HOL4-Beagle, an implementation of an higher order to first order translation

Gauthier Thibault

September 2, 2013

Abstract

The interactive theorem prover HOL4 is enhanced with an external automated theorem prover Beagle. Performance of this architecture is evaluated against problems solved by an internal prover METIS_TAC during HOL4 build. It is shown that Beagle proves 82% of 300 translated theorems in 15 seconds of real time on one single CPU.

The necessary work involves an implementation of a sound translation of the HOL4 logic to Beagle formalism: typed first-order arithmetic. The monomorphisation step of this translation will be presented in details. Ideas on how to complete the project are briefly exposed, such as a generic Beagle proof output and a HOL4 proof replayer.

Contents

1 Introduction

1.1 Motivation

Tell a story about the context.

1.2 Structure of this paper

The project main focus was on HOL4-Beagle interaction the internal first-order prover METIS_TAC From section 3 to section 5, three different tools are presented, the interactive theorem prover HOL4, the TFF(TPTP) format and the automated theorem prover Beagle. From then, we will show how to enhance HOL4 with the help of Beagle. Our goal is to reduce the need for human guidance in HOL4 by letting Beagle solve problems dealing with linear integer arithmetic.

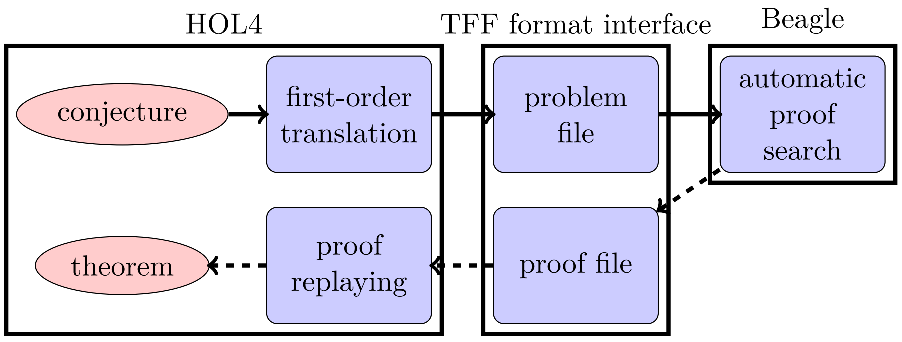
2 Interaction between HOL4 and Beagle

2.1 Goal

The main objective of the project is to create a function called BEAGLE_TAC so that it improves METIS_TAC(an internal first order prover) which doesnt know anything about arithmetic. BEAGLE_TAC will take a conjecture as well as user-provided theorems and returns the conjecture as a theorem.

2.2 Interaction diagram

This is how BEAGLE_TAC would work.



Remark. Only the first line is currently implemented. So BEAGLE_TAC can now only be used as an oracle, it means that the conjecture is true if you believe the HOL4 translation and Beagle are sound.

2.3 Interface

The TFF format will be used as an interface between HOL4 and Beagle. This has some advantages:

1. The problem file created by HOL4 can be given to any other automated theorem prover that can read TFF problems
2. The proof file create by Beagle (work in progress) could be tested within any other interactive theorem prover that can read TFF proofs.

3 The HOL4 interactive theorem prover

3.1 General ideas

HOL4 deals with higher order logic which make it easy for mathematicians to express their formulas. It is based on the functional programming and type checking language SML.

3.2 HOL4 type and formula representation

Definition. (HOL4 types)
Let $\mathfrak{B}$ be the set of basic types. (e.g *num*, *bool*,...)
Let $\mathfrak{P}$ be the set of polymorphic types. (e.g *a*,*b*,...)
Let $\mathfrak{Op}$ be the set of operators. (e.g fun,*list*,...)
The set of all types is defined inductively by:

$$type, ty1, \dots, tyN, \dots := b \mid p \mid (op, [ty1, \dots, tyM])$$

where $b \in \mathfrak{B}$, $p \in \mathfrak{P}$ and $op \in \mathfrak{Op}$.

Remark. $(fun, [a, b])$ is the HOL4 internal representation of $a \to b$.

Remark. To say that a variable f has type t, we write $f : t$.

Definition. (HOL4 Formula)
Let C be the set of constants. (e.g. =,+,!,...)
Let V be the set of variables. (e.g. x,y,...)
let N be the set of natural numbers. (e.g 0, $(SUC0)$, $SUC(SUC0)$),...)
This definition is based on the type lambda-calculus:

$$f, g := c \mid v \mid (g : ty1 \to ty2)f : ty2 \mid \lambda v.f \text{ where } c \in C \text{ and } v \in V$$

To easily deals with numbers, this alternative definition will be used:

$$f, g := c \mid v \mid n \mid (g : ty1 \to ty2)f : ty2 \mid \lambda v.f \text{ where } c \in C \text{ , } v \in V \text{ and } n \in N$$

Remark. A variable has a name and a type that can be accessed by calling $name\_of$ or $type_of$.

Example. $\forall x,\ x = 0$ is represented in HOL by $(!)\lambda x.((=)x)0$.

3.3 Useful SML types

Here is a list of SML types created in HOL4 and their intended use:

1. Term (*term*)
   This is the type of HOL4 formulas described above.

2. Goal ($goal = (term\ list * term)$)
   Used for conjectures in a sequent form.

3. Theorem (*thm*)
   Theorems are represented in a sequent form. A theorem may be created by any function that returns a theorem.

4. Rule ($rule = thm \to thm$)
   Using a rule is the most common way to create theorems. Rules can be used to simulate steps in a derivation.

5. Conversion ($conv = term \to thm$)
   Takes a term and returns a theorem of the form A —- term = term. Conversions will be used to rewrite terms inside a theorem.

6. Tactic ($tactic = goal \to goal\ list * (thm\ list \to thm)$)
   Takes a goal and returns a goal list and a way to reconstruct the proof when every goal in the goal list happens to be proved. Its used in interactive mode, so that the user can construct the proof backwards.

7. Problem ($thm\ list * goal$) This is not a type by itself but it will represent the problem we want to solve.

Remark. Except for the *thm* type, there is no control whether you used the type in the intended way or not.

3.4 Soundness

There are initially a small number of SML functions that returns a theorem and a small number of theorems. You only need to check them to be convinced of the soundness of HOL4. Indeed, the SML type-checking system (which you need to trust) will guarantee that any new variable of type *thm* is a theorem.

3.5 One proof example

Here is an example of a derivation and how it can be proved in HOL4 using forward and backward reasoning.

$$\dfrac{\dfrac{\dfrac{\dfrac{\neg\exists x\ x = 2x \vdash \neg\exists x\ x = 2x \qquad \dfrac{x = 2x \vdash x = 2x}{x = 2x \vdash \exists x\ x = 2x}\,\exists_i}{\neg\exists x\ x = 2x, x = 2x \vdash \bot}\,\neg_e}{\neg\exists x\ x = 2x \vdash \neg x = 2x}\,\neg_i}{\neg\exists x\ x = 2x \vdash \forall x\ \neg x = 2x}\,\forall_i}{\vdash \neg(\exists x\ x = 2x) \to \forall x\ \neg(x = 2x)}\,\to_i$$

Forward proof (using rules):

```
val th1=ASSUME (x=2*x);
val th2=EXISTS (?x:num.x=2*x,x:num) th1;
val th3=ASSUME (~(?x:num. x=2*x));
val th4=NOT_ELIM th3;
val th5=MP th4 th2;
val th6=NEG_DISCH (x=2*x) th5;
val th7=GEN x:num th6;
val th8=DISCH ~?x. x = 2 * x th7;
```

Backward interactive proof (using tactics):

```
g(‘~(?x:num. x=2*x) ==> !x:num. ~(x=2*x)‘);
e(DISCH_TAC);
e(GEN_TAC);(*possible to use because x is not free in hypothesis*)
e(DISCH_TAC);
val th1=ASSUME(~(?x:num. x=2*x));
e(MP_TAC(th1));
e(REWRITE_TAC []);
e(EXISTS_TAC x:num);
e(RES_TAC);
val th8=top_thm();
```

4 The TFF(TPTP) format

The TPTP website (Thousands problems for theorem provers) provides a database for theorem provers to be tested against (see here). A TPTP format is human-readable and non specific prover dependent. The syntax can be checked of a problem can be checked (here) and you can also called some theorem prover on them (here).

4.1 TFF format

The TFF format is one of the format used to write these problems, it supports first-order, typed and arithmetic formulas.

Definition. (TFF type)
Let D be the set of defined type. (e g. $\$o$,$\$int$,…)

let U be the set of user type.
A TFF type is defined inductively by :

$$type, ty1, \ldots, tyN, \ldots := d \mid u \mid (ty1 * \ldots * tyM) > type$$

where $d \in D$ and $u \in U$.

4.2 Derivation format

The derivation format is just a use of a TPTP format to represents derivations, to do so you need to tell which formula was created with which rule was used with which parents. For more information, see (here).

5 The Beagle automated theorem prover

Beagle supports CNF(TPTP), FOF(TPTP), and TFF(TPTP) format as input. It can deals with typed first-order formula and linear integer arithmetic. if Beagle ends, it outputs a result either Unsatisfiable, Satisfiable or Unknown. The proof output is still under development.

5.1 Beagle type and clause

Definition. (Type) The Beagle representation for type uses TFF type.

Definition. (Atom) An atom is term that as boolean type.

Definition. (Literal)

Definition. (Clause) A clause is a disjunction of literals.

5.2 Translation into a clause set

Beagle parsed the TFF file and then normalize it into a clause set using techniques such as skolemisation.

Remark. The normalisation part isnt used in our project as the output from HOL4 is already a clause set.

5.3 Rules

Here are some stats of the usage of different rules in Beagle on a random problem.

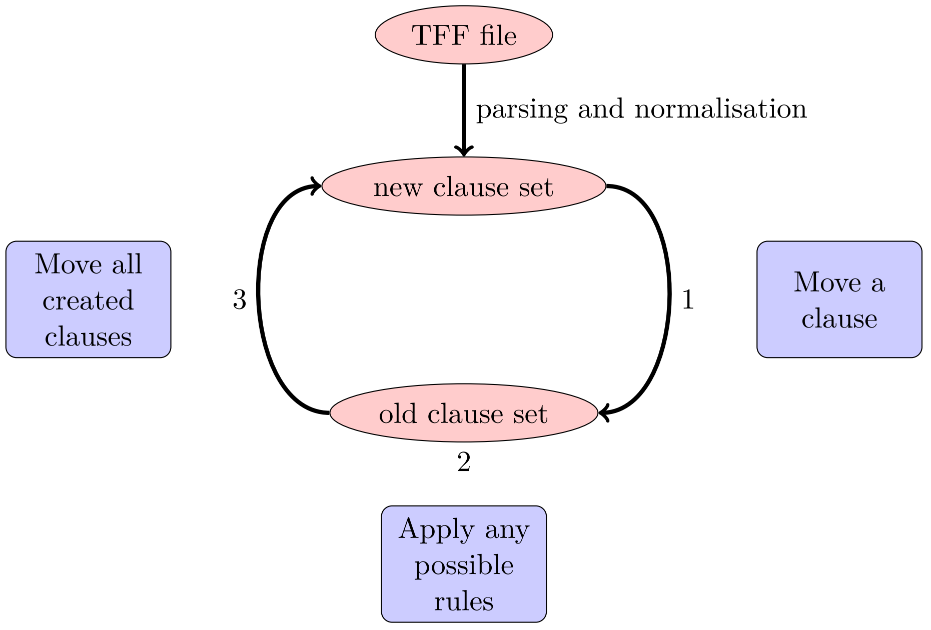
```
Inference rules
----------------------
#Ref:           0
#Sup:           34 (Superposition uses paramodulation see paper)
```

```
#Fact:           0
#Define:         0
#Split:          63 (A special rule)
#Close:          0

Simplification rules
----------------------
#Subsume:        10
#Demod:          7
#Tautology:      30
#SimpNegUnit:    2
#BGTheorySimp:   0

Other
----------------------
#QE:               52
#BGSolverCalls:    4 (Background solvers:
                      one of them deals with arithmetic)
#Ordered Literals: 3163
```

5.4 Derivation search

5.4.1 Saturation loop (no Split)

Beagle uses two clause sets, the new clause set (NCS) and old clause set (OCS). The OCS is initially empty. The NCS is initially the one created by the normalisation step.
Lets consider that the Split rule is not used ,the main loop can be represented by this diagram:



These are the steps of the saturation loop:

1. Move a clause from the new clause set into the old clause set (using a good ordering)
2. Derive all consequences from the application of any rule (except for the Split rule) to any subset of the old clause set. You only to check the subsets that contains the new clauses.
3. Move all created consequences into the new clause set.

Statement. *After each interaction loop, the old clause set is closed under the reunion of the old clause set and the new clause set.*

Definition. (Saturated clause set)

5.4.2 A special rule: Split

Definition. (saturation loop state) Lets represent a saturation loop state by a triplet (clause, OCS ,NCS) where clause is the clause which was moved from NCS to OCS.

Definition. (Split) A Split is defined by this inference step.
Let $a_1 \vee a_2 \vee \dots a_N$ a clause.

$$\frac{(a_1 \vee a_2 \vee \dots a_N, OCS, NCS)}{(a_1, OCS, NCS), (a_2, OCS, NCS), \dots, (a_N, OCS, NCS)}\,\textit{Split}$$

Remark. This is the only rule which creates new saturation loop from one saturation loop, other rules only modifies the saturation they are in.

Remark. The proof can be represented by a tree with each node being a saturation loop.

5.4.3 Termination

There are mostly three possibilities that can happen in that point.

1. Every leaf old clause set contains a contradiction then the clause set is **unsatisfiable**. (Beagle rule set is refutation-complete)

2. There exists a saturated clause set which doesnt contains any contradiction.

   If the inital clause set was ”sufficiently” complete (see [ref]), then the status of this problem is **satisfiable**.

   If the inital clause set was not ”sufficiently” complete, then the status of this problem is **unknown**.

3. The computation may go on forever. (The first order logic is undecidable)

6 HOL4 problem translation

6.1 Code location

The code for this translation can be found at `https://github.com/barakeel/HOLtoTFF` (see Readme to install). This chapter follows exactly how the code is organized. All examples can be found in test.sml.

6.2 Notation and convention

Term of type bool are called formulas and often written f.
We write u[t] when a term t is free in a term u.

6.3 Atoms for a higher order formula

A higher order formulas can be represented in a way that looks like a first order formula. From then, atoms can be defined for a higher order formula.

Definition. (Atom) Let L be the set of logical operators. ($\Rightarrow$,$\wedge$,...)
A term t in a formula f is said to be an atom if it satisfies all this conditions:

- Its operator is not a logical operator.

- For each term that includes this term, his operator is a logical operator.

6.4 Translation order

The translation happen in this order:

1. Monomorphisation. (may be done at anytime but it happens first to improve efficiency. see [ref])
2. Theorems insertion and negation of the conclusion (tactic). The whole problem (theorem list,goal) is rewritten into one single term.
3. CNF conversion.
4. Lambda-lifting conversion.
5. CNF conversion.
6. Boolean arguments conversion.
7. CNF conversion.
8. Numeral variables conversion.
9. CNF conversion.
10. Clause set tactic.
11. Higher order conversion.
12. Numeral functions axioms.
13. Boolean bound variables conversion.
14. Printing.

    Types.

    Variables or constants.

    True and false constants.

    Non-linear integer arithmetic.

6.5 Monomorphisation

In this step, provided theorems containing polymorphic types are instantiated in a clever way.

Example. if we have the theorem $!x : a.x = x$ and the goal $42 = 42$, it is "clever" instantiate the first theorem to num. One way of doing this is finding a substitution that match the type of =: $a \to a \to bool$ with the type of =: $num \to num \to bool$ and applying this substitution to the theorem.

Remark. If the goal contains polymorphic types, they will be treated as new fresh types with non polymorphic value.

6.5.1 Arithmetic for type substitutions

Definition. (Subtitution) function form type to type

(Domain)

Definition. (Compatibility)
Two substitutions σ_1 , σ_2 are said compatible when:

$$\forall x \in D(\sigma_1) \cap D(\sigma_2),\ \sigma_1(x) = \sigma_2(x)$$

Definition. (Addition)

$$"+" : \Sigma \times \Sigma \to P(\Sigma)$$

If σ_1 and σ_2 are compatible :

$$\sigma_1 + \sigma_2 := \{\sigma_1, \sigma_2, \sigma_1 \cup \sigma_2\}$$

If σ_1 and σ_2 are not compatible :

$$\sigma_1 + \sigma_2 := \{\sigma_1, \sigma_2\}$$

Definition. (Multiplication) Let i and j be positive integers:

$$* : P(\Sigma) \times P(\Sigma) \to P(\Sigma)$$

$$\{\sigma_1, \ldots, \sigma_i\} * \{\rho_1, \ldots, \rho_j\} := \bigcup(\sigma_i + \rho_j)$$

Theorem. *(Increasing)*
$*$ is an increasing function: if $\sigma_1 < \sigma_2$ and $\rho_1 < \rho_2$ then $\sigma_1 * \rho_1 < \sigma_2 * \rho_2$.

Example. (Code)

```
mult_subst
  [[:a ↦ :bool], [:b ↦ :bool]]
  [[:b ↦ :num] , [:c ↦ :num]];
val it =
  [
 [:a ↦ :bool, :b ↦ :num],
 [:a ↦ :bool],
 [:a ↦ :bool, :c ↦ :num],
 [:b ↦ :num],
 [:b ↦ :bool],
 [:c ↦ :num],
 [:b ↦ :bool,:b ↦ :bool]
  ]
```

Definition. (Substitution order)
Let σ_1,σ_2 two substitions, $<$ is defined by :

$$\sigma_1 < \sigma_2 \iff D(\sigma_1) \subseteq D(\sigma_2) \wedge (\forall x \in D(\sigma_1),\ \sigma_1(x) = \sigma_2(x))$$

Remark. $[] < [a \rightarrow bool]$ but $[a \rightarrow a] \not< [a \rightarrow bool]$.

Definition. (Maximal substitution)
A substitution σ is said maximal in a set of substitution Σ if it satisfies all these conditions:

- $\sigma \in \Sigma$
- $\forall \rho \in \Sigma,\ \sigma \not< \rho$

Example. (Code) Using the result of the example above.

```
get_maxsubstl it;
val it = [
 [:a ↦ :bool, :b ↦ :num],
 [:a ↦ :bool, :c ↦ :num],
 [:b ↦ :bool,:b ↦ :bool]]
```

6.5.2 Matching substitutions

Definition. (Type matching) Let us call *match_type* any function that takes two types ty_1, ty_2 and returns a substitution σ such as $\sigma(ty_1) = ty_2$ if it is possible.

Let $([th_1, \ldots, th_p], goal)$ be a representation of the problem.
Let $th \in [th_1, \ldots, th_p]$.
Let $C = \{c_1, \ldots, c_m\}$ the set of constants in th.
For each constant c_i, let $substl_i$ be a list of matching substitutions
Let $D = \{d_1, \ldots, d_n\}$ the set of constants in that appears in the problem.
The creation of matching substitutions follow this procedure:

1. For all constants c_i in $\{c_1, \ldots, c_m\}$, find a constant d of the same name in $\{d_1, \ldots, d_n\}$ and try to apply *match_type* to their respective types. If it succeed, add the created substitution to $substl_i$.

2. Compute every possible association of these different substitution lists that is, multiply every non empty substitution list. (see Multiplication section 6.3.1)

The result is a part of all possible substitutions for theorem th which will be called $\mathcal{M}(C, D)$.

Theorem. *(Increasing)*
$\mathcal{M}$ is an increasing function:
Let C_1, C_2, D_1, D_2 be sets of constants.
If $C_1 \subseteq C_2$ and $D_1 \subseteq D_2$ then $\mathcal{M}(C_1, D_1) \subseteq \mathcal{M}(C_2, D_2)$

Proof. It is a corollary of the fact that $*$ for substitutions is an increasing function. □

Remark. Constants will appear in different theorems with the same meaning where as free variables may use with different meanings. Therefore, constants are a way to link theorems together, thats why we chose to monomorphise with the help of constants.

6.5.3 Substitutions composition loop.

Let p be the number of theorems in the initial problem.
Let Sll_0 be $[[], \dots, []]$ be the list of initial substitution list of each initial theorem.
Let $Cll_0 = [C_{0.0}, \dots, C_{0.p}]$ be the list of initial constants list for each theorem.
Let $Cll_i = [C_{i.0}, \dots, C_{i.p}]$ be the list of constants at step i list for each theorem.
Let D_O be the set of all constants present in the initial problem.
We can recursively define:

$$Sll_{i+1} = [\mathcal{M}(C_{i.0}, D_i), \dots, \mathcal{M}(C_{i.p}, D_i)]$$

$$Cll_{i+1} = [\mathcal{I}(\mathcal{M}(C_{i.0}, D_i), C_{i.0}), \dots, \mathcal{I}(\mathcal{M}(C_{i.p}, D_i), C_{i.p})]$$

$$D_{i+1} = D_i \cup \mathcal{C}(Cll_{i+1})$$

where $\mathcal{C}$ concatenate a list and $\mathcal{I}(S, C)$ returns the set of all instantiated constants $c \in C$ with every possible substitution $s \in S$.

Remark. $\mathcal{I}$ is an increasing function.

Remark. The constant $=$ is currently removed all sets as the type of $=$ is almost always derived from other constants. It improves efficiency although it removes some possibilities.

Theorem. *(Fix point)*
If $size(C_{i+1.j}) \leq size(C_{i.j})$, then $C_{i.j} = C_{i+1.j}$.

Proof. (Fix point)
We only need to prove that $C_{i.j} \subseteq C_{i+1.j}$.
By construction:
If $i = 0$,

$$C_{0.j} \subseteq \mathcal{I}(\mathcal{M}(C_{0.j}, D_i)) = C_{1.j}$$

If $i > 0$,

$$C_{i+1.j} = \mathcal{I}(\mathcal{M}(C_{i.j}, D_i))$$
$$C_{i.j} = \mathcal{I}(\mathcal{M}(C_{i-1.j}, D_{i-1}))$$

By the recursive definition it stands that $D_{i-1} \subseteq Di$. By invoking a recursive hypothesis $C_{i-1.j} \subseteq C_{i.j}$, it can be deduced that $C_{i.j} \subseteq C_{i+1.j}$ because $\mathcal{I}$ and $\mathcal{M}$ are increasing functions. □

6.5.4 Termination

The previous procedure has two different outcomes:

1. A fix point is found, id est: $\exists \text{i}, \forall j,\ 1 \leq j \leq p \Rightarrow C_{i,j} = C_{i+1,j}$. In the implementation, initial theorems are instantiated with maximal substitutions in their respective member of Sll_i.

2. It keeps growing: $\forall i, \exists j,\ 1 \leq j \leq p \wedge C_{i,j} \subsetneq C_{i+1,j}$ set strictly grows when i increase. In the implementation, the loop stops when the number of substitutions inside Sll_i is greater than 15 then initial theorems are instantiated with maximal substitutions in their respective member of Sll_{i-1}.

Theorem. *The substitution composition loop for this list of sets of substitutions doesn't find a fix point.*

Remark. (Non-completeness)
If a fix point is found, it doesn't guarantee that the goal can be proved (since for example we don't consider bound function variables). If a fix point is not found, the goal may still be proved.

6.5.5 Dependency graph

Definition.

Theorem. *(No loop theorem)*

Proof. □

6.5.6 Example

Here is an example from the set of problems we tested, where you need repeat the substitutions composition loop twice to find a fix point.

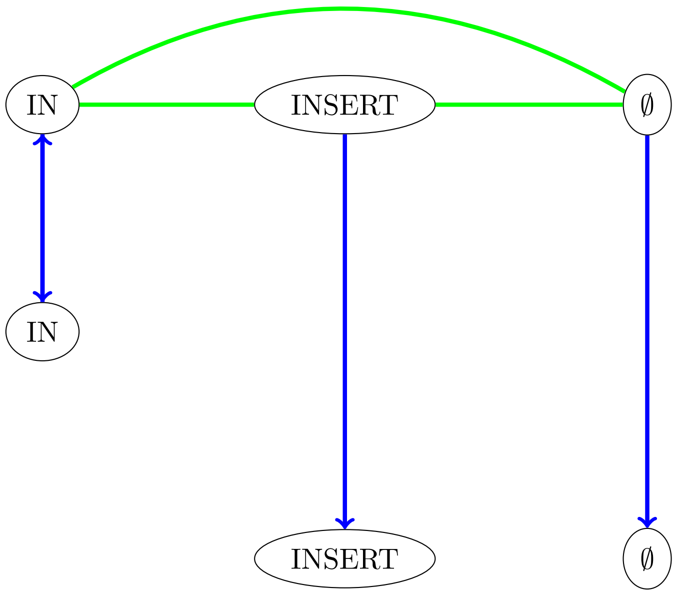
Problem.

```
val th1 = ∀ x y. x ∈ { y }  ⟺  (x = y)
val th2 = ∀ P x. x ∈ P  ⟺  P x
val thml = [th1,th2]
val goal = ([], ∀ x. (x = z)  ⟺  { z } x)
```

Remark. As previously stated in [ref], equality is removed from all set of constants.

Type of each constant in this problem.
Theorem 1:
IN : $c \rightarrow (c \rightarrow bool) \rightarrow bool$
INSERT : $c \rightarrow (c \rightarrow bool) \rightarrow c \rightarrow bool$
$\emptyset$: $c \rightarrow bool$
Theorem 2:
IN : $a \rightarrow (a \rightarrow bool) \rightarrow bool$
Goal
INSERT : $num \rightarrow (num \rightarrow bool) \rightarrow num \rightarrow bool$
$\emptyset$: $num \rightarrow bool$

Dependency graph.



Constants in the same row are from the same theorem.
Blue directed links connects two constants that verifies this properties:
They are from different theorems.
They have the same name.
The type of the final one is a subtype of the initial one.
Green undirected links connects two constants that verifies this properties :

They are from the same theorem.
They share a common polymorphic type.

Matching substitutions.
To compute substitutions induced by the graph for one theorem we look at every possible combination of the substitution(see multiplcation section... ref) induced by each constant in this theorem.
This graph induces 5 substitutions (The two empty substitutions guarantees that we don't erase constants):

```
first theorem  : [], [:c↦:a],  [:c↦:num]
second theorem : [], [:a↦:c]
```

Substitutions composition loop.
New nodes(constants) are created by instantiating old ones with each possible substitutions. Then, we can repeat the previous process with respect to the new graph and obtain the following substitution.

```
first theorem : [],
[:a↦:c], [:a↦:c,:c↦:num],
[:a↦:c,:c↦:a], [:a↦:num],
[:a↦:num,:c↦:num],
[:a↦:num,:c↦:a], [:c↦:a],
[:c↦:num]
```

```
second theorem : [],
[:a↦:num,:c↦:num],
[:a↦:c, :c↦:num],
[:c↦:num], [:a↦:num],
[:a↦:num,:c↦:a], [:a↦:c],
[:a↦:c, :c↦:a], [:c↦:a]
```

If the process is repeated again, the same set of substitutions is returned. Hence, a fix point is found in two steps which is not surprising if we look at the initial graph.

Maximal substitutions.
To improve efficiency, each theorem will be instantiated with maximal substitutions only.

```
first theorem :
[:a↦:c,:c↦:num],
[:a↦:c,:c↦:a],
[:a↦:num,:c↦:num],
[:a↦:num,:c↦:a],
```

```
second theorem :
[:a↦:num,:c↦:num], [:a↦:c,
:c↦:num], [:a↦:num,:c↦:a],
[:a↦:c, :c↦:a],
```

6.6 Normalisation

The code makes use of a already define function called CNF_CONV made by [ref], which deals with these problems:

1. Beta-reduction
2. Skolemisation
3. Logical operator conversion: elimination of the ”IF THEN ELSE” operator ”?!” (exists unique) operator, etc
4. Normalisation: The result is given in conjunctive normal form.

6.7 Lambda-lifting conversion

Remove a lambda abstraction by replacing it by a fresh variable and its definition.
Let $abs = \lambda x_1 \ldots x_n.\ t$.
Let g be a fresh variable in f.
The lambda-lifting conversion $\mathcal{L}$ is defined by:

$$f[abs] \longrightarrow (!x_1 \ldots x_n.\ g\ x_1 \ldots x_n = t) \Rightarrow f[abs := g]$$

This conversion is repeated for every abstraction in a repeated top-down approach.

Remark. There are different ways to deals with remaining lambda-abstraction after beta-reduction. Combinators could also be used as in METIS_TAC and in [ref].

Example. (Code)

6.8 Boolean argument conversion

Remove boolean arguments, except $true$ and $false$.
Let t be an operand(argument) of type *bool* (see remark below).
The boolean argument conversion $\mathcal{B}_a$ is defined by:

$$\mathcal{B}_a(f[t]) = (t \Rightarrow f[t := true]) \wedge (\neg t \Rightarrow f[t := false])$$

This conversion is repeated for every boolean arguments in a repeated top-down approach. If a boolean argument is bound, we goes down the formula tree till we find a formula where it is free and do the conversion here.

Code test:

Remark. We need to be careful not to consider $\wedge$ as a function symbol in $A \wedge B$. To avoid this, we start our research for boolean arguments from the atom sets and take the first we find.

Code test:

6.9 Numeral variable conversion

Since Beagle doesnt have a numeral type but an integer type, numeral variables should be told that they are positive.
Let n be numeral variable in the formula.
The numeral variable conversion $\mathcal{N}_v$ is defined by:

$$\begin{aligned}\mathcal{N}_v(f[n]) &= 0 \leq n \wedge f\\ \mathcal{N}_v(\forall n.f) &= 0 \leq n \Rightarrow f\\ \mathcal{N}_v(\exists n.f) &= 0 \leq n \wedge f\end{aligned}$$

This conversion is repeated for every numeral variables. Code test:

6.10 Higher order conversion

Let x be a variable, a constant or a number.
Let t_1, t_2 be two terms.
Let c_a be any arithmetic constant.
Let *App* be a fresh variable in the overall formula,that verifies $\mathit{App}xy = xy$.
The higher order conversion $\mathcal{H}$ is recursively defined on atoms by:

$$\begin{aligned}\mathcal{H}(x) &= x\\ \mathcal{H}((c_a t)t_2) &= (c_a\mathcal{H}(t))\mathcal{H}(t_2)\\ \mathcal{H}(t_1t_2) &= \mathit{App}\ \mathcal{H}(t_1)\mathcal{H}(t_2)\end{aligned}$$

Code test:

6.11 Numeral function axiom

Let f be a free variable in a formula P,
Let n be the arity at which f is used in P.

if f is used as a function that returns a numeral, then the numeral function axiom is:

$$\forall x_1 \dots x_n.\ (0 \leq x_1 \wedge \dots \wedge 0 \leq x_n) \Rightarrow (0 \leq f x_1 \dots x_n)$$

For each numeral function, the numeral function axiom is added to the hypothesis. Code test:

6.12 Boolean bound variables conversion

6.13 Printing

The TFF file created by the program tries to be user-friendly. It keeps most of the variable names provided by the user and gives information about the types.

6.13.1 Variables

Variables are translated using three injective dictionaries, one for bound variables, one for free variables and one for constants.

The constants dictionary is not used for arithmetic constants such as (=, +, ...) because they need to be translated in a special way since they are expected to match with the TFF constants (=, +, ...).

6.13.2 Types

Definition. (Inductive mapping)
Let us define a recursively a mapping $\mathcal{F}$: HOL4 types $\times\ \mathbb{N} \to$ TFF types. The number $n \in \mathbb{N}$ represents the arity at which a variable of this type is used.
if the arity is 0:

$$\begin{aligned}
\mathcal{F}(ty) &= ty \\
\mathcal{F}((fun, [ty_1, ty_2]), 0) &= \mathcal{F}(ty_1, 0)\_F\_\mathcal{F}(ty_2, 0) \\
\mathcal{F}((prod, [ty1, ty2]), 0) &= \mathcal{F}(ty_1, 0)\_P\_\mathcal{F}(ty_2, 0) \\
\mathbb{F}((op, [ty_1, \ldots, ty_n]), 0) &= opI\mathbb{F}(ty1, 0)\_\ldots\_\mathbb{F}(ty_n, 0)I
\end{aligned}$$

if the arity is $a > 0$, $m = a - 1$ and $type = ty_1 \to ty_2 \to \ldots \to ty_a$:

$$\mathcal{F}(type) = (\mathcal{F}(ty_1, 0) * \mathcal{F}(ty_2, 0) * \ldots * \mathcal{F}(ty_m, 0)) > \mathcal{F}(ty_a, 0)$$

Example. $\mathcal{F}((a \to b) \to c \to d, 2) = (a\_F\_b * c) > d$

Remark. This translation may not be injective, but we force injectivity by numbering the TFF type if it was already used.

6.13.3 Non linear integer arithmetic

To prevent non linear arithmetic from happening (because Beagle doesnt support non linear arithmetic), non linear terms are translated using a fresh constant for multiplication instead of $\times$.

6.14 Soundness and non-completeness of the HOL4 translation

Statement. *(Soundness)*
The code quarantees that if we prove the last HOL4 representation of the problem then we prove the problem in its original form.

Remark. The printing part wont be sound (in the previous sense) till we create the proof replayer.

Statement. *(Non completeness)*
The translation is not complete meaning that if Beagle cant prove because of the non-completeness of the substitutions composition loop [ref]. part obviously the number of possible instantiation maybe infinite. Lambda - lifting is incomplete as shown in this paper. The higher order conversion doesnt pass the definition of the App operator see section ... to Beagle since it has higher order. The printing part is not controlled what so ever.

7 Beagle proof output

A complete step by step TFF derivation (direct acyclic graph) would be a nice output. But an other idea would be to print only relevant steps which would lead to a smaller proof and counter intuitively an easier proof reconstruction as discussed in the next section.

8 HOL4 proof replayer

Once we get a reasonable proof output from Beagle, a HOL4 parser may be created to get a HOL4 representation of the proof.
Then, the first idea is to simulate each steps of the proof by a rule. For example, the Split rule may be simulated by the case disjunction rule. But these may leads to very long proof, and will be very dependent on the development of Beagle.
An other idea would be to create an all purpose HOL4 rule such as metis in SledgeHammer for HOL/Isabelle [ref]. Then use only important lemmas of the Beagle proof as a guide to reconstruct the proof. Although it would be slower than a direct reconstruction, the flexibility gain could outweigh the cost.

9 Results

Our code was tested on 297 problems that METIS_TAC (a HOL4 internal prover) solves when building HOL4.
The flags just below were set up to see which part of the code is used. We can see that it is evenly distributed so each part of the code was really necessary.

```
Polymorph   : 147
Lambda-lift : 73
Boolean     : 85
Numeral     : 172
Higher Order: 99
```

This is the status Beagle returns for each problems.

```
Unsatisfiable : 241
Unkown        : 15
Satisfiable   : 2
Time out      : 23
Parsing error : 11
```

The missing 1 problem is a code errors that occurs in the HOL4 translation that prevents the program from generating a TFF(TPTP) file.

10 Future works

Lets summarize here what could be done to complete and improve the HOL4-Beagle interaction.

1. Improving the translation in HOL4.

   Automatically generated theorems from HOL4 theories. (based on some heuristic which would involved the number of commons constants) [ref]

   Direct translation for integers, rationals and reals since Beagle supports them also. (Only numerals are currently supported)

2. Improving Beagle.

   Improving the proof output so that it keep tracks of parents and rules used for each step.

   Creation of a counter-examples from a saturated clause set if the clause set is satisfiable.

3. Creation of a HOL4 proof replayer.

11 Acknowledgements

References

[1] Kananaskis-7-description.pdf

[2] Kananaskis-8-tutorial.pdf

[3] Kananaskis-5-logic.pdf

[4] Hurd, ”Gandalf” TPHOLs 1999

[5] Paulson, Computer Laboratory, University of Cambridge, U.K., Meng, National ICT, Australia, ”Automation for Interactive Proof”, ”Lightweight relevance filtering”, IJCAR 2004,

[6] Paulson, Computer Laboratory, University of Cambridge, U.K., Meng, National ICT, Australia, ”Translating Higher-Order Clauses to First-OrderClauses”, 2007

[7] Cezary Kaliszyk, Josef Urban, ”Learning-Assisted Automated Reasoning with Flyspeck”, 2013

[8] Sascha Böhme, ”Proving Theorems of Higher-Order Logic with SMT Solvers”, 2012

[9] Jasmin Christian Blanchette, ”Automatic Proofs and Refutations for Higher-Order Logic”, 2012

[10] Jasmin Christian Blanchette, Sascha Böhme, Andrei Popescu, Nicholas Smallbone, ”Encoding Monomorphic and Polymorphic Types”

[11] Jasmin Christian Blanchette1, Andrei Paskevich, ”TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism”