Challenges Implementing a HOL System in Haskell or: How I Learned to Stop Worrying and Love the Monad

Evan Austin    Perry Alexander    Nick Frisby

The University of Kansas
Information and Telecommunication Technology Center

Midwest Verification Day - September 16, 2010
Follow the LCF approach of a small logical kernel

Kernel implements ADT for theorems and functions for primitive inference rules

Take a bootstrapping approach to build a large system on the small, trusted kernel
Follow the LCF approach of a small logical kernel

Kernel implements ADT for theorems and functions for primitive inference rules

Take a bootstrapping approach to build a large system on the small, trusted kernel

Starting with HOL90, all have been implemented in Standard ML (or one of its derivatives)
Challenges for a New Implementation

- Can it match the existing functionality?
- Can it match the existing trust?
- Can it match (or improve) the existing performance?
HOL Light vs. HaskHOL

HOL Light

- Implemented in OCaml
- Based on a typed λ-calculus
- Logical kernel is composed of 10 primitive rules and an extensible system for constants, definitions, and axioms
HOL Light

- Implemented in OCaml
- Based on a typed $\lambda$-calculus
- Logical kernel is composed of 10 primitive rules and an extensible system for constants, definitions, and axioms

HaskHOL

- Implemented in Haskell
- Extends HOL Light logical kernel with monad and surrounding structure
Can HaskHOL match the functionality of the HOL Light kernel?
Can HaskHOL match the functionality of the HOL Light kernel?

OCaml $\rightarrow$ Haskell
The First Challenge

Can HaskHOL match the functionality of the HOL Light kernel?

OCaml $\rightarrow$ Haskell

Strict/Impure $\rightarrow$ Lazy/Pure
The First Challenge

Can HaskHOL match the functionality of the HOL Light kernel?

OCaml $\rightarrow$ Haskell

Strict/Impure $\rightarrow$ Lazy/Pure

Potential Concerns

- Lack of side effects
- Effect of laziness on error handling
The HolM Monad

Need:

- Strict error handling
- Global context
The HolM Monad

Need:

- Strict error handling
- Global context

The HolM Monad

type HolM m = StateT (HolContext m) (ErrorT String m)
runHolT x = runErrorT (evalStateT x initCtxt)
Definition

\[
\frac{t}{t \vdash t} \quad \text{assume}
\]

Fails when \( t \) is not a proposition.
Example

Definition

\[
\frac{t}{t \vdash t}
\]

assume

Fails when \( t \) is not a proposition.

HOL Light Implementation

let ASSUME tm =
if Pervasives.compare (type_of tm) bool_ty = 0
then Sequent([tm],tm)
else failwith "ASSUME: not a proposition"
HOL Light Implementation

```ocaml
define ASSUME (tm : term) : sequent
  =
  if type_of tm = bool
  then Sequent([tm],tm)
  else failwith "ASSUME: not a proposition"
```

HaskHOL Implementation

```haskell
define cASSUME ' :: (Monad m) => HolTerm -> HolM m
  = do
      ty <- type_of tm
      if ty == tybool
      then return $ Sequent([tm],tm)
      else throwError "assume: term not a prop"
```
Example

HOL Light Implementation

let ASSUME tm =  
  if Pervasives.compare (type_of tm) bool_ty = 0  
  then Sequent([tm],tm)  
  else failwith "ASSUME: not a proposition"

HaskHOL Implementation

cASSUME’ :: (Monad m) => HolTerm -> HolM m Theorem  
cASSUME’ tm =  
  do { ty <- type_of tm  
      ; if ty == tybool  
        then return $ Sequent [tm] tm  
        else throwError "assume: term not a prop"  
    }
Can HaskHOL match the trust provided by the HOL Light kernel?
The Second Challenge

Can HaskHOL match the trust provided by the HOL Light kernel?

OCaml $\rightarrow$ Haskell
The Second Challenge

Can HaskHOL match the trust provided by the HOL Light kernel?

OCaml $\rightarrow$ Haskell

Different module systems
Can HaskHOL match the trust provided by the HOL Light kernel?

OCaml $\rightarrow$ Haskell

Different module systems

Different run-time systems
The Second Challenge

Can HaskHOL match the trust provided by the HOL Light kernel?

OCaml $\rightarrow$ Haskell

Different module systems

Different run-time systems

Potential Concerns

- Maintain proper restrictions on access to kernel
- Prevent unnecessary growth in trusted computing base
OCaml's module system:

- **Signature** - Specification of the types and values exposed
- **Structure** - Implementation satisfying the signature
OCaml’s module system:

- **Signature** - Specification of the types and values exposed
- **Structure** - Implementation satisfying the signature

**HOL Light Signature**

```ocaml
module type Hol_kernel = sig
    type thm
    val types: unit -> (string * int) list
end
```

**HOL Light Structure**

```ocaml
module Hol : Hol_kernel = struct
    type thm = Sequent of (term list * term)
    let the_type_constants = ref ["bool",0; "fun",2]
    let types () = ! the_type_constants
end
```
Module Systems

OCaml’s module system:

- **Signature** - Specification of the types and values exposed
- **Structure** - Implementation satisfying the signature

HOL Light Signature

```ocaml
defmodule type Hol_kernel = sig
  type thm
  val types : unit -> (string * int)list
end
```

HOL Light Structure

```ocaml
defmodule Hol : Hol_kernel = struct
  type thm = Sequent of (term list * term)
  let the_type_constants = ref ["bool",0; "fun",2]
  let types() = !the_type_constants
end
```
Module Systems

Need:

- Hide Theorem constructor
- Relegate data access to provided functions
- Hide ”helper” functions
Module Systems

Need:

- Hide Theorem constructor
- Relegate data access to provided functions
- Hide "helper" functions

Haskell’s module system:

- Provides capacity for enumerating allowed exports
- Can be used in conjunction with Cabal to hide entire modules from user
module HaskHOL.Kernel.Kernel
  ( Theorem,
   types)
where

data Sequent a c = Seq [a] c deriving (Eq, Ord)
type Theorem = Sequent HolTerm HolTerm

types :: (Monad m) => HolM m TypeConstants
types = do ctxt <- get
           return $ typeConstants ctxt
Run-time Systems:

- Differences outside of scope of preliminary research
- Haskell used for very large number of projects
- Including: Cryptol, Agda, Ivor, etc.
- Provides a "reasonable" place to stand

Other Concerns:

- Kernel needs to be extended to include monad, context, and resulting structure
- Only used to replicate behavior of OCaml, so kernel should remain consistent and valid
- Concerns about exposing context can be eliminated with top-level mutable state work arounds or extensible records
Run-time Systems:  
- Differences outside of scope of preliminary research  
- Haskell used for very large number of projects  
- Including: Cryptol, Agda, Ivor, etc.  
- Provides a ”reasonable” place to stand

Other Concerns:  
- Kernel needs to be extended to include monad, context, and resulting structure  
- Only used to replicate behavior of OCaml, so kernel should remain consistent and valid  
- Concerns about exposing context can be eliminated with top-level mutable state work arounds or extensible records
The Third Challenge

Can HaskHOL match the performance of HOL Light?

Naive HaskHOL Implementation: 1.38s average over 10 tests

1 2.2 GHz Core 2 Duo, 4 GB RAM, OS X 10.6.4, GHC 6.12.3

E. Austin, P. Alexander, N. Frisby (KU)
The Third Challenge

Can HaskHOL match the performance of HOL Light?

Our Test

prove ("(p \land q \iff q \land p) \land"
     "((p \land q) \land r \iff p \land (q \land r)) \land"
     "(p \land (q \land r) \iff q \land (p \land r)) \land"
     "(p \land p \iff p) \land"
     "(p \land (p \land q) \iff p \land q)") \tITAUT_TAC
The Third Challenge

Can HaskHOL match the performance of HOL Light?

**Our Test**

```haskell
prove "(p \&& q <=> q \&& p) \&& 
    ((p \&& q) \&& r <=> p \&& (q \&& r)) \&& 
    (p \&& (q \&& r) <=> q \&& (p \&& r)) \&& 
    (p \&& p <=> p) \&& 
    (p \&& (p \&& q) <=> p \&& q)" tITAUT_TAC
```

"Naive" HaskHOL Implementation: 1.38s average over 10 tests

---

1 2.2 GHz Core 2 Duo, 4 GB RAM, OS X 10.6.4, GHC 6.12.3
Why So Slow?

Cause:

- Exorbitant amount of time being spent parsing.
- Expression parser being rebuilt approximately 24,000 times
Why So Slow?

Cause:

- Exorbitant amount of time being spent parsing.
- Expression parser being rebuilt approximately 24,000 times
- True Cause: Results of constant functions with monadic computations are not being cached.

Example:

```haskell
rTRUTH :: (Monad m) => HolM m Theorem
rTRUTH = do lth <- rSYM =<< dTRUE_DEF
             rth <- cREFL "p: bool . p"
             rEQ_MP lth rth
```

E. Austin, P. Alexander, N. Frisby (KU)   HOL in Haskell   MVD 2010 15 / 21
Why So Slow?

Cause:

- Exorbitant amount of time being spent parsing.
- Expression parser being rebuilt approximately 24,000 times
- True Cause: Results of constant functions with monadic computations are not being cached.

Example

```haskell
rTRUTH :: (Monad m) => HolM m Theorem
rTRUTH = do lth <- rSYM =<< dTRUE_DEF
           rth <- cREFL "\p: bool . p"
           rEQ_MP lth rth
```
OneShots

Solution:
Use a version of OneShots (shared on-demand IO actions).

```haskell
mkOneShot :: MonadIO m => m a -> IORef (m a)
mkOneShot m = ref where
    ref = unsafePerformIO . newIORef $ do
        x <- m
        liftIO $ writeIORef ref (return x)
        return x

unOneShot :: MonadIO m => IORef (m b) -> m b
unOneShot ref = join (liftIO (readIORef ref))
```
OneShots

Solution:
Use a version of OneShots (shared on-demand IO actions).

Making oneShots

```haskell
mkOneShot :: MonadIO m => m a -> IORef (m a)
mkOneShot m = ref where
  ref = unsafePerformIO . newIORef $ do
    x <- m
    liftIO $ writeIORef ref (return x)
  return x
```

Using oneShots

```haskell
unOneShot :: MonadIO m => IORef (m b) -> m b
unOneShot ref = join (liftIO (readIORef ref))
```
OneShots

Solution:
Use a version of OneShots (shared on-demand IO actions).

Making OneShots

\[ \text{mkOneShot :: MonadIO} \ m \Rightarrow m \ a \to IORef (m \ a) \]
\[ \text{mkOneShot} \ m = \text{ref} \ where \]
\[ \text{ref} = \text{unsafePerformIO} . \text{newIORef} \$ \ do \]
\[ x \leftarrow m \]
\[ \text{liftIO} \$ \text{writeIORef} \text{ref} \ (\text{return} \ x) \]
\[ \text{return} \ x \]

Using OneShots

\[ \text{unOneShot :: MonadIO} \ m \Rightarrow IORef (m \ b) \to m \ b \]
\[ \text{unOneShot} \ \text{ref} = \text{join} \ (\text{liftIO} \ (\text{readIORef} \ \text{ref})) \]
proveOnce

Use Template Haskell to control the process for the user:

```
proveOnce

rTRUTH :: ContextMonad m => HolM m Theorem
rTRUTH = $(proveOnce [|
    lth <- rSYM =<< dTRUE_DEF
    rth <- cREFL "p: bool . p"
    rEQ_MP lth rth |])
```

Key Notes:
- Local transformation only
- Does not modify the proof code
- Inserts a NOINLINE pragma to guard against common sub-term elimination
- Safe as long as the monadic computation is side effect free
proveOnce

Use Template Haskell to control the process for the user:

proveOnce

rTRUTH :: ContextMonad m => HolM m Theorem
rTRUTH = $(proveOnce [|
  lth <- rSYM =<< dTRUE_DEF
  rth <- cREFL "\\p:boolean . p"
  rEQ_MP lth rth |])

Key Notes:
- Local transformation only
- Does not modify the proof code
- Inserts a NOINLINE pragma to guard against common sub-term elimination
- Safe as long as the monadic computation is side effect free
Results:

- "Naive" HaskHOL Implementation: 1.38s average over 10 tests

Compare with HOL Light: 2.35s average over 10 tests

Checkpointed HOL Light: ?
Results:

- "Naive" HaskHOL Implementation: 1.38s average over 10 tests
- Optimized HaskHOL Implementation: 0.55s average
Results:

- "Naive" HaskHOL Implementation: 1.38s average over 10 tests
- Optimized HaskHOL Implementation: 0.55s average
- Compiled Optimized HaskHOL Implementation: 0.13s average
Results:

- "Naive" HaskHOL Implementation: 1.38s average over 10 tests
- Optimized HaskHOL Implementation: 0.55s average
- Compiled Optimized HaskHOL Implementation: 0.13s average

- Compare with HOL Light: 2.35s average over 10 tests
Results:

- "Naive" HaskHOL Implementation: 1.38s average over 10 tests
- Optimized HaskHOL Implementation: 0.55s average
- Compiled Optimized HaskHOL Implementation: 0.13s average

- Compare with HOL Light: 2.35s average over 10 tests
- Checkpointed HOL Light: ?
Conclusions

- Can it match the existing functionality?

  Yes (So Far).

  Assumedly so.

  Potential topic for further exploration

- Can it match (or improve) the existing performance?

  Absolutely yes.
Conclusions

- Can it match the existing functionality?
  - Yes (So Far).
Conclusions

- Can it match the existing functionality?
  - Yes (So Far).
- Can it match the existing trust?

Assumedly so. Potential topic for further exploration.

Absolutely yes.
Conclusions

- Can it match the existing functionality?
  - Yes (So Far).

- Can it match the existing trust?
  - Assumedly so.
  - Potential topic for further exploration
Conclusions

- Can it match the existing functionality?
  - Yes (So Far).

- Can it match the existing trust?
  - Assumedly so.
    - Potential topic for further exploration

- Can it match (or improve) the existing performance?
Conclusions

- Can it match the existing functionality?
  - Yes (So Far).
- Can it match the existing trust?
  - Assumedly so.
  - Potential topic for further exploration
- Can it match (or improve) the existing performance?
  - Absolutely yes.
Future Challenges

- Formally prove kernel validity
- Explore additional optimizations presented by laziness and parallelization in Haskell
- Get it polished and get people using it.
Questions?