OpenTheory

Package Management for Higher Order Logic Theories

Joe Hurd

Galois, Inc.
joe@galois.com

PLMMS 2009
Friday 21 August 2009

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.
    - *"Proving in the large."*

galois

Theory Engineering

- Think of a theory as a module in a weird programming language that implements a set of theorems:

example	**module**	$\sim$	**theory**	**example**
$\alpha \to \alpha$	type	$\sim$	theorem	$\vdash t = t$
$\lambda x.\ x$	value	$\sim$	proof	refl t

- Theory engineering is to proving as software engineering is to programming.

Software Engineering for Theories

An incomplete list of software engineering techniques applicable to the world of theories:

- **Standards:** Programming languages, basis libraries.
- **Abstraction:** Module systems to manage the namespace and promote reuse.
- **Multi-Language:** Tight/efficient (e.g., FFIs) to loose/flexible (e.g., SOAs).
- **Distribution:** Package repos with dependency tracking and automatic installation.

The OpenTheory Project

- The goal of the OpenTheory project is to apply software engineering principles to theories of higher order logic.
- The initial case study for the project is Church's simple theory of types, extended with Hindley-Milner polymorphism.
  - The logic implemented by HOL4, HOL Light and ProofPower.
- By focusing on a concrete case study we aim to investigate the issues surrounding:
  - Exchanging theories between theorem prover implementations.
  - Building a common library of higher order logic theories.
  - Discovering design techniques for theories that compose well.
  - Installing and upgrading theories while respecting their dependencies.

OpenTheory Articles

- A theory of higher order logic consists of:
    1. An import list of theorems Γ that the theory requires.
    2. An export list of theorems Δ that the theory provides.
    3. A formal proof $\Gamma \vdash \Delta$ that the theorems in Δ logically derive from the theorems in Γ.
- This talk will introduce the OpenTheory article file format for higher order logic theories.
- This is a standards-based approach to theories, to:
    - enable simple import and export between theorem prover implementations;
    - reduce storage requirements by compressing theories; and
    - think about composability of theories.

Tactic Proof Scripts

Porting theories between higher order logic theorem provers is currently a painful process of editing scripts that call proof tactics:

Code (Typical HOL Light tactic script proof)

```
let NEG_IS_ZERO = prove
    (`!x. neg x = Zero <=> x = Zero`,
     MATCH_MP_TAC N_INDUCT THEN
     REWRITE_TAC [neg_def] THEN
     MESON_TAC [N_DISTINCT]);;
```

Difficulty: Every theorem prover implements a subtley different set of tactics, the behaviour of which evolves across versions.

galois

Theorem Provers in the LCF Design

- Higher order logic theorem provers are just functional programs, where one of the modules is the logical kernel:

Code (The opentheory logical kernel)

```
type thm

(* refl t yields the theorem |- t = t *)
val refl : Term.term -> thm

[...10 other primitive inferences...]
```

- **Key Idea:** The `thm` type is abstract, so the only way to create a theorem is to use the primitive inferences of the logic.
- Tactics and other proof procedures must eventually expand to primitive inferences.

galois

Compiled Theories

- **Approach:** Instead of storing the source tactic script, store a compiled version of the theory by fully expanding the tactics to a primitive inference proof.
- **Benefit:** The logic will never change, so the compiled theories will never suffer from bit rot.
  - Whereas tactic scripts can break every time the tactics change.
- **Benefit:** The compiled proof need only store the inferences that contribute to the proof.
  - Whereas tactic scripts often explore many dead ends before finding a valid proof.
- **Drawback:** Once the theory has been compiled to a proof, it is difficult to change it.
  - So theories should be compiled only when they are stable enough to be archived.

Representing Proofs

- Not all higher order theorem provers build explicit proof objects for theorems.
- However, every tactic in the theorem prover is a function that calls lower-level tactics, all the way down to the primitive inference functions in the logical kernel.
- Thus the proof of a theorem in a higher order logic theorem prover can be represented as a call tree in a functional programming language.
- The OpenTheory article format is a direct representation of this call tree.

Proofs as Stack-Based Programs

- Articles represent call trees in functional programming languages as programs in a stack-based language.
- The theorem prover interprets this stack-based program, and simulates the primitive inference calls that are described by the stack-based program.
- When the theorem prover has finished interpreting the program, it will have simulated the entire proof of the theorems exported by the article.
- The stack-based program representation of proofs is easy to read, and easy to generate by instrumenting the inference functions in the theorem prover.

Stack Operations

- Articles are programs in a stack-based language.
- They are a sequence of commands, one per line.
- Most commands build up data objects to be used as function arguments or return values.

Definition (The “var” article command)

```
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
```

galois

Article Data Objects

Different kinds of data appear in call trees representing proofs, and these are defined in the article format.

Definition (Article data objects)

```
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 name        (* A special object marking a
                            function call                   *)
```

galois

Call Stack Operations

Definition (The "call" and "return" article commands)

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

    Stack:  Before:  Oname n :: i :: stack
            After:   i :: 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
```

galois

Article Exports

In addition to the stack, programs reading articles also maintain a list of theorems that will be exported from theory.

Definition (The "save" article command)

```
save
    Pop a theorem th; add th to the list of
    theorems that the article will export.

    Stack:  Before:  Othm th :: stack
            After:   stack

    Export list:  Before:  saved
                  After:   saved @ [th]
```

galois

Constructing Theorems

The `thm` command constructs a theorem with given hypotheses and conclusion.

Definition (The "thm" article command)

```
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
```

But wait! Theorems can't be constructed from their hypotheses and conclusion, they must be proved using primitive inferences. What's going on?

galois

Constructing Theorems (The Real Story)

- The `thm` just gives the specification for the theorem to be constructed—it doesn't say how it should be proved.
- Theorems are proved by the following methods (in order of preference):
  1. The theorem might already be on the export list of the theory.
  2. The current function might be a primitive inference rule, in which case the result theorem is proved by simulating the inference using the input arguments.
  3. The theorem might be inside a data object on the stack.
  4. If none of the previous rules apply, assert the theorem as an axiom and add it to the import list of the article.

The Dictionary

- In addition to the stack and the export list, programs reading articles also maintain a dictionary mapping integers to data objects.
- Data objects need only be constructed once, saved in the dictionary and then used multiple times.
- Without the dictionary, data objects with a great deal of memory sharing could expand exponentially in articles.

Adding to the Dictionary

Definition (The "def" article command)

```
def
    Pop a number k; peek an object x; update the
    dictionary so that key k maps to object x.

    Stack:  Before:  Oint k :: x :: stack
            After:   x :: stack

    Dictionary:  Before:  dict
                 After:   dict[k |-> x]
```

galois

Reading the Dictionary

Definition (The “ref” article command)

```
ref
    Pop a number k; look up key k in the dictionary
    to get an object x; push the object x.

    Stack:  Before:  Oint k :: stack
            After:  dict[k] :: stack

    Dictionary:  Before:  dict
                 After:   dict
```

Removing from the Dictionary

Definition (The "remove" article command)

```
remove
    Pop a number k; look up key k in the dictionary
    to get an object x; push the object x; delete
    the entry for key k from the dictionary.

    Stack:  Before:  Oint k :: stack
            After:   dict[k] :: stack

    Dictionary:  Before:  dict
                 After:   dict[entry k deleted]
```

Generating Articles from HOL Light

- We instrumented HOL Light v2.20 to emit articles for each of the theory files in the distribution.
- Each primitive inference (and selected other functions) generates `call` and `return` article commands with the argument and return values.
  - Exceptions are trapped and an `0error` return value is generated, and then the exception is re-raised.
- The theorems left on the stack are treated as the export list of the article.
- For each article a dictionary is maintained of all types and terms constructed.

galois

HOL Light Articles

HOL Light theory	article (Kb)	gzip'ed article (Kb)
num	1,820	227
arith	27,469	2,884
wf	29,277	3,222
calc_num	3,922	374
normalizer	2,845	300
grobner	2,417	257
ind-types	10,625	1,274
list	12,368	1,485
realax	23,628	2,519
calc_int	2,844	314
realarith	16,275	1,326
real	30,031	3,179
calc_rat	2,555	289
int	40,617	3,465
sets	168,586	17,514
iter	207,324	17,557
cart	20,351	2,076
define	82,185	8,157

Compressing Articles

- The articles generated by HOL Light are compressed by the following post-processing steps:
  1. Adding explicit `save` commands to the exported theorems, instead of leaving them on the stack.
  2. Not adding data objects to the dictionary that are only used once.
  3. Removing data objects from the dictionary on their last use.
  4. Eliminating all function calls where the result does not contribute to the exported theorems.
- Trick: By storing dependency pointers with each data object, the garbage collector takes care of dead inference elimination automatically as the article is read.

galois

Compressing the HOL Light Articles

HOL Light theory	article (Kb)	comp. (Kb)	comp. ratio	gzip'ed article (Kb)	gzip'ed comp. (Kb)	comp. ratio
num	1,820	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	2,555	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%

HOL Light Article Summary

Concatenating all the HOL Light theories in turn generates an article exporting 129,888 theorems, and depending on 3 axioms:

Axioms (The HOL Light axioms)

types: *bool fun ind*

consts: $\forall \; \wedge \; = \; \Longrightarrow \; \exists \; \mathit{ONE\_ONE} \; \mathit{ONTO} \; \mathit{select} \; \neg$

thms: $\vdash \forall t.\ (\lambda x.\ t\ x) = t$

$\vdash \forall P, x.\ P\ x \Longrightarrow P\ (\mathit{select}\ P)$

$\vdash \exists f.\ \mathit{ONE\_ONE}\ f \wedge \neg \mathit{ONTO}\ f$

galois

Article Summaries

- Until now we have been focused on the details of the proof format.
- Now let us focus on the interface to the article, called summaries, $\Gamma \vdash \Delta$:
  - Γ: The set of axioms that the theory depends on.
  - Δ: The set of theorems that the theory exports.
- Reducing the export set is always safe:

$$\mathrm{filter}_{\Delta'} \, (\Gamma \vdash \Delta) \;=\; \Gamma \vdash (\Delta \cap \Delta')$$

- Also, stack-based languages are concatenative:

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

Mapping Constant Names

Definition (The "const" article command)

```
const
    Pop a type ty; pop a name n; push a constant
    with name (interpret_const_name n) and type ty.

    Stack:  Before:  Otype ty :: Oname n :: stack
            After:   Oterm (mk_const (n',ty)) :: stack
                     where n' = interpret_const_name n
```

The `interpret_const_name` function is present to handle the situation where theorem provers have given the same constant different names.

galois

Theory Interpretations

- The interpret_const_name and interpret_type_name functions can be used creatively to simulate theory interpretations.
- The same article can be re-run with different interpretations to bind the dependencies to different theorems in the local context, and generate different exports.
- This provides a limited theory substitution operator.

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

Theory Operations

- We have presented three theory operations:
  1. reducing the exported theorems;
  2. concatenation;
  3. interpreting constant and type names.
- Theory Engineering Challenge: Design theories that can be applied in many contexts using the above operations.
- From this perspective, theories are like ML functors, which map modules to modules:

ML module	$\sim$	**HOL theory**
types	$\sim$	types
values	$\sim$	constants
type judgements	$\sim$	theorems
implementation	$\sim$	proof

galois

Example I

Code (A Haskell type class instance)

```
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
```

What's missing here?

Missing Dependency: Require <= to be a total order on elements.

Missing Export: Can guarantee that <= is a total order on elements.

Example I — Adding Properties to Type Classes

Create a theory containing an uninterpreted type `T` and constant `cmp`, and an axiom that `cmp` is a total order over `T`.

Axioms (Type class example theory)

```
types:
  T
consts:
  cmp total_order
thms:
  |- total_order cmp
```

When the theory is applied, the type `T` and constant `cmp` will be interpreted to a concrete type and total order.

Example I — Adding Properties to Type Classes

Theory (Type class example theory)

```
consts:
  cmp_list
thms:
  |- cmp_list NIL l2 = T /\
     cmp_list (CONS h1 t1) NIL = F /\
     cmp_list (CONS h1 t1) (CONS h2 t2) =
       if cmp h1 h2 then
         if cmp h2 h1 then cmp_list t1 t2 else T
       else F
  |- total_order cmp_list
```

We retain the definition of `cmp_list` from the Haskell type class instance, but we also know that it is a total order (if `cmp` is).

Example II

- Harrison's thesis showed how to mechanize the construction of the real numbers using the positive route:

$$\mathbb{Z}^+ \rightsquigarrow \mathbb{Q}^+ \rightsquigarrow \mathbb{R}^+$$

- After this step there remain three similar constructions:

$$\mathbb{Z}^+ \rightsquigarrow \mathbb{Z} \qquad \mathbb{Q}^+ \rightsquigarrow \mathbb{Q} \qquad \mathbb{R}^+ \rightsquigarrow \mathbb{R}$$

- This is a perfect application for theory interpretation.

Example II — Defining Negative Number Types

Axioms (Negative number example theory)

```
types:
  P
consts:
  leP addP subP multP
thms:
  |- !x. leP x x
  |- !x y. leP x y /\ leP y x ==> x = y
  |- !x y z. leP x y /\ leP y z ==> leP x z
  |- !x y. leP x y \/ leP y x
  |- !x y. addP x y = addP y x
  |- !x y z. addP (addP x y) z = addP x (addP y z)
  |- !x x' y y'.
       leP x x' /\ leP y y' ==> leP (addP x y) (addP x' y')
  |- !x y. ~leP (addP x y) x
  |- !x y. ~leP y x ==> addP x (subP y x) = y
  |- !x y. multP x y = multP y x
  |- !x y z. multP (multP x y) z = multP x (multP y z)
```

Example II — Defining Negative Number Types

Theory (Negative number example theory)

```
types:
  N
consts:
  zero le add neg sub mult inject
thms:
  |- !x. le x x
  |- !x y. le x y /\ le y x ==> x = y
  |- !x y z. le x y /\ le y z ==> le x z
  |- !x y. le x y \/ le y x

  |- !x. add zero x = x
  |- !x. add x zero = x
  |- !x y. add x y = add y x
  |- !x y z. add (add x y) z = add x (add y z)
  |- !x y z. add x y = add x z = (y = z)
  |- !x y z. le (add x y) (add x z) = le y z
  |- !x x' y y'.
       le x x' /\ le y y' ==> le (add x y) (add x' y')
```

Example II — Defining Negative Number Types

Theory (Negative number example theory)

```
more thms:
  |- neg zero = zero
  |- !x. neg x = zero = (x = zero)
  |- !x. neg (neg x) = x
  |- !x. add x (neg x) = zero
  |- !x. add (neg x) x = zero

  |- sub x y = add x (neg y)
  |- !x y. add x (sub y x) = y

  |- !x. mult zero x = zero
  |- !x. mult x zero = zero
  |- !x y. mult x y = mult y x
  |- !x y z. mult (mult x y) z = mult x (mult y z)
```

Example II — Defining Negative Number Types

Theory (Negative number example theory)

```
even more thms:
  |- !x y. leP x y = le (inject x) (inject y)
  |- !x y. inject (addP x y) = add (inject x) (inject y)
  |- !x y.
       ~leP x y ==>
       inject (subP x y) = sub (inject x) (inject y)
  |- !x y. inject (multP x y) = mult (inject x) (inject y)

  |- !x. ~(inject x = zero)
  |- !x y. ~(inject x = neg (inject y))

  |- !p.
       (!x. p (inject x)) /\ p zero /\
       (!x. p (neg (inject x))) ==> !x. p x
```

galois

Summary

- This talk has presented the OpenTheory project, which aims to apply software engineering principles to theories of higher order logic.
- The article format for higher order logic theories is now stable.
- The next challenge: installing and upgrading theories with automatic dependency management.
- The project web page:

  `http://gilith.com/research/opentheory`