Documentation Lean Hammer

Phillip Lippe

May 3, 2019

Contents

1 Specific pipeline DTT to FOL

1.1 Approach

The first version of the Lean hammer implements a direct translation from DTT to FOL. The approach follows the Coq hammer paper by introducing an intermediate representation

1.2 Implementation

1.2.1 Code structure

The code is split into multiple files

- `leanhammer.lean`: Main file combining all functions for testing. In here, we include examples of translation, and show the usage of the overall system.
- `problem_translation.lean`: Summarizes the steps for translating a problem into FOF. Note that this file should mostly be independent of the actual encoding of the first-order logic.
- `premise_selection.lean`: Implementation of strategies for selecting the most relevant premises/axioms. Still in a very early form. Might be moved into C/C++ code for performance gain.
- `import_export.lean`: Handling functions for the import and export of files (communication to first-order provers)

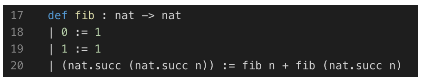
Next to the general files, there are several functions that directly interact with the underlying FOF encoding. Currently, only the TPTP3 encoding is fully supported that can be processed by theorem provers like E. The implementation for that is structured as follows:

- `tptp.lean`: Summarizes data structures for simple first-order formula and their representation in the TPTP format
- `translation_tptp.lean`: Functions for translating expressions into FOF. This include the hammer functions *F*, *C* and *G*.
- `simplification_tptp.lean`: Simplification functions for TPTP encoded first-order formula

1.2.2 Encoding of inductive declarations

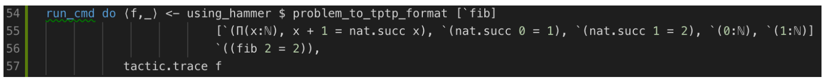
1.3 Examples

The file `leanhammer.lean` contains a few examples of translating DTT to FOL. For explanation purpose, Figure 1a shows a sample translation example where we want to prove that the second Fibonacci number is equals to 2.



```
def fib : nat -> nat
| 0 := 1
| 1 := 1
| (nat.succ (nat.succ n)) := fib n + fib (nat.succ n)
```

(a) Sample inductive declaration



```
run_cmd do ⟨f,_⟩ <- using_hammer $ problem_to_tptp_format [`fib]
                        [`(Π(x:ℕ), x + 1 = nat.succ x), `(nat.succ 0 = 1), `(nat.succ 1 = 2), `(0:ℕ), `(1:ℕ)]
                        `((fib 2 = 2)),
            tactic.trace f
```

(b) Sample translation problem

Figure 1: FOL translation example in Lean. The code can be found in the file `leanhammer.lean`. The left side shows the inductive declaration of `fib`, and the right the call for translating this into plain FOL in TPTP encoding.

The main function that is used is `problem_to_tptp_format`. This function takes as first argument a list of declarations that are eventually relevant for the conjecture and should be included in the translation. In our case, we want to translate the inductive declaration of the Fibonacci numbers as shown in Figure 1a. The second argument is a list of expressions that should be translated as clauses. Here, we for example have to define that 0 and 1 are natural numbers. If we would leave these out, we will not be able to find a proof as the clause implementing `fib` in FOL has as pre-condition that the argument is of type `nat`. The last expression we enter to the translation process is of course the conjecture. By tracing the output of the function, we see the translated construct in text form, which can be used for other theorem provers.

To use this translation, copy output to the file `test_problem.tptp` (note that the name of the file can be changed). Run a theorem prover like E on this file by executing the command `./eprover --auto --tptp3-in` *`file/to/`*`test_problem.tptp`. If everything worked out correctly, E finds a proof in a few milliseconds.

1.4 Open issues

- Up till now, the user has to specify by himself which declarations and expressions should be translated besides the conjecture. This process should be automated by a retrieving all possible clauses/declarations that are somehow connected with the conjecture, and then filtering/ranking these to only take the N most relevant ones. Code based on an old version of Lean can be found here, and the file `premise_selection.lean` might be a good starting point.

- The translation of inductive declarations works fine in the Lean hammer. However, plain declarations like:
  `def sum_two (x:ℕ) (y:ℕ) :ℕ := x+y`
  cause troubles in the translation problem. Lean represents these functions as lambda expressions although we need it in pi notation (for all `x` and `y`, `sum_two x y = x + y`). This translation needs to be integrated into the hammer. For debugging, a quick fix is to write all declarations inductively.

- Currently, the interaction to the theorem prover is in a debug stage where the user needs to copy the output of Lean into a file, and run E on it. In future, this process should be automated by an IO import/export to theorem provers.

- In relation to the previous points, once the IO communication with the theorem provers is automated, a proof reconstruction needs to be implemented. This would help the user to understand the proof, and provide him a way of implementing this proof in Lean.

- Another missing point is the translation of inductive types like `list`. Currently, these types are not supported.

2 General pipeline

To support theorem provers of different input levels (TF0, TF1, TH0, TH1), the generalized version of the Lean hammer is planed to follow a pipeline which successively translates the Dependent Type Theory to FOL. The pipeline is visualized in Figure 2.

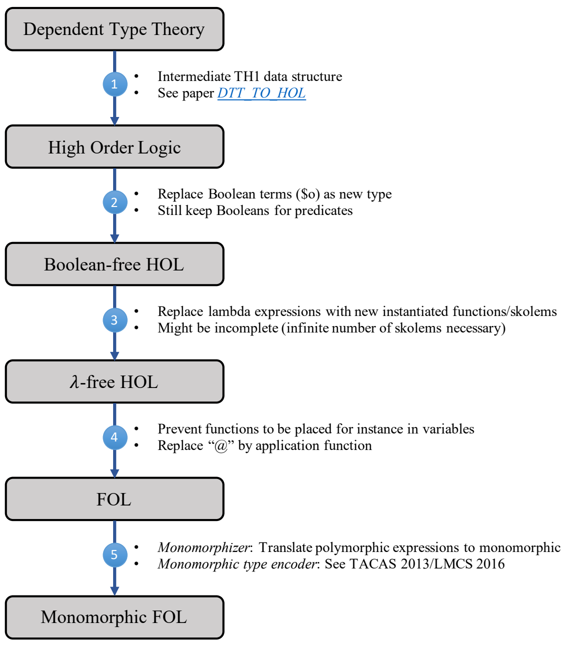

Dependent Type Theory
1
Intermediate TH1 data structure
See paper DTT_TO_HOL
High Order Logic
2
Replace Boolean terms ($o) as new type
Still keep Booleans for predicates
Boolean-free HOL
3
Replace lambda expressions with new instantiated functions/skolems
Might be incomplete (infinite number of skolems necessary)
λ-free HOL
4
Prevent functions to be placed for instance in variables
Replace "@" by application function
FOL
5
Monomorphizer: Translate polymorphic expressions to monomorphic
Monomorphic type encoder: See TACAS 2013/LMCS 2016
Monomorphic FOL

Figure 2: General overview of the translation pipeline from Dependent Type Theory to different logic forms.

2.1 Dependent Type Theory to High Order Logic

Paper DTT to HOL: paper. We introduce an intermediate data structure based on TH1 which can be parsed.

2.2 High Order Logic to Boolean-free HOL

2.3 Boolean-free HOL to λ-free HOL

2.4 λ-free HOL to FOL

2.5 FOL to monomorphic FOL

A FOL examples

```
fof('_fresh.265.4460',
axiom,
(t(a(a('has_one.one',
'nat'),
'nat.has_one'),
'nat'))).

fof('_fresh.265.4459',
axiom,
(t(a(a('has_zero.zero',
'nat'),
'nat.has_zero'),
'nat'))).

fof('_fresh.265.4455',
axiom,
((a(a(a('bit0',
'nat'),
'nat.has_add'),
a(a('has_one.one',
'nat'),
'nat.has_one'))
= a('nat.succ',
a(a('has_one.one',
'nat'),
'nat.has_one'))))).

fof('_fresh.265.4454',
axiom,
(t(a('nat.succ',
a(a('has_one.one',
'nat'),
'nat.has_one')),
'nat'))).

fof('_fresh.265.4453',
axiom,
((a(a('has_one.one',
'nat'),
'nat.has_one')
= a('nat.succ',
a(a('has_zero.zero',
'nat'),
'nat.has_zero'))))).

fof('_fresh.265.4452',
axiom,
(t(a('nat.succ',
a(a('has_zero.zero',
'nat'),
'nat.has_zero')),
'nat'))).
```

```
fof('_fresh.265.4450',
axiom,
(! [V1 /* _fresh.265.4451 */] :
((t(V1,
'nat'))
=> (t(a('fib',
V1),
'nat'))))).

fof('_fresh.265.4447',
axiom,
((a('fib',
a(a('has_zero.zero',
'nat'),
'nat.has_zero'))
= a(a('has_one.one',
'nat'),
'nat.has_one')))).

fof('_fresh.265.4444',
axiom,
((a('fib',
a(a('has_one.one',
'nat'),
'nat.has_one'))
= a(a('has_one.one',
'nat'),
'nat.has_one')))).

fof('_fresh.265.4437',
axiom,
(! [V1 /* _fresh.265.4438 */] :
((t(V1,
'nat'))
=> ((a('fib',
a('nat.succ',
a('nat.succ',
V1)))
= a(a(a(a('has_add.add',
'nat'),
'nat.has_add'),
a('fib',
V1)),
a('fib',
a('nat.succ',
V1)))))))).

fof('problem_conjecture',
conjecture,
((a('fib',
a(a(a('bit0',
'nat'),
'nat.has_add'),
a(a('has_one.one',
'nat'),
'nat.has_one')))
= a(a(a('bit0',
'nat'),
'nat.has_add'),
a(a('has_one.one',
'nat'),
'nat.has_one'))))).
```