How I Learned to Stop Worrying and Implement Dedukti Myself

Michael Färber

June 25, 2022

Section 1

Introduction

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 1: Vouloir le beurre et l'argent du beurre.

My Background

- 2019–2020: member of DEDUCTEAM
- extraction of proofs from proof assistent Isabelle
- proofs quite large $\rightarrow$ reimplement Dedukti (DK) to make it faster
- result: proof checker Kontroli (KO)

My Background

- 2019–2020: member of DEDUCTEAM
- extraction of proofs from proof assistent Isabelle
- proofs quite large $\rightarrow$ reimplement Dedukti (DK) to make it faster
- result: proof checker Kontroli (KO)

Goal of this Talk

I want to give you some basic knowledge of the Dedukti implementation. This should lower the barrier for you to tackle fun projects such as:

Processing DK theories

- transform DK theories to a proof blockchain
- machine 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)

Section 2

Preliminaries

Terms

Table 1: Definition of terms t, u.

$t, u :=$	description	examples
$\mid s$	sort	$\mathtt{Type}$, $\mathtt{Kind}$ (the type of $\mathtt{Type}$)
$\mid c$	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	$\mathtt{Type}$, $\mathtt{Kind}$ (the type of $\mathtt{Type}$)
$\mid c$	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	$\Pi x : nat.\, vec\,x \to vec\,x$

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

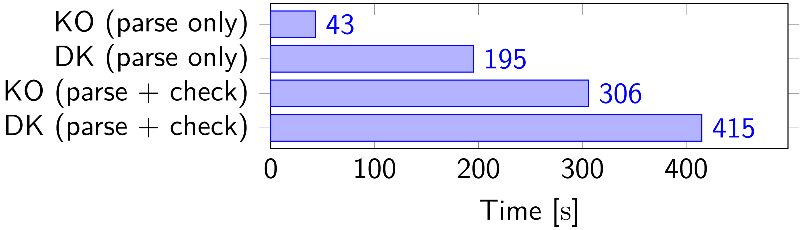

Figure 2: Processing the Isabelle/HOL dataset with Kontroli & Dedukti.

- Parsing can take up to half the total proof checking time
- Automatically generated parser can become a performance bottleneck

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

Pitfalls of Dedukti's Syntax

- The Dedukti syntax is relatively "low-level".
- Yet, parsing it is still far from trivial.

```
f : A    # a constant, variable, or start of a quantifier?
```

Pitfalls of Dedukti's Syntax

- The Dedukti syntax is relatively "low-level".
- Yet, parsing it is still far from trivial.

```
f : A    # a constant, variable, or start of a quantifier?
```

```
f : A
    B    # an application
```

Pitfalls of Dedukti's Syntax

- The Dedukti syntax is relatively "low-level".
- Yet, parsing it is still far from trivial.

```
f : A    # a constant, variable, or start of a quantifier?

f : A
    B    # an application

f : A
    B
 -> C    # a product
```

There can be surprises lurking at the end ...

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

Existing parsers

OCaml: the parser in `dkcheck`

- automatically generated
- good error reporting
- supports full DK syntax (by definition)

Rust: the parser in `kocheck` (`dedukti_parse`)

- hand-written
- lazy and strict parsing with and without scoping
- highly 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!

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
    use dedukti_parse::{Lazy, Symb};
    let cmds = Lazy::<_, Symb<String>, String>::new(lines);

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

Section 4

Proof Checking

Proof 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 types of l and r are convertible.
3. Add it to *global context* Γ (initially empty).

Proof 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 types of l and r are convertible.
3. Add it to *global context* Γ (initially empty).

What we need

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

Proof 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 types of l and r are convertible.
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 whether two terms are convertible ($\Gamma \vdash l \equiv_{\beta\mathcal{R}} r$)?

Section 5

Type Checking & Inference

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"

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

- 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.

Type Checking of Rewrite Rules

How to check that a rewrite rule containing variables preserves types?

With Type Annotations

- 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.

Type Checking of Rewrite Rules

How to check that a rewrite rule containing variables preserves types?

With Type Annotations

- 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

- 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 in this case
- Highly complex (I do not really understand how it works)
- Not needed if types for all variables given

Section 6

Convertibility Check

Convertibility Check

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

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

Table 3: Constraints.

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

Convertibility Check

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

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

Table 3: Constraints.

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

Section 7

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!

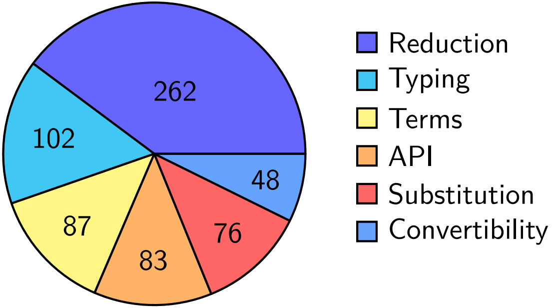

Figure 3: Lines of code of all parts of the Kontroli kernel.

Reduction: Challenges

- Laziness: $\mathit{ite} \top\ T\ F \hookrightarrow T$, $\mathit{ite} \perp\ T\ F \hookrightarrow F$
  (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 *)
}
```

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 Rewriting

- 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

Section 8

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)

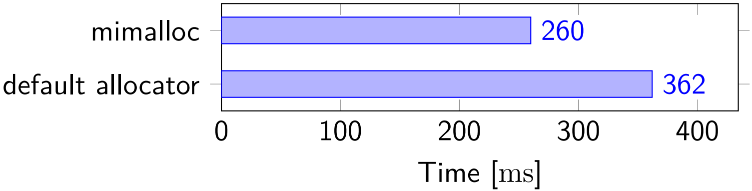

Figure 4: Kontroli checking the Matita dataset using different allocators.

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

Section 9

Conclusion

Conclusion

- The representation of terms is crucial for performance.
- Parsing is an important performance bottleneck.
- Parsing is hard $\rightarrow$ use an existing parser.
- Bidirectional type checking can be omitted if types of rewrite rule variables are annotated by hand.
- Regular type and convertibility checking are easy.
- First-order rewriting is enough for many theories, e.g. HOL.
- Evaluation is hairy due to lazy evaluation, memoization, ...
- Sharing of constants & terms saves time & memory.
- The memory allocation strategy has a large impact.