The HERMIT in the Machine

Neil Sculthorpe

(joint work with Andrew Farmer, Andy Gill and Ed Komp)

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

Nottingham, England
28th August 2012

Motivation

- There is often a trade-off between the clarity and efficiency of a program.

Motivation

- 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).

Motivation

- 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).
- 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

Motivation

- 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).
- 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

Motivation

- 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).
- 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
- But they all operate on Haskell source code (or some variant).

Motivation

- 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).
- 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
- But they all operate on Haskell source code (or some variant).
- Haskell source code?

Motivation

- 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).
- 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
- But they all operate on Haskell source code (or some variant).
- Haskell source code? Haskell 98?

Motivation

- 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).
- 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
- But they all operate on Haskell source code (or some variant).
- Haskell source code? Haskell 98? Haskell 2010?

Motivation

- 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).
- 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
- But they all operate on Haskell source code (or some variant).
- Haskell source code? Haskell 98? Haskell 2010? GHC-extended Haskell?

Motivation

- 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).
- 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
- But they all operate on Haskell source code (or some variant).
- Haskell source code? Haskell 98? Haskell 2010? GHC-extended Haskell? Which extensions?

Motivation

- 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).
- 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
- But they all operate on Haskell source code (or some variant).
- Haskell source code? Haskell 98? Haskell 2010? GHC-extended Haskell? Which extensions?
- Alternative: GHC Core, GHC's intermediate language

GHC Core

```
data ModGuts = ModGuts {_ :: [CoreBind], ...}
data CoreBind = NonRec Id CoreExpr
              | Rec [(Id, CoreExpr)]
data CoreExpr = Var Id
              | Lit Literal
              | App CoreExpr CoreExpr
              | Lam Id CoreExpr
              | Let CoreBind CoreExpr
              | Case CoreExpr Id Type [CoreAlt]
              | Cast CoreExpr Coercion
              | Tick (Tickish Id) CoreExpr
              | Type Type
              | Coercion Coercion
type CoreAlt = (AltCon, [Id], CoreExpr)
data AltCon = DataAlt DataCon | LitAlt Literal | DEFAULT
```

What is HERMIT?

What is HERMIT?

- Haskell Equational Reasoning Model-to-Implementation Tunnel

What is HERMIT?

- Haskell Equational Reasoning Model-to-Implementation Tunnel
- A scriptable toolkit for interactive transformation of GHC Core programs.

What is HERMIT?

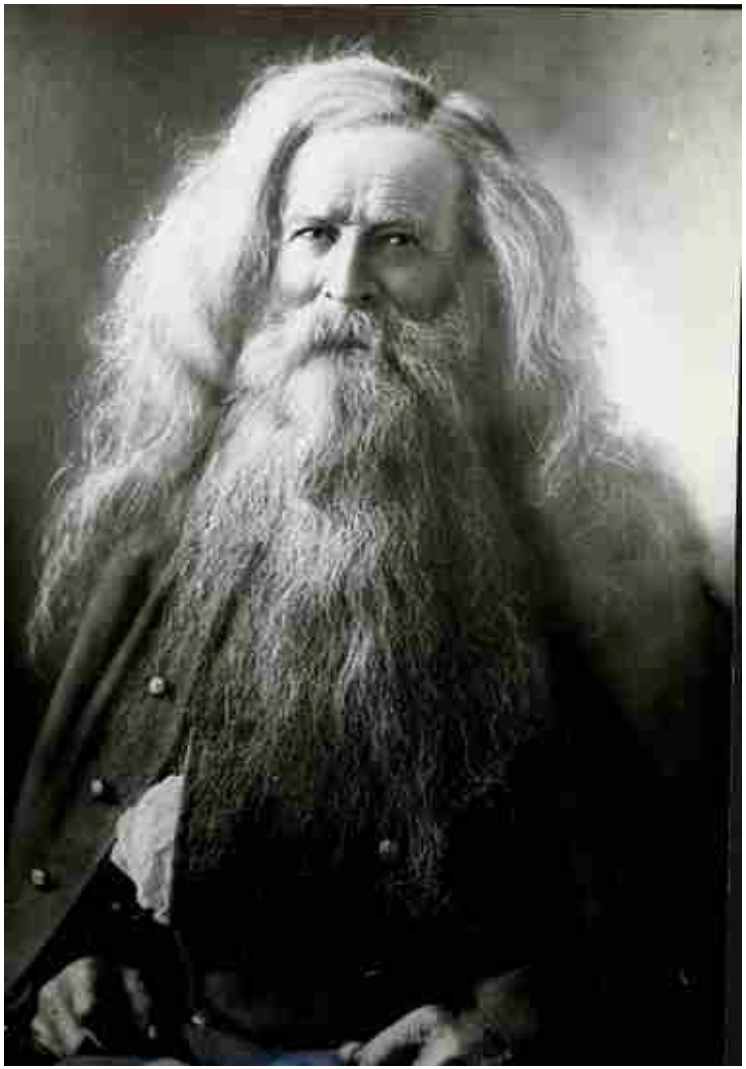
- Haskell Equational Reasoning Model-to-Implementation Tunnel
- A scriptable toolkit for interactive transformation of GHC Core programs.
- Under development at the University of Kansas, Lawrence.

What is HERMIT?

- Haskell Equational Reasoning Model-to-Implementation Tunnel
- A scriptable toolkit for interactive transformation of GHC Core programs.
- Under development at the University of Kansas, Lawrence.
- Not to be confused with:

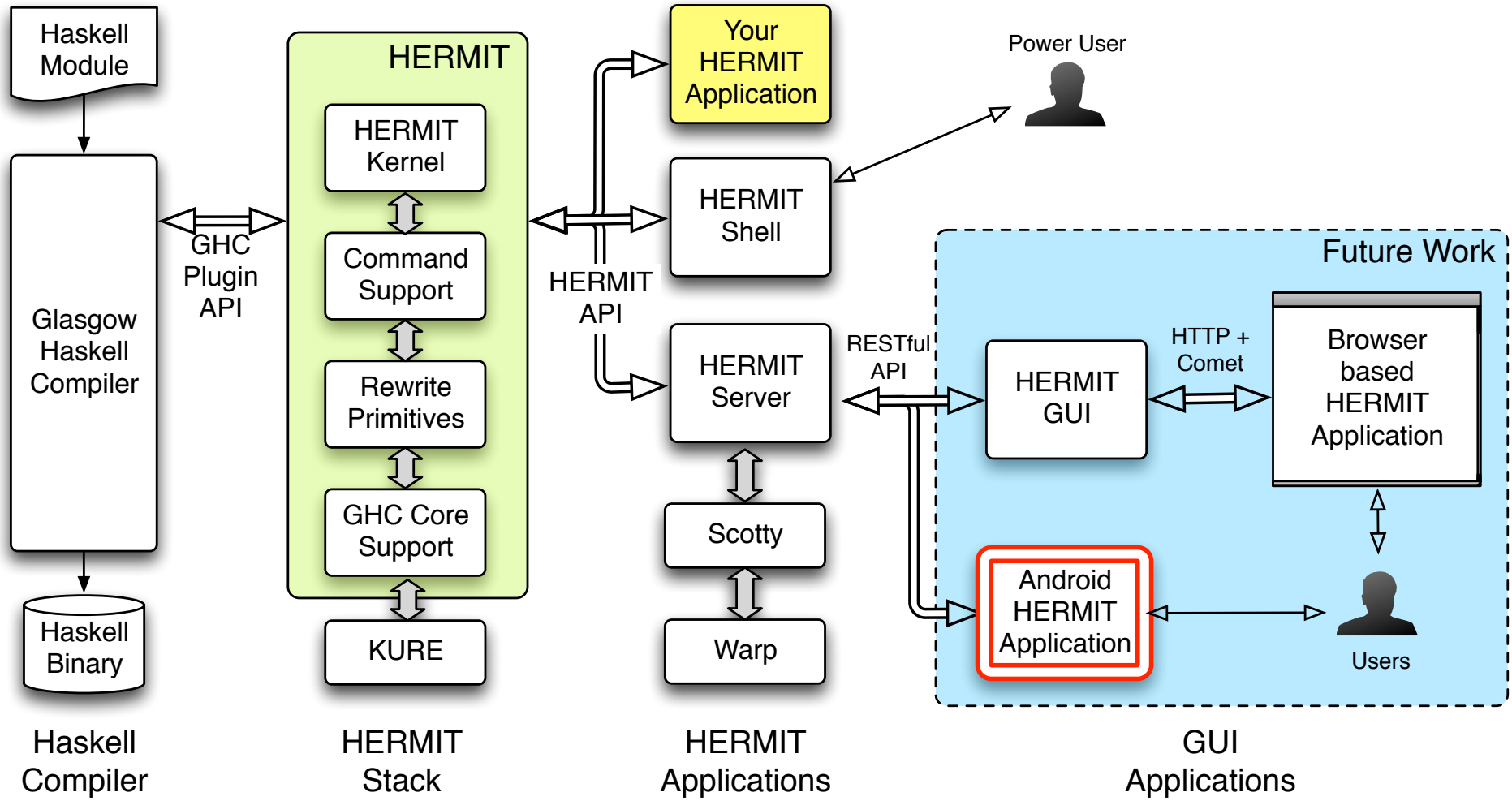
What is HERMIT?

- Haskell Equational Reasoning Model-to-Implementation Tunnel
- A scriptable toolkit for interactive transformation of GHC Core programs.
- Under development at the University of Kansas, Lawrence.
- Not to be confused with: The Kansas Hermit (1826–1909), also from Lawrence.



(image from `http://www.angelfire.com/ks/larrycarter/LC/OldGuardCameron.html`)

The HERMIT Project



Downloading and Running HERMIT

HERMIT requires GHC 7.4.

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

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

```
ghc Main.hs -fforce-recomp -O2 -dcore-lint
            -fsimple-list-literals -fplugin=HERMIT
```

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.

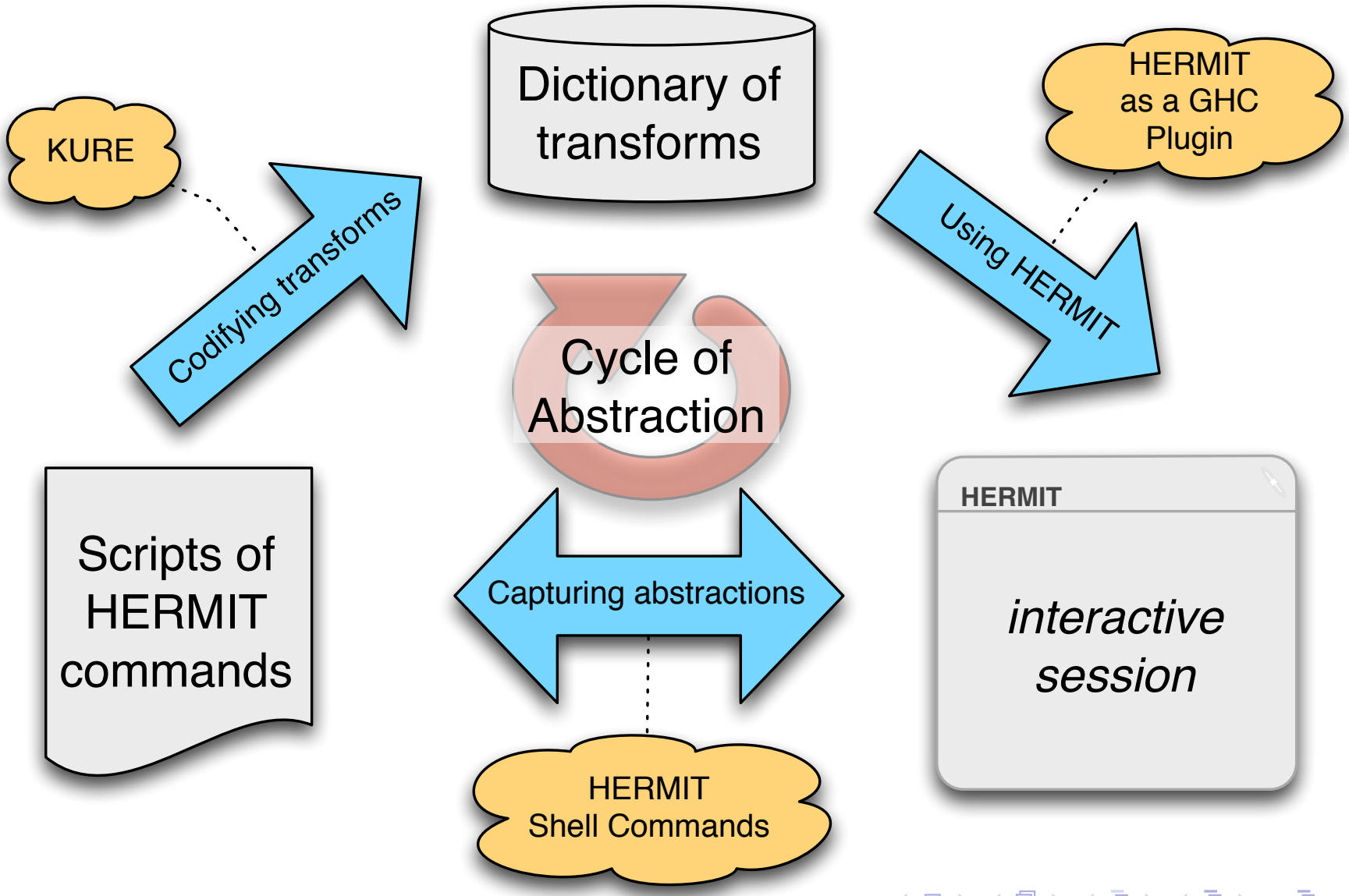
```
fib :: Int → Int
fib n = if n < 2
        then 1
        else fib (n − 1) + fib (n − 2)
```

```
fib :: Int → Int
fib n = if n < 2 then 1
                 else (if (n − 1) < 2 then 1
                                      else fib (n − 1 − 1) + fib (n − 1 − 2)
                      )
                      +
                      (if (n − 2) < 2 then 1
                                      else fib (n − 2 − 1) + fib (n − 2 − 2)
                      )
```

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
  - consider '*foo*
  - 0, 1, 2, ...
- Version control, e.g.
  - log
  - back
  - step
  - save "myscript.hss"

Developing Transformations



GHC RULES

GHC RULES

- GHC language feature allowing custom optimisations

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 #-}

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

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

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

Adding Transformations to HERMIT

Two main ways:

- Using KURE
  - very expressive
  - currently requires recompiling HERMIT
- 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

Demonstration: Fast Reverse

Consider transforming the slow (quadratic) version of reverse to the fast (linear) version:

```
rev :: [a] → [a]
rev [ ]      = [ ]
rev (x : xs) = rev xs ++ [x]
```

```
rev :: [a] → [a]
rev as = let work :: [a] → [a] → [a]
             work [ ]      ys = ys
             work (x : xs) ys = work xs (x : ys)
         in
             work as [ ]
```

HERMIT Summary

- A GHC plugin for interactive transformation of GHC Core programs
- Still early in development
- Next step: an equational reasoning framework that only allows correctness preserving transformations
- Publications:
  - *The HERMIT in the Machine* (Haskell '12) — describes the HERMIT implementation
  - *The HERMIT in the Tree* (submitted to IFL '12) – describes our experiences mechanising existing program transformations