Composable Packages for Higher Order Logic Theories

Joe Hurd
Galois, Inc.
joe@galois.com

Abstract

Interactive theorem proving is tackling ever larger formalization and verification projects, and there is a critical need for theory engineering techniques to support these efforts. One such technique is effective package management, which has the potential to simplify the development of logical theories by precisely checking dependencies and promoting re-use. This paper introduces a domain-specific language for defining composable packages of higher order logic theories, which is designed to naturally handle the complex dependency structures that often arise in theory development. The package composition language functions as a module system for theories, and the paper presents a well-defined semantics for the supported operations. Preliminary tests of the package language and its toolset have been made by packaging the theories distributed with the HOL Light theorem prover. This experience is described, leading to some initial theory engineering discussion on the ideal properties of a reusable theory.

1 Introduction

Interactive theorem proving has grown from toy examples to major formalization and verification projects in mathematics and computer science. Recent examples include: the 20 man-year verification of the seL4 operating system kernel [17]; the CompCert project, which verified an optimizing compiler from a large subset of C to PowerPC assembly code [18]; and the Flyspeck project, which aims to mechanize a proof of the Kepler sphere-packing conjecture [10].

Just as the term software engineering was coined in 1968 [20] 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 (*"proving in the large"*). One software engineering technique that can be applied to proof development is effective package management. Modern operating systems bundle software into packages that carry around a list of their dependencies—these are checked at installation time to ensure that the system can properly support the package. The same techniques have been applied to manage Haskell packages [5], which allows much deeper checking at installation time by type checking new packages in the context of the packages they depend on. Extending this idea to logical theories offers the promise of complete dependency checking at installation time, by precisely matching the required theorems to previously installed packages and checking proofs to ensure consistency.

The goal of the OpenTheory project is to transfer the benefits of package management to aid the development of logical theories.[1] Making effective use of package management techniques in logical theory development will require new tools and techniques for at least the following tasks:

1. Designing *theory languages* portable across theorem prover implementations.
2. Discovering *design techniques* for reusable theories.
3. *Uploading*, *installing* and *upgrading* theory packages from online repositories.
4. Building a *standard theory library*.

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

The approach taken by the OpenTheory project is to uncover the underlying theory engineering issues by tackling the above tasks for a concrete case study logic: Church's simple theory of types [8], extended with Hindley-Milner style type variables (which in the interest of brevity will be simply referred to as *higher order logic* in this paper). Higher order logic is a good testing ground for refining theory engineering concepts for two reasons. Firstly, it is a simple logic, so the focus naturally shifts to properties of the theories rather than the underlying logic. Secondly, the logic is implemented by three interactive theorem provers with mature theory libraries formalizing a large collection of concepts from mathematics and computer science, providing source material to experiment with packaging schemes and dependency management.

The three theorem provers are HOL Light [11], HOL4 [22] and ProofPower [16], which are all descended from the LCF theorem prover [9] and share its basic design of building proof tools on top of a small trusted kernel implementing the primitive inferences of the logic. A 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. Each of the three theorem provers uses a different (and extensible) source language for high level proof steps, and there is currently no better way of transferring theories between them at the source level than manually translating the theorem statements and creating new high-level proofs for them. However, it is relatively easy as a one-off task to implement a set of proof rules in one theorem prover that implement the primitive inferences of another. To continue the compiler analogy, the hardware (logic) of the three theorem provers is the same, so even though they implement subtly different assembly languages (primitive inferences) they can simulate each other.[2] With these simulation rules in place, representing higher order logic theories at the level of primitive inferences is portable across theorem provers, and previous work introduced the *article* file format for this purpose [13, 14].

An article is a compact representation of a higher order logic theory, consisting of a formal proof that a set of theorems logically follow from a set of assumptions. The article format was designed to simplify theory import and export for theorem prover implementations, but these are just two of the operations that must be supported for a theory to function effectively as a reusable component in logical theory development. Viewing theories as modules in a programming language, a module system for theories would ideally support at least the following operations:

- Renaming type operators and constants in theories, either to avoid namespace clashes or to bind the arguments of a *functor theory*.
- Forming compound theories by satisfying the assumptions of one theory with the theorems of others.
- Restricting the exported theorems of a theory.

The main contribution of this paper is a language for constructing compound theories by combining basic articles, which supports all the above operations with a well-defined semantics. The theory language forms the core of a domain-specific language for defining composable packages of higher order logic theories, which is designed to naturally handle the complex dependency structures that often arise in theory development.

The language for defining composable packages of higher order logic theories is now stable, and tools for processing packages are included with the OpenTheory toolset.[3] Package tools exist for displaying meta-information, querying dependencies, pretty-printing assumptions and theorems, and compiling to

[2] The type class extension of Isabelle/HOL [21] makes it a later version of the hardware, so although it can simulate the assembly language of the others, they cannot simulate it.

[3] The OpenTheory toolset is available for download at `http://gilith.com/software/opentheory`

articles. Preliminary tests of the package language and its toolset have been made by packaging the theories distributed with the HOL Light theorem prover. This experience is described, including some initial discussion on what makes a reusable theory.

The remainder of the paper is structured as follows: Section 2 presents the language and semantics for constructing compound theories; Section 3 extends this to a complete language for defining composable packages; Section 4 discusses the experience of packaging HOL Light theories; and finally Sections 5–7 examine related work, summarize and consider future directions.

2 Constructing Compound Theories

A theory $\Gamma \triangleright \Delta$ of higher order logic consists of:

1. A set Γ of assumption sequents.
2. A set Δ of theorem sequents.
3. A formal proof that the theorems in Δ logically derive from the assumptions in Γ.

A higher order logic theory can be serialized using the article file format; this section shows how compound theories can be constructed by combining basic articles.

2.1 Type Operators and Constants

To promote clarity when reasoning about theory operations, it is sometimes helpful to be explicit about the type operators and constants that appear in a theory $\Gamma \triangleright \Delta$. Suppose the sequents in Γ (resp. Δ) contain references to the set t (resp. u) of type operators and the set c (resp. d) of constants. There will be naturally be a large overlap between the type operators and constants in Γ and Δ, but the ones that only appear in Δ are of special interest, being the *theory definitions*, so let $u' := u - t$ and $d' := d - c$. It is now possible to annotate the theory $\Gamma \triangleright \Delta$ with its type operators and constants by writing it as the formula

$$\forall t, c.\ \Gamma(t, c) \implies \exists u', d'.\ \Delta(t \cup u', c \cup d')\ .$$

This annotated view shows theories to be similar to functors in the ML module system [19], with the following interpretation: if a set of types t and values c that satisfy the signature Γ is provided as input, the functor will generate a set of types u' and values d' that satisfy the signature Δ. However there are two significant differences between higher order logic theories and Standard ML functors: the 'theorem prover' in Standard ML is the type checker, and so the signatures are restricted to be type judgments, whereas theories can specify arbitrary higher order logic properties; and the particular representation of the types u' and construction of the values d' can have a great effect on the performance of the resulting code, whereas for theories it little matters how the type operators u' and constants d' are constructed, so long as they satisfy their properties Δ.

The annotated view of theories is also useful for extracting the correct side conditions on theory operations. For example, the (left-biased) conjunction operation

$$(\Gamma_1 \triangleright \Delta_1) \wedge (\Gamma_2 \triangleright \Delta_2) \ = \ (\Gamma_1 \cup (\Gamma_2 - \Delta_1)) \triangleright (\Delta_1 \cup \Delta_2)$$

on theories is valid only when the two sets of theory definitions are disjoint, which can be verified by manipulating the formulas of the annotated view. At a critical point the proof requires a step of the form

$$(\exists x.\ P(x)) \wedge (\exists y.\ Q(y)) \iff \exists x, y.\ P(x) \wedge Q(y)$$

which is valid only if x and y are distinct. Without this side condition it would be possible to create the definition $\mathsf{c} = \mathsf{true}$ in one theory and the definition $\mathsf{c} = \mathsf{false}$ in another, conjoin the two theories, and then prove $\mathsf{true} = \mathsf{false}$.

2.2 Interpreting Theories

The ability to rename the type operators and constants in a theory is essential to bind a theory to a local context, apply it as a functor, or just avoid namespace clashes. An *interpretation* consists of a renaming function for type operators, and another one for constants. It is possible to apply an interpretation σ to a variety of different types of object t, but the application is always written $t\sigma$. There is an identity interpretation id that does nothing, and a composition operator $\cdot \circ \cdot$ that satisfies $t(\sigma \circ \rho) = (t\sigma)\rho$.

The annotated view of theories from Section 2.1 is again used to extract the side conditions from the theory interpretation operation:

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

Manipulating the formulas shows that this can succeed only if:

- the theory assumptions and the theory definitions are kept disjoint

$$t\sigma \cap u'\sigma = c\sigma \cap d'\sigma = \emptyset \;;$$

- and there are no collisions between the theory definitions

$$|u'\sigma| = |u'| \;\wedge\; |d'\sigma| = |d'| \;.$$

Apart from type operators and constants, the only other named objects in a higher order logic theory are type variables and term variables. The scope of both type and term variables is the sequent, since free type and term variables in sequents are implicitly universally quantified, and proof rules can be used to consistently rename them. The scope of type operators and constants is the theory, and interpretations can be used to consistently rename them. The end result is that theories are *nameless*: whether a theory can be applied in a context is oblivious to the particular choice of names that it contains.

2.3 Theory Language

Without further ado, here is the grammar for the theory language:

theory ← `article "`*filename*`";`
| { *theory** }
| `local` *theory* `in` *theory*
| `interpret` { *interpretation** } `in` *theory*
| `import` *package-instance* `;`

The `article` keyword is used to read a theory stored as an article file. The $\{\cdots\}$ grouping is used to sequentially compose theories, simulating the standard mode in a theorem prover where assumptions of theories in the sequence can be satisfied by previously proved theorems. The `local` keyword is simply sequential composition of two theories, except that the theorems of the local theory are not exported. The `interpret` keyword is used to interpret a theory with a renaming of some type operators and constants, as described in Section 2.2. Finally, the `import` keyword imports a previously loaded theory (in the form of a *package-instance*, which will be explained in Section 3).

To make this precise, suppose that a *theory* expression θ is evaluated in a context where the theorem set Φ is available to satisfy assumptions, and interpretation σ is in effect. The result of the evaluation,

written $[\![\theta]\!]_{\Phi,\sigma}$, satisfies the following set of structural recursion rules:

$$\begin{array}{rcl}
[\![\texttt{article "f";}]\!]_{\Phi,\sigma} & = & \mathsf{readArticle}\ \Phi\ \sigma\ f \\
[\![\{\ []\ \}]\!]_{\Phi,\sigma} & = & \emptyset \triangleright \emptyset \\
[\![\{\ \theta_1 :: \theta_2\ \}]\!]_{\Phi,\sigma} & = & \mathsf{let}\ \Gamma_1 \triangleright \Delta_1 = [\![\theta_1]\!]_{\Phi,\sigma}\ \mathsf{in} \\
 & & \mathsf{let}\ \Gamma_2 \triangleright \Delta_2 = [\![\{\ \theta_2\ \}]\!]_{\Phi\cup\Delta_1,\sigma}\ \mathsf{in} \\
 & & \Gamma_1 \cup \Gamma_2 \triangleright \Delta_1 \cup \Delta_2 \\
[\![\texttt{local}\ \theta_1\ \texttt{in}\ \theta_2]\!]_{\Phi,\sigma} & = & \mathsf{let}\ \Gamma_1 \triangleright \Delta_1 = [\![\theta_1]\!]_{\Phi,\sigma}\ \mathsf{in} \\
 & & \mathsf{let}\ \Gamma_2 \triangleright \Delta_2 = [\![\theta_2]\!]_{\Phi\cup\Delta_1,\sigma}\ \mathsf{in} \\
 & & \Gamma_1 \cup \Gamma_2 \triangleright \Delta_2 \\
[\![\texttt{interpret}\ \{\ \rho\ \}\ \texttt{in}\ \theta]\!]_{\Phi,\sigma} & = & [\![\theta]\!]_{\Phi,\sigma\circ\rho} \\
[\![\texttt{import p;}]\!]_{\Phi,\sigma} & = & \mathsf{importPackageInstance}\ p
\end{array}$$

Setting down the evaluation semantics helps to explain the design of the theory composition language. It crucially relies on a readArticle function that can read an article file containing a theory $\Gamma \triangleright \Delta$, and while running the proof:

- use the interpretation σ to rename type operators and constants that appear; and
- use the theorems in Φ to satisfy assumptions that are made.

If successful,[4] this will result in the theory $\Gamma\sigma - \Phi \triangleright \Delta\sigma$. Performing the theory interpretation and assumption satisfaction while running a proof complicates the implementation of the article reader, but there is no alternative for theorem provers in the LCF design: the strictly limited functionality of the logical kernel offers no support for theory operations after the fact.

The semantics of sequential composition $\{\ \theta_1 :: \theta_2\ \}$ is left-biased theory conjunction (discussed in Section 2.1), and the `local` version simply forgets the theorems of the local theory.

The `interpret` keyword is easily handled by composing the interpretations from the context and argument.

Finally, the `import` keyword pulls in a previously loaded theory $\Gamma \triangleright \Delta$. No proofs are run in an import operation, so the current context is ignored and the resulting theory is $\Gamma \triangleright \Delta$.

3 Packaging Theories

The previous section presented a language for constructing theories, and the logical next step is to package theories for distribution and installation. This section will build on the theory language to present a complete package language for theories.

3.1 Theory Dependencies

It is expected that theory packages will be browsed by users, and so a desirable feature is that they have a coherent subject. For example, a theory package defining the trigonometric functions (sin, cos, etc.) and proving the standard identities ($\forall\theta.\ \sin^2\theta + \cos^2\theta = 1$, etc.) would be easy for a user to understand, and therefore to know whether it will help in their theory development.

Unfortunately, the desire to create coherent packages can result in cycles in the package dependency graph. Figure 1 shows a simple example of how this can occur. Going anti-clockwise from the bottom left, here is the sequence of definitions and theorems:

[4] i.e., there are no name clashes as described in Section 2.2.

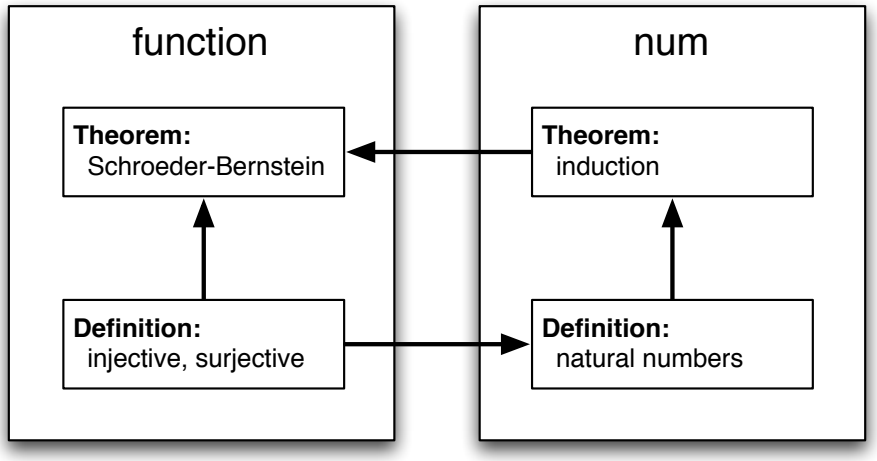

Figure 1: Example theory dependency graph.

1. The definition of what it means for a function to be injective and surjective (`function-def`).
2. The construction of the natural number type using the Axiom of Infinity, which refers to the injective and surjective properties (`num-def`).
3. Proving that mathematical induction holds for the type of natural numbers (`num-induction`).
4. Proving the Schroeder-Bernstein theorem, which states that if there is an injective function $\alpha \to \beta$, and if there is also a surjective function $\alpha \to \beta$, then there is a function $\alpha \to \beta$ that is both injective and surjective. The proof of this theorem uses mathematical induction (`schroeder-bernstein`).

The dependency between the definitions and theorems is shown with arrows in Figure 1. The most coherent split into theory packages is:

- a package called `function` containing `function-def` and `schroeder-bernstein`; and
- a package called `num` containing `num-def` and `num-induction`.

However, this results in a cyclic dependency between the two packages.

This tension between logical dependency and coherent theories occurs whenever a more complex theory is invoked to prove results of a simpler theory. For another example from the formalization of mathematics, the real numbers are constructed on top of the natural numbers, but the whole field of analytic number theory is based on the idea of using real analysis to prove theorems about the natural numbers.

Going back to the example, one solution would be to split the two packages into four small packages: `function-def`; `num`; `num-def`; and `schroeder-bernstein`. There are no cycles in the dependency graph between these small packages. However, usability is likely to suffer if users have to spend time looking for related packages. A user working with injective and surjective functions might have never heard of the Schroeder-Bernstein theorem, and so miss out on the opportunity to simplify a proof.

The solution adopted is to split theory packages to eliminate cycles in the dependency graph, but permit theory packages to require the existence of other *package instances* and combine them using the theory language. This allows the `function` and `num` packages to be defined as before, but without a cyclic dependency. The cost of obtaining cycle-free coherent packages is that there are now six packages

```
name: function
version: 1.0
description: Function theory

require function-def {
  package: function-def-1.0
}

require schroeder-bernstein {
  import: function-def
  package: schroeder-bernstein-1.0
}

theory {
  import function-def;
  import schroeder-bernstein;
}
```

Figure 2: The `function` package.

to maintain (both the original two and the split four), which may in time lead to package pollution with negative consequences for user browsing.

Figure 2 shows how the `function` package is expressed in the package language. A full description of the package language will be given in Sections 3.2–3.4, but it might be helpful to see an example first. The `require` blocks ensure that instances of the split packages are present, and binds the instances to names. The `import` constraint in the `schroeder-bernstein` block ensures that this package instance is dependent on the `function-def` instance. The `theory` block contains an expression in the theory language of Section 2.3 that combines the required package instances. In this case the `function` package simply re-exports the theorems of the required packages.

3.2 Package Meta-Data

At the top of a package there is a list of *tag*s that serve as package meta-data:

$$\textit{tag} \quad \leftarrow \quad \textit{name}\texttt{:}\ \textit{value}$$

The two most important tag names are `name` and `version`, the values for which together determine the full name of the package as *`name-version`*. For example, the full name of the example package in Figure 2 is `function-1.0`.

3.3 Package Instances

After the package meta-data there is a sequence of *package-instance-spec* blocks, which specify the package instances that are assumed to be present, and bind them to *package-instance* names.

Here is the grammar for a *package-instance-spec* block:

$$\begin{aligned} \textit{package-instance-spec} \quad \leftarrow \quad & \texttt{require}\ \textit{package-instance}\ \{ \\ & \quad \texttt{import:}\ \textit{package-instance}^* \\ & \quad \texttt{interpret:}\ \textit{interpretation}^* \\ & \quad \texttt{package:}\ \textit{package-name} \\ & \} \end{aligned}$$

The `package` constraint is the full name of the package that this package instance is derived from. The `interpret` constraints specify the interpretation that must be applied to the package in this instance, the `import` constraints list the package instances that must be used to satisfy the assumptions of this package instance. In summary, a list of *package-instance-spec*s specifies a connection graph between theory packages with particular interpretations.

In the example package in Figure 2, the two *package-instance-spec*s specify that two package instances deriving from `function-def-1.0` and `schroeder-bernstein-1.0` must already be present, bound to the *package-instance* names `function-def` and `schroeder-bernstein`. The absence of `interpret` constraints requires that the identity interpretation must be used to derive the package instances from the packages, and the `import` constraint requires that the theorems from the `function-def` instance are used to satisfy the assumptions of the `schroeder-bernstein` instance.

In a context where the theorem set Φ is available to satisfy assumptions, and interpretation σ is in effect, the concrete syntax for *package-instance-spec* evaluates to the theory

$$\bigcup \Gamma_i \cup \Gamma \triangleright \Delta$$

where:

- the `import` *package-instance* names are bound to the theories $\Gamma_i \triangleright \Delta_i$;
- the `interpret` rules are the interpretation ρ; and
- the `package` evaluates to the theory $\Gamma \triangleright \Delta$ in a context where the theorem set $\bigcup \Delta_i \cup \Phi$ is available to satisfy assumptions, and interpretation $\rho \circ \sigma$ is in effect (as will be described in Section 3.4).

3.4 Package Language

Putting everything together, here is the grammar for the language of theory packages:

$$\begin{array}{rcl} \textit{package} & \leftarrow & \textit{tag}^* \\ & & \textit{package-instance-spec}^* \\ & & \texttt{theory}\ \{\ \textit{theory}\ \} \end{array}$$

In a context where the theorem set Φ is available to satisfy assumptions, and interpretation σ is in effect, the concrete syntax for *package* is evaluated to a theory in two steps:

1. Use the same context to evaluate the *package-instance-spec*s and bind the resulting theories to *package-instance* names, as described in Section 3.3.
2. Use the same context to evaluate the theory expression in the `theory` block, as described in Section 2.3. The importPackageInstance function looks up its argument *package-instance* name in the binding created in the first step, and returns the corresponding theory.

4 Packaging HOL Light Theories

To test the expressivity of the theory package language and the practicality of its toolset, an experiment was performed to package the theorems distributed with the HOL Light theorem prover [11].[5] HOL Light provides a good test of a theory package language, because it has no (official) theory infrastructure. Instead there is an OCaml file `hol.ml`, which contains a long sequence of let bindings for theorems and

```
name: hol-light-trivia-one-def
version: 2009.8.24
description: HOL Light definition of the unit type.

theory { article "trivia-one-def.art"; }
```

Figure 3: A package wrapping a HOL Light theory chunk.

proof tools built upon the logical kernel. Executing the file proves all the theorems, some 6,336 of them, and the resulting proof consists of 766,421 primitive inferences (with maximal sub-proof sharing).

The starting point for a packaging HOL Light theorems is given by the structure of `hol.ml`, the body of which consists of a sequence of `loads "t.ml";;` directives, where each `t.ml` is an unofficial theory containing a coherent set of definitions and theorems (and associated proof tools). Previous work reported on converting each `t.ml` file into a proof article `t.art` [13], which can be lifted to a package as shown in Figure 3. This experiment used these packages, and goes further by splitting an initial prefix of the `t.ml` files into multiple proof article chunks to investigate the potential of using the package language to assemble them into reusable theory packages.

4.1 Splitting HOL Light Theories

Just looking at the theories that result from the `t.ml` files, the first obstacle to creating reusable theories comes from the use of *pro forma theorems*: special purpose theorems designed to speed up proof tools, and clearly not intended for human consumption. From a theory engineering perspective, pro forma theorems pollute a clean theory interface, but they are necessary to support any use of their associated proof tool in a later theory. A promising approach to a long term solution is the lazy theorem infrastructure of Boulton[4], where the pro forma theorems would be proved once inside each theory that used the associated proof tool; to avoid their pollution in the present experiment the pro forma theorems for each theory were split off into their own sub-theory.

In principle, the free form of OCaml bindings in HOL Light allows arbitrary intertwining of theory dependencies of the kind discussed in Section 3.1, but in fact this was rarely observed. In this limited experiment, there was only one occasion when the degree of intertwining prevented a natural splitting into theories. It occurred because all of the arithmetic operations are defined in terms of numerals, but the definition of the numerals used the addition operation, so after the numerals are defined the definition of addition is rewritten to use the numerals. This interdependency prevented a clean split into a theory of addition and a theory of numerals; the solution taken was to change the HOL Light source to define the numerals directly using primitive recursion, and then define addition using numerals.

4.2 Assembling Reusable Theories

Once the HOL Light theories have been split into chunks, the package language can be used to assemble them into reusable theories. For example, the package shown in Figure 3 covers the raw HOL Light theory chunk that constructs the unit type, and evaluates to the theory shown in Figure 4. This theory suffers from two main drawbacks from a reusability point of view:

- many of the assumptions are pro forma theorems specific to HOL Light proof tools; and

[5]This experiment used HOL Light version 2.20++, snapshot release on 24 August 2009. HOL Light is available for download at `http://www.cl.cam.ac.uk/~jrh13/hol-light`.

```
input-types: -> bool
input-consts: ! /\ = ? T select
assumed:
  |- T
  {.} |- (!) P
  {.} |- (?) P
  {..} |- p /\ q
  |- t = (t = T)
  |- (?) = \P. P ((select) P)
defined-types: unit
defined-consts: one one_ABS one_REP
thms:
  |- ?b. b
  |- one = select x. T
  |- (!a. one_ABS (one_REP a) = a) /\
     !r. r = (one_REP (one_ABS r) = r)
```

Figure 4: A raw HOL Light theory chunk.

```
name: unit-def
version: 1.0
description: Definition of the unit type

require hol-light-thm {
  package: hol-light-thm-2009.8.24
}

require hol-light-trivia-one-def {
  import: hol-light-thm
  package: hol-light-trivia-one-def-2009.8.24
}

require hol-light-trivia-one-alt {
  import: hol-light-thm
  import: hol-light-trivia-one-def
  package: hol-light-trivia-one-alt-2009.8.24
}

theory { import hol-light-trivia-one-alt; }
```

Figure 5: A reusable theory package.

- the theorems contain too much detail on the construction of the unit type, rather than its interface.

Suppose the proofs of the pro forma theorems are collected into a package called `hol-light-thm`, and a minimal theorem interface for the unit type is derived from its definition in a package called `hol-light-trivia-one-alt`. The package shown in Figure 5 assembles these HOL Light theory chunks into the reusable theory shown in Figure 6. This theory also defines the unit type and its single element, but contains no pro forma theorems in its assumptions, just standard definitions of boolean operators. Also, no details of the construction leak through to its theorems, just a minimal interface of its defining property.

```
input-types: -> bool
input-consts: ! /\ = ==> ? T select
assumed:
  |- !t. (\x. t x) = t
  |- T = ((\p. p) = \p. p)
  |- (!) = \P. P = \x. T
  |- (==>) = \p q. (p /\ q) = p
  |- !P x. P x ==> P ((select) P)
  |- (/\) = \p q. (\f. f p q) = \f. f T T
  |- (?) = \P. !q. (!x. P x ==> q) ==> q
defined-types: unit
defined-consts: one
thms:
  |- !v. v = one
```

Figure 6: A reusable theory.

5 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 [27]. More recently, Obua and Skalberg [23] instrumented HOL4 and HOL Light to export theories in XML format that could be imported into the Isabelle/HOL theorem prover. The present work differs from this line of proof recording work by its focus on the theory as the central concept, independent of any particular theorem prover implementation.

From this point of view, the most related work is the AWE project [3], which builds on the explicit proof terms in Isabelle [2]. 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.

Many theorem provers implement a theory infrastructure that offers some of the functionality of a module system. ProofPower has a sophisticated system for building and navigating a hierarchy of theories which contain both logical data and information for tools such as parsers, pretty printers and proof tools [16]. Locales offer the functionality of nested modules, and these were first implemented in the Isabelle theorem prover [15] at the granularity of theorems and later refined to structured proofs [1]. If theories are modules, then theory interpretations are functors, and these are implemented in the EVES [6], IMPS [7], PVS [24] and Specware [26] theorem provers. The novel contribution of the present work is demonstrating how a natural collection of theory operators can be soundly integrated with a higher order logic theorem prover in the LCF design.

Another approach to higher order logic theory operations is to extend the logic so that theories can be directly represented with theorems [25, 12]. The goal of the present work is to implement a theory infrastructure on top of the existing logic, but extending the logic has the significant advantage of supporting theory operations without replaying proofs.

6 Summary

This paper introduced a domain-specific language for defining composable packages of higher order logic theories, designed to support the development of coherent theory packages with cycle-free dependencies.

The package composition language functions as a module system for theories, with a well-defined semantics for the supported operations.

Preliminary tests packaging the theories distributed with the HOL Light theorem prover gave promising results for expressivity of the package language and practicality of the toolset on a real-world set of theories.

The development of the package composition language and the experimentation work revealed some desirable properties of a reusable theory package:

- a clear topic (e.g., trigonometric functions);
- assumptions that are satisfied by the theorems of other reusable theory packages;
- a carefully chosen set of theorems, presenting an abstract interface to the theory (hiding construction details).

Although derived from experimenting with a particular case study of higher order logic theories, these properties are structural and thus apply to a general class of logical theories.

7 Future Work

There is scope for future work in developing and extending the theory operations. For example, theories would have more power as functors if constants could be interpreted as arbitrary terms (with no free variables or additional type variables), instead of simply renaming them to other constants.

It would also be interesting to investigate design techniques for package sets. For example, does it make sense to put theorem prover specific pro forma theorems into a separate package, or should pro forma theorems be proved as needed in proof articles? This is the design choice between dynamic and static linking of theories. For an individual theory it makes sense to statically link to eliminate unused pro forma theorems: for the reusable unit theory in Figure 5, statically linking `hol-light-thm` results in a proof requiring 585 primitive inferences, while dynamically linking requires 9,055. However, for a set of packages requiring the same pro forma theorems, dynamically linking might result in smaller proofs and thus more efficient package management.

Now that the package format has been developed, the hope is that the full benefits of package management can be transferred to theory development, including searchable repositories of packages, and installation with automatic dependency resolution. The next challenge is to build enough package management infrastructure for people to contribute to building a standard library of reusable theory packages.

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. The paper was greatly improved by comments from the anonymous referees of the VERIFY workshop.

References

[1] Clemens Ballarin. Locales and locale expressions in Isabelle/Isar. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs: Third International Workshop, TYPES 2003*, volume 3085 of *Lecture Notes in Computer Science*, pages 34–50. Springer, May 2004.

[2] 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.

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

[4] Richard J. Boulton. Lazy techniques for fully expansive theorem proving. *Formal Methods in System Design*, 3(1/2):25–47, August 1993.

[5] 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.

[6] D. Craigen, S. Kromodimoeljo, I. Meisels, B. Pase, and M. Saaltink. EVES: An overview. Technical Report CP-91-5402-43, ORA Corporation, 1991.

[7] William M. Farmer. Theory interpretation in simple type theory. In Jan Heering, Karl Meinke, Bernhard Möller, and Tobias Nipkow, editors, *Higher-Order Algebra, Logic, and Term Rewriting, First International Workshop (HOA '93)*, volume 816 of *Lecture Notes in Computer Science*, pages 96–123. Springer, 1994.

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

[9] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[10] 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.

[11] 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.

[12] Peter V. Homeier. The HOL-Omega logic. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference (TPHOLs 2009)*, volume 5674 of *Lecture Notes in Computer Science*, pages 244–259. Springer, August 2009.

[13] Joe Hurd. OpenTheory: Package management for higher order logic theories. In Gabriel Dos Reis and Laurent Théry, editors, *PLMMS '09: Proceedings of the ACM SIGSAM 2009 International Workshop on Programming Languages for Mechanized Mathematics Systems*, pages 31–37. ACM, August 2009.

[14] Joe Hurd. OpenTheory article format (version 4). Available for download at `http://gilith.com/research/opentheory/article.html`, March 2010.

[15] 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.

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

[17] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220. ACM, October 2009.

[18] 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.

[19] 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.

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

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

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

[23] 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.

[24] Sam Owre and N. Shankar. Theory interpretations in PVS. Technical Report SRI-CSL-01-01, SRI International, April 2001.

[25] Norbert Völker. HOL2P - A system of classical higher order logic with second order polymorphism. In Klaus Schneider and Jens Brandt, editors, *20th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 334–351. Springer, September 2007.

[26] Stephen Westfold. Integrating Isabelle/HOL with Specware. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings*, number 364/07 in Department of Computer Science, University of Kaiserslautern Technical Reports, August 2007.

[27] 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.