The HERMIT in the Tree

Neil Sculthorpe Andrew Farmer Andy Gill

Functional Programming Group
Information and Telecommunication Technology Center
University of Kansas
{neil,afarmer,andygill}@ittc.ku.edu

24th Symposium on
Implementation and Application of Functional Languages
Oxford, England
31st 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

```
type CoreProg = [CoreBind]
data CoreBind = NonRec Var CoreExpr
              | Rec [(Var, CoreExpr)]
data CoreExpr = Var Var
              | Lit Literal
              | App CoreExpr CoreExpr
              | Lam Var CoreExpr
              | Let CoreBind CoreExpr
              | Case CoreExpr Var Type [CoreAlt]
              | Cast CoreExpr Coercion
              | Tick CoreTickish CoreExpr
              | Type Type
              | Coercion Coercion
type CoreAlt  = (AltCon, [Var], 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?

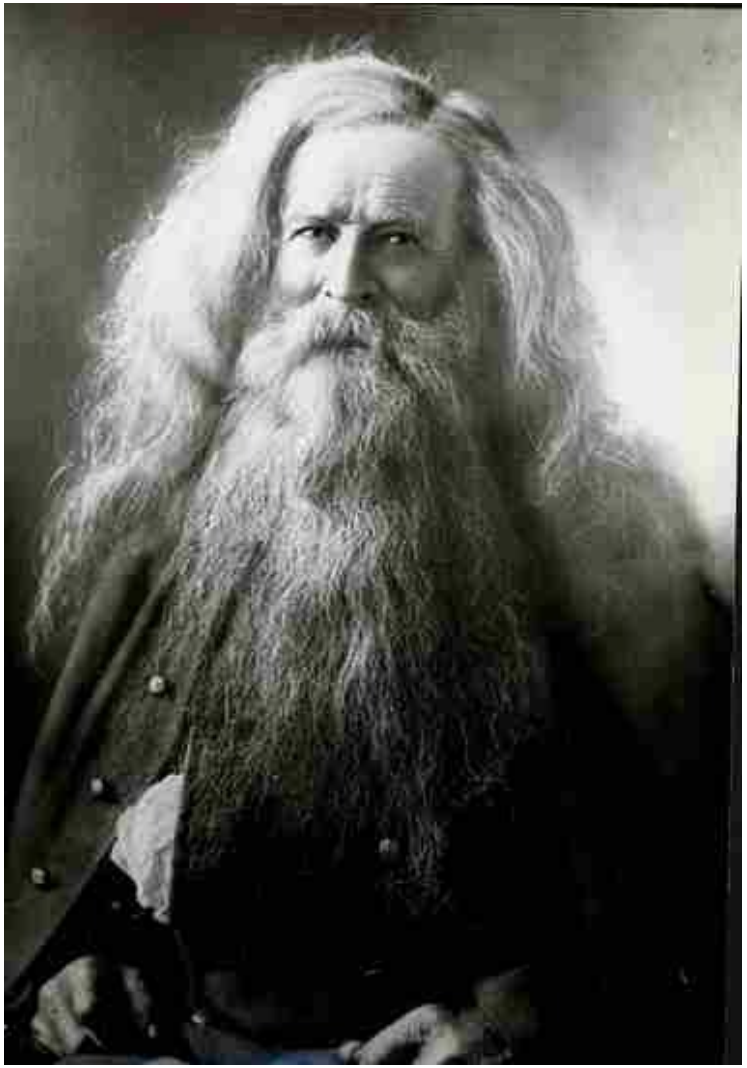
- Haskell Equational Reasoning Model-to-Implementation Tunnel
- A scriptable toolkit for interactive transformation of GHC Core programs.
- Under development at the University of Kansas, Lawrence.

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`)

Downloading and Running HERMIT

HERMIT requires GHC 7.4 or 7.6 (7.6 recommended)

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
            -fsimple-list-literals -fplugin=HERMIT
            -fplugin-opt=HERMIT:main:Main:
```

Demonstration: Unrolling Fibonacci

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

$$\textbf{data}\ Nat = \mathrm{Zero} \mid \mathrm{Succ}\ Nat$$

$$\begin{aligned}
&fib :: Nat \to Nat \\
&fib\ \mathrm{Zero} &&= \mathrm{Zero} \\
&fib\ (\mathrm{Succ}\ \mathrm{Zero}) &&= \mathrm{Succ}\ \mathrm{Zero} \\
&fib\ (\mathrm{Succ}\ (\mathrm{Succ}\ \mathrm{n})) &&= fib\ (\mathrm{Succ}\ \mathrm{n}) + fib\ \mathrm{n}
\end{aligned}$$

Demonstration: Unrolling Fibonacci

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

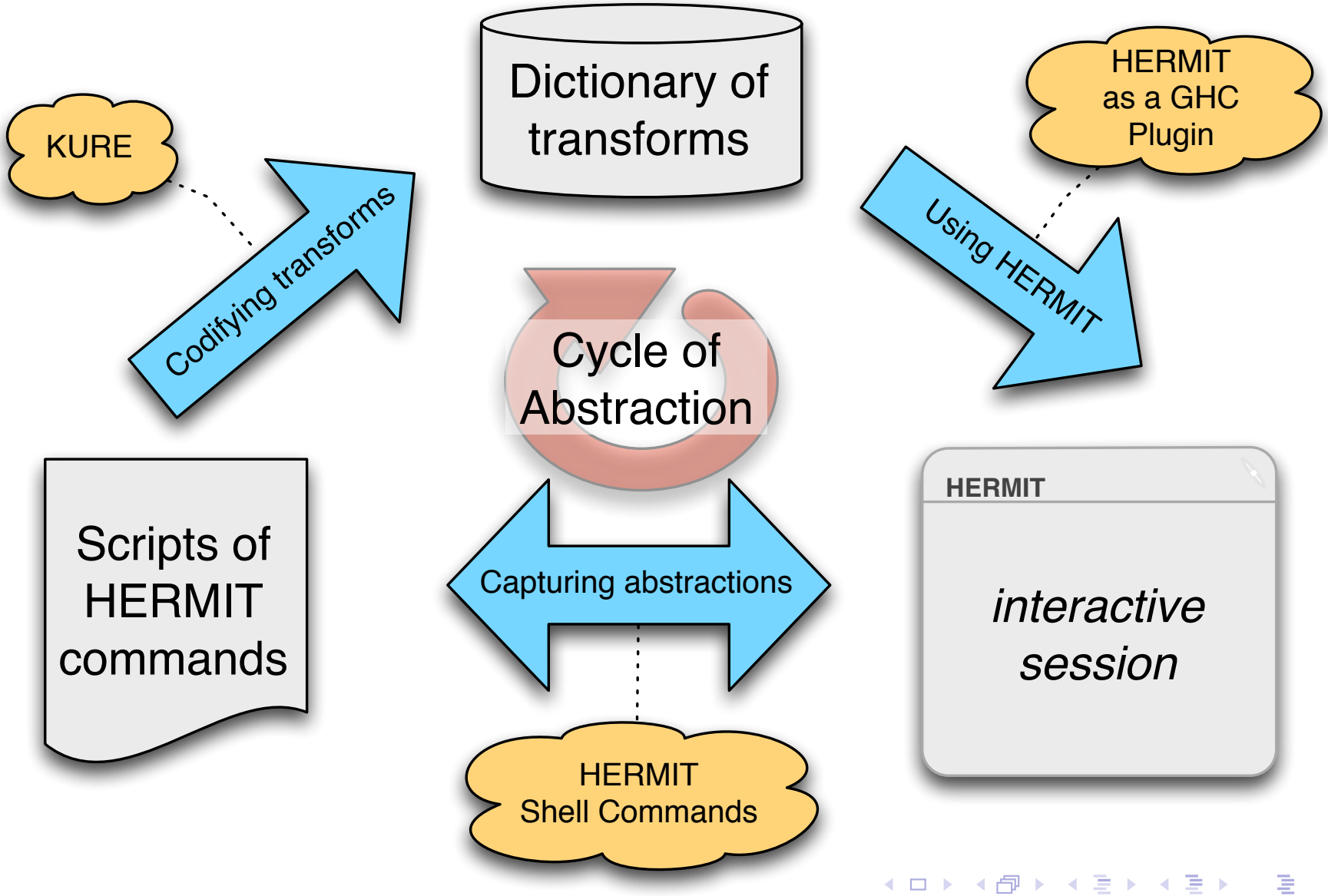
```
data Nat = Zero | Succ Nat

fib :: Nat → Nat
fib Zero              = Zero
fib (Succ Zero)       = Succ Zero
fib (Succ (Succ n))   = (case Succ n of
                           Zero          → Zero
                           Succ Zero     → Succ Zero
                           Succ (Succ m) → fib (Succ m) + fib m)
                         +
                         (case n of
                           Zero          → Zero
                           Succ Zero     → Succ Zero
                           Succ (Succ m) → fib (Succ m) + fib m)
```

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



Transformations Mechanised

We've been using HERMIT to mechanise some established program transformations:

- Concatenate Vanishes [Wad89]
- Tupling Transformation [Pet84]
- Worker/Wrapper [GH09]

Transformations Mechanised

We've been using HERMIT to mechanise some established program transformations:

- Concatenate Vanishes [Wad89]
- Tupling Transformation [Pet84]
- Worker/Wrapper [GH09]

In the process we've discovered that concatenate vanishes and tupling transformation can be expressed as instances of worker/wrapper.

Tupling Transformation: Fib

$$
\begin{aligned}
&\mathit{fib} :: \mathit{Nat} \to \mathit{Nat} \\
&\mathit{fib}\ \mathsf{Zero} &&= \mathsf{Zero} \\
&\mathit{fib}\ (\mathsf{Succ}\ \mathsf{Zero}) &&= \mathsf{Succ}\ \mathsf{Zero} \\
&\mathit{fib}\ (\mathsf{Succ}\ (\mathsf{Succ}\ \mathsf{n})) &&= \mathit{fib}\ (\mathsf{Succ}\ \mathsf{n}) + \mathit{fib}\ \mathsf{n}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{fib} :: \mathit{Nat} \to \mathit{Nat} \\
&\mathit{fib}\ \mathsf{n} = \textbf{let}\ \mathit{work} :: \mathit{Nat} \to (\mathit{Nat}, \mathit{Nat}) \\
&\qquad\qquad \mathit{work}\ \mathsf{Zero} \qquad = (\mathsf{Zero}, \mathsf{Succ}\ \mathsf{Zero}) \\
&\qquad\qquad \mathit{work}\ (\mathsf{Succ}\ \mathsf{m}) = \textbf{let}\ (\mathsf{x}, \mathsf{y}) = \mathit{work}\ \mathsf{m} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ (\mathsf{y}, \mathsf{x} + \mathsf{y}) \\
&\qquad \textbf{in} \\
&\qquad\qquad \mathit{fst}\ (\mathit{work}\ \mathsf{n})
\end{aligned}
$$

Summary

- HERMIT is a GHC plugin for interactive transformation of GHC Core programs
- Currently we're using it to mechanise known program transformations
- Next step: an equational reasoning framework that only allows correctness preserving transformations
- For HERMIT implementation details, see:
  - "The HERMIT in the Machine" (Haskell '12) [FGKS12]

References

Andrew Farmer, Andy Gill, Ed Komp, and Neil Sculthorpe.
The HERMIT in the machine: A plugin for the interactive transformation of GHC core language programs.
In *Haskell Symposium*, pages 1–12. ACM, 2012.

Andy Gill and Graham Hutton.
The worker/wrapper transformation.
Journal of Functional Programming, 19(2):227–251, 2009.

Alberto Pettorossi.
A powerful strategy for deriving efficient programs by transformation.
In *LISP and Functional Programming*, pages 273–281. ACM, 1984.

Philip Wadler.
The concatenate vanishes.
Technical report, University of Glasgow, 1989.