OpenTheory: Package Management for Higher Order Logic Theories

Joe Hurd
Galois, Inc.
joe@galois.com

Abstract

Interactive theorem proving has grown from toy examples to major projects formalizing mathematics and verifying software, and there is now a critical need for theory engineering techniques to support these efforts. This paper introduces the OpenTheory project, which aims to provide an effective package management system for logical theories. The OpenTheory article format allows higher order logic theories to be exported from one theorem prover, compressed by a stand-alone tool, and imported into a different theorem prover. Articles naturally support theory interpretations, which is the mechanism by which theories can be cleanly transferred from one theorem prover context to another, and which also leads to more efficient developments of standard theories.

Keywords theory management, formalizing mathematics

1. Introduction

Interactive theorem provers have grown far beyond toy examples, and are nowadays being used for impressive verification projects in mathematics and computer science. Recent examples include: the CompCert project, which verified an optimizing compiler from a large subset of C to PowerPC assembly code [12]; and the Flyspeck project, which aims to mechanize a proof of the Kepler sphere-packing conjecture [6].

Just as the term software engineering was coined in 1968 [14] to give a name to techniques for developing increasingly large programs, there is now a need for *theory engineering* techniques to develop increasingly large proofs. An interactive theorem prover in the LCF design can be seen as a compiler from a source language program containing high-level proof steps to an assembly language program consisting of primitive inferences in the logic. Viewed this way, it is instructive to see which software engineering principles can be applied to proving in the large.

One software engineering principle that immediately applies is to have a standardized source language for high-level proof steps, to eliminate the problem of improvements to the underlying proof tools breaking legacy proofs. The declarative proof language of Mizar [18] pioneered this approach, and it has been refined in the Isar mode of the Isabelle theorem prover [19]. Likewise, the software engineering principle of using modules to effectively reuse proofs and avoid namespace pollution has been implemented in many theorem provers, notable examples being theory interpretations in IMPS [5] and locales in Isabelle [10].

This paper illustrates how another principle of software engineering can be applied to proof development: effective package management of theories. Modern operating systems bundle software into packages that explicitly track their dependencies, to avoid being installed into a system that does not provide the support that they require. The `apt-get install` command in the Debian distribution of the Linux operating system takes this one step further, recursively installing packages until all dependencies are satisfied and the requested package can be installed. Similarly, `cabal install` does the same for modules in the Haskell programming language [3]. By contrast, transplanting a theory from one theorem prover to another often involves manually translating the theorem statements and creating new proofs for them in the local high-level proof step language.

The OpenTheory project aims to transfer the benefits of effective package management to logical theories.[1] The initial case study of the project is higher order logic [4],[2] which is implemented by three interactive theorem provers that currently have no good way to exchange theories: HOL Light [7], HOL4 [16] and ProofPower [11]. To continue the compiler analogy, the hardware (logic) of the three theorem provers is the same, so even though they all implement subtly different assembly languages (primitive inferences), they can all simulate each other.[3] This paper presents the file format for packages of higher order logic theories, called *articles*, which support the following:

- export from and import into theorem provers without explicit proof objects;
- offline concatenation and compression operations using a stand-alone tool;
- clean transfer from one theorem prover context to another, using a restricted form of theory interpretation.

The remainder of the paper is structured as follows: Section 2 presents the article file format for higher order logic theories; Section 3 describes the operations that can be performed on theory packages in article format; Section 4 demonstrate the results of archiving and compressing the theories distributed with the HOL Light theorem prover; Section 5 illustrates the utility of theory

[1] The OpenTheory project homepage is `http://gilith.com/research/opentheory`

[2] In this paper, the term *higher order logic* is exclusively used to refer to simple type theory with Hindley-Milner type polymorphism.

[3] The type class extension of Isabelle/HOL [15] makes it a later version of the hardware, so although it can simulate the assembly language of the others, they cannot simulate it.

`error`	Construct an error object
⟨number⟩	Construct a non-negative integer
"⟨string⟩"	Construct a quoted string
`nil`	Construct an empty list
`cons`	Add an object to a list
`type_var`	Construct a type variable
`type_op`	Construct a type operator
`var`	Construct a term variable
`const`	Construct a constant
`comb`	Construct a function application
`abs`	Construct a λ-abstraction
`thm`	Find a theorem
`call`	Simulate a function call
`return`	Simulate a return from a function call
`def`	Add an object to the dictionary
`ref`	Look up an object in the dictionary
`remove`	Remove an object from the dictionary
`pop`	Pop an object from the stack
`save`	Save a theorem onto the export list

Table 1. The complete list of article commands.

packages with two examples; and finally Sections 6–8 examine related work, consider future directions and summarize.

2. Articles of Proof

This section introduces the OpenTheory article file format for higher order logic theory packages. For articles, the design choice was made to rely only on the primitive inferences of the supported theorem provers. As a consequence, articles cannot be edited, so theories should only be packaged into articles when they are stable enough to be archived.

An alternative approach, taken by the Common HOL Project, is to package the ML source file that generated the higher order logic theory.[4] Packaging theories in source form has the advantage that they can be edited, but at the cost of being dependent on a set of standardized proof tools.

Not all of the supported higher order logic theorem provers build explicit proof objects for theorems. However, every proof tool in the theorem prover is a function that calls lower-level proof tools, all the way down to the primitive inference functions in the logical kernel. Thus the proof of a theorem can be represented as an ML function call tree, and the article format is a direct encoding of the function call tree generated by the ML source file that generated the theory.

This choice of format makes it easy to export theories from a theorem prover: all that is required is for some functions (at least the primitive inferences) to be instrumented to log their arguments and return value to a file. To read in an article created in this way, the theorem prover steps through the file and simulates the primitive inference functions that it encounters. This process is made more precise in the remainder of this section: for every last detail the reader is referred to the OpenTheory Article Format [9].

2.1 File Format

At the most basic level, an OpenTheory article is a text file using the UTF-8 character set. Every line in an article file is either a comment, in which case the first character must be #, or one of the commands in Table 1. The commands in the article file are processed by a stack-based virtual machine.

As it reads the article file, the virtual machine maintains the following data structures:

- A stack containing values of type `object`.
- A dictionary mapping keys of type `int` to values of type `object`.
- An import list of type `thm list`.
- An export list of type `thm list`.

Initially the stack, dictionary, import list and export list are all empty. The type of `object` stored on the stack and in the dictionary is defined as:

```
datatype object =
    Oerror                  (* An error value    *)
  | Oint of int             (* A number          *)
  | Oname of string         (* A name            *)
  | Olist of object list    (* A list (or tuple)
                               of objects        *)
  | Otype of type           (* A higher order
                               logic type        *)
  | Oterm of term           (* A higher order
                               logic term        *)
  | Othm of thm             (* A higher order
                               logic theorem     *)
  | Ocall of string         (* A special object
                               marking a
                               function call     *)
```

The virtual machine reads the article file line by line. Comments are discarded, and commands are immediately executed. As a result of executing a command, the stack, dictionary, import list or export list may be altered. After a command has been executed it is discarded. When the virtual machine has finished processing all the lines in the article file, the import list and export list of theorems are the result of reading the article (the stack and dictionary are discarded).

2.2 Simulating Call Trees

The article file is a direct encoding of the function call tree generated by the ML source file that generated the theory, and as the virtual machine reads the article it performs a simulation of the function calls using its stack. Here is the specification of the `call` and `return` article commands:

```
call
    Pop a name n; pop an object a; push the function
    call marker Ocall n; push the argument value a.

    Stack:  Before:  Oname n :: a :: stack
            After:   a :: Ocall n :: stack

return
    Pop a name n; pop an object r; pop objects from
    the stack up to and including the top function
    call marker Ocall n; push the return value r.

    Stack:  Before:  Oname n :: r :: ... ::
                       Ocall n :: stack
            After:   r :: stack
```

Every function must return a value, to ensure that the `Ocall` objects are properly removed from the stack. If the ML function returns an exception, then this must be trapped by the instrumenting code and logged as returning with an `Oerror` value.

Some of the function calls being simulated will be primitive inferences, and some will be higher-level proof tools. By examining an article, it is possible to profile proof tools by seeing what lower-level proof tools they call. One simple metric would be the number of primitive inferences that a proof tool makes.

2.3 Constructing Types and Terms

Function arguments and return values are built up on the stack by other article commands. For example, here is the specification

[4] Personal communication from Mark Adams.

of the `var` command for constructing a higher order logic term variable:

```
var
    Pop a type ty; pop a name n; push a variable with
    name n and type ty.

    Stack:  Before:  Otype ty :: Oname n :: stack
            After:   Oterm (mk_var (n,ty)) :: stack
```

It is often the case that the representation of higher order logic types and terms in ML uses sharing of memory locations. Naively building highly shared types and terms can require exponentially more stack commands than the number of memory locations used in its ML representation.

The article dictionary maintained by the virtual machine is used to solve this problem. If an object is going to be required multiple times, it is constructed once, and associated with an integer key in the dictionary using the `def` command. Subsequent uses look up the object in the dictionary with the key using the `ref` command, instead of constructing it again. On the last use of an object, the `remove` command can be used instead of `ref`, which removes the association from the dictionary in addition to looking up the object.

In practice the theorem prover that is instrumented to export an article file will not know how many times an object will be used in the future. It can either choose to use the dictionary on a per-object basis to avoid the exponential cases, or store all newly constructed objects in the dictionary in case they will be later required. The first approach requires no additional storage in the theorem prover, but the second approach will usually result in smaller article files.

2.4 Theorems

The `save` command reads an `Othm th` object from the top of the stack, and adds the theorem `th` to the export list of the article. Theorems are constructed using the `thm` command, which has the following specification:

```
thm
   Pop a term c; pop a list of terms h; push the
   theorem h ⊢ c with hypothesis h and conclusion c.

   Stack:  Before:  Oterm c ::
                      Olist [Oterm h1, ..., Oterm hn] ::
                        stack
           After:   Othm ([h1, ..., hn] ⊢ c) :: stack
```

The hypothesis and conclusion terms only provide a specification of the result theorem—the theorem is constructed using the first one of the following methods that succeeds:

1. Look for the result theorem on the export list of the article.
2. If the current function is a primitive inference rule, the result theorem is proved by simulating the inference using the argument value.
3. Look for the result theorem inside an object on the stack.
4. Assert the result theorem as an axiom, and add it to the import list of the article.

2.5 Example Article

Here is a tiny annotated article that exports the theorem $\vdash \mathsf{T}$:

```
# Construct the hypothesis list
nil
# Construct the conclusion term
"T"
"bool"
nil
type_op
const
# Construct the theorem
thm
# Export the theorem
save
# Clean up the stack
pop
```

This tiny example illustrates several aspects of articles: each line of the article is a command; the command arguments are popped from the stack and the result (if any) is pushed onto the stack; and clean articles leave nothing on the stack when they end.

REQUIRES
types: $\mathsf{t}_1\ \mathsf{t}_2 \ldots \mathsf{t}_i$
consts: $\mathsf{c}_1\ \mathsf{c}_2 \ldots \mathsf{c}_m$
thms: $\gamma_1\ \gamma_2 \ldots \gamma_p$
PROVIDES
types: $\mathsf{u}_1\ \mathsf{u}_2 \ldots \mathsf{u}_j$
consts: $\mathsf{d}_1\ \mathsf{d}_2 \ldots \mathsf{d}_n$
thms: $\delta_1\ \delta_2 \ldots \delta_q$

Figure 1. The general form of an article summary.

3. Theory Packages

The previous section was concerned with the details of the article format, focusing on how its design makes it practical to export theories from one theorem prover and import them into another. By contrast, this section examines articles as a package for higher order logic theories, independently of any particular theorem prover, and focuses on the theory operations that they support.

3.1 Summaries

The result of reading an article file is the import and export lists of theorems, which can be written as $\Gamma \vdash \Delta$, to indicate that the theorems in the export list Δ can be logically derived from the theorems in the import list Γ. In addition to the theorems Γ, the article also depends on the set of type operators and constants that appear in Γ. Also, in addition to proving the theorems in Δ, the article must also define the type operators and constants that appear in Δ but not in Γ. This information is captured in an article *summary*, which has the general form shown in Figure 1.

Informally, an article can be thought of as a representation of the following parameterized theory:

$$\forall \mathsf{t}, \mathsf{c} : \Gamma.\ \exists \mathsf{u}, \mathsf{d} : \Delta\ .$$

This is similar to a functor in the ML module system [13], where if a set of types t and values c that satisfy a signature Γ is provided as input, the functor will generate a set of types u and values d that satisfy a signature Δ. However there are two significant differences: the "theorem prover" in Standard ML is the type checker, and so the signatures are restricted to be type judgments, whereas articles can specify arbitrary higher order logic properties; and the particular representation of the types u and construction of the constants d has an effect on the performance of the resulting code, whereas for articles it little matters how the types u and constants d are constructed, so long as they satisfy their properties Δ.

3.2 Theory Operations

Considering articles as a package for a parameterized theory, it is interesting to consider the effect of article file operations on the summary.

Firstly, it is possible to remove some `save` commands from the article file, which has the effect of filtering theorems from the export list:

$$\text{filter}_{\Delta'}\ (\Gamma \vdash \Delta)\ =\ \Gamma \vdash (\Delta \cap \Delta')\ .$$

Secondly, the article format is concatenative, and when concatenating two article files it may be the case that the first article exports theorems that are imported by the second. Since the export list is always checked before adding a theorem to the import list, concatenation may result in some required theorems being removed:

$$(\Gamma_1 \vdash \Delta_1) \cdot (\Gamma_2 \vdash \Delta_2) \;=\; \Gamma_1 \cup (\Gamma_2 - \Delta_1) \vdash \Delta_1 \cup \Delta_2 \; .$$

Finally, it is possible to rename the type operators and constants that appear in the article summary. The easiest point at which to do this is when constructing a type operator or constant with the `type_op` and `const` commands. This provides a limited theory substitution operator:

$$(\Gamma \vdash \Delta)\sigma \;=\; \Gamma\sigma \vdash \Delta\sigma \; .$$

Of the three theory operations presented here, the theory substitution operator is the most practically useful for importing theories in article format into a theorem prover. The requirements of the article can be completely renamed to match type operators and constants in the local context, so that every theorem in the import list can be proved by theorems in the current environment.

3.3 Compression

The previous section considered the effect on the summary of performing operations on the article file. This section considers a different problem: reducing the size of the article file while keeping the summary constant.[5]

The first technique for compressing articles is *dead inference elimination*, that is, removing function calls for which the return value never contributes to the proof of an exported theorem. Perhaps surprisingly, this occurs a great deal in LCF theorem provers, usually because a proof tool throws an exception (which is represented in the article as returning an error value). An implementation trick helps here: by storing dependency pointers with each object, the ML garbage collector automatically eliminates dead inferences as the article is read.

The second technique for compressing articles is to use the dictionary in an optimal way, never constructing any object more than once. For types and terms this has a predictable effect, but arranging for a theorem to use a previously proved version can cut out a long sequence of inferences.

4. Results

Table 2 presents the results of applying the compression algorithm described in Section 3.3 to the theories distributed with the HOL Light theorem prover [7].[6] The table presents the compression ratios for both the raw articles and also the articles compressed using the `gzip` program.

The HOL Light source code was instrumented to log function calls to an article file for all the primitive inferences plus a select number of higher-level proof tools. For each theory a dictionary was maintained of all previously constructed objects; this choice results in smaller article files. Concatenating all of the general article files together results in a gigantic theory providing 129,888 theorems, but requiring only the three standard axioms shown in Figure 2 (the Axioms of Extensionality, Choice and Infinity, respectively).

From the results it is possible to derive several conclusions. Firstly, there is a high level of compression despite a maximal dictionary being used, indicating that there are many dead inferences made while processing theories that can be safely eliminated. Secondly, the similar compression ratio for the raw articles and the articles compressed by `gzip` shows the complementary nature of application and generic compression algorithms. Finally, the absolute compressed sizes of the theories are not large, demonstrating that expanding theories to primitive inferences and archiving them is feasible.

These HOL Light theories were read into HOL4 by a tool for reading articles implemented by Quinn Yee Qin Teh,[7] simultaneously demonstrating the software engineering benefits of a clear file format standard for theories, and the transfer of theories from one of the target theorem provers to another.

REQUIRES
types: bool fun ind
consts: $\forall \wedge = \implies \exists$ ONE_ONE ONTO select $\neg$
thms: $\vdash \forall t.\ (\lambda x.\ t\ x) = t$
$\vdash \forall P, x.\ P\ x \implies P$ (select P)
$\vdash \exists f.$ ONE_ONE $f \wedge \neg$ONTO f

Figure 2. The HOL Light axioms.

```
instance Ord a => Ord [a] where
  []   <= _    = True
  _:_  <= []   = False
  x:xs <= y:ys = if x <= y then
                   if y <= x then xs <= ys else True
                 else False
```

Figure 3. A Haskell type class instance lifting the <= comparison operator from a type to lists of the type.

5. Examples

5.1 Functional Properties

The first example illustrates how properties can be added to type class instances in the Haskell functional programming language. Figure 3 shows an example Haskell type class instance, which lifts the <= comparison operator for a type to lists of the type. However, the only properties that can be expressed in a type class instance are those that can be encoded as type judgments, so for instance, it is not possible to mechanically check the desirable property that if the element comparison operator is a total order then so is the generated list comparison operator.

Figure 4 shows an article summary illustrating how this type class instance can be represented as a higher order logic theory, including the lifting of the property that the comparison operator is a total order.

For clarity, not all the requirements of the theory are shown, only those that cannot be satisfied by the standard theories distributed with HOL Light. Also, instead of naming both comparison operators <=, the element comparison operator is called le and the list comparison operator leList. Despite these surface differences, it is clear that this pattern can be used to augment type class instances to generate functions with mechanically checked properties.

5.2 Negative Numbers

The second example demonstrates the utility of a higher order logic theory that generates a new type on a realistic case study in formalized mathematics. Figure 5, which is modified from Harrison's thesis [8], shows a path for constructing the real numbers from the natural numbers, via the positive integers, rationals and reals.

[5] More precisely, the summary must provide the same theorems and require a subset of the dependencies in the original article.

[6] This experiment used HOL Light version 2.20, snapshot release on 25 May 2009. HOL Light is available for download at `http://www.cl.cam.ac.uk/~jrh13/hol-light`.

[7] Personal communication from Michael Norrish.

HOL Light theory	article (Kb)	comp. (Kb)	comp. ratio	gzip'ed article (Kb)	gzip'ed comp. (Kb)	comp. ratio
num	1,821	813	56%	227	113	51%
arith	27,469	7,548	73%	2,884	1,015	65%
wf	29,277	6,330	79%	3,222	861	74%
calc_num	3,922	1,570	60%	374	203	46%
normalizer	2,845	688	76%	300	92	70%
grobner	2,417	748	70%	257	103	60%
ind-types	10,625	4,422	59%	1,274	599	53%
list	12,368	4,870	61%	1,485	673	55%
realax	23,628	7,989	67%	2,519	1,070	58%
calc_int	2,844	861	70%	314	119	63%
realarith	16,275	4,684	72%	1,326	589	56%
real	30,031	9,346	69%	3,179	1,217	62%
calc_rat	2,556	1,166	55%	289	157	46%
int	40,617	9,546	77%	3,465	1,249	64%
sets	168,586	30,315	83%	17,514	4,048	77%
iter	207,324	32,422	85%	17,557	4,199	77%
cart	20,351	3,632	83%	2,076	495	77%
define	82,185	16,409	81%	8,157	2,175	74%

Table 2. Benchmarking the compression algorithm on HOL Light theories.

REQUIRES

types: t
consts: (le : t → t → bool) totalOrder
thms: ⊢ totalOrder le

PROVIDES

consts: (leList : t list → t list → bool)
thms:
⊢ leList [] l_2 = T ∧
leList (h_1 :: t_1) [] = F ∧
leList (h_1 :: t_1) (h_2 :: t_2) =
if le h_1 h_2 then
if le h_2 h_1 then leList t_1 t_2 else T
else F
⊢ totalOrder leList

Figure 4. A higher order logic theory illustrating a pattern for representing type class instances with properties.

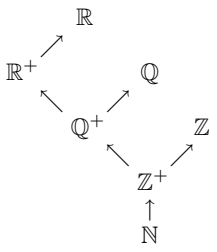

Figure 5. Constructing the real numbers from the natural numbers.

The constructions of the integers, rationals and reals from their positive elements follow the same pattern, and it is possible to generalize the construction into a higher order logic theory. Figure 6 shows the summary for an article that requires a type p with operations that satisfy the usual set of arithmetic properties. It provides a new type n with operations that satisfy arithmetic properties, plus extra zero and neg operators that behave as expected. In addition, the inject function embeds the p type into the n type, and is shown to be a homomorphism with respect to the arithmetic operations. This allows special properties of the p type to be carried over to the n type, such as the density of rational numbers.

Again, for clarity not all the requirements of the theory are shown, only those that cannot be satisfied by the standard theories distributed with HOL Light. Incidentally, this construction is normally performed by quotienting pairs of positive numbers, but in this formalization a different route was taken by defining the n type as a datatype:

$$\mathsf{n} \equiv \mathsf{negative\ p} \mid \mathsf{zero} \mid \mathsf{positive\ p}\ .$$

This led to a style of proof with much case splitting, but was generally smooth except for the associativity of addition. This one theorem was difficult enough that the quotient route is recommended for similar formalizations.

5.3 Example Compression

To confirm that there is no significant difference between the example theories and the theories distributed with HOL Light, Table 3 presents the results of applying the compression algorithm described in Section 3.3 to the example theories.

6. Related Work

Recording and replaying proofs from LCF theorem provers is not new: Wong's pioneering *Recording and checking HOL proofs* in 1995 appears to be the first [20]. As is necessary in a theorem prover without explicit proof objects, the approach involved instrumenting the logical kernel to emit primitive inferences, and the proofs were checked for validity by an independent proof checker. No attempt was made to compress the proofs, although the reading phase made two passes to avoid storing theorems past their last use.

More recently, Obua and Skalberg [17] instrumented HOL4 and HOL Light to export theories into the Isabelle/HOL theorem prover. The proofs were stored in an XML document, which preserves sharing, and peephole optimizations could be made on the proofs. This achieves good compression: they quote 21Mb for all the HOL Light theories after being compressed by the `gzip` program, which compares to 18Mb in this paper. The present work differs from this line of proof recording work by its focus on the theory package as the central concept, independent of any particular theorem prover.

REQUIRES

types: p
consts: leP addP subP multP
thms: $\vdash \forall x.\ \mathsf{leP}\ x\ x$
$\vdash \forall x, y.\ \mathsf{leP}\ x\ y \wedge \mathsf{leP}\ y\ x \implies x = y$
$\vdash \forall x, y, z.\ \mathsf{leP}\ x\ y \wedge \mathsf{leP}\ y\ z \implies \mathsf{leP}\ x\ z$
$\vdash \forall x, y.\ \mathsf{leP}\ x\ y \vee \mathsf{leP}\ y\ x$
$\vdash \forall x, y.\ \mathsf{addP}\ x\ y = \mathsf{addP}\ y\ x$
$\vdash \forall x, y, z.\ \mathsf{addP}\ (\mathsf{addP}\ x\ y)\ z = \mathsf{addP}\ x\ (\mathsf{addP}\ y\ z)$
$\vdash \forall x, x', y, y'.\ \mathsf{leP}\ x\ x' \wedge \mathsf{leP}\ y\ y' \implies \mathsf{leP}\ (\mathsf{addP}\ x\ y)\ (\mathsf{addP}\ x'\ y')$
$\vdash \forall x, y.\ \neg\mathsf{leP}\ (\mathsf{addP}\ x\ y)\ x$
$\vdash \forall x, y.\ \neg\mathsf{leP}\ y\ x \implies \mathsf{addP}\ x\ (\mathsf{subP}\ y\ x) = y$
$\vdash \forall x, y.\ \mathsf{multP}\ x\ y = \mathsf{multP}\ y\ x$
$\vdash \forall x, y, z.\ \mathsf{multP}\ (\mathsf{multP}\ x\ y)\ z = \mathsf{multP}\ x\ (\mathsf{multP}\ y\ z)$

PROVIDES

types: n
consts: zero le add neg sub mult (inject : p → n)
thms: $\vdash \forall x.\ \mathsf{le}\ x\ x$
$\vdash \forall x, y.\ \mathsf{le}\ x\ y \wedge \mathsf{le}\ y\ x \implies x = y$
$\vdash \forall x, y, z.\ \mathsf{le}\ x\ y \wedge \mathsf{le}\ y\ z \implies \mathsf{le}\ x\ z$
$\vdash \forall x, y.\ \mathsf{le}\ x\ y \vee \mathsf{le}\ y\ x$
$\vdash \forall x.\ \mathsf{add}\ \mathsf{zero}\ x = x$
$\vdash \forall x.\ \mathsf{add}\ x\ \mathsf{zero} = x$
$\vdash \forall x, y.\ \mathsf{add}\ x\ y = \mathsf{add}\ y\ x$
$\vdash \forall x, y, z.\ \mathsf{add}\ (\mathsf{add}\ x\ y)\ z = \mathsf{add}\ x\ (\mathsf{add}\ y\ z)$
$\vdash \forall x, y, z.\ \mathsf{add}\ x\ y = \mathsf{add}\ x\ z = (y = z)$
$\vdash \forall x, y, z.\ \mathsf{le}\ (\mathsf{add}\ x\ y)\ (\mathsf{add}\ x\ z) = \mathsf{le}\ y\ z$
$\vdash \forall x, x', y, y'.\ \mathsf{le}\ x\ x' \wedge \mathsf{le}\ y\ y' \implies \mathsf{le}\ (\mathsf{add}\ x\ y)\ (\mathsf{add}\ x'\ y')$
$\vdash \mathsf{neg}\ \mathsf{zero} = \mathsf{zero}$
$\vdash \forall x.\ \mathsf{neg}\ x = \mathsf{zero} = (x = \mathsf{zero})$
$\vdash \forall x.\ \mathsf{neg}\ (\mathsf{neg}\ x) = x$
$\vdash \forall x.\ \mathsf{add}\ x\ (\mathsf{neg}\ x) = \mathsf{zero}$
$\vdash \forall x.\ \mathsf{add}\ (\mathsf{neg}\ x)\ x = \mathsf{zero}$
$\vdash \forall x, y.\ \mathsf{sub}\ x\ y = \mathsf{add}\ x\ (\mathsf{neg}\ y)$
$\vdash \forall x, y.\ \mathsf{add}\ x\ (\mathsf{sub}\ y\ x) = y$
$\vdash \forall x.\ \mathsf{mult}\ \mathsf{zero}\ x = \mathsf{zero}$
$\vdash \forall x.\ \mathsf{mult}\ x\ \mathsf{zero} = \mathsf{zero}$
$\vdash \forall x, y.\ \mathsf{mult}\ x\ y = \mathsf{mult}\ y\ x$
$\vdash \forall x, y, z.\ \mathsf{mult}\ (\mathsf{mult}\ x\ y)\ z = \mathsf{mult}\ x\ (\mathsf{mult}\ y\ z)$
$\vdash \forall x, y.\ \mathsf{leP}\ x\ y = \mathsf{le}\ (\mathsf{inject}\ x)\ (\mathsf{inject}\ y)$
$\vdash \forall x, y.\ \mathsf{inject}\ (\mathsf{addP}\ x\ y) = \mathsf{add}\ (\mathsf{inject}\ x)\ (\mathsf{inject}\ y)$
$\vdash \forall x, y.\ \neg\mathsf{leP}\ x\ y \implies \mathsf{inject}\ (\mathsf{subP}\ x\ y) = \mathsf{sub}\ (\mathsf{inject}\ x)\ (\mathsf{inject}\ y)$
$\vdash \forall x, y.\ \mathsf{inject}\ (\mathsf{multP}\ x\ y) = \mathsf{mult}\ (\mathsf{inject}\ x)\ (\mathsf{inject}\ y)$
$\vdash \forall x.\ \neg(\mathsf{inject}\ x = \mathsf{zero})$
$\vdash \forall x, y.\ \neg(\mathsf{inject}\ x = \mathsf{neg}\ (\mathsf{inject}\ y))$
$\vdash \forall p.\ (\forall x.\ p\ (\mathsf{inject}\ x)) \wedge p\ \mathsf{zero} \wedge (\forall x.\ p\ (\mathsf{neg}\ (\mathsf{inject}\ x))) \implies \forall x.\ p\ x$

Figure 6. A general theory constructing a number system from its positive elements.

example theory	article (Kb)	comp. (Kb)	comp. ratio	gzip'ed article (Kb)	gzip'ed comp. (Kb)	comp. ratio
`example-ord-1`	3,167	1,250	61%	393	173	56%
`example-ord-2`	15,866	3,782	77%	1,971	521	74%
`example-pos-1`	74,912	15,756	79%	6,609	2,056	69%

Table 3. Benchmarking the compression algorithm on HOL Light example theories.

From this point of view, the most related work is the AWE project [2], which builds on the explicit proof terms in Isabelle [1]. Though tied to one theorem prover, it nevertheless focuses on the theory as the central concept, and has developed sophisticated mechanisms for theory interpretation based on rewriting proof terms. The present work differs from AWE by being theorem prover independent, and also by its technique of processing proofs one step at a time rather than requiring the whole proof to be in memory, which may allow it to scale up more effectively.

7. Future Work

There are further opportunities to compress theories in article format. For example, by renaming variables that are used in proofs but never appear in a summary theorem, it may be possible to generate more object sharing.

There is scope for future work in developing more theory operations, especially more sophisticated theory interpretations. For example, it would be useful to specialize constants to arbitrary terms (with no free variables or additional type variables), instead of simply renaming them to other constants.

Now that the package format has been developed, the hope is that the full software engineering benefits of package management can be applied to higher order logic, including searchable repositories of packages, and installation with automatic dependency resolution. The goal is to develop the `opentheory install` command!

8. Summary

This paper has presented the OpenTheory article format for packaging higher order logic theories. Articles can be effectively exported from and imported into theorem provers without explicit proof objects, and they carry around their own dependencies in the form of automatically derived summaries. Articles are nameless, and thus can be cleanly transplanted into a different theorem proving context, proving the dependencies using theorems in the local environment. Finally, the examples in the paper show how the restricted form of theory interpretations in articles can be employed to efficiently develop realistic theories.

Acknowledgments

The OpenTheory project was initiated in 2004 as a result of discussions between Rob Arthan and the author, and the work since then has been guided by feedback from many other people, including John Harrison, Rebekah Leslie, John Matthews, Michael Norrish and Konrad Slind.

References

[1] Stefan Berghofer and Tobias Nipkow. Proof terms for simply typed higher order logic. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics, 13th International Conference: TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 38–52. Springer, August 2000.

[2] Maksym Bortin, Einar Broch Johnsen, and Christoph Lüth. Structured formal development in Isabelle. *Nordic Journal of Computing*, 13:1–20, 2006.

[3] Duncan Coutts, Isaac Potoczny-Jones, and Don Stewart. Haskell: Batteries included. In Andy Gill, editor, *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 125–126. ACM, September 2008.

[4] William M. Farmer. The seven virtues of simple type theory. *Journal of Applied Logic*, 6:267–286, 2008.

[5] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11:213–248, 1993.

[6] Thomas C. Hales. Introduction to the Flyspeck project. In Thierry Coquand, Henri Lombardi, and Marie-Françoise Roy, editors, *Mathematics, Algorithms, Proofs*, number 05021 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[7] John Harrison. HOL light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer, 1996.

[8] John Harrison. *Theorem Proving with the Real Numbers (Distinguished dissertations)*. Springer, 1998.

[9] Joe Hurd. OpenTheory article format (version 3). Available for download at `http://gilith.com/research/opentheory/article.html`, December 2007.

[10] F. Kammüller. Modular reasoning in Isabelle. In David A. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction (CADE-17)*, volume 1831 of *Lecture Notes in Computer Science*. Springer, June 2000.

[11] D. J. King and R. D. Arthan. Development of practical verification tools. *ICL Systems Journal*, 11(1), May 1996.

[12] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*, pages 42–54. ACM, January 2006.

[13] David MacQueen. Modules for Standard ML. In Robert S. Boyer, Edward S. Schneider, and Jr. Guy L. Steele, editors, *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 198–207. ACM, August 1984.

[14] P. Naur and B. Randell, editors. *Software Engineering*. Scientific Affairs Division, NATO, October 1968.

[15] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[16] Michael Norrish and Konrad Slind. A thread of HOL development. *The Computer Journal*, 41(1):37–45, 2002.

[17] Steven Obua and Sebastian Skalberg. Importing HOL into Isabelle/HOL. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning, Third International Joint Conference (IJCAR 2006)*, volume 4130 of *Lecture Notes in Computer Science*, pages 298–302. Springer, August 2006.

[18] Piotr Rudnicki. An overview of the Mizar project. Notes to a talk at the workshop on Types for Proofs and Programs, June 1992.

[19] Markus Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99*, volume 1690 of *Lecture Notes in Computer Science*, pages 167–184. Springer, September 1999.

[20] W. Wong. Recording and checking HOL proofs. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 353–368. Springer, September 1995.