Scalable LCF-style proof translation

Cezary Kaliszyk[1] and Alexander Krauss[2]

[1] University of Innsbruck
cezary.kaliszyk@uibk.ac.at
[2] QAware GmbH
krauss@in.tum.de

Abstract. All existing translations between proof assistants have been notoriously sluggy, resource-demanding, and do not scale to large developments, which has lead to the general perception that the whole approach is probably not practical. We aim to show that the observed inefficiencies are not inherent, but merely a deficiency of the existing implementations. We do so by providing a new implementation of a theory import from HOL Light to Isabelle/HOL, which achieves decent performance and scalability mostly by avoiding the mistakes of the past. After some preprocessing, our tool can import large HOL Light developments faster than HOL Light processes them. Our main target and motivation is the Flyspeck development, which can be imported in a few hours on commodity hardware. We also provide mappings for most basic types present in the developments including lists, integers and real numbers. This papers outlines some design considerations and presents a few of our extensive measurements, which reveal interesting insights in the low-level structure of larger proof developments.

1 Introduction

The ability to exchange formal developments between different proof assistants has been on the wishlist of users for as long as these systems exist.

Most systems are designed in such a way that their proofs can, in principle, be exported into a form that could be checked or imported by some other system. Implementations of such proof translations exist between various pairs of systems. However, they all have in common that they are very expensive in terms of both runtime and memory requirements, which makes their use impractical for anything but toy examples.

For instance, Obua and Skalberg [12] describe a translation from HOL Light and HOL4 to Isabelle/HOL. Their tool is capable of replaying the HOL Light standard library, but this takes several hours (on 2010 hardware). Similarly, Keller and Werner [10] import HOL Light proofs in Coq and report that the standard library requires ten hours to load in Coq. Note that the standard library takes less than two minutes to load in plain HOL Light, so there is roughly a factor 300 of overhead involved. Other descriptions of similar translations report comparable overhead [14,11].

So should we conclude that this sort of blowup is inherent in the approach and that proof translations must necessarily require lots of memory and patience? This paper aims to refute this common belief and show that the bad performance is merely a deficiency of the existing implementations. More specifically, we make the following contributions:

- We describe a new implementation of a proof translation from HOL Light into Isabelle/HOL, which performs much better than previous tools: After some preprocessing, the HOL Light standard library can be imported into Isabelle/HOL in 30 seconds, which reduces the overhead factor from 300 down to about 0.4.
- Large developments are routinely handled by our tool, such that we can import major parts of the formalized proof of the Kepler Conjecture, developed in the Flyspeck [3] project on normal hardware. Here, the overhead factor is a bit larger (with optimizations a factor of roughly 1.2).
- By providing better mappings of HOL Light concepts to Isabelle concepts, we obtain more natural results; in particular little effort is needed to map compatible but differently defined structures like integers and real numbers.
- We present various results obtained during our measurements, which yield some empirical insights about the low-level structure of larger formal developments.

Our work shares some visions with Hurd's OpenTheory project [6] but has a slightly different focus:

- Our translation is able to work directly on the sources of any HOL Light development, without requiring any modification, such as adding special proof recording commands. This is crucial when dealing with large developments, where such modifications would create significant work and versioning problems.
  In contrast, the OpenTheory setup still needs manual arrangements to the sources, which hinders its use out of the box. Therefore, it is also hard to assess its scalability to developments of Flyspeck's size.
- We do not focus on creating small, independent, reusable packages. Instead, our approach assumes a large development, which is treated as a whole. By default, all definitions and top-level theorems are converted, but filtering to a subset is easy.

This paper is structured as follows. In Section 2, we give an overview of the architecture of our translation. In Section 3 we discuss memoization strategies that allow reducing the time and memory requirements of the translation processes. In Section 4 we present the statistics of inference steps involved in the translation and interesting statistics about HOL Light and Flyspeck discovered using our experiments and finally we conclude in Section 5 and present some outlook on the future work.

2 Architecture overview

In this section we explain the four basic steps in our proof translation: the collection of theorems that need to be exported; the instrumented kernel used to export the inference steps performed in HOL Light; the offline proof processor, and the importer itself. The four components are presented schematically in Figure 1.

2.1 Collecting Theorems to Export

The first issue in implementing a proof translation mechanism is to choose which theorems are relevant and what are their canonical names. While many other proof assistants

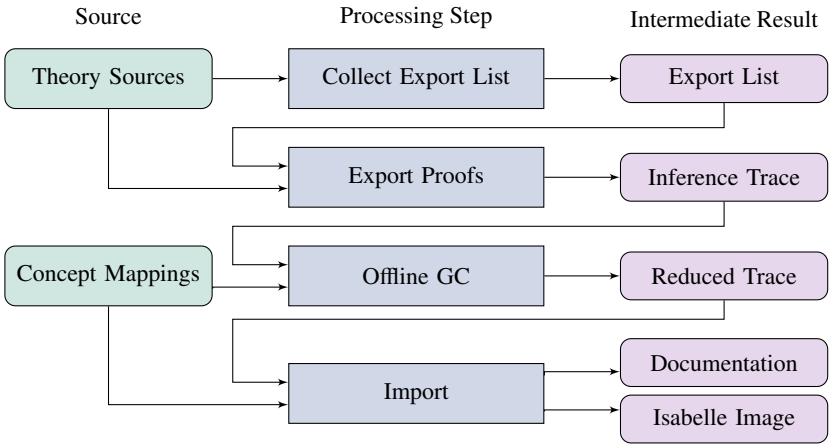


Fig. 1. The architecture of the translation mechanism.

manage a list of named lemmas explicitly in some form of table, HOL Light simply represents lemmas as OCaml values of a certain type, bound on the interactive toplevel.

We first tried to follow a strategy similar to Obua and Skalberg [12], where statements to export are collected heuristically, by detecting certain idioms in the theory sources, such as the use of the function prove in the context of a toplevel binding. However, this rather superficial analysis is quite incomplete and fails to detect non-standard means of producing lemmas occur frequently in larger developments. It also produces false positives for bindings local to a function or module.

Instead of such guesses based on surface syntax, it is more practical to rely on the existing update_database [5] functionality, which can produce a list of name-theorem pairs accessible from the toplevel by analyzing OCaml's internal data structures. The result can be saved to a file and loaded into a table that maps statements to names, before starting the actual export. Whenever a new theorem object is created, we can then look up the corresponding name, if any, in the table.

2.2 Exporting the inference trace

To export the performed proofs, we use a patched version of the HOL Light kernel, which is modified to record all inference steps. Our recorded HOL Light session is a sequence of steps according to the grammar in Fig. 2. Each step constructs a new type, term or theorem, and the new object is implicitly assigned an integer id from an increasing sequence (separately for each kind of object), by which it can be referenced later. The arguments of proof steps are either identifiers (such as names of variables or constants), or references to previous proof steps, with an optional tag denoting the deletion of the referenced object.

The trace is generated on the fly and piped to a *gzip* process, which compresses it and writes it to disk.

⟨Step⟩ ::=		A step is either:
	⟨Typ⟩	a type construction step
	| ⟨Trm⟩	a term construction step
	| ⟨Thm⟩	a theorem construction step
⟨Typ⟩ ::=		A type step is either:
	TyVar ⟨id⟩	a named type variable
	| TyApp ⟨id⟩ ⟨ref⟩*	a type application with a list of subtypes
⟨Trm⟩ ::=		A term step is either:
	Var ⟨id⟩	a named variable
	| Const ⟨id⟩ ⟨ref⟩	a named constant with a type, the type is necessary for polymorphic constants
	| App ⟨ref⟩ ⟨ref⟩	an application
	| Abs ⟨ref⟩ ⟨ref⟩	an abstraction; in valid HOL Light abstractions the first subterm is always a term that is a variable
⟨Thm⟩ ::=		A term step is either:
	| Refl ⟨ref⟩	Reflexivity of a term
	| Trans ⟨ref⟩ ⟨ref⟩	Transitivity of two theorems
	| Comb ⟨ref⟩ ⟨ref⟩	Application of two theorems
	| Abs ⟨ref⟩ ⟨ref⟩	Abstraction of a term and a theorem
	| Beta ⟨ref⟩	β-reduction of a term
	| Assum ⟨ref⟩	Assumption of a term
	| Eqmp ⟨ref⟩ ⟨ref⟩	Equality *modus-ponens* of two theorems
	| Deduct ⟨ref⟩ ⟨ref⟩	Anti-symmetric deduction of two theorems
	| Inst ⟨ref⟩ ⟨ref⟩*	Type substitution with a theorem and a list of types
	| Subst ⟨ref⟩ ⟨ref⟩*	Term substitution with a theorem and a list of terms
	| Axiom ⟨ref⟩	Axiom with a term
	| Defn ⟨id⟩ ⟨ref⟩	Definition of an identifier to a term
	| TyDef ⟨id⟩ ⟨id⟩ ⟨id⟩ ⟨ref⟩ ⟨ref⟩ ⟨ref⟩	Type definition with three identifiers, two terms and a proof of existence
	| Save ⟨id⟩ ⟨ref⟩	Assign a name to a theorem

Fig. 2. The grammar of our export format. Identifiers ⟨id⟩ are strings with the same characters allowed as in the input system and references ⟨ref⟩ are integers.

Since the type thm of theorems is an abstract datatype, we can achieve the numbering simply by adding an integer tag, which is filled from a global proof step counter when the theorem is constructed.

Hence, the proof steps are recorded in the order in which they occur. This instrumentation is local to the kernel (module `fusion.ml`), which encapsulates all theorem construction and destruction operations. No changes are required in other files.

For terms and types this is not so easy, as in OCaml even for abstract objects matching is possible. This means that we cannot add tags without breaking client code. However, just writing out terms naively would destroy all sharing and lead to significant blowup. To make the process manageable, we need to preserve at least some of the

sharing present in memory. Thus we employ memoization techniques that partly recover the sharing of types and terms. Section 3 describes these techniques; all of them produce traces in the format described above.

There is also a choice of the basic proof steps that get written. We have decided to export the minimal set of inference rules that covers HOL Light. This is in opposition to Obua and Skalberg's export where some of the proof rules get exported as multiple steps (Deduct gets exported as three rules that are easier to import) whereas other rules get optimized proofs. We found out that such *ad-hoc* optimizations yield only very small improvements in efficiency (below 2%), but unnecessarily complicate the code and make it more prone to errors.

2.3 Offline garbage collection

While our trace records the creation of all relevant objects, it does not contain information about their deletion. However, this information is equally important if we do not want to create a memory leak in the Isabelle import, which would have to keep all objects, even though they have long been garbage-collected in the HOL Light process.

While the information about object deletion is not directly present in the trace, it can easily be recovered by marking the last reference to each object specially as an indication that the importer should drop the reference to the object after using it.

We do this in a separate processing step, which in addition performs an offline garbage collection on the whole graph and throws away all objects that do not contribute to the proofs of a named theorem.

The offline processor takes two inputs: the inference trace and a list of theorems that should be replaced by Isabelle theorems during import. The program first removes all the steps that are not needed when importing. This includes theorems that were created but never used (for example by proof search procedures that operate on theorems), theorems that are mapped to their Isabelle counterparts, and their transitive dependencies. Next, the last occurrences of a reference to any type, term or theorem is marked, so that the importer can (after using the object as a dependency) immediately forget it. Currently, our offline processor reads the whole dependency graph into memory. This is feasible even for developments of the size of Flyspeck. Hypothetical, for larger developments this could be replaced by an algorithm that does several passes on the input and requires less memory.

The output of this step has the same format as the raw trace. We call it the *reduced trace*.

2.4 Import

The actual import into Isabelle now amounts to replaying the reduced trace generated in the previous step. In addition, the import is configured with the mappings of types, constants and theorems to the respective Isabelle concepts.

Replaying the trace is conceptually straightforward: we simply replay step by step, keeping integer-indexed maps to look up the required objects needed for each proof step.

We use Isabelle's `cterm` type (an abstract type representing type-checked terms) to store terms, and similarly for types. This avoids repeated type-checking of terms and reduces import runtime by a factor of two, with a slight increase in memory use.

We do not attempt to generate theory source text, which was a major bottleneck and source of problems in Obua and Skalberg's approach, since re-parsing the generated theories is time-consuming, and the additional layer of build artifacts only makes the setup unnecessarily complex with little benefit. The only advantage of the generated theories were that users could use them to inspect the imported material. For this purpose, we instead generate a documentation file (in LaTeX and HTML), which lists lemma names and statements. Excerpts from the documentation can be seen in Figures 3 and 4. We also provide the complete rendered documentation for the Flyspeck import at http://cl-informatik.uibk.ac.at/~cek/import/.

thm RIGHT_OR_DISTRIB:
∀(p::bool) (q::bool) r::bool. ((p ∨ q) ∧ r) = (p ∧ r ∨ q ∧ r)
thm FORALL_SIMP:
∀t::bool. (∀x::'a. t) = t
thm EXISTS_SIMP:
∀t::bool. (∃x::'a. t) = t
thm EQ_CLAUSES:
∀t::bool. (True = t) = t ∧ (t = True) = t ∧ (False = t) = (¬ t) ∧ (t = False) = (¬ t)

Fig. 3. Cropped printout of the HTML documentation of the imported HOL Light library.

thm opposite_hypermap_plain:

$\forall H.\ plain\_hypermap\ H \longrightarrow plain\_hypermap\ (opposite\_hypermap\ H)$

thm opposite_components:

$\forall H\ x.\ dart\ (opposite\_hypermap\ H) = dart\ H \wedge node\ (opposite\_hypermap\ H)\ x = node\ H\ x \wedge face\ (opposite\_hypermap\ H)\ x = face\ H\ x$

thm opposite_hypermap_simple:

$\forall H.\ simple\_hypermap\ H \longrightarrow simple\_hypermap\ (opposite\_hypermap\ H)$

thm hypermap_eq_lemma:

$\forall H.\ tuple\_hypermap\ H = (dart\ H,\ edge\_map\ H,\ node\_map\ H,\ face\_map\ H)$

Fig. 4. Cropped printout of the LaTeX generated PDF documentation of Flyspeck imported to Isabelle. The documentation can be generated with or without type annotations, and we chose to present the type annotations in Fig. 3 and no types here.

After replaying the trace in Isabelle, it can be used interactively or saved to an image (using Isabelle's standard mechanisms) for later use.

Concept Mappings In general it is desirable to map HOL Light concepts to existing Isabelle concepts whenever they exist instead of redefining them during the import. This makes it easier to use the imported results and combine them with existing ones. This is a form of *theory interpretation*.

There are two scenarios: First, if the definition of the constant or type (typedef) from the original system can be derived in the target system, it is enough to replace the definition with the derived theorem. Second, if a definition or type is not defined in the same way and the original definition cannot be derived, the procedure is more involved. This is the common common case for more complex definitions.

Consider the function HD which represents returns the first element of a fiven non-empty list. It's result on the empty list is some arbitrary unknown value, but while HOL Light makes use of the Hilbert operator ϵ, Isabelle/HOL uses list_rec undefined, which is an artifact of the tools used. Both these terms represent "arbitrary" values, but they are not provably equal. However, since there are no theorems in HOL Light that talk about the head of an empty list, we can get away with it.

In general, we can replace any set of characteristic properties from which all translated results are derived and which is provable in the target system. This need not be the actual definition of the constant. This also means that some lemmas that are merely used to derive the characteristic properties will not be translated. This requires some dependency analysis which is hard to do during the actual export or import, which merely write and read a stream. It is therefore done in the offline processor.

Obua and Skalberg's import attempted to resolve mappings during import, which would make mapping non-trivial concepts like the real numbers a tedious trial-and-error experience.

Mapping of constants can be provided simply by giving a theorem attribute in Isabelle, for example to map the HOL Light constant FST to the Isabelle/HOL constant fst it is enough to prove the following:

```
lemma [import_const FST]:
  "fst = (λp::'A × 'B. SOME x::'A. ∃y::'B. p = (x, y))"
  by auto
```

and similarly to map a type:

```
lemma [import_type prod ABS_prod REP_prod]:
  "type_definition Rep_prod Abs_prod (Collect
    (λx::'A ⇒ 'B ⇒ bool. ∃a b. x = Pair_Rep a b))"
  using type_definition_prod[unfolded Product_Type.prod_def] by simp
```

There is one more issue that we need to address. Even if the the natural numbers of HOL Light are mapped to the natural numbers of Isabelle, the binary representation of nat in Isabelle is different from that of num in HOL Light. To make them identical, a set of rewrite rules is applied that rewrites the constant NUMERAL applied to bits, to the Isabelle version thereof.

3 Time and Memory comparison of Flyspeck vs Import

In this Section we discuss the memoization techniques used to reduce the import time and memory footprint of the processing steps as well as the import time and give some comparisons of the processes involved in Import with the original ones of HOL Light.

As we have noticed in Section 2 when writing the trace the sharing between terms (and types) is lost. The number and size of types in a typical HOL Light development is insignificant in comparison with the number and size of terms; which is why we will focus on optimizing the terms; however the same principles are used for types.

There are many ways in which the problem can be addressed; the simplest is to simply write out all the terms. This is equivalent to not doing any sharing; and came out to be infeasible in practice. Even when writing a compressed trace after 3 weeks runtime, the compressed trace was 500GB, without having got past HOL Light's Multivariate library (a prerequisite of Flyspeck).

So is sharing always good? Then the ultimate goal would be to achieve complete sharing, where the export process keeps all the terms that have been written so far, and whenever a new term is to be written it is checked against the present ones. However, keeping all terms in memory obviously does not scale, as the memory requirements would grow linearly with the size of the development. (In fact, this was true in Obua and Skalberg's implementation, where recorded proofs were kept in-memory eternally.) Moreover, the import stage becomes equally wasteful, since terms are held in memory between any two uses, where it would be much cheaper to rebuild them when needed. Running Flyspeck with this strategy requires 70GB RAM for the export and 120GB for the import stage.

Instead, we employ a least-recently-used strategy, which keeps only a fixed number of N entries in a cache and discards those that were not referenced for the longest time. While this does not necessarily reflect the actual lifecycle of terms in the original process, it seems to be a good enough approximation to be useful. In particular, it avoids wasting memory by keeping unused objects around for a long time. Our data structures are built on top of OCaml's standard maps in a straightforward manner, and all operations are $O(\log N)$.

We show the impact of the size of the term cache on the export time and memory footprint in Figure 5. The graph shows two datasets: the core of HOL Light and the VOLUME_OF_CLOSED_TETRAHEDRON lemma from Flyspeck. We chose the former, as it represents a typical HOL Light development, consisting of a big number of regular size lemmas, and we chose the latter, as it creates a big proof trace. The big trace is created using the REAL_ARITH decision procedure which implements the Gröbner bases procedure [4].

As we can see the size of the term cache has very little impact on the time and size of HOL Light standard library. However for a proof which constructs a big number of bigger terms using a very small cache increases the export time and trace size exponentially. This means that the size needs to be adjusted to the size of the terms produced by the decision procedures and if this is not known a biggest possible term cache size is advisable.

We performed a similar experiment for caching proofs. The first idea is to reuse the proof number if a theorem with same statement has already been derived. This can

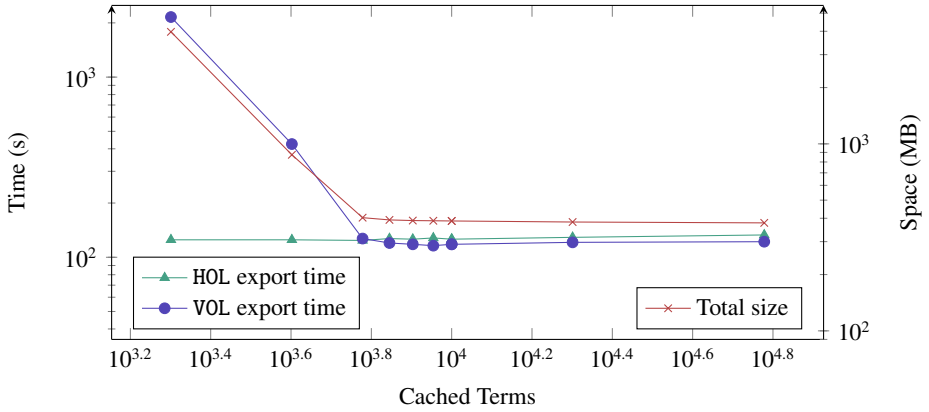


Fig. 5. Time required to write the proof trace and the size of the resulting trace as a function of the term cache size. We compare the time of the HOL Light standard library (HOL) and the VOLUME_OF_CLOSED_TETRAHEDRON theorem (VOL).

however cause problems if there are some mappings performed between the two theorems. A constant or type mapping is performed together with some theorem mappings, and the new type or constant will be used from the point of this theorem. This means that unnamed theorems used before a mapping cannot be reused after the mapping. To overcome this difficulty we chose to clear the theorem cache at every named theorem. This may lose a small amount of sharing but prevents issues with adding constant and type maps. We have computed the impact that the size of the proof cache has on the resulting trace and the export time in Figure 6.

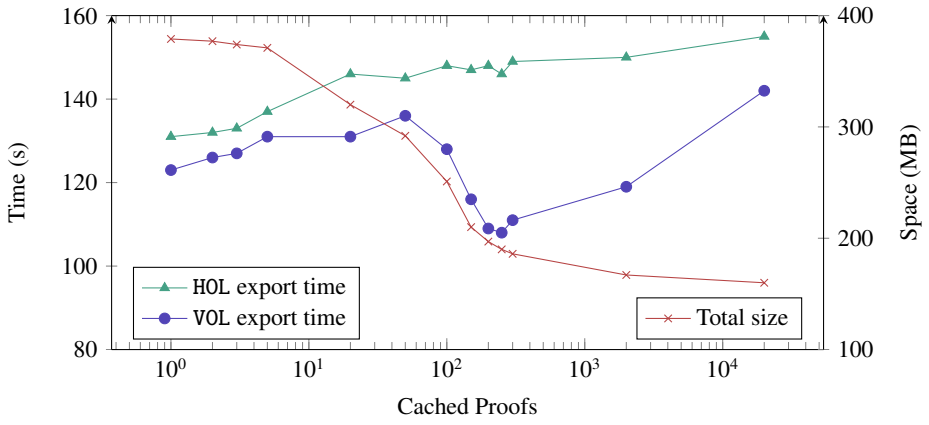


Fig. 6. Time required to write the proof trace and the size of the resulting trace as a function of the proof cache size. We compare the time of the HOL Light standard library (HOL) and the VOLUME_OF_CLOSED_TETRAHEDRON theorem (VOL).

As we can see for typical proof steps (HOL Light standard library) with more sharing the export time increases however not significantly. This is not the case for a proof which constructs big intermediate results. In such a case there is an optimal number of proofs to cache and with a bigger number the complexity of comparing the proofs with the cache increases the time. Since in a typical HOL Light development the optima for the different proofs may differ, we instead choose to use a small proof cache. In Section 4 we will see that the decrease in the space does not lead to a significant decrease in the Import time. We have compared the memory usage of the OCaml process writing the trace with the memory of the PolyML process doing the Import (Fig. 7 and the two are roughly comparable, with two exceptions. PolyML is much less conservative when allocating memory, and quickly uses all available memory, however most of it is reclaimed by major collections. Also due to the garbage collection running in a separate thread in PolyML major collections happen more often than in OCaml.

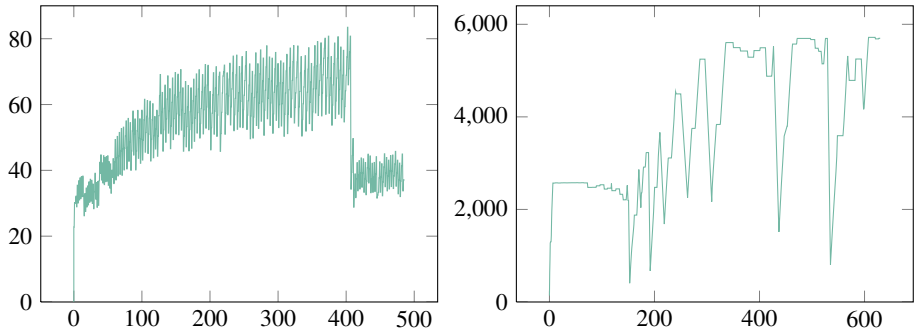


Fig. 7. Comparing the export memory (left) with import memory (right) for the core HOL Light together with the VOLUME_OF_CLOSED_TETRAHEDRON theorem. Export memory is computed with OCaml garbage collection statistics in millions of life words. Import memory in bytes.

4 Statistics over Flyspeck

In this section we look at various statistics that can be discovered when analyzing the proofrecorded Flyspeck.

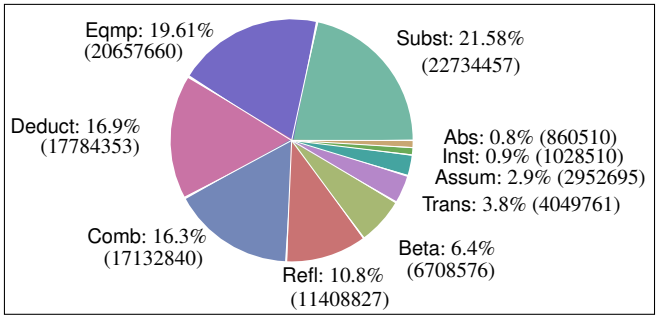
4.1 Dependencies and steps statistics

We first look at the dependencies between theorems, terms and types. The following table shows the total number of theorem steps, term steps and type steps in the proof trace. The table includes four rows, first two are for full sharing, second two are for term caching.

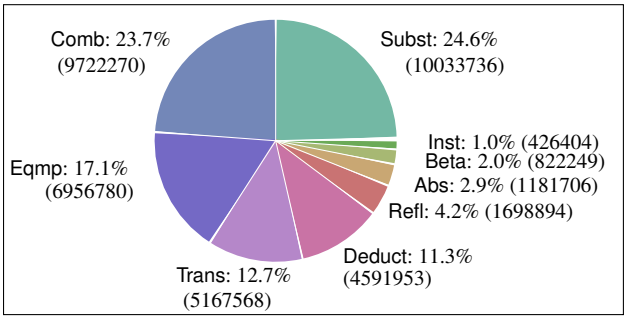
Strategy	Types	Terms	Theorems
Full sharing	173,800	51,448,207	146,120,269
Full sharing processed	130,936	13,066,288	40,802,070
Term caching	173,800	101,846,215	420,253,109
Term caching processed	146,710	23,318,639	194,541,803

We first notice that the number of types is insignificant relative to the number of terms or theorems, and the number of types that can be removed by processing is only 16–25%. For terms, the number of terms with full sharing is 50% of that with caching, which means that the overhead is still quite big; and the overhead reduces only to 45% with the offline GC. Sharing has the biggest effect on theorems: the shared theorems are 34% of the cached ones. Finally, offline GC lets reduce the number of terms and theorems roughly a quarter of the recorded ones, which is a huge improvement.

Next we will look at the exact inference steps that were derived but not needed. We have computed the numbers of inference steps of each kind and their percentages for the steps that were derived but the offline processor could remove them:

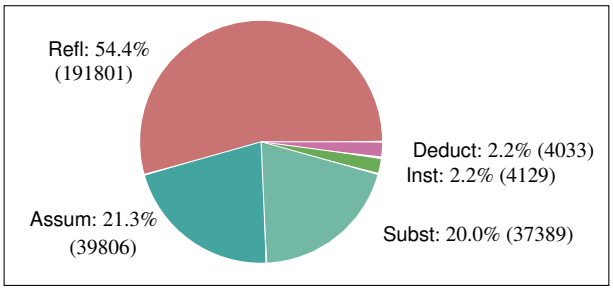


We compare the above to the needed steps (the steps left by the garbage collection):



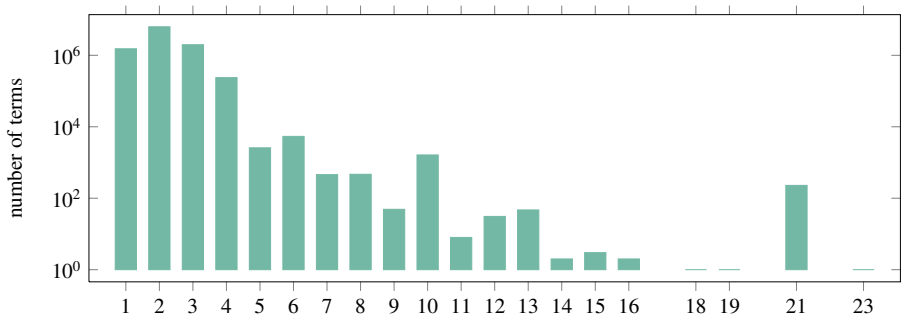
In both cases the methods that do complicated computation (Subst, Eqmp, Comb) dominate; however for the unneeded diagram the very cheap methods Refl and Assum occur much more often.

We next compare it to the steps that were derived multiple times. 2,515,531 theorem steps have been derived multiple times with exactly same dependencies. It is quite interesting, that many of these steps are the same ones repeated over and over again. This suggests that the steps are performed repeatedly by decision procedures or even by the simplifier. The unique repeated steps are only 187,164 which is just 7% of the repeated steps! We have analyzed the kinds of steps that have been derived multiple times:



Here we see a dominance of Refl and Assum which are the two cheapest steps. However the third steps — Subst — is not a cheap one; and this is where the main advantage of caching or sharing comes in.

We have also looked at the complexity of individual Subst steps; even if the maximum is 23, the average number of term pairs in a subsitution is 2.098, and the distribution is as follows:



We see that there are 229 substitutions with 21 simultaneously substituted terms; however the time consumed by these substitutions is caused by the size of the terms involved.

4.2 Theorem name statistics

There are 622 theorems that have been assigned more than one name. In certain cases useful theorems from another module have been given a different name in a later one; however in certain cases theorems have been rederived, sometimes in completely different ways. We have computed the statistics:

Unique statements	Number of names	Canonical name
1	16656	
2	567	
3	46	
4	5	
5	2	Trigonometry1.ups_x, Sphere.aff
7	1	Trigonometry.UNKNOWN
11	1	REAL_LE_POW

In certain cases, theorems with meaningful names have been assigned also random names; but there are a few cases where meaningful names have been assigned to theorems twice. There are multiple possible reasons for this and we present here a few common cases, from the more trivial ones to more involved ones.

- REAL_LT_NEG2 and REAL_LT_NEG both derived using the real decision procedure in the same file just 70 lines further.
- Topology.LEMMA_INJ and Hypermap.LEMMA_INJ the proof is an exact copy, still both are processed.
- REAL_ABS_TRIANGLE and Real_ext.ABS_TRIANGLE first one derived using a decision procedure, second one with a complete proof.
- INT_NEG_NEG and INT_NEGNEG both derived using the quotient package from real number theorems called differently.
- CONVEX_CONNECTED_1_GEN and CONNECTED_CONVEX_1_GEN where one is derived from the other.

4.3 Translation time and size statistics

Given the best cache sizes computed in Section 3 and the statistics over the steps we have computed the times and sizes of the various stages of Import evaluated on the whole Flyspeck development:

Phase	Time	Size
Collect Export List	4h	18MB
Export Proofs	10h	3692MB
Offline GC	48m	1089MB
Import without optimizations	54h	

The created Isabelle image can be loaded in a few seconds and memory allocated by the underlying PolyML system is 2.7GB which is almost same as the 2.7GB allocated by HOL Light. The time required to create the image is by a factor of 13 longer than the time required to run Flyspeck. This is still quite a lot, and is possible to do it once, but not if such a development must be translated routinely. As we have discovered in Section 4.1, this is caused by decision procedures; in fact REAL_ARITH (an implementation of Gröbner bases) is what creates the huge terms and substitutions that constitute a big part of the proof trace and import time.

To further optimize the Import time we tried to map the two most expensive calls to this decision procedure to a similar decision procedure in Isabelle. The algebra tactic [2] can solve the goals in Isabelle in a similar time as that needed in HOL Light:

Phase	Time	Size
Offline GC	48m	964MB
Import with optimizations	4.5h	

The reduced proof trace is 11% smaller, but the import time becomes roughly equal to processing of the theories with HOL Light.

5 Conclusion

We have presented a new implementation of a theory import from HOL Light to Isabelle/HOL, designed to achieve a decent performance. The translation allows mapping the concepts to their Isabelle counterparts obtaining natural results. By analyzing the proof trace of Flyspeck we have also presented a number of statistics about a low-level structure of a big formal development.

The code of our translation mechanism has been included in Isabelle together with a component for loading the core HOL Light automatically and the documentation for it. The formalization includes the mappings of all the basic types present in both developments including types that are not defined in the same way, such as lists, integers and real numbers. In total, 97 constructors have been mapped with little effort. It is easy to generate similar components for other HOL Light developments. The code is 5 times smaller than the code of Obua and Skalberg's Import.

In our development we defined a new format for proof exchange traces, despite the existence of other exchange formats. We have tried writing the proof trace in the OpenTheory format, and it was roughly 10 times bigger. For the proof traces whose sizes are measured in gigabytes such an optimization does make sense; however is it conceivable to share the Import code with other formats.

The processed trace generated by this work is already used by machine learning tools for HOL Light to provide proof advice [7,8].

5.1 Future Work

The most obvious future work is testing our export on other HOL Light developments, including the most interesting ones which are not formalized in Isabelle, for example the developments from Wiedijk's 100 theorems list [13] and Hilbert Axiom Geometry. Similarly the work can be extended to work with different pairs of provers, in particular not HOL-based ones, or the Common HOL Platform [1] intended as a minimal base for sharing HOL proofs.

A different line of work could be to automate the mapping of results of decision procedures. We have tried to export the steps performed by REAL_RING, REAL_ARITH, or REAL_FIELD as a single proof step. Unfortunately for each of these, there exists at least one goal that it solved, but which could not be solved by algebra; this needs to be investigated further.

The statistics performed on the repository allow for an easy discovery of duplicate proofs and multiple names given to same theorems. This can be used to streamline the original developments. Conversely, for imported libraries that match ones present in the target system, analyzing the theorems that are not present may lead to the discovery of interesting intermediate lemmas.

References

1. Mark Adams. Introducing HOL Zero - (extended abstract). In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *ICMS*, volume 6327 of *Lecture Notes in Computer Science*, pages 142–143. Springer, 2010.
2. Amine Chaieb and Tobias Nipkow. Proof synthesis and reflection for linear arithmetic. *J. Autom. Reasoning*, 41(1):33–59, 2008.
3. Thomas C. Hales, John Harrison, Sean McLaughlin, Tobias Nipkow, Steven Obua, and Roland Zumkeller. A revision of the proof of the Kepler conjecture. *Discrete & Computational Geometry*, 44(1):1–34, 2010.
4. John Harrison. Automating elementary number-theoretic proofs using Gröbner bases. In Frank Pfenning, editor, *Automated Deduction (CADE 21)*, volume 4603 of *Lecture Notes in Computer Science*, pages 51–66, Bremen, Germany, 2007. Springer.
5. John Harrison and Roland Zumkeller. update_database module. Part of the HOL Light distribution.
6. Joe Hurd. The OpenTheory standard theory library. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2011.
7. Cezary Kaliszyk and Josef Urban. Initial experiments with external provers and premise selection on HOL Light corpora. In Pascal Fontaine, Renate Schmidt, and Stephan Schulz, editors, *PAAR*. to appear, 2012.
8. Cezary Kaliszyk and Josef Urban. Learning-assisted automated reasoning with Flyspeck. *CoRR*, abs/1211.7012, 2012.
9. Matt Kaufmann and Lawrence C. Paulson, editors. *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*. Springer, 2010.
10. Chantal Keller and Benjamin Werner. Importing HOL Light into Coq. In Kaufmann and Paulson [9], pages 307–322.
11. Alexander Krauss and Andreas Schropp. A mechanized translation from higher-order logic to set theory. In Kaufmann and Paulson [9], pages 323–338.
12. Steven Obua and Sebastian Skalberg. Importing HOL into Isabelle/HOL. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 298–302. Springer, 2006.
13. Freek Wiedijk. Formalizing 100 theorems. http://www.cs.ru.nl/~freek/100/.
14. Wai Wong. Recording and checking HOL proofs. In E. Thomas Schubert, Phillip J. Windley, and Jim Alves-Foss, editors, *TPHOLs*, volume 971 of *Lecture Notes in Computer Science*, pages 353–368. Springer, 1995.