Documentation Lean Hammer

Phillip Lippe

June 7, 2019

Contents

General remarks

This documentation contains a description of a hammer designed for the Dependent Type Theory of Lean. The aim is to use fast theorem provers of lower-level logics (like FOL, TH0 etc.) to support the user of finding proofs in Lean.

The current code can be found on this github repository https://github.com/phlippe/Lean_hammer. It is recommended to take a look at the code while reading this documentation. The code is commented for giving low-level details, but the high-level overview can be found in this documentation.

Overview

In general, there are two development branches for the Lean hammer. When this project started, we focused on a similar approach to which is already implemented in the Coq proof assistant [1]. In there, we have a pipeline designed for translating Dependent Type Theory to First-order logic with equality. This approach is described in Section 1.
The second approach, which was introduced later and is still in construction, is designing a pipeline which allows a step-wise translation to FOL where we can stop at any time. Thereby, we would first translate DTT to e.g. the high-order logic TH1. This can be translated to e.g. TH0 or TF1, and finally we end up in TF0 or plain FOL. This would allow us to use a much greater variety of theorem provers, and maybe makes the translation process more tractable. The current state of this idea is presented in Section 2.

Contact

If you have any questions about the code/design/etc., do not hesitate to contact me by phillip.lippe@googlemail.com.

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 [1] by introducing an intermediate representation. Details are explained along the code description.

1.2 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 $\mathcal{F}$, $\mathcal{C}$ and $\mathcal{G}$ of [1] (described in detail below).
- `simplification_tptp.lean`: Simplification/optimization of terms for TPTP encoded first-order formula. Here for example terms that only occur together (like `has_add(nat)`), are replaced with a new constant name.

All the mentioned steps in the previous descriptions will be explained below.

1.3 Low-level translation

In this section, the translation of single units (here Lean expressions) is described. This part is highly inspired by [1].

1.3.1 Translating expressions into intermediate representations

The Coq paper relies on an intermediate representation called CIC_0 in which the DTT expressions are encoded with the help of the functions $\mathcal{F}$, $\mathcal{C}$ and $\mathcal{G}$. In this implementation of the Lean hammer, a similar approach is used. The data structures of the intermediate representations, which are closely related to FOL, can be found in `tptp.lean`. An axiom for example consists of a name and a formula in FOL, which again can consist of multiple formula or terms. The set of all axioms and constants which were introduced in the translation process, are summarized in the data structure `hammer_state`. To this structure, axioms are successively added during the translation.
For the export of this data structure to other theorem provers, the function `to_format` is defined for the structures `folform` and `folterm` which returns the corresponding TPTP FOL representation of the formula/terms as a string. This can be used to export the translated axioms and use them as input to e.g. E [3].

The main translation happens in the functions `hammer_c`, `hammer_g` and `hammer_f` which implement $\mathcal{C}$, $\mathcal{G}$ and $\mathcal{F}$ of [1]. In general, the functions have the following purpose:

- $\mathcal{F}$ encodes propositions as FOL formulas. Therefore, it takes a Lean expression which has to be of type `Prop` (e.g. `1+1=2`), and translates it into the previously defined data structures. Note that for expressions that are not propositions, we can only include them as type guard with $\mathcal{G}$. For more details how declarations/other expressions in Lean are used, see Section 1.4.1.

- $\mathcal{C}$ takes a Lean expression, and translates it into a FOL term. The function $\mathcal{F}$ makes use of it while translating (e.g. for translating `1+1=2`, it calls $\mathcal{C}$ on the expression `1+1` and `2`).
- $\mathcal{G}$ encodes type guards on FOL terms. This includes for example that `1` is of type $\mathbb{N}$. The corresponding proposition used for this is noted by T here, and we add new axioms as for example $t(1, \mathbb{N})$. When translating declarations, these guards are used to limits the possible input arguments to which the declaration/function holds. For an example, see Appendix A.1 formula `_fresh.265.4450`. Note that these type constraints get redundant if we use language that include types, as e.g. TF0.

The implementation of these functions are closely related to the definitions in [1]. For details, we refer to the code.

1.3.2 Term optimizations

The translation process described in the previous section might be inefficient in some cases. Take for example the number `2`. From Lean, the expression is translated to:

```
a(a(a('bit0','nat'),'nat.has_add'),a(a('has_one.one','nat'),'nat.has_one'))
```

This term already contains 5 applications, and is most times not necessary. Therefore, we apply the following simplification rule:

> For a term `abc('test', 'c')` where `abc` is not used anywhere else in combination with variables (neither `abc('test',V1)`, `abc(V1,'c')` nor `abc(V1,V2)`), we can simplify it to a new constant `'abc_test_c'`.

An example output for the optimized output is shown in Appendix A.1.1. Currently, the name is just the concatenation of the term names that were combined for the new constant. To improve readability and guarantee unique term names for any user input, the name generation should be replaced by a combination of unique constant name and a small part for readibility (e.g. `const_xxxx_two_nat` for the expression of `2`). Further optimization ideas with which this hammer could possibly be extended, are described in Section 5 of [1].

1.4 High-level translation

This section describes the translation process from the perspective of Lean formula/expressions. This includes for example declarations and other expressions that might be necessary for the proof in FOL.

1.4.1 Encoding axiom expressions

Despite the function $\mathcal{F}$ as described in Section 1.3.1, some expressions in Lean might not be of type `Prop`. The implementation to this task can be found in the function `translate_axiom_expression` in the file `translate_axiom_expression.lean`. In general, we translate expressions that are not of type `Prop` to type guard expressions. For example, the expression $1 : \mathbb{N}$ would be translated to $t(1, \mathbb{N})$.
We also take care of equality relations. An expressions that states $a = b$ in Lean will be translated in the same formula in FOL, but we apply $\mathcal{C}$ on both the left and the right term.

1.4.2 Encoding inductive declarations

In case of an inductive declaration, a pre-processing step is required to encode it as a list of axioms (one axiom per constructor). For this, the tactic `tactic.get_eqn_lemmas_for tt e` is applied on the declaration name `e` which extracts all equation lemmas for an inductive declaration. In case of the Fibonacci example which is shown in Figure 1a, we would therefore get three axioms (which can be seen in Appendix A.1 axioms `_fresh.265.4447`, `_fresh.265.4444` and `_fresh.265.4437`). Each of these axioms is translated by the procedure described in Section 1.4.1.

1.5 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.
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

```
17  def fib : nat -> nat
18  | 0 := 1
19  | 1 := 1
20  | (nat.succ (nat.succ n)) := fib n + fib (nat.succ n)
```

(a) Sample inductive declaration

```
54  run_cmd do (f,_) <- using_hammer $ problem_to_tptp_format [`fib]
55                                     [`(Π(x:ℕ), x + 1 = nat.succ x), `(nat.succ 0 = 1), `(nat.succ 1 = 2), `(0:ℕ), `(1:ℕ)]
56                                     `((fib 2 = 2)),
57            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.

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 [3] 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.

1.6 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.

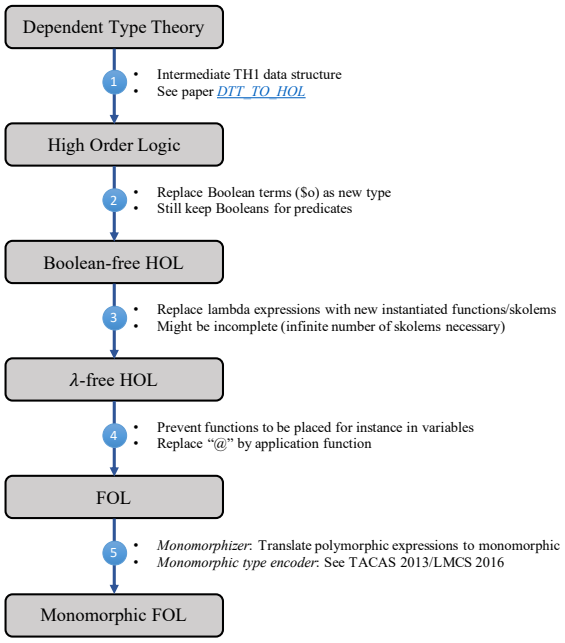

Figure 2: General overview of the translation pipeline from Dependent Type Theory to different logic forms.

2.1 Dependent Type Theory to High Order Logic

The first step of our pipeline is parsing the Dependent Type Theory of Lean to TH1. We decided to take TH1 as highest-level language as most high-order theorem prover can deal with this language. This section first describes the data structures itself, and then continues with discussing the translation process.

2.1.1 Data structures

holtype The data structure `holtype` represents type expressions in TH1. The following types are distinguished:

- The standard types `$o` (boolean), `$i` (individual) and `$type` (type class) as well as the numerical types `$int`, `$rat` and `$real` are implemented as constants
- Local types can be created by `ltype`. It is intended that these types are applied when using previously defined types in axioms. Example:

  ```
  thf(bird_type, type, (bird : $tType)).
  thf(tweety_type, type, (tweety: bird)).
  ...
  ```

  The usage of `bird` in the second type definition should be done by `ltype`. Note that the usage of `bird` in the first line is not included in a `holtype` data item, but in a full `type_definition`.
- `functor` provides support for defining functions with multiple parameters. Example:

  ```
  thf(map_type, type, (map : $tType > $tType > $tType)).
  ```

  The functor type should be defined as:

  ```
  holtype.functor [holtype.type, holtype.type] holtype.type
  ```

  Note that we use a list instead of a recursive structure in order to simplify encodings for other logic levels. In TF0 and TF1, we for example have to write:

  ```
  tff(map_type, type, (map : ($tType * $tType) > $tType)).
  ```
- The deep binder option is thought to be used for type signatures with type arguments. An example use-case is:

```
thf(bird_lookup_type, type, (bird_lookup: !>[A:$tType, B:$tType] : ((map@A@B) > A > B))).
```

- At last, we also need to support partial applications as term arguments. Thus, in the previous example, `map@A@B` should be encoded by `holtype.partial_app`.

holterm This datastructure represents term expressions in TH1. Supported expressions are:

- Constants can be encoded by `const`. This also includes function names like `map`. Alternatively, a local constant can be defined where the second argument specifies the type of the constant.
- Proofs (aka constants of a type whose type is `Prop`) are explicitly handled by the term option `prf`.
- Variables in a formula are specified by a natural number instead of a name. We use this as counter throughout the formula to create variable names (i.e. `X1`, `X2`, etc.).
- Applications of a function on parameters is encoded by `app`. We use a recursive pattern to support function with more than one input parameters. Hence, the term `map@A@B` (or `map(A,B)`) is translated to:

  `map@A@B  =>  app(app(map, A), B)`
- Lambda expressions are specified by the name and type of the input parameter, and the return expressions. An example translation from TH1 is as follows:

  `^ [Q : $o] : $false  =>  holterm.lambda (‘Q‘, holtype.o, holterm.bottom)`
- As languages like TH1 define separate keywords for `$true` and `$false`, they are also included here as options (called `top` and `bottom` respectively). Note that these terms are not valid for Boolean-free HOL.

holform The data structure `holform` combines the previous two data structures to formula.

- Type constraints are encoded by `holform.T`. This can be used if a type check is used within another formula. However, type definitions should not be defined here, but rather by the explicit data structure `type_def`.
- Equality is encoded by `holform.eq`.

  `A = B  =>  holform.eq (A, B)`
- Standard logical operators such as negation, implication, disjunction and conjunction are provided as well.
- The quantifiers $\forall$ and $\exists$ take a name and type as input to specify the parameter/variable of the quantifier. The last input is the term over which the quantifier should hold. Example:

  `forall x : nat, 2*x=x+x  =>  holform.all (‘x‘, holtype.i, holform.eq (...))`
- For terms that cannot be converted to any of these formula options, we use `holform.P` which represents the provability of a term.

We collect all formula in a data structure called `hammer_state`. We thereby distinguish between axioms and type definitions. An axiom is specified by a name (note that this name can be arbitrary and does not influence the proof process), and its formula itself which is assumed to be true. It encodes examples like:

```
thf('test_axiom',axiom, 1+1=2).
```

A type definition in contrast only specifies the type of a constant throughout the proof. The first name (`def_name`) is an arbitrary name for debugging as for axiom. The second specifies the type name that is defined here. The last argument is a `holtype` argument that encodes the explicit type. Example:

```
thf('test_type_def',type,(myType : $o > $tType > $i)).
```

2.1.2 Translation

The translation of DTT to HOL was implemented with inspiration of the translation of DTT to HOL paper [2], and the CoqHammer paper [1]. The translation process resembles the structure of the pipeline from DTT to FOL, but was extended and/or simplified at certain points.

Lean expressions to HOL
Implementation in `translation_hol.lean`
Similar to the approach for the specific pipeline, we again implement the function $\mathcal{F}$, $\mathcal{C}$ and $\mathcal{G}$ but $\mathcal{G}$ does this time not introduce a type constraint, but returns the `holtype` for a given expression. Therefore, each function converts a given expression to another data structure. We describe here the single steps of each of the three functions:

$\mathcal{G}$ **: expr** $\rightarrow$ **holtype**

- The type $\mathbb{N}$ is translated to `$int`. Note that this assignment is currently a test idea and might not hold for all cases as `$int` also includes negative numbers. Change this if needed.
- The type `Prop` (or also referred to as `Sort 0`) is translated to `$o`
- Any type of form `Sort x` with `x` $= 1, 2, ...$ is currently translated to `$type`. For some cases, it might be necessary to translate types with `x` > 1 to separate instances.
- Partial applications like `A@B` are translated as such by using `holtype.partial_app`. Note that we would translate `A` and `B` by $\mathcal{C}$ first to get term expressions.
- λ-expressions within types are translated by a deep binder as the result of it (which is assumed to be a type) depends on the input parameters. We therefore introduce a new variable and translate its type with $\mathcal{G}$. Next, we translate the inner expression of the lambda by $\mathcal{G}$, and finally combine everything into `holtype.dep_binder`. Note that if the inner expression is a lambda expression as well, we combine both binders into one.
- Standard function types such as $\mathbb{N} \rightarrow \mathbb{N}$ are handled by `holtype.functor`. To translate this kind of expression, we first translate both the left and right part by $\mathcal{G}$, and combine them into a functor structure. In cases where we have more than two arguments, e.g. $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$, we translate it in the default order $\mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$ but combine the parameter list afterwards. Note that pi-expressions are included in the expressions `‘(%%l → %%r)`.
- Besides the ones mentioned above, any constant expression is translated to a local type `holtype.ltype` with the expressions name itself.
- If we get an expression that does not fit to any of the mentioned cases, we simply use the type `$i` (default type for any object/individual)

$\mathcal{C}$ **: expr** $\rightarrow$ **holterm**

- We translate constants simply into `holterm.const`, except they are a proof. In this case, we translate it into `holterm.prf`. Local constants are translated with the same principle, only that the type of the constant (determined by $\mathcal{G}$) is added for `holterm.local_const`
- Applications can also be translated to its corresponding data structures `holterm.app`, except if one of them is a proof which leads to either the proof itself or the function applied on it.
- Pi expressions for `holterms` **are not implemented yet** as it is not clear yet how to deal with them the best way in TH1 (for pi expressions in formula, look at function $\mathcal{F}$).
- In contrast to FOL, lambda expressions can now simply be converted to `holterm.lambda`. For this, we create a new variable term, and determine its type by $\mathcal{G}$. The inner expression is again recursively translated by $\mathcal{C}$.

$\mathcal{F}$ **: expr** $\rightarrow$ **holform**

- Pi expressions are translated as $\forall$ formula as we assume the inner expressions to be from type `Prop` (or at least be translatable to `holform`). We therefore again introduce a new variable, and translate its type by $\mathcal{G}$.
- Logical relations/operators such as `and`, `or`, `imp`, `iff` and `not` are explicitly integrated into the data structure `holform`
- Equalities are simply translated into `holform.eq` with the left and right side being translated to `holterm` by $\mathcal{C}$.
- In case that the expression does not fit to any of the cases mentioned above, we use `holform.P` representing the provability of a term.

Axiom expressions
Implementation in `problem_translation_hol.lean`
When translating axiom expressions, we distinguish between two cases on a high-level view: equality and type definitions. If the expression to translate is a equality `L=R` (e.g. `1+1=2`), we check whether the right-hand side expression is:

- a proof. In this case, we can simply translate type (so the actual `Prop`-expression) by $\mathcal{F}$, and add it as axiom.
- of type `Prop`: Then we replace the equality of the whole expression by a biconditional `L`$\Leftrightarrow$`R`, translate both `L` and `R` by $\mathcal{F}$, and finally add this statement as new axiom.

- any other type: If none of the previous cases applies, we can simply apply $\mathcal{C}$ on each `L` and `R`, and create the new axiom with the equality relation.

Equalities are a special cases where we can translate them directly into HOL. However, for expressions which do not have a equality on the highest level, we can do the same translation process we did for the right-hand side. Instead of the biconditional in case of `Prop`, we can simply add $\mathcal{F}$ (`R`) to the clause/axiom database. However, if `R` is from any other type, we now add the axiom as type definition. As an example, consider the fibonacci function as shown in Figure 1. If we now pass the expression `'fib` to the translation process, we would get the following type definition (with arbitrary name):

```
thf('_fresh.758.915',type,(fib : $int > $int)).
```

This type definition expresses that `fib` takes as input parameter an integer (here we map $\mathbb{N}$ to `$int`, might have to be changed) and return as well an integer. The recursive, inductive declarations of this functions need to be passed to the translation function as well as the expression `'fib`. Note that for inductive declarations, we use the same functions as for FOL. The full example with the same input as in Figure 1 is shown in Section A.2.

2.1.3 Open issues

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

References

[1] Łukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory. *Journal of automated reasoning*, 61(1-4):423–453, 2018.

[2] Bart Jacobs and Tom Melham. Translating dependent type theory into higher order logic. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, pages 209–229, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[3] Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.

A Example translations

A.1 FOL examples

The non-optimized output for the example of proving that `fib 2 = 2` is shown below:

```
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'))))).
```

A.1.1 Optimized FOL export

The optimized version get rids of some constant term applications, and replaces it with a new name. For debugging purpose, the names are right now the concatenation of the term names that were combined for the new constant:

```
fof('_fresh.325.1061',
axiom,
(t('const_.a_.a_.c_.has_one.one.-.c_.nat.-.c_.nat.has_one',
'nat'))).

fof('_fresh.325.1060',
axiom,
(t('const_.a_.a_.c_.has_zero.zero.-.c_.nat.-.c_.nat.has_zero',
'nat'))).
```

```
fof('_fresh.325.1056',
axiom,
(('const_.a_.a_.a_.c_.bit0.-.c_.nat.-.c_.nat.has_add.-.c_.const_.a_.a_.c_.has_one.one.-.c_.nat.-.c_.nat.has_one'
= a('nat.succ',
'const_.a_.a_.c_.has_one.one.-.c_.nat.-.c_.nat.has_one')))).

fof('_fresh.325.1055',
axiom,
(t(a('nat.succ',
'const_.a_.a_.c_.has_one.one.-.c_.nat.-.c_.nat.has_one'),
'nat'))).

fof('_fresh.325.1054',
axiom,
(('const_.a_.a_.c_.has_one.one.-.c_.nat.-.c_.nat.has_one'
= a('nat.succ',
'const_.a_.a_.c_.has_zero.zero.-.c_.nat.-.c_.nat.has_zero')))).

fof('_fresh.325.1053',
axiom,
(t(a('nat.succ',
'const_.a_.a_.c_.has_zero.zero.-.c_.nat.-.c_.nat.has_zero'),
'nat'))).

fof('_fresh.325.1051',
axiom,
(! [V1 /* _fresh.325.1052 */] :
((t(V1,
'nat'))
=> ((a(a('const_.a_.a_.c_.has_add.add.-.c_.nat.-.c_.nat.has_add',
V1),
'const_.a_.a_.c_.has_one.one.-.c_.nat.-.c_.nat.has_one')
= a('nat.succ',
V1))))).

fof('_fresh.325.1048',
axiom,
(! [V1 /* _fresh.325.1049 */] :
((t(V1,
'nat'))
=> (t(a('fib',
V1),
'nat'))))).

fof('_fresh.325.1045',
axiom,
((a('fib',
'const_.a_.a_.c_.has_zero.zero.-.c_.nat.-.c_.nat.has_zero')
= 'const_.a_.a_.c_.has_one.one.-.c_.nat.-.c_.nat.has_one'))).

fof('_fresh.325.1042',
axiom,
((a('fib',
'const_.a_.a_.c_.has_one.one.-.c_.nat.-.c_.nat.has_one')
= 'const_.a_.a_.c_.has_one.one.-.c_.nat.-.c_.nat.has_one'))).

fof('_fresh.325.1035',
axiom,
(! [V1 /* _fresh.325.1036 */] :
((t(V1,
'nat'))
=> ((a('fib',
a('nat.succ',
a('nat.succ',
V1)))
= a(a('const_.a_.a_.c_.has_add.add.-.c_.nat.-.c_.nat.has_add',
a('fib',
V1)),
a('fib',
a('nat.succ',
V1))))))).

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

A.2 HOL example

```
thf('_fresh.758.915',type,(fib : $int > $int)).

thf('_fresh.758.919',axiom,(((((bit0@nat)@nat.has_add)@((has_one.one@nat)@nat.has_one))=
(nat.succ@((has_one.one@nat)@nat.has_one))))).

thf('_fresh.758.918',axiom,((((has_one.one@nat)@nat.has_one)=(nat.succ@((has_zero.zero@nat)@nat.has_zero))))).

thf('_fresh.758.917',axiom,(! [V1:$int] : (((((has_add.add@nat)@nat.has_add)@V1)@((has_one.one@nat)@nat.has_one))=
(nat.succ@V1)))).

thf('_fresh.758.912',axiom,(((fib@((has_zero.zero@nat)@nat.has_zero))=((has_one.one@nat)@nat.has_one)))).

thf('_fresh.758.909',axiom,(((fib@((has_one.one@nat)@nat.has_one))=((has_one.one@nat)@nat.has_one)))).

thf('_fresh.758.903',axiom,(! [V1:$int] : ((fib@(nat.succ@(nat.succ@V1)))=
((((has_add.add@nat)@nat.has_add)@(fib@V1))@(fib@(nat.succ@V1)))))).

thf('problem_conjecture',conjecture,(((fib@(((bit0@nat)@nat.has_add)@((has_one.one@nat)@nat.has_one)))=
(((bit0@nat)@nat.has_add)@((has_one.one@nat)@nat.has_one))))).
```