Composable Packages for Higher Order Logic Theories

Joe Hurd

Galois, Inc.
joe@galois.com

VERIFY Workshop
Wednesday 21 July 2010
Interactive theorem proving is growing up.

- The FlySpeck project is driving the HOL Light theorem prover towards a formal proof of the Kepler sphere-packing conjecture.
- The seL4 project recently completed a 20 man-year verification of an operating system kernel in the Isabelle theorem prover.

There is a need for **theory engineering** techniques to support these major verification efforts.

- Theory engineering is to proving as software engineering is to programming.
- “Proving in the large.”
The goal of the OpenTheory project is to transfer the benefits of package management to logical theories.\(^1\)

The initial case study for the project is Church’s simple theory of types, extended with Hindley-Milner style type variables.

- The logic implemented by HOL4, HOL Light and ProofPower.

By focusing on a concrete case study we aim to investigate the issues surrounding:

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

---

\(^1\)OpenTheory was started in 2004 with Rob Arthan.
A theory $\Gamma \vdash \Delta$ of higher order logic consists of:

1. A set $\Gamma$ of assumption sequents.
2. A set $\Delta$ of theorem sequents.
3. A formal proof that the theorems in $\Delta$ logically derive from the assumptions in $\Gamma$.

Theories (including their proofs) can be directly represented as OpenTheory article files.

- A format designed to simplify theory import and export for theorem prover implementations.

This talk will present a language for building up from article files to theory packages.

- We’ll see toy case studies that demonstrate the concepts, but the true test will be whether it scales up—watch this space!
Note that both the input assumptions and output theorems of a theory are sequent sets.

We can therefore connect the output theorems of one theory to satisfy the input assumptions of another:

\[ A \cup B \cup C_{IN} \triangleright S \cup C_{OUT} \]
Theory Interpretations

- A theory $\Gamma \triangleright \Delta$ can be instantiated in any context where the assumptions $\Gamma$ hold. This is called theory interpretation.

**Example:** The theory

$$\{\vdash id = \lambda x. x\} \triangleright \{\vdash \forall x. id \; x = x\}$$

can be applied in any context with a constant id having the assumed property.

- Constants and type operators can be consistently renamed

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

allowing theories to be instantiated in even more contexts.
What Can Go Wrong?

- When connecting together theories, the connection graph must not contain any loops!
  - Theories are representations of proofs, which are directed *acyclic* graphs.
  - In this aspect proofs are more like combinational circuits than programs.
- A set of theorems must not have incompatible definitions for the same constant or type operator.
  - Example: The two theories
    \[
    \emptyset \triangleright \{\vdash c = 0\} \quad \text{and} \quad \emptyset \triangleright \{\vdash c = 1\}
    \]
    are individually fine, but must never be imported into the same context.
A Language for Theories

The following theory language allows article files and theory packages to be combined into a new theory:

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

Incompatible definition clashes are prevented by:

- Limiting the scope of contexts using the `local` construct.
- Renaming constant and type operators using `interpret` blocks.

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

theory { article "hol-light-unit-def.art"; }
Theory Package Example


input-types: \( \rightarrow \) bool
input-consts: ! /\ = ? T select
assumed:
\[
\begin{align*}
|- \; T \\
\{.\} \; |- \; (!) \; P \\
\{.\} \; |- \; (?) \; P \\
\{..\} \; |- \; p \; /\ \; q \\
|- \; t \; = \; (t \; = \; T) \\
|- \; (?) \; = \; \backslash P. \; P \; ((select) \; P)
\end{align*}
\]
defined-types: unit
defined-consts: one one_ABS one_REP
thms:
\[
\begin{align*}
|- \; ?b. \; b \\
|- \; one \; = \; select \; x. \; T \\
|- \; (!a. \; one_ABS \; (one_REP \; a) \; = \; a) \; /\ \\
\quad \; !r. \; r \; = \; (one_REP \; (one_ABS \; r) \; = \; r)
\end{align*}
\]
Theory Package Design

- **Well-designed** theory packages have:
  1. A clear **topic**.
     - Example: Trigonometric functions.
  2. A simple set of **assumptions**.
     - Satisfied by well-designed packages.
  3. A carefully chosen set of **theorems**.
     - No junk.
     - A minimal interface if the package makes definitions.
  4. **No axioms**.
     - No assumptions about defined constants>Type operators.

- **Theory Engineering Challenge**: Construct a standard library of well-designed theory packages, available to all the HOL theorem prover implementations.
**Problem:** Complex theory dependencies can result in cycles in the package dependency graph.

- **Definition:** injective, surjective
- **Definition:** natural numbers
- **Theorem:** Schroeder-Bernstein
- **Theorem:** induction

**Solution:** Permit compilation theory packages which contain previously loaded theory packages.
Theory Package Instances

• An imported *package-instance* refers to a required theory package, specified as a *package-instance-spec*:

  \[
  \text{package-instance-spec} \leftarrow \text{require package-instance} \{ \\
  \quad \text{import: package-instance}^* \\
  \quad \text{interpret: interpretation}^* \\
  \quad \text{package: package-name} \\
  \}
  \]

• A list of *package-instance-specs* specify a connection graph between theory packages.

• Each *package-instance-spec* may only import earlier *package-instance-specs*, to ensure the absence of loops.
Theory Packages

- We can now define the grammar for theory packages:

  \[
  \text{package} \leftarrow \text{tag}\ast \\
  \text{package-instance-spec}\ast \\
  \text{theory} \left\{ \text{theory} \right\}
  \]

- Tags are package meta-data:

  \[
  \text{tag} \leftarrow \text{name: value}
  \]
Theory Package Example II

**Theory Package (unit-def-1.0)**

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

require hol-light-aux {
}

require hol-light-unit-def {
    import: hol-light-aux
}

require hol-light-unit-alt {
    import: hol-light-aux
    import: hol-light-unit-def
}

theory { import hol-light-unit-alt; }

Joe Hurd  
Composable Packages for Higher Order Logic Theories
Theory Package Example II

Theory Package Summary (unit-def-1.0)

input-types: \rightarrow 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
Symbol Tables Considered Harmful

- To make it easy to reason about theory package instances, we would like package instantiation to be a pure function

\[ \text{package-instance-spec} \rightarrow \Gamma \triangleright \Delta \]

- Possible because the package management tool implements a purely functional logical kernel (an idea of Freek Wiedijk).

- Constants and type operators contain their definitions, instead of being inserted in a symbol table, so definitions are referentially transparent:

\[
\begin{align*}
\text{(let } c \equiv \text{define } \phi \text{ in } f \ c \ c) & \equiv (f \ (\text{define } \phi) \ (\text{define } \phi)) \\
\end{align*}
\]
Referential transparency means there is no difference in functionality between instantiating a theory package multiple times in the same way or instantiating it once and reusing.

However, there will likely be a big difference in performance (article files are measured in megabytes).

Challenge: Detecting when two `package-instance-specs` would result in the same theory.

The logical kernel similarly aims to share subterms as much as possible, in computing free variables, substitutions, etc.
This talk presented a language for combining and packaging theories.

The next challenge: build the package management infrastructure for people to contribute to building a standard library of theories.

The project web page:

http://gilith.com/research/opentheory
The concrete syntax for `package-instance-spec` evaluates to
the theory
\[ \bigcup \Gamma_i \cup \left( \Gamma \sigma - \bigcup \Delta_i \right) \triangleright \Delta \sigma \]
where:
- the imported `package-instance-specs` evaluate to \( \Gamma_i \triangleright \Delta_i \);
- the *interpretation* rules are the renaming \( \sigma \); and
- the *package-name* is the theory \( \Gamma \triangleright \Delta \).
Here is how the concrete syntax for theory is evaluated in a context with theorems $\Phi$ and renaming $\sigma$:

\[
\begin{align*}
\text{[article } "[\Gamma \triangleright \Delta]";\text{]}_{\Phi,\sigma} &= \Gamma \sigma - \Phi \triangleright \Delta \sigma \\
\text{[} \{ [] \} \text{]}_{\Phi,\sigma} &= \emptyset \triangleright \emptyset \\
\text{[} \{ \theta_1 :: \theta_2 \} \text{]}_{\Phi,\sigma} &= \text{let } \Gamma_1 \triangleright \Delta_1 = [\theta_1]_{\Phi,\sigma} \text{ in} \\
&\quad \text{let } \Gamma_2 \triangleright \Delta_2 = [\{ \theta_2 \}]_{\Phi \cup \Delta_1,\sigma} \text{ in} \\
&\quad \Gamma_1 \cup \Gamma_2 \triangleright \Delta_1 \cup \Delta_2 \\
\text{[local } \theta_1 \text{ in } \theta_2\text{]}_{\Phi,\sigma} &= \text{let } \Gamma_1 \triangleright \Delta_1 = [\theta_1]_{\Phi,\sigma} \text{ in} \\
&\quad \text{let } \Gamma_2 \triangleright \Delta_2 = [\theta_2]_{\Phi \cup \Delta_1,\sigma} \text{ in} \\
&\quad \Gamma_1 \cup \Gamma_2 \triangleright \Delta_2 \\
\text{[interpret } \{ \rho \} \text{ in } \theta\text{]}_{\Phi,\sigma} &= [\theta]_{\Phi,\sigma \circ \rho} \\
\text{[import } [\Gamma \triangleright \Delta]\text{]}_{\Phi,\sigma} &= \Gamma \triangleright \Delta
\end{align*}
\]

Note that importing a package-instance ignores the theory context; its context is fixed by the package-instance-spec.