f\ g$ with pattern $ite\ \top\ T\ F$

1. We convert the term $ite\,(eq\ 0\ 1)\,f\ g$ to a machine state, where `term` $= ite$ and `stack` $= [eq\ 0\ 1, f, g]$.
2. Matching with $ite\ \top\ T\ F$ evaluates $eq\ 0\ 1$ to $\bot$; we update the stack.
3. $\bot$ does not match $\top$, so the term does *not* match the pattern.

Because we updated the stack, subsequent pattern matches with this machine will *not* need to evaluate $eq\ 0\ 1$ again.

Decision Trees

Accelerate matching with many overlapping rewrite rules

Example (from the DK Sudoku solver)

```
[x] getc 1 (c x _ _ _ _ _ _ _ _) --> x
[x] getc 2 (c _ x _ _ _ _ _ _ _) --> x
[x] getc 3 (c _ _ x _ _ _ _ _ _) --> x
[x] getc 4 (c _ _ _ x _ _ _ _ _) --> x
[x] getc 5 (c _ _ _ _ x _ _ _ _) --> x
[x] getc 6 (c _ _ _ _ _ x _ _ _) --> x
[x] getc 7 (c _ _ _ _ _ _ x _ _) --> x
[x] getc 8 (c _ _ _ _ _ _ _ x _) --> x
[x] getc 9 (c _ _ _ _ _ _ _ _ x) --> x.
```

My experience

When only few rewrite rules on the same head symbol are defined, decision trees do not pay off → I did not implement them

Higher-Order Pattern Matching

- Example: *forall* $(\lambda x.\top) \hookrightarrow \top$
- Encoding of CoC uses it
- FOL & HOL-like theories do not need it $\rightarrow$ I did not implement this

Higher-Order Pattern Matching

- Example: $\mathit{forall}\,(\lambda x.\top) \hookrightarrow \top$
- Encoding of CoC uses it
- FOL & HOL-like theories do not need it $\rightarrow$ I did not implement this

Hint: I am happy to accept pull requests for Kontroli . . . :)

Section 6

Sharing & Memory

Sharing

- Implicit in many FP languages (such as OCaml, Haskell, . . .)
- Explicit in other languages (such as Rust, C, . . .)
- Saves time & memory
- Due to implicitness, easy to break

Without Sharing

```
let a = "zero" in
let b = "zero" in
a = b && not (a == b) (* slow: character-wise comparison *)
```

With Sharing

```
let a = "zero" in
let b = a in
a = b && a == b (* fast: comparison by memory address *)
```

Sharing in Dedukti

Shared constants

- Map all equal parsed constants to a *single* canonical constant
- To compare constants, compare *only* pointer addresses

Shared terms

- Reuse existing terms instead of keeping new terms whenever possible
- Example: to substitute t with σ, when $\sigma t = t$, then return t, not σt
- To determine whether $t = u$, compare addresses of t and u first

Memory allocation

- Proof checking (de-)allocates lots of memory, mostly for terms
- Memory allocator manages where objects are written to in memory
- mimalloc: memory allocator originally written for proof assistant Lean
- Using mimalloc boosts speed with minimal effort (3 lines added)

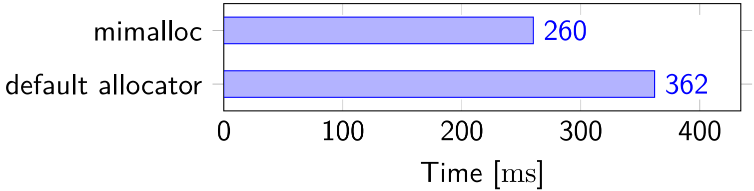

Figure 5: Kontroli checking the Matita dataset using different allocators.

When using garbage collection, similar gains *might* be obtained by tuning it.

Section 7

Conclusion

Conclusion

- Kontroli reimplements Dedukti, focussing on kernel size & performance.
- The representation of terms is crucial for performance.
- Parsing is an important performance bottleneck.
- Parsing is hard $\rightarrow$ use an existing parser.
- Reduction is hairy due to lazy evaluation, memoization, ...
- Higher-order matching is not needed for many theories, e.g. HOL.
- Sharing of constants & terms saves time & memory.
- The memory allocation strategy has a large impact.

Conclusion

- Kontroli reimplements Dedukti, focussing on kernel size & performance.
- The representation of terms is crucial for performance.
- Parsing is an important performance bottleneck.
- Parsing is hard $\rightarrow$ use an existing parser.
- Reduction is hairy due to lazy evaluation, memoization, ...
- Higher-order matching is not needed for many theories, e.g. HOL.
- Sharing of constants & terms saves time & memory.
- The memory allocation strategy has a large impact.

Have fun playing with Dedukti / Kontroli!