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

Thibault Gauthier

September 13, 2013

Abstract

The interactive theorem prover HOL4 is interfaced with an external automated theorem prover Beagle. The necessary work involves the implementation of a sound translation from the polymorphic higher-order logic of HOL4 to the monomorphic first-order logic of Beagle.

Performance of the HOL4-Beagle system is evaluated against problems solved by the internal prover METIS_TAC during HOL4 build. It is shown that Beagle proves 80% of 271 translated theorems in 15 seconds of real time on one CPU.

.

Contents

1 Introduction

1.1 Context

Interactive theorem provers are used by researchers for certifying proofs of problems such as the Kepler conjecture and modelling complex algorithms or systems. The main weakness of these tools is that they need a lot of human guidance. To solve this problem, internal automated methods were created but they are usually limited to specific problems. Therefore, it is a natural thing to enhance them with automated theorem provers which tackle a wide range of first-order problems automatically.

1.2 Interactive provers and automated provers comparison

We present here the strength and the weakness of both these tools.

	Interactive theorem provers	Automated theorem provers
Provers	HOL4, HOL/Isabelle, HOL light, Coq, ...	Beagle, SPASS, Vampire, E-prover, E-Darwin, ...
Input	Mathematical problems directly match their higher-order logic format.	Mathematical problems need to be translated to their first-order logic format.
Efficiency	Need human guidance. Have internal automated procedures but are very specialized.	Automatically prove a wide range of first-order problems.
Reliability	Are based on easily verifiable small kernel.	Proving their relatively long code is a bit harder and has to be repeated for each new version. Another approach is to check their proof output ”a posteriori”.

Remark. It is worth to mention that there exists other kind of automated theorem provers such as SMT solvers.

1.3 General problem

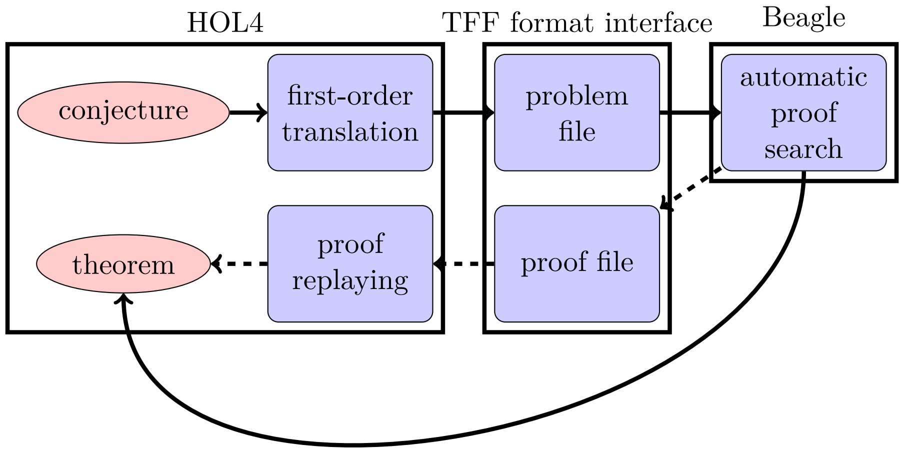
1.3.1 Problem statement

Problem:
HOL4 is an interactive theorem prover. It has an internal first-order prover METIS_TAC. But this internal prover does not deal with arithmetic without user-provided theorems.

Solution:
A new tactic called BEAGLE_TAC is created. It calls an external prover Beagle that was designed to cope with arithmetic problems.

1.3.2 The HOL4-Beagle interaction diagram

This is how BEAGLE_TAC globally works.



Remark. The plain arrows are currently implemented, so BEAGLE_TAC can only be used as an oracle. The first-order translation was proved, thus the conjecture is true if Beagle returns the conjecture as a theorem and if you believe that the printing part (second arrow) and Beagle are sound.

Remark. 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 created by Beagle (work in progress) could be tested within any other TFF proof checker.

1.4 Structure of this report

Up to Section 4, we present the different tools we used in our project. The Section 5 (Translation) explains the main part of our project. Then, we put the problem into perspective in Section 6,7,8.

2 The TFF(TPTP) format

The TPTP website (Thousands Problems for Theorem Provers) (`http://www.cs.miami.edu/~tptp/`) provides a database for theorem provers to be tested against. A TPTP format is human-readable and non specific prover dependent.
The syntax of a problem can be checked online there:
`http://www.cs.miami.edu/~tptp/cgi-bin/SystemB4TPTP`.
Some automated theorem provers can be called on any problem there:
`http://www.cs.miami.edu/~tptp/cgi-bin/SystemOnTPTP`.

2.1 TFF format

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

Definition. (TFF type)
A TFF type is inductively defined by :

$$type, ty_1, \ldots, ty_n, \ldots := d \mid u \mid (ty_1 * \ldots * ty_m) > type$$

where d is a defined type and u a user-defined type.

3 The HOL4 interactive theorem prover

3.1 General ideas

HOL4 is an interactive theorem prover that deals with higher-order logic, which is close to the language mathematicians use to express their formulas. It is based on the functional programming and type checking language SML. To learn more about programming in HOL4, you can read the Kananaskis documentation.

3.2 HOL4 type and formula representation

Definition. (HOL4 types)
The set of all types is inductively defined by:

$$type, ty_1, \ldots, ty_n, \ldots := b \mid p \mid (op, [ty_1, \ldots, ty_m])$$

where b is a basic type, p is a polymorphic type and op is a type operator.

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)
This definition is based on the typed λ-calculus:

$$f, g := c \mid v \mid n \mid (g)\ f \mid \lambda v.f$$

where c is a constant, v is a variable and n a positive integer.
There are type constraints when building $(g)\ f$.

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

Example. $\forall x,\ x = 0$ is represented in HOL4 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 ($rule = thm \rightarrow thm$)
   Using a rule is the most common way to create theorems. Rules can be used to simulate steps in a derivation.

5. Conversion ($conv = term \rightarrow thm$)
   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 ($tactic = goal \rightarrow goal\ list * (thm\ list \rightarrow thm)$)
   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 use a type in the intended way or not.

3.4 Soundness

Since *thm* is an SML abstract type, only reserved functions can return the *thm* type. The number of these functions is small and all these functions are simple enough, so they can be checked by a human. Then, new variables of type *thm* can only be created by one of these functions. That is why every new variable of type *thm* is a theorem.

3.5 One easy proof

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

$$\dfrac{\dfrac{\dfrac{A \vdash A \qquad B \vdash B}{A, B \vdash A \wedge B}\wedge_i}{A \vdash B \Rightarrow (A \wedge B)}\Rightarrow_i}{\vdash A \Rightarrow (B \Rightarrow (A \wedge B))}\Rightarrow_i$$

```
(* forward proof *)
val th1 = ASSUME ``A:bool``;
val th2 = ASSUME ``B:bool``;
val th3 = CONJ th1 th2;
val th4 = DISCH ``B:bool`` th3;
val th5 = DISCH ``A:bool`` th4;
```

```
(* backward proof (using the interactive goal stack) *)
g(`A ==> B ==> A /\ B `);
e(DISCH_TAC);
e(DISCH_TAC);
e(CONJ_TAC);
e(ACCEPT_TAC th1);is
e(ACCEPT_TAC th2);
```

3.6 Derivation format

You can use the TFF format to encode derivations. For more information, look at `http://www.cs.miami.edu/~tptp/TPTP/QuickGuide/Derivations.html`.is

4 The Beagle automated theorem prover

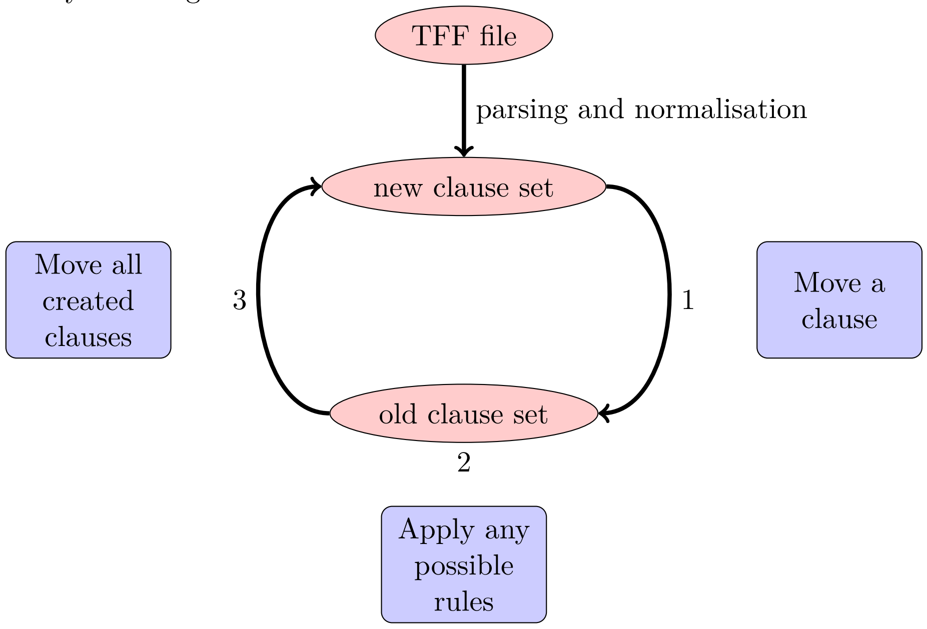
Beagle is an automated theorem prover that deals with typed first-order arithmetic formulas. It translates the problem into a clause set (e.g. $\{x \vee g(x), f(y) \vee \neg x\}$).
It returns that the clause set is either unsatisfiable, satisfiable or unknown.

Remark. To look deeper in the Beagle calculus, which contains rules such as superposition, you can read [1].

4.1 Derivation search in Beagle

4.1.1 Saturation loop (no Split)

Beagle uses two clause sets, the new clause set (NCS) and the old clause set (OCS). The OCS is initially empty. The NCS is initially the one created by the normalisation step.
Let us 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 have to check the subsets that contain the moving clause.
3. Move all created consequences into the new clause set.

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

4.1.2 A special rule: Split

Definition. (saturation loop state) Let us represent a saturation loop state by a triplet (c, OCS ,NCS) where c 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.

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

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

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

4.1.3 Termination

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

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

2. If there exists a saturated clause set which does not contains any contradiction:

   If the initial clause set was sufficiently complete, then the status of this problem is **satisfiable**.

   If the initial 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)

5 HOL4 problem translation

5.1 Code location

The code for this translation is located at `https://github.com/barakeel/HOLtoTFF`. Some examples can be found in paperexample.sml.

5.2 Notation

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

5.3 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 these conditions:

- Its operator is not a logical operator.
- For each term that includes this term, his operator is a logical operator.

5.4 Translation order

The translation order was thought carefully. It happens in this order:

1. Monomorphisation. (may be done at anytime but it happens first to improve efficiency [9].)
2. Goal negation.
   The goal is rewritten to ([hypotheses, conclusion negation],false). Then a conjunction of the new goal and the theorem list returns a single term.
3. CNF conversion.
4. λ-lifting conversion then CNF conversion.
5. Boolean arguments conversion then CNF conversion.
   This step is necessary because the TFF format does not accept boolean variables as arguments.
6. Numeral variables conversion then CNF conversion.
7. Clause set tactic.
   Transform the term into a clause set.
8. Higher-order conversion.
   This conversion occurs almost at the end of the translation, it allows us to limit the number of problems that are considered higher-order.(e.g. $\exists f.\ \forall x.\ f\ x = 0$ is not a higher-order formula.)
9. Numeral functions axioms.
   It has to be done after the higher-order conversion because the *App* operator produced by the higher-order conversion may be a numeral function.
10. Boolean bound variables conversion.
    This step is necessary because the TFF format does not accept quantified boolean variables.
11. Printing.

    Variables or constants.

Types.

True and false constants.

Non-linear integer arithmetic.

5.5 Monomorphisation

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

Example. if we are given a theorem $!x : a.\ C\ x\ x$ and a goal $C\ 42\ 42$, it is ”clever” to instantiate the theorem with the substitution $[a \mapsto \mathit{num}]$. One way of doing this is to find a substitution that matches the type of $C : a \to a \to \mathit{bool}$ to the type of $C : \mathit{num} \to \mathit{num} \to \mathit{bool}$ and apply this substitution to the theorem.

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

5.5.1 Dependency graph

In this section, we will present a theoretical representation of our monomorphisation implementation.

For every problem, a dependency graph can be defined.

Definition. (Dependency graph)

- Nodes are constants in the theorem list or in the goal. Two same constants are represented by different nodes if they are in different theorems or one is in the goal and the other is not.

- Substitution directed links connect two different constants that verify this properties:

  The initial one does not belong to the goal.

  They have the same name.

  The type of the destination is a type instance of the type of the origin.

- Binding undirected links connect two constants that verify this properties:

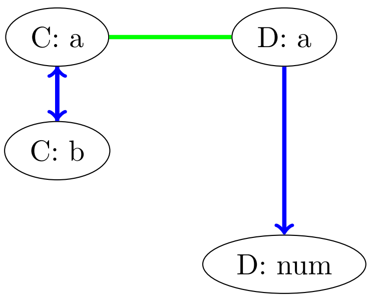
  They are from the same theorem.

  They share a common polymorphic type.

Each substitution links infers a substitution. When there is no ambiguity, we will identify the link and the substitution.

Remark. When drawing a dependency graph or its supergraph, we will draw constants from the same theorem (or goal) in the same row.

Example of a dependency graph.

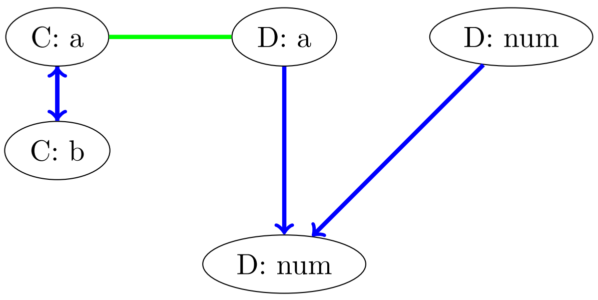


Definition. (A substitution transformation) The substitution transformation $\mathcal{T}_s$ of a dependency graph G for a substitution link σ from c to c', is defined by:

$$\mathcal{T}_s(G) = G \cup \sigma(c)$$

where the created image stays in the same theorem.

New dependency graph using the right substitution link as a substitution transformation.

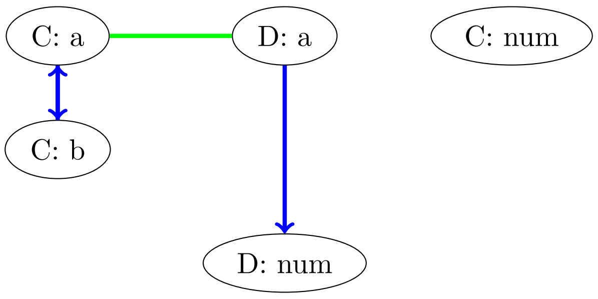


Definition. (A binding transformation) The binding transformation $\mathcal{T}_b$ of a graph G, for a binding link from c to c' and a substitution link σ from c' to c'', is defined by:

$$\mathcal{T}_b(G) = G \cup \sigma(c)$$

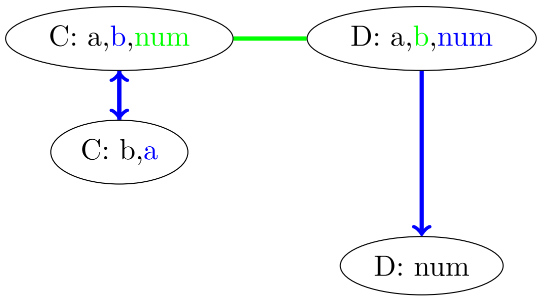
where the created image stays in the same theorem.

New dependency graph using the binding link and the right substitution link as a binding transformation.



Definition. (A graph transformation) The graph transformation $\mathcal{T}$ is defined by applying all possible substitution transformations and binding transformations simultaneously to a graph G.

New supergraph using the graph transformation

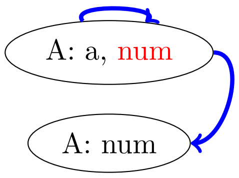


Definition. (Supergraph of a dependency graph) Let G be a dependency and G_i the dependency graph created after i transformation $\mathcal{T}$. The supergraph SG_i of G_i is defined by:

- Nodes are sets composed of a node in the initial graph G (head node) and all its derived nodes in G_i (tail nodes). They will be called a supernodes.
- Substitution undirected links connect two nodes sn_1 and sn_2 when there is at least one substitution link, between a constant in sn_1 and a constant in sn_2, in G_i.
- Binding undirected links connect two nodes sn_1 and sn_2 when there is at least one binding link , between a constant in sn_1 and a constant in sn_2, in G_i.

Definition. (Redundancy)
A substitution link is redundant if it has the same origin and induce the same substitution as a substitution link inside the supernode of its origin. From this point, we will not draw redundant substitution link.

Supergraph at step 1 with a redundant substitution link.
A substitution link inside the super node is exceptionally represented.

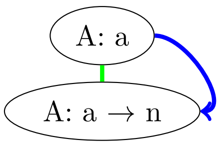


Definition. (Directable)
A graph G is directable if every binding link b can be oriented, i.e. these two things does not happen:
- The two nodes that b connects are origins of substitution links.
- The two nodes that b connects are destinations of substitution links.

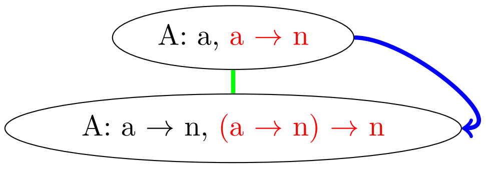
Definition. (Partial order induced by a directable graph)
Let G be a directable dependency graph with no loop. We will say that a node n_1 is greater than another node n_2, if there is a path from n_1 to n_2. This order induces an order over supernodes considering only their head.

Theorem. *(Loop theorem)*
There exists a dependency graph G for which the transformation $\mathcal{T}$ does not find a fix point.

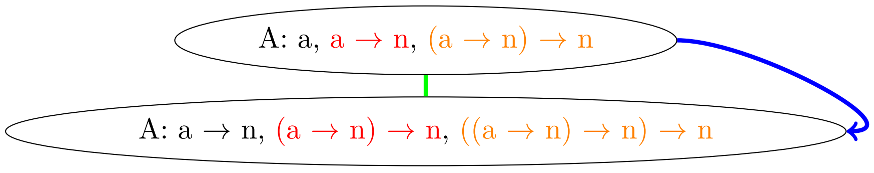
Initial supergraph.



Step 1.



Step 2.



Proof. (Idea) Let us take the previous example and induct on the number of arrows in (a → ... n) → n. ☐

Theorem. *(Conjecture)*
let $G = SG$ be a directable graph.
Let p the length of the longest path in G. (Dependency links have length 0)
Let $G_p = \mathcal{T}^g(G)$.
G_p is a fix point if:
- G_p has no loop.
- SG_p has no substitutions arrows.

Theorem. *(Conjecture)*
let SG=G be a directable graph.
Let p the length of the longest path in G. (Dependency links have length 0)
Let $G_p = \mathcal{T}^g(G)$.
If G_p is not a fix point then forall $i > 0$, G_i is not a fix point.

Conclusion.
From each constant in a theorem, a list of substitutions can be extracted from the dependency graph. They are combined in every possible way to create a list of substitutions for the theorem. Eventually, each theorem in the problem is instantiated with all its possible substitutions.

5.5.2 A practical dependency graph example

This example comes from one of the many problems we tested.
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)
```

Type of each constant in this problem.

Theorem 1:

IN : $c \to (c \to bool) \to bool$

INSERT : $c \to (c \to bool) \to c \to bool$

$\emptyset$: $c \to bool$

Theorem 2:

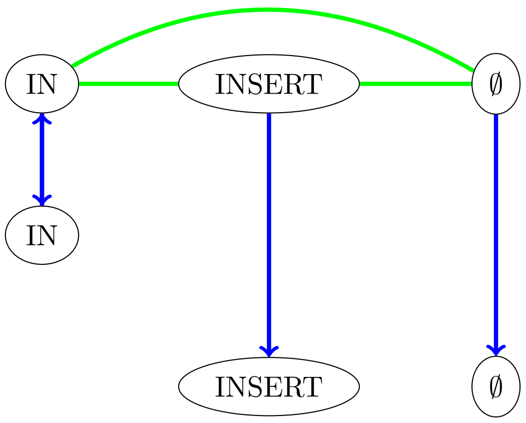
IN : $a \to (a \to bool) \to bool$

Goal

INSERT : $num \to (num \to bool) \to num \to bool$

$\emptyset$: $num \to bool$

Dependency graph.



Remark. If we apply the graph transformation $\mathcal{T}$ twice, the graph becomes a fix point. This is not surprising if we considered its longest path to have length 2.

5.6 Normalisation

The code uses an already defined function called CNF_CONV, 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.

5.7 λ-lifting conversion

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

$$\mathcal{L}(f[abs]) = (!x_1 \dots x_n.\ g\ x_1 \dots x_n = t) \Rightarrow f[abs := g]$$

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

```
Example:
fun_conv ``P (\x y. x = f):bool``;
val it = |- P (\x y. x = f) <=> ?f1. (!x y. f1 x y <=> (x = f)) /\ P f1
```

5.8 Boolean arguments conversion

The reason we do this conversion is that in the TFF format, boolean type is reserved for predicates. This transformation removes 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 go down the formula tree till we find a formula where it is free and repeat the conversion there.

```
Example:
bool_conv ``P (!x. P (z:bool) /\ x):bool``;
val it =
|- P (!x. P z /\ x) <=>
(z ==> ((!x. P T /\ x) ==> P T) /\ (~(!x. P T /\ x) ==> P F)) /\
(~ z ==> ((!x. P F /\ x) ==> P T) /\ (~(!x. P F /\ x) ==> P F))
```

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.

5.9 Numeral variables conversion

Since Beagle does not have a numeral type but an integer type, every numeral variable should be told that they are positive.
Let n be a numeral variable in the formula.
The numeral variable conversion $\mathcal{N}_v$ is defined by:

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

This conversion is repeated for every numeral variables.

```
Example:
num_conv ``!x. ?y. x - y = 0``;
val it =
   |- (!x. ?y. x - y = 0) <=> !x. (0 <= x) ==> ?y. (0 <= y) /\ (x - y = 0)
```

5.10 Higher-order conversion

This translation converts every function (variable or constant used with a strictly positive arity) to first-order variable (variable or constant used with arity 0). This translation is applied as soon as there is a higher-order term (a quantified function or a function used with different arities) in the formula .

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 that verifies $\mathit{App}\ x\ y = x\ y$.
The higher-order conversion $\mathcal{H}$ is recursively defined on atoms by:

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

```
Example:
app_conv "App" ``!f. f x = g x y + f y``;
val it =
    [!f x. App f x = f x, !f x. App f x = f x]
|- (!f. f x = g x y + f y) <=>
!f. App f x = App (App g x) y + App f y
```

Remark. Arithmetic constants are left as functions so that Beagle can use them. It makes this conversion incomplete because it prevents them to be instantiated in a universally quantified formula.

5.11 Numeral functions axioms

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 its associated 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.

5.12 Boolean bound variables conversion

The translation $\mathcal{B}_v$ remove boolean bound variables by instantiated them.

$$\mathcal{B}_v(\forall x : \mathit{bool}.f[x]) = (x \Rightarrow f[T]) \wedge (\neg x \Rightarrow f[F])$$

5.13 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.

5.13.1 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 the TFF constants (=, +, ...).

5.13.2 Types

Definition. (Inductive mapping)
Let us define recursively a mapping $\mathcal{F}$: HOL4 types $\times\,\mathbb{N} \rightarrow$ 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:

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

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

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

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

Remark. One goal of this translation was to preserve the type information in a string, so that the user can easily recognized where it comes from. This translation may not be injective, but we force injectivity by numbering the TFF type if it was already used.

5.13.3 True and false constants

The TFF format does not accept quantified boolean variables and boolean variables as arguments, so we need to be carefully when translating our last boolean arguments. If true(respectively false) appears as an argument, it is printed $btrue$(respectively $bfalse$) and the axiom $btrue \neq bfalse$ is added. The TFF variable $btrue$ and $bfalse$ will have a new type ”bool” which is different from the defined type ”\$o” for TFF booleans because
If true(respectively false) appears as a predicate, it is printed as a defined TFF constant $\$true$ (respectively $\$false$).

5.13.4 Non linear integer arithmetic

To prevent non linear arithmetic from happening (because Beagle does not support non linear arithmetic), non linear terms are translated using a fresh constant for multiplication instead of $\times$.

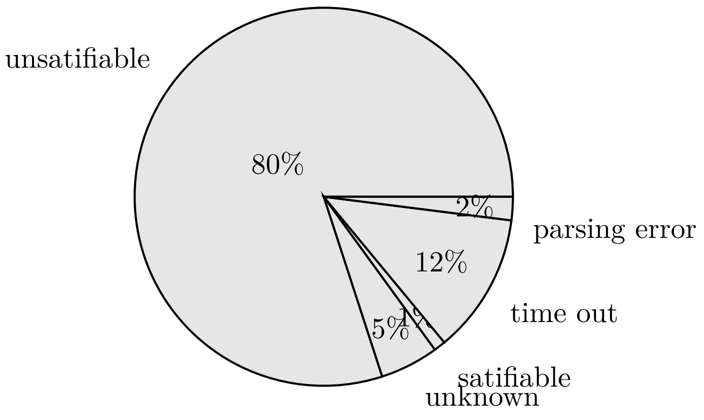
5.14 Soundness and non-completeness of the HOL4 translation

Statement. *(Soundness)*
If the last HOL4 representation of the problem is proved then the problem in its original form is proved.

Statement. *(Non-completeness)*
The translation is not complete for at least two reasons:
- Our monomorphisation is not complete. In particular, the monomorphisation may not reach a fix point. If it reaches a fix point, the type information may not pass through constants only.
- Our higher conversion is not complete (arithmetic constants are not converted).

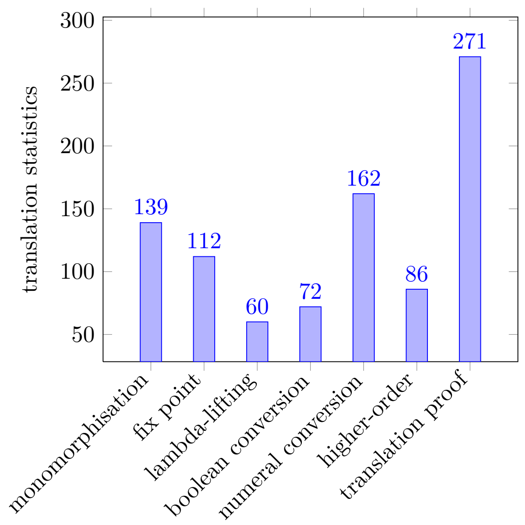
6 Results

Our code was tested on 271 problems that METIS_TAC solves when building HOL4. This is how BEAGLE_TAC performs on general problems.



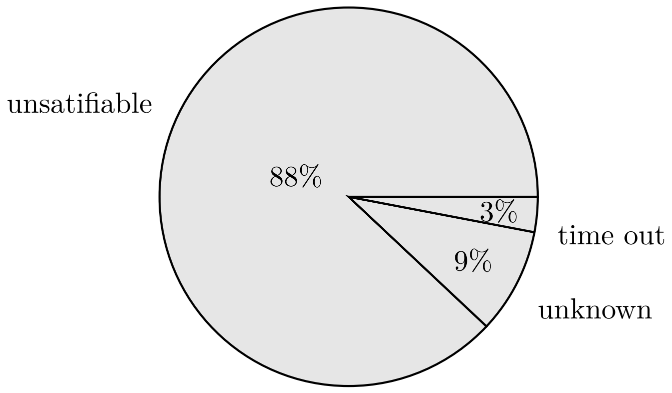
An unsatisfiable result means the conjecture was proved since our starting strategy was to negate the goal.
A parsing error or a time out result usually means that the problem was too complex for Beagle to solve.
A satisfiable or unknown result usually means that our translation didn't give enough information for Beagle.

This diagram represents which part of the code was used and how successful it was.

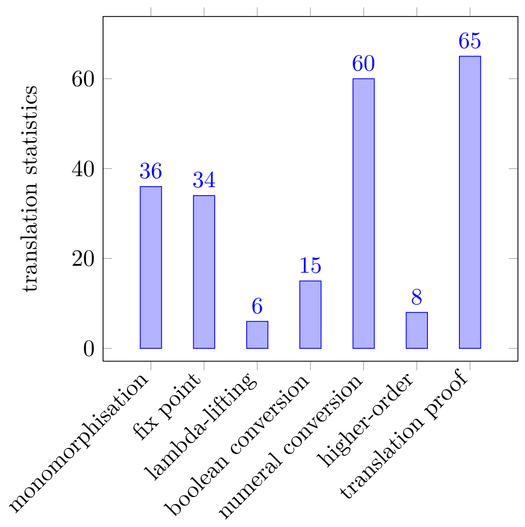


Every conversion (λ-lifting, boolean conversion, numeral conversion and higher-order conversion) is used more than 1 time out of 5, so the code has been tested with a good variety of problems.
The second column shows that 80 percent of the monomorphisation steps reaches a fix point.
The last column shows that the translation inside HOL4 was proven 100 percent of the time.

To test BEAGLE_TAC's ability to deal with arithmetic without user-provided theorems, we searched through all the problems and removed used-provided arithmetic theorems and see if BEAGLE_TAC could prove them whereas METIS_TAC cannot.
There were 65 problems involving arithmetic. All the 79 user-provided arithmetic theorems were removed. This is how BEAGLE_TAC performs on these arithmetic problems.



These are the characteristics of these arithmetic problems.



7 Contribution and related works

7.1 Contribution

BEAGLE_TAC will be added to the HOL4 library as a TFF converter and and a oracle. It will enables HOL4 user to omit giving arithmetic theorems to ”prove” arithmetic goals.

7.2 Related works

Here are four different researchers who implemented higher-order to first-order translations:

- Joe Hurd developed a translation [5] in his internal prover METIS_TAC. It encodes type variables as first-order logic variables and uses combinators to remove λ-abstractions.

- Paulson and Meng implemented three different higher-order to first-order translations [9] from HOL/Isabelle to the logic of first-order prover. One of those involves forgetting about the typing information. It leads to an increase of efficiency and a lost of soundness. A ”a posterirori” check of the proof is necessary.

- Sascha Böhme wrote a translation [4] from HOL/Isabelle's higher-order logic to the first-order logic of SMT solvers which is a monomorphic typed first-order logic. It performs a monomorphisation step by finding a fix point in a set of special terms. It uses λ-lifting. It is the translation that has the most similarity with the one we explained in this report.

- Kaliszyk and Urban developed an automatic tool [6] based on HOL light and the Flyspeck project. It includes implementations of sound translations of the HOL Light logic to ATP formalisms: untyped first-order, polymorphic typed first-order, and typed higher-order.

Comparatively, our translation outputs monomorphic typed first-order logic formulas in a clause set form. It performs monomorphisation by finding a fix point in a set of constants. It uses lambda-lifting and its higher-order conversion is tweaked to deal with arithmetics. Boolean arguments are instantiated. Non linear-arithmetic terms are treated in a special way as in [4].

Remark. To see many different ways of dealing with polymorphic types, you can read [3].

Here are two projects which make use of higher-order to first-order conversion and give ideas how to enhance HOL4 further more:

- Chantal Keller developed a cooperation between the interactive theorem prover Coq and SAT/SMT solvers [7].

- Jasmin Blanchette developed a tool to generate counterexamples [2].

8 Future works

We present here what could be done to complete and improve the HOL4-Beagle interaction.

1. Improving the translation in HOL4:
   - A first improvement will be to generate theorems from HOL4 theories. It would be based on some heuristic which would involve the number of common constants between the problem and the generated theorem [8].
   - A second improvement will be to translate integers, rationals and reals since Beagle supports them.
   - A third improvement will be to do a less complete but more efficient higher-order conversion by not converting functions that always appear with the same arity.

2. Improving the proof output of Beagle:
   A complete step by step TFF derivation (direct acyclic graph) would be a nice output. But another 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.

3. Creation of a HOL4 proof replayer:
   - Once Beagle creates a reasonable proof output, 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.
Another idea would be to create an all purpose HOL4 rule such as metis in SledgeHammer for HOL/Isabelle [8]. Only important lemmas will be used to guide the reconstruction of the proof. Although it would be slower than a direct reconstruction, the flexibility gain could outweigh the cost.

9 Acknowledgements

During the 3 months and half I spent at NICTA, everyone I met gave me a very warm welcome. My supervisor, Michael Norrish, always led me to a good approach to the problem. Peter Baumgartner and his student Josh Bax spent time explaining me how Beagle works and reviewing some of my errors. I am really thankful to them. Chantal Keller, my supervisor in France, put my problem into perspective and guided me to write my report. It helped me a lot.

References

[1] Peter Baumgartner and Uwe Waldmann. Hierarchic superposition with weak abstraction. Research Report MPI-I-2013-RG1-002, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, June 2013.

[2] Jasmin Blanchette. *Automatic Proofs and Refutations for Higher-order Logic*. PhD thesis, 2012.

[3] Jasmin Christian Blanchette, Sascha Böhme, Andrei Popescu, and Nicholas Smallbone. Encoding monomorphic and polymorphic types. In Nir Piterman and Scott A. Smolka, editors, *TACAS*, Lecture Notes in Computer Science, pages 493–507. Springer, 2013.

[4] Sascha Böhme. *Proving Theorems of Higher-Order Logic with SMT Solvers*. PhD thesis, Technische Universität München, 2012.

[5] Joe Hurd. First-order proof tactics in higher-order logic theorem provers. 2003.

[6] Cezary Kaliszyk and Josef Urban. Learning-assisted automated reasoning with flyspeck. *CoRR*, abs/1211.7012, 2012.

[7] C. Keller. *A Matter of Trust: Skeptical Communication Between Coq and External Provers*. PhD thesis, École Polytechnique, June 2013.

[8] Jasmin C. Blanchette Lawrence C. Paulson. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In S. Schulz G. Sutcliffe, E. Ternovska, editor, *IWIL-2010*, 2010.

[9] Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *J. Autom. Reasoning*, 40(1):35–60, 2008.