Packaging Theories of Higher Order Logic

Joe Hurd

Galois, Inc.
joe@galois.com

Theory Engineering Workshop
Tuesday 9 February 2010

galois

Talk Plan

galois

Motivation

- Interactive theorem proving is growing up.
- It has moved beyond toy examples of mathematics and program verification.
  - The FlySpeck project is driving the HOL Light theorem prover towards a formal proof of the Kepler sphere-packing conjecture.
  - The CompCert project used the Coq theorem prover to verify an optimizing compiler from a large subset of C to PowerPC assembly code.
- 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 OpenTheory Project

- The OpenTheory project aims to apply software engineering principles to the development of higher order logic 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.
  - Uploading, installing and upgrading theory packages from online repositories.
  - Discovering design techniques for reusable theories.
  - Building a standard library of higher order logic theories.

[1] OpenTheory was started in 2004 with Rob Arthan.

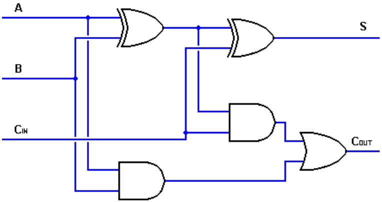
Theory Definition

- A theory $\Gamma \vdash \Delta$ 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.
- Theories 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.

galois

Connecting Theories

- 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:



- In this example, some basic theories have been connected together to produce the compound theory

$$A \cup B \cup C_{\mathrm{IN}} \vdash S \cup C_{\mathrm{OUT}} .$$

galois

Theory Interpretations

- A theory $\Gamma \vdash \Delta$ can be applied in any context where the assumptions Γ hold. This is called theory interpretation.
- Example: The theory

$$\{\mathrm{id} = \lambda x.\ x\} \vdash \{\forall x.\ \mathrm{id}\ x = x\}$$

  can be applied in any context with a constant id having the assumed property.
- Constants and type operators can be consistently renamed

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

  allowing theories to be applied 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 way 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

$$\{\} \vdash \{c = 0\} \qquad \text{and} \qquad \{\} \vdash \{c = 1\}$$

    are individually fine, but must never be imported into the same context.

galois

A Language for Theories

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

$$
\begin{array}{lcl}
\textit{theory} & \leftarrow & \texttt{article "}\textit{filename}\texttt{";} \\
& | & \{\ \textit{theory*}\ \} \\
& | & \texttt{local}\ \textit{theory}\ \texttt{in}\ \textit{theory} \\
& | & \texttt{interpret}\ \{\ \textit{interpretation*}\ \}\ \texttt{in}\ \textit{theory} \\
& | & \texttt{import}\ \textit{package-instance}\texttt{;}
\end{array}
$$

- Incompatible definition clashes are prevented by:
  - Limiting the scope of contexts using the `local` construct.
  - Renaming constant and type operators using `interpret` blocks.

Theory Package Instances

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

```
package-instance-spec  ←  require package-instance {
                             import: package-instance*
                             interpret: interpretation*
                             package: package-name
                           }
```

- A list of *package-instance-spec*s specify a connection graph between theory packages.
- Each *package-instance-spec* may only import earlier *package-instance-spec*s, to ensure the absence of loops.

Theory Packages

- We can now define the grammar for theory packages:

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

- Tags are package meta-data:

$$\textit{tag} \;\leftarrow\; \textit{name} \, \texttt{:} \; \textit{value}$$

Theory Package Example

Theory Package (hol-light-trivia-one-def-2009.8.24)

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

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

galois

Theory Package Example

Theory Package Summary (hol-light-trivia-one-def-2009.8.24)

```
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)
```

Theory Package Design

- Well-designed theory packages have:
  - a clear topic (e.g., trigonometric functions);
  - a simple set of assumptions (i.e., satisfied by standard packages);
  - a carefully chosen set of theorems (no junk, and a minimal interface if the package makes definitions);
  - and it should go without saying: no axioms!
- Theory Engineering Challenge: Construct a standard library of well-designed theory packages, available to all the theorem prover implementations.

galois

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-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; }
```

Theory Package Example II

Theory Package Summary (unit-def-1.0)

```
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
```

galois

Symbol Tables Considered Harmful

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

$$\textit{package-instance-spec} \rightarrow \Gamma \vdash \Delta \ .$$

- 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:

$$(\mathrm{let}\ c = \mathrm{define}\ \phi\ \mathrm{in}\ f\ c\ c) \ \equiv\ (f\ (\mathrm{define}\ \phi)\ (\mathrm{define}\ \phi))$$

Efficient Sharing

- 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-spec*s would result in the same theory.
- The logical kernel similarly aims to share subterms as much as possible, in computing free variables, substitutions, etc.

Summary

- 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`

galois

Package Instance Semantics

- The concrete syntax for *package-instance-spec* evaluates to the theory

$$\bigcup \Gamma_i \cup \left(\Gamma\sigma - \bigcup \Delta_i \right) \vdash \Delta\sigma$$

  where:
  - the imported *package-instance-spec*s evaluate to $\Gamma_i \vdash \Delta_i$;
  - the *interpretation* rules are the renaming σ; and
  - the *package-name* is the theory $\Gamma \vdash \Delta$.

Theory Semantics

- Here is how the concrete syntax for *theory* is evaluated in a context with theorems Φ and renaming σ:

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

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