OpenTheory
Package Management for Higher Order Logic Theories

Joe Hurd
Galois, Inc.
joe@galois.com

Galois Tech Seminar
Tuesday 13 January 2009
Talk Plan

1. Proof Articles
2. HOL Light Case Study
3. Theory Engineering
4. Grand Vision
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.
Informally, a theory is a set of theorems, together with their proofs.

Think of a theory as a module in a weird programming language:

\[
\text{module } \sim \text{ theory} \quad \text{example} \\
\text{function type } \sim \text{ theorem} \quad \vdash t = t \\
\text{function definition } \sim \text{ proof} \quad \text{refl } t
\]

Theory engineering is to proving as software engineering is to programming. “Proving in the large.”

What can software engineering techniques do for us in the world of theories?
There are 3 major interactive theorem provers implementing exactly the same variant of higher order logic.

- The variant is Church’s simple theory of types, extended with Hindley-Milner polymorphism.

The theorem provers are:

- HOL4, developed by Konrad Slind and Michael Norrish;
- HOL Light, developed by John Harrison;
- and ProofPower, developed by Rob Arthan.

Isabelle/HOL is not on the list, because its powerful type class mechanism is an extension of the logic.

Porting theories between these theorem provers is currently a painful process of editing scripts that call proof tactics.
Tactic Proof Scripts

Each higher order logic theorem prover compiles its own tactic script language into a sequence of primitive logical inferences.

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: Each of the theorems provers has a slightly different set of tactics, the behaviour of which evolves across versions.
Compiled Theories

- **Idea:** Port the compiled inference proof, not the source tactic script.
- **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.
This talk will introduce OpenTheory articles, which are compiled higher order logic theories.

The OpenTheory project was started by Joe Hurd and Rob Arthan to exchange theories between higher order logic theorem provers.

The format of articles is now stable, and the focus of the project is on design techniques for articles that compose well.

The complete article format can be found at the project web page:

http://opentheory.gilith.com
Theorem Provers in the LCF Design

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

```
(* 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 one is to use the primitive inferences of the logic.
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.
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 *)
| Onum of num (* A number *)
| Oname of name (* A name *)
| Olist of object list (* A list (or tuple) of objects *)
| Otype of hol_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 *)
```
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
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
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?
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 be a theory export previously proved in the article (see next slide).
  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, the theorem is asserted as an axiom and becomes a dependency of the theory.
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]
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: Onum k :: x :: stack
After: x :: stack

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

After: dict[k] :: stack

Dictionary: Before: dict

After: dict[entry k deleted]
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 Error 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.
## HOL Light Articles

<table>
<thead>
<tr>
<th>hol-light theory</th>
<th>article (Kb)</th>
<th>gzip’ed article (Kb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>1,821</td>
<td>227</td>
</tr>
<tr>
<td>arith</td>
<td>25,878</td>
<td>2,736</td>
</tr>
<tr>
<td>wf</td>
<td>29,139</td>
<td>3,210</td>
</tr>
<tr>
<td>calc_num</td>
<td>3,903</td>
<td>372</td>
</tr>
<tr>
<td>normalizer</td>
<td>2,845</td>
<td>300</td>
</tr>
<tr>
<td>grobner</td>
<td>2,412</td>
<td>256</td>
</tr>
<tr>
<td>ind-types</td>
<td>10,520</td>
<td>1,262</td>
</tr>
<tr>
<td>list</td>
<td>11,997</td>
<td>1,440</td>
</tr>
<tr>
<td>realax</td>
<td>23,530</td>
<td>2,510</td>
</tr>
<tr>
<td>calc_int</td>
<td>2,844</td>
<td>314</td>
</tr>
<tr>
<td>realarith</td>
<td>15,981</td>
<td>1,311</td>
</tr>
<tr>
<td>real</td>
<td>29,081</td>
<td>3,078</td>
</tr>
<tr>
<td>calc_rat</td>
<td>2,536</td>
<td>287</td>
</tr>
<tr>
<td>int</td>
<td>39,463</td>
<td>3,371</td>
</tr>
<tr>
<td>sets</td>
<td>146,661</td>
<td>15,517</td>
</tr>
<tr>
<td>iter</td>
<td>144,682</td>
<td>13,254</td>
</tr>
<tr>
<td>cart</td>
<td>18,968</td>
<td>1,948</td>
</tr>
<tr>
<td>define</td>
<td>79,573</td>
<td>8,005</td>
</tr>
</tbody>
</table>
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.
## Compressing the HOL Light Articles

<table>
<thead>
<tr>
<th>hol-light theory</th>
<th>article (Kb)</th>
<th>comp. (Kb)</th>
<th>comp. ratio</th>
<th>gzip’ed article (Kb)</th>
<th>gzip’ed comp. (Kb)</th>
<th>comp. ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>1,821</td>
<td>790</td>
<td>57%</td>
<td>227</td>
<td>113</td>
<td>51%</td>
</tr>
<tr>
<td>arith</td>
<td>25,878</td>
<td>6,940</td>
<td>74%</td>
<td>2,736</td>
<td>966</td>
<td>65%</td>
</tr>
<tr>
<td>wf</td>
<td>29,139</td>
<td>6,037</td>
<td>80%</td>
<td>3,210</td>
<td>854</td>
<td>74%</td>
</tr>
<tr>
<td>calc_num</td>
<td>3,903</td>
<td>1,518</td>
<td>62%</td>
<td>372</td>
<td>200</td>
<td>47%</td>
</tr>
<tr>
<td>normalizer</td>
<td>2,845</td>
<td>660</td>
<td>77%</td>
<td>300</td>
<td>91</td>
<td>70%</td>
</tr>
<tr>
<td>grobner</td>
<td>2,412</td>
<td>718</td>
<td>71%</td>
<td>256</td>
<td>102</td>
<td>61%</td>
</tr>
<tr>
<td>ind-types</td>
<td>10,520</td>
<td>4,287</td>
<td>60%</td>
<td>1,262</td>
<td>590</td>
<td>54%</td>
</tr>
<tr>
<td>list</td>
<td>11,997</td>
<td>4,586</td>
<td>62%</td>
<td>1,440</td>
<td>645</td>
<td>56%</td>
</tr>
<tr>
<td>realax</td>
<td>23,530</td>
<td>7,699</td>
<td>68%</td>
<td>2,510</td>
<td>1,063</td>
<td>58%</td>
</tr>
<tr>
<td>calc_int</td>
<td>2,844</td>
<td>825</td>
<td>71%</td>
<td>314</td>
<td>118</td>
<td>63%</td>
</tr>
<tr>
<td>realarith</td>
<td>15,981</td>
<td>4,598</td>
<td>72%</td>
<td>1,311</td>
<td>580</td>
<td>56%</td>
</tr>
<tr>
<td>real</td>
<td>29,081</td>
<td>8,604</td>
<td>71%</td>
<td>3,078</td>
<td>1,164</td>
<td>63%</td>
</tr>
<tr>
<td>calc_rat</td>
<td>2,536</td>
<td>1,130</td>
<td>56%</td>
<td>287</td>
<td>155</td>
<td>46%</td>
</tr>
<tr>
<td>int</td>
<td>39,463</td>
<td>8,935</td>
<td>78%</td>
<td>3,371</td>
<td>1,203</td>
<td>65%</td>
</tr>
<tr>
<td>sets</td>
<td>146,661</td>
<td>24,839</td>
<td>84%</td>
<td>15,517</td>
<td>3,460</td>
<td>78%</td>
</tr>
<tr>
<td>iter</td>
<td>144,682</td>
<td>24,260</td>
<td>84%</td>
<td>13,254</td>
<td>3,342</td>
<td>75%</td>
</tr>
<tr>
<td>cart</td>
<td>18,968</td>
<td>3,336</td>
<td>83%</td>
<td>1,948</td>
<td>469</td>
<td>76%</td>
</tr>
<tr>
<td>define</td>
<td>79,573</td>
<td>15,787</td>
<td>81%</td>
<td>8,005</td>
<td>2,156</td>
<td>74%</td>
</tr>
</tbody>
</table>
Concatenating all the HOL Light theories in turn generates an article exporting 126,555 theorems, and depending on 3 axioms:

**Axioms (The HOL Light axioms)**

- **types:**
  ```
  bool fun ind
  ```
- **consts:**
  ```
  ! /
  ```
  ```
  = ==> ?
  ```
  ```
  ONE_ONE ONTO select ~
  ```
- **thms:**
  ```
  |- !t. (\x. t x) = t
  |- !P x. P x ==> P ((select) P)
  |- ?f. ONE_ONE f /
  ```
  ```
  ~ONTO f
  ```
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, $\Gamma \vdash \Delta$:
- $\Gamma$: The set of axioms that the theory depends on.
- $\Delta$: The set of theorems that the theory exports.

Reducing the export set is always safe:

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

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.
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:

  \[
  \text{ML module} \sim \text{HOL theory}
  \]

  \[
  \begin{align*}
  \text{types} & \sim \text{types} \\
  \text{values} & \sim \text{constants} \\
  \text{type judgements} & \sim \text{theorems} \\
  \text{implementation} & \sim \text{proof}
  \end{align*}
  \]
Example 1

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 $\leq$ to be a total order on elements.

**Missing Export:** Can guarantee that $\leq$ is a total order on elements.
Example I — Defining Type Class Properties

Definition (Total orders)

\[- \text{refl}_\text{rel} \ r = \ !x. \ r \ x \ x \]
\[- \text{antisym}_\text{rel} \ r = \ !x \ y. \ r \ x \ y \ \&\& \ r \ y \ x \ \Rightarrow \ x = y \]
\[- \text{trans}_\text{rel} \ r = \ !x \ y \ z. \ r \ x \ y \ \&\& \ r \ y \ z \ \Rightarrow \ r \ x \ z \]
\[- \text{total}_\text{rel} \ r = \ !x \ y. \ r \ x \ y \ \lor \ r \ y \ x \]
\[- \text{pre}_\text{order} \ r \iff \ \text{refl}_\text{rel} \ r \ \&\& \ \text{trans}_\text{rel} \ r \]
\[- \text{partial}_\text{order} \ r \iff \ \text{pre}_\text{order} \ r \ \&\& \ \text{antisym}_\text{rel} \ r \]
\[- \text{total}_\text{order} \ r \iff \ \text{partial}_\text{order} \ r \ \&\& \ \text{total}_\text{rel} \ r \]
Example I — Adding Properties to Type Classes

Create a theory containing an uninterpreted type $T$ and constant $\text{cmp}$, and an axiom that $\text{cmp}$ 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 $\text{cmp}$ 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:

\[ \mathbb{Z}^+ \leadsto \mathbb{Q}^+ \leadsto \mathbb{R}^+ \]

- After this step there remain three similar constructions:

\[ \mathbb{Z}^+ \leadsto \mathbb{Z} \quad \mathbb{Q}^+ \leadsto \mathbb{Q} \quad \mathbb{R}^+ \leadsto \mathbb{R} \]

- 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 \land le y x ==> x = y
    |- !x y z. le x y \land le y z ==> le x z
    |- !x y. le x y \lor 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’ \lor le y y’ ==> le (add x y) (add x’ y’)

Joe Hurd
OpenTheory
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:

\[ |- \ !x \ y. \ leP \ x \ y = le (inject \ x) (inject \ y) \]
\[ |- \ !x \ y. \ inject (addP \ x \ y) = add (inject \ x) (inject \ y) \]
\[ |- \ !x \ y. \ \neg leP \ x \ y ==> inject (subP \ x \ y) = sub (inject \ x) (inject \ y) \]
\[ |- \ !x \ y. \ inject (multP \ x \ y) = mult (inject \ x) (inject \ y) \]
\[ |- \ !x. \ \neg (inject \ x = zero) \]
\[ |- \ !x \ y. \ \neg (inject \ x = neg (inject \ y)) \]
\[ |- \ !p. \]
\[ (!x. \ p (inject \ x)) \ \lor \ p \ zero \ \lor \]
\[ (!x. \ p (neg (inject \ x)))) ==> !x. \ p \ x \]
OpenTheory Platform

- Build up a collection of reusable theories in article format.
- Store them in publically-accessible repositories.
- A local installer supports queries of the form: *Build a theory with new types and constants satisfying a set of theorems.*
  - It uses article summaries from the repositories to reduce the input query to alternative dependencies.
  - Tries to automatically resolve the dependencies using locally installed theories, or by calling itself recursively down to some fixed depth.
- Eventually, just want an apt-get or cabal-install style user interface:

  ```
  opentheory install complex-numbers
  ```
The talk has presented the OpenTheory project for managing higher order logic theories.

The project web page:

http://opentheory.gilith.com