HERMIT: An Equational Reasoning Model to Implementation Rewrite System for Haskell

Andy Gill

HERMIT is joint work with
Andrew Farmer, Nick Frisby, Ed Komp, Neil Sculthorpe,
Robert Blair, Jan Bracker, Patrick Flor, Adam Howell,
Ryan Scott, Mike Stees, Brad Torrence and Michael Tabone

Functional Programming Group
Information and Telecommunication Technology Center
University of Kansas
andygill@ku.edu

13th July 2014

Compilers Should not be Black Boxes

- We improve spam filters by scripting.

- Can we fix our compiler using scripting?

Remote Shell for our Haskell compiler?

- There is often a trade-off between the clarity and efficiency of a program.
- Useful to transform a clear program (specification) into an efficient program (implementation).
- This idiom has many instantiations: faster code; using a different interface; space usage; semi-formal verification.
- We want to mechanise such transformations on Haskell programs:
  - less time-consuming and error prone than pen-and-paper reasoning
  - no need to modify the source file
- Several existing transformation systems for Haskell programs, e.g. HaRe, HERA, PATH, Ultra. They all operate on Haskell source code.
- **We take a different approach, and provide commands to transforming GHC Core, GHC's intermediate language.**

Demonstration: Unrolling Fibonacci

As a first demonstration, let's transform the *fib* function by unrolling the recursive calls once.

$$
\begin{aligned}
&\mathit{fib} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{fib}\ \mathsf{n} = \textbf{if}\ \mathsf{n} < 2 \\
&\qquad\qquad \textbf{then}\ 1 \\
&\qquad\qquad \textbf{else}\ \mathit{fib}\ (\mathsf{n} - 1) + \mathit{fib}\ (\mathsf{n} - 2)
\end{aligned}
$$

Demonstration: Unrolling Fibonacci

As a first demonstration, let's transform the *fib* function by unrolling the recursive calls once.

$$
\begin{aligned}
&\mathit{fib} :: \mathit{Int} \rightarrow \mathit{Int} \\
&\mathit{fib}\ \mathsf{n} = \textbf{if}\ \mathsf{n} < 2 \\
&\qquad\qquad \textbf{then}\ 1 \\
&\qquad\qquad \textbf{else}\ \mathit{fib}\ (\mathsf{n} - 1) + \mathit{fib}\ (\mathsf{n} - 2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{fib} :: \mathit{Int} \rightarrow \mathit{Int} \\
&\mathit{fib}\ \mathsf{n} = \textbf{if}\ \mathsf{n} < 2\ \textbf{then}\ 1 \\
&\qquad\qquad\qquad \textbf{else}\ (\textbf{if}\ (\mathsf{n} - 1) < 2\ \textbf{then}\ 1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \mathit{fib}\ (\mathsf{n} - 1 - 1) + \mathit{fib}\ (\mathsf{n} - 1 - 2) \\
&\qquad\qquad\qquad\qquad) \\
&\qquad\qquad\qquad\qquad + \\
&\qquad\qquad\qquad\qquad (\textbf{if}\ (\mathsf{n} - 2) < 2\ \textbf{then}\ 1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \mathit{fib}\ (\mathsf{n} - 2 - 1) + \mathit{fib}\ (\mathsf{n} - 2 - 2) \\
&\qquad\qquad\qquad\qquad)
\end{aligned}
$$

First Demo

First Demo

- `resume` .... resume the compile
- `binding-of 'main` .... goto the `main` definition
- `binding-of 'fib` .... goto the `fib` definition
- `remember "myfib"` .... remember a definition
- `show-remembered` .... show what has been remembered
- any-call (unfold-remembered "myfib") .... try unfold "myfib"
- `bash` .... bash a syntax tree with simple rewrites
- `top` .... go back to the top of the syntax tree
- `load-and-run "Fib.hss"` .... load and run a script

What did we do?

HERMIT requires a recent ghc (I am using GHC 7.8.2)

1. `cabal update`
2. `cabal install hermit`
3. `hermit Main.hs`

The `hermit` command just invokes GHC with some default flags:

```
% hermit Main.hs
ghc Main.hs -fforce-recomp -O2 -dcore-lint
            -fexpose-all-unfoldings
            -fsimple-list-literals -fplugin=HERMIT
            -fplugin-opt=HERMIT:main:Main:
```

HERMIT Use Cases

- We want to explore the use of the worker/wrapper transformation for program refinement
  - We need mechanization to be able to scale the idea to larger examples
  - Are working on large case study: Low Density Parity Checker (LDPC)
  - Transforming math equations into Kansas Lava programs
- HERMIT is for library writers
  - Authors show equivalence between clear (specification) code, and efficient (exported) code.
- HERMIT is a vehicle for prototyping GHC passes
  - Optimization: Stream Fusion
  - Optimization: SYB
  - Staging: Translating Core into CCC combinators. (Elliott, et. al.)
- Hope to use for teaching program refinement and optimization
- (Your project goes here)

Highest Level Architecture

We draw inspiration from UNIX and operating systems.

Three levels

- Shell Level . . . . (UNIX Shell style commands)
- Rewrite Level . . . . (UNIX man(2) system commands)
- Stratego-style library for rewrites . . . . (DSL for rewrites)

Shell Level

UNIX Shell style commands

- Dynamically typed, variable arguments
- Help (man) for each command
- Control flow commands (';', retry, etc.)

Rewrite Level

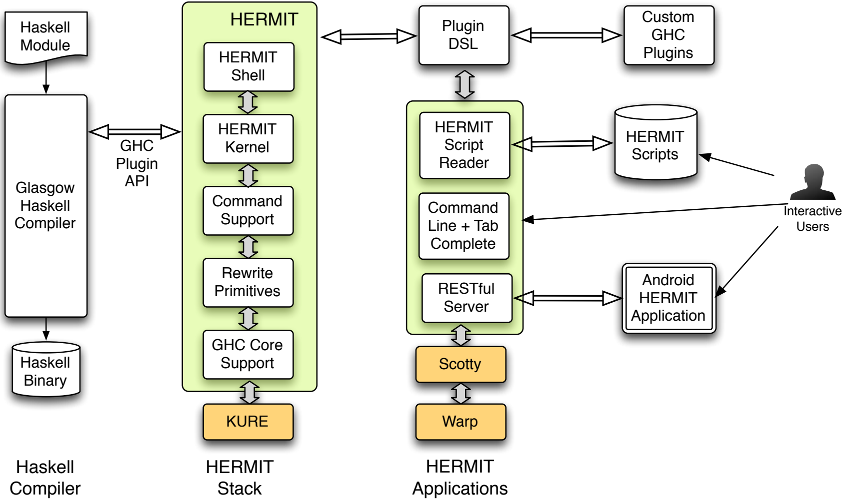
UNIX man(2) system commands

- Haskell functions, strongly typed
- Think type :: `CoreExpr` $\rightarrow$ `M CoreExpr`
- Higher-order functions for tunneling into expressions
- Many function tunnel into GHC (example: substExpr)
- Allow, all GHC "RULES" are directly invokable.

Stratego style library for rewrites

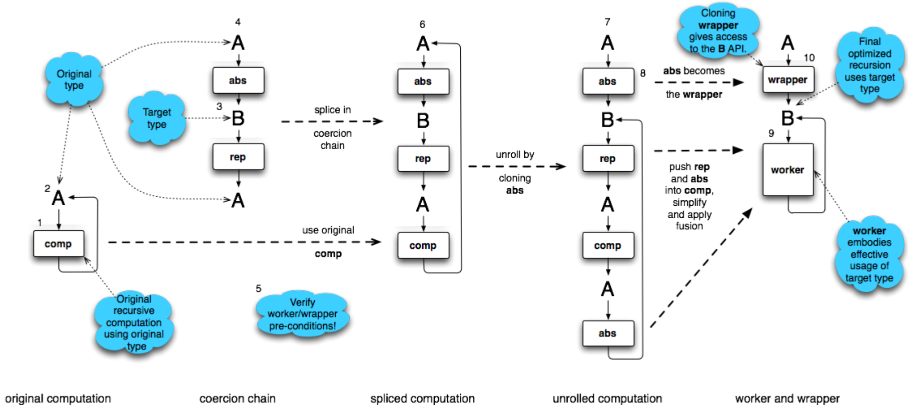
- Haskell DSL call KURE
- Basic idea: rewrites can succeed or fail
- Higher-order combinators for search, catching fail, retry
- Both levels reflect the Stratego API

Lifting the Lid on the HERMIT Project



HERMIT Commands

- Core-specific rewrites, e.g.
  - beta-reduce
  - eta-expand '*x*
  - case-split '*x*
  - inline
- Strategic traversal combinators (from KURE), e.g.
  - any-td *r*
  - repeat *r*
  - innermost *r*
- Navigation, e.g.
  - up, down, left, right, top
  - binding-of '*foo*
  - app-fun, app-arg, let-body, . . .
- Version control, e.g.
  - log
  - back
  - step
  - save "myscript.hss"


Original type
Target type
4
A
abs
3
B
rep
A
2
A
1
comp
Original recursive computation using original type
splice in coercion chain
use original comp
5
Verify worker/wrapper pre-conditions!
6
A
abs
B
rep
A
comp
unroll by cloning abs
7
A
abs
8
B
rep
A
comp
A
abs
Cloning wrapper gives access to the B API.
abs becomes the wrapper
push rep and abs into comp, simplify and apply fusion
A
10
wrapper
B
9
worker
Final optimized recursion uses target type
worker embodies effective usage of target type
original computation
coercion chain
spliced computation
unrolled computation
worker and wrapper

Creating Worker and Wrapper for last

```
last :: [a] -> a
last =
```

```
        \ v -> case v of
                [] -> error "last: []"
                (x:xs) -> case xs of
                           [] -> x
                           (_:_) -> last xs
```

Creating Worker and Wrapper for last

```
last :: [a] -> a
last =




last_work :: a -> [a] -> a
last_work = \ x xs ->
        (\ v -> case v of
                 []     -> error "last: []"
                 (x:xs) -> case xs of
                            []    -> x
                            (_:_) -> last xs) (x:xs)
```

Create the worker out of the body and an invented coercion

Creating Worker and Wrapper for last

```
last :: [a] -> a
last = \ v -> case v of
                 []     -> error "last: []"
                 (x:xs) -> last_work x xs

last_work :: a -> [a] -> a
last_work = \ x xs ->
        (\ v -> case v of
                   []     -> error "last: []"
                   (x:xs) -> case xs of
                               []     -> x
                               (_:_) -> last xs) (x:xs)
```

Invent the wrapper which calls the worker

Creating Worker and Wrapper for last

```
last :: [a] -> a
last = \ v -> case v of
                []     -> error "last: []"
                (x:xs) -> last_work x xs

last_work :: a -> [a] -> a
last_work = \ x xs ->
        (\ v -> case v of
                  []     -> error "last: []"
                  (x:xs) -> case xs of
                              []    -> x
                              (_:_) -> last xs) (x:xs)
```

These functions are mutually recursive

Inline Wrapper

```
last :: [a] -> a
last = \ v -> case v of
                [] -> error "last: []"
                (x:xs) -> last_work x xs

last_work :: a -> [a] -> a
last_work = \ x xs ->
        (\ v -> case v of
                [] -> error "last: []"
                (x:xs) -> case xs of
                           [] -> x
                           (_:_) -> last xs) (x:xs)
```

We now inline *last* inside *last_work*

Inline Wrapper

```
last :: [a] -> a
last = \ v -> case v of
                []     -> error "last: []"
                (x:xs) -> last_work x xs

last_work :: a -> [a] -> a
last_work = \ x xs ->
        (\ v -> case v of
                  []     -> error "last: []"
                  (x:xs) -> case xs of
                               []    -> x
                               (_:_) ->
                 (\ v -> case v of
                            []     -> error "last: []"
                            (x:xs) -> last_work x xs) xs) (x:xs)
```

last_work is now trivially recursive.

Simplify *work*

```
last :: [a] -> a
last = \ v -> case v of
                []     -> error "last: []"
                (x:xs) -> last_work x xs

last_work :: a -> [a] -> a
last_work = \ x xs ->
       (\ v -> case v of
                  []     -> error "last: []"
                  (x:xs) -> case xs of
                            []    -> x
                            (_:_) ->
               (\ v -> case v of
                          []     -> error "last: []"
                          (x:xs) -> last_work x xs) xs) (x:xs)
```

We now simplify the worker

Simplify *work*

```
last :: [a] -> a
last = \ v -> case v of
                []      -> error "last: []"
                (x:xs) -> last_work x xs

last_work :: a -> [a] -> a
last_work = \ x xs ->



                            case xs of
                              []      -> x
                              (x:xs) -> last_work x xs
```

Reaching our efficient implementation

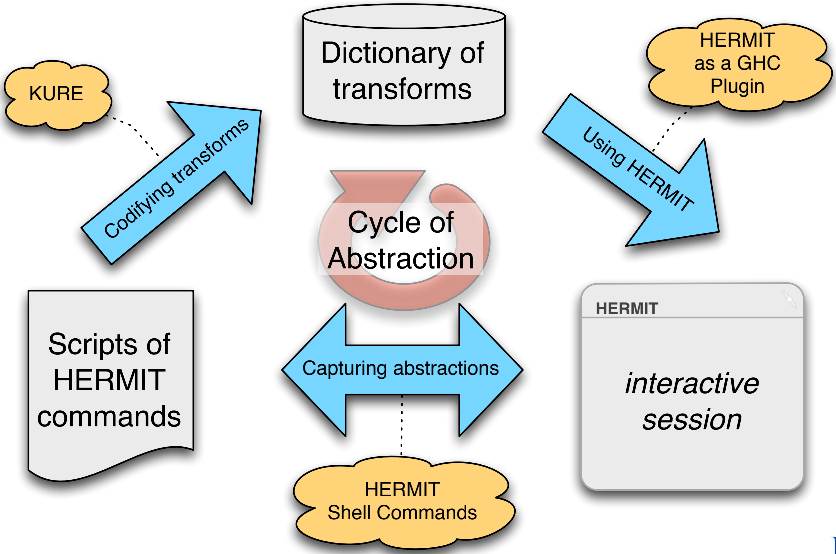
Second Demo

Second Demo

- `flatten-module` .... create one big rec group
- `fix-intro` .... introduce a fix
- `split-1-beta last [| wrap |] [| unwrap |]` .... apply worker/wrapper
- `unfold ['g,'wrap,'unwrap]` .... unfold a set of bindings
- `prove-lemma last-assumption` .... open a proof
- `lhs (...)` .... Apply a rewrite to the left-hand-side of a proof
- `end-proof` .... check for α-equivalence

Pause for breath

Developing Transformations



Adding Transformations to HERMIT

Three ways to add a transform:

- Using Shell
  - Direct
  - No Argument Passing
  - Trying to avoiding "Yet another language"
  - (At some point the Shell will be replaced with a GHCI prompt)
- Using GHC Rules
  - lightweight (can be included in the source code of the object program)
  - no need to recompile HERMIT
  - limited by the expressiveness of RULES
- Using KURE
  - very expressive
  - Requires learning new DSL

GHC RULES

- GHC language feature allowing custom optimisations
- e.g.

{-# RULES "map/map" $\forall$ f g xs. $map\ f\ (map\ g\ xs) = map\ (f \circ g)\ xs$ #-}

- HERMIT adds any RULES to its available transformations
  - allows the HERMIT user to introduce new transformations
  - HERMIT can be used to test/debug RULES

What do we want our KURE DSL to do?

Consider the first case rewriting rule from the Haskell 98 Report.

(*a*) `case e of { alts } = (\v -> case v of { alts }) e`
where v is a new variable

Writing a rule that expresses this syntactical rewrite is straightforward.

```
-- Template Haskell based solution
rule_a :: ExpE -> Q ExpE
rule_a (CaseE e alts) = do
  v <- newName "v"
  return $ AppE (mkLamE [VarP v] $ CaseE (VarE v) alts) e
rule_a _ = fail "rule_a not applicable"
```

KURE is a DSL that allows the structured promotion of locally acting rules into globally acting rules.

Basis of a Rewrite DSL

Combinator	Purpose
`id`	identity strategy
`fail`	always failing strategy
$\mathcal{S}$ `<+` $\mathcal{S}$	local backtracking
$\mathcal{S}$ `;` $\mathcal{S}$	sequencing
`all(`$\mathcal{S}$`)`	apply $\mathcal{S}$ to each immediate child
`<`$\mathcal{S}$`>` *term*	apply $\mathcal{S}$ to *term*, giving a *term* result

Stratego Examples

Try a rewrite, and if it fails, do nothing.

```
try(s) = s <+ id
```

Repeatedly apply a rewrite, until it fails.

```
repeat(s) = try(s ; repeat(s))
```

Apply a rewrite in a topdown manner.

```
topdown(s) = s ; all(topdown(s))
```

New function for constant folding on an `Add` node.

```
EvalAdd : Add(Int(i),Int(j)) -> Int(<addS>(i,j))
```

DSL Formula

- Propose a small set of primitives;
- Unify these combinators round a small number of type(s);
- Postulate the monad that implements the primitives;
- Wrap some structure round this monad, our principal type.

After this, the primitives in this shallow embedding are easy to implement, using the monad, typically

- Construction of our type, the atoms of our solution;
- Combinators for our type, to compose solutions;
- Execution of our type, to give a result.

What is our Principal Type?

$$\boxed{\mathcal{T}\ t_1\ t_2}$$

$$\mathcal{R}\ t = \mathcal{T}\ t\ t$$

Basic Operations in KURE

Combinator	Type	
`id`	$\forall t_1.$	$\mathcal{T}\ t_1\ t_1$
`fail`	$\forall t_1, t_2.$	$\mathcal{T}\ t_1\ t_2$
$\mathcal{S}$ `<+` $\mathcal{S}$	$\forall t_1, t_2.$	$\mathcal{T}\ t_1\ t_2 \rightarrow \mathcal{T}\ t_1\ t_2 \rightarrow \mathcal{T}\ t_1\ t_2$
$\mathcal{S}$ `;` $\mathcal{S}$	$\forall t_1, t_2, t_3.$	$\mathcal{T}\ t_1\ t_2 \rightarrow \mathcal{T}\ t_2\ t_3 \rightarrow \mathcal{T}\ t_1\ t_3$

The KURE Monad

We list our requirements, then build our monad.

We want the ability to

- Represent failure
- create new global binders
- have a context / understand binders

For historic reasons, we pull the environment out explicitly.

Implementation of Translate

```
data Translate c m a b = Translate
     { -- | Apply a 'Translate' to a value
       --   and its context.
       apply :: c -> a -> m b}

-- | The primitive  way of building a 'Translate'.
translate :: (c -> a -> m b) -> Translate c m a b
translate = Translate

-- | A 'Translate' that shares the same source
--   and target type.
type Rewrite c m a = Translate c m a a

-- | The primitive way of building a 'Rewrite'.
rewrite :: (c -> a -> m a) -> Rewrite c m a
rewrite = translate
```

Translate and the Category Zoo

```
instance Functor m => Functor (Translate c m a)
instance Applicative m => Applicative (Translate c m a)
instance Alternative m => Alternative (Translate c m a)
instance Monad m => Monad (Translate c m a)
instance MonadCatch m => MonadCatch (Translate c m a)
instance MonadPlus m => MonadPlus (Translate c m a)
instance Monad m => Category (Translate c m)
instance MonadCatch m => CategoryCatch (Translate c m)
instance Monad m => Arrow (Translate c m)
instance MonadPlus m => ArrowZero (Translate c m)
instance MonadPlus m => ArrowPlus (Translate c m)
instance Monad m => ArrowApply (Translate c m)
instance (Monad m, Monoid b) => Monoid (Translate c m a b)
```

Lenses in KURE

```
-- | A 'Lens' is a way to focus on a sub-structure
-- of type @b@ from a structure of type @a@.
newtype Lens c m a b = Lens (Translate c m a ((c,b), b -> m a))

-- | Apply a 'Rewrite' at a point specified by a 'Lens'.
focusR :: Monad m => Lens c m a b -> Rewrite c m b -> Rewrite c m a

-- | Apply a 'Translate' at a point specified by a 'Lens'.
focusT :: Monad m => Lens c m a b -> Translate c m b d
                                  -> Translate c m a d
```

Where are we?

- KURE allow us to build rewrite engines out of small parts.
- We can perform shallow and deep transformations over a single type.

Most abstract syntax trees are constructed of trees of multiple types.
Challenge – Can we extend our typed rewrites to work over multiple types?

What is the type of `all`?

$$\texttt{all} \ :: \forall t_1.\ \mathcal{R}\ t_1 \rightarrow \mathcal{R}\ t_1$$

OR

$$\texttt{all} \ :: \forall t_1, t_2.\ \mathcal{R}\ t_1 \rightarrow \mathcal{R}\ t_2$$

We use a local Universe

```
-- | Core is the sum type of all nodes in the AST that
-- we wish to be able to traverse.

data Core = GutsCore  ModGuts      -- ^ The module.
          | ProgCore  CoreProg     -- ^ A program
          | BindCore  CoreBind     -- ^ A binding group.
          | DefCore   CoreDef      -- ^ A recursive definition.
          | ExprCore  CoreExpr     -- ^ An expression.
          | AltCore   CoreAlt      -- ^ A case alternative.
```

KU

Example – β-reduction

This is the code for our β-reduction combinator.

```
betaReduce :: RewriteH CoreExpr
betaReduce = setFailMsg ("Beta-reduction failed: " ++ ...) $
    do App (Lam v e1) e2 <- idR
       return $ Let (NonRec v e2) e1
```

HERMIT Problems

What went wrong? What could be better?

- The commands, and the way they act, are still low, low level
- There are way too many commands!
- Want higher-level combinators for worker/wrapper (contextually aware)
- The Shell language has grown legs, and walked away (want GHC)
- Focus on correctness, not speed (`-set-auto-corelint`)

Larger Example: Deriving a better century

- We selected the chapter *Making a Century* from the textbook *Pearls of Functional Algorithm Design*.
- The book is entirely dedicated to reasoning about Haskell programs, with each chapter calculating an efficient program from an inefficient specification program.

The program in *Making a Century* computes the list of all arithmetic expressions formed from ascending digits, where juxtaposition, addition, and multiplication evaluate to 100. For example, one possible solution is

$$100 = 12 + 34 + 5 \times 6 + 7 + 8 + 9$$

The derivation of an efficient program involves a substantial amount of equational reasoning, and the use of a variety of proof techniques, including fold/unfold transformation, structural induction, fold fusion, and numerous auxiliary lemmas.

What happened while deriving a better century

- During mechanization we discovered that several auxiliary properties in the textbook are stated as assumptions without proof.
  - we suspect that they are deemed either “obvious” or “uninteresting”.
- Assumption 6.2 also had a simple proof, but it relied on arithmetic properties of Haskell’s built-in `Int` type (specifically, that `m == n` $\Longrightarrow$ `m <= n`).
- Two proof techniques are used in the textbook that HERMIT does not directly support.
  - The first is the fold fusion law, which needs implication, which we do not support.
  - The second involves postulating the existence of an auxiliary function.
  - We did manage to run the postulated function backwards, to verify the calculation.
- We have a plugin that provides the fold fusion law as a primitive.

KU

Length of Calulations for Century

Calculation	Textbook Lines	HERMIT Commands		
		Transformation	Navigation	Total
solutions	16	12	7	19
expand	19	18	20	38
Lemma 6.5	not given	4	4	8
Lemma 6.6	not given	2	1	3
Lemma 6.7	not given	2	0	2
Lemma 6.8	7	5	8	13
Lemma 6.9	1	4	4	8
Lemma 6.10	not given	23	13	36
Total	43	70	57	127

HERMIT Summary

- A GHC plugin for interactive transformation of GHC Core programs
- HERMIT is still in development
- Can run different scripts for different modules
- Current step: an equational reasoning framework that only allows correctness preserving transformations (Reading, Writing, and Arithmetic)
- Publications:
  - *The HERMIT in the Machine* (Haskell '12) — describes the HERMIT implementation
  - *The HERMIT in the Tree* (IFL '12) — describes our experiences mechanising existing program transformations
  - *KURE: A Haskell embedded strategic programming language with custom closed universes.* (JFP) — describes our DSL for rewrites.
  - *Reasoning with the HERMIT: Tool Support for Compile-time Equational Reasoning on Haskell Programs* (drafted)

```
cabal install hermit
```

KU