# A Language for Domain Specific Optimizations in Haskell

Andrew Farmer and Andy Gill

Information and Telecommunication Technology Center
The University of Kansas
{afarmer,andygill}@ittc.ku.edu

**Abstract.** Haskell programmers, especially those writing reusable libraries, spend considerable effort on directing the Glasgow Haskell Compiler's (GHC) optimizer in order obtain faster code. Many desired domain-and-program-specific optimizations cannot be expressed by GHC's current means of directing the optimizer. This paper presents a domain-specific language (DSL) for specifying these optimizations as custom compiler plugins using the infrastructure provided by the recently developed HERMIT plugin for GHC. This optimization DSL is used to implement an improved Stream Fusion deforestation optimization.

## 1 Introduction

The Glasgow Haskell Compiler (GHC) [GHC Team, 2013] offers an abundance of compiler flags and source-level pragma directives that enable the programmer to control the compilation of her program. Many of these "knobs and dials" are for controlling GHC's optimizer, allowing the programmer to try and direct optimization toward a desired result.

The right compiler flags can offer dramatic performance improvements for little cost to the programmer, but with the finesse of a sledgehammer. As an example, aggressive tuning of GHC's inliner can improve the performance of generic programming libraries by an order of magnitude, yet can have an adverse effect on other parts of the program, increasing code size and duplicating work [Magalhães et al., 2010].

GHC also offers source-level pragma statements to direct inlining [Peyton Jones and Marlow, 2002] and specialization [Peyton Jones, 2007], allowing functions to be annotated to guide the optimizer. Additionally, the programmer can use RULES pragmas [Peyton Jones et al., 2001] to extend GHC's optimizer with domain-specific rewrite rules, allowing the optimizer to make use of of the programmer's high-level knowledge about the structure of the program. In general, pragmas give the programmer more fine-grained control than flags, but with limited expressivity. Pragmas cannot be placed on bindings which are not present in the source, such as those introduced by other compiler passes [Jones and Launchbury, 1991, Peyton Jones, 2007]. The implementation of type classes using implicit dictionary arguments can lead to unexpected mutual recursion that prevents desired inlining [Jones, 1995]. RULES pragmas are limited in the syntactic constructs that can be matched, meaning many valid rules are inexpressible in the system.

```
data ModGuts  = ModGuts { _ :: [CoreBind], ...  }

data CoreBind = NonRec Var CoreExpr | Rec [CoreDef]

data CoreDef  = Def Var CoreExpr

data CoreExpr = Var Var | Lit Literal | Type Type
              | App CoreExpr CoreExpr | Lam Var CoreExpr
              | Let CoreBind CoreExpr | Case CoreExpr Var Type [CoreAlt]
              | Cast CoreExpr Coercion | Coercion Coercion
              | Tick CoreTickish CoreExpr

type CoreAlt  = (AltCon, [Var], CoreExpr)
```

Fig. 1: GHC Core.

To summarize, attempting to guide the optimizer via command line flags or source program annotations means that some optimization opportunities will be missed, because the knowledge required to exploit them cannot be expressed. This paper introduces a domain-specific language (DSL) for specifying custom optimizer plugins for GHC, allowing the programmer to express new optimizations at a high level of abstraction.

Specifically, this paper makes the following contributions.

- A domain-specific language for manipulating and augmenting GHC's optimization pipeline. (Section 3)
- A principled set of strategic-programming combinators for pattern-matching on and unfolding function calls. (Section 4)
- An implementation of Stream Fusion which optimizes `concatMap` using the transformation suggested by Coutts [2010] (Section 5.1), then extended to optimize more general uses (Section 5.2).

## 2 HERMIT and KURE

In this section we briefly overview HERMIT, KURE, and GHC's intermediate language, Core.

### 2.1 Core

GHC's front end transforms Haskell source into an intermediate language called Core (Fig. 1). The optimizer is written as a pipeline of Core-to-Core passes which manipulate this Core program. Core is an implementation of System $F_C^{\uparrow}$ [Sulzmann et al., 2007, Yorgey et al., 2012], which is System F [Pierce, 2002] extended with let-bindings, constructors, type coercions and algebraic and polymorphic kinds. Types in Core are explicitly passed as arguments, but never returned.

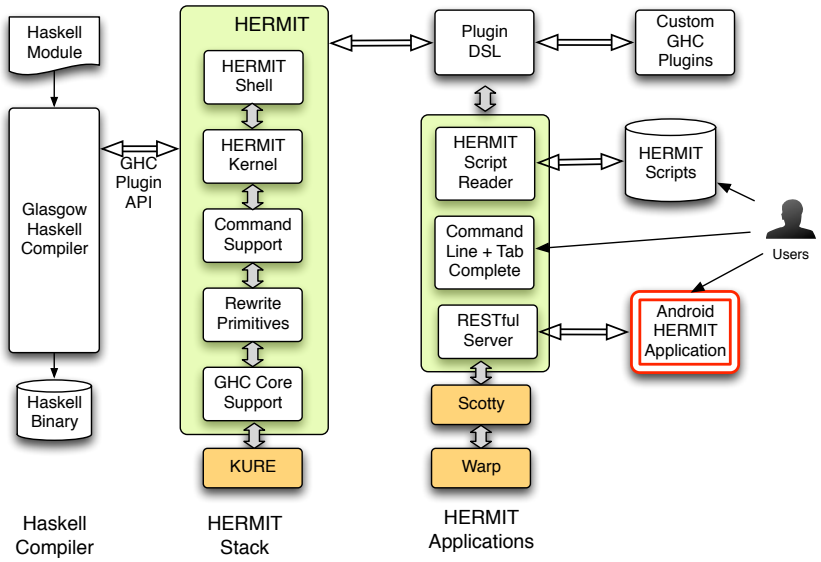


Fig. 2: Architecture of HERMIT

### 2.2 HERMIT

HERMIT is a GHC plugin that allows the programmer to interact with the GHC Core representation of their program *during compilation*. Fig. 2 presents the architecture of the HERMIT system. More details of HERMIT's implementation and use can be found in [Farmer et al., 2012] and [Sculthorpe et al., 2013b], respectively.

A key contribution of this paper is to develop HERMIT's capabilities for non-interactive use. Specifically, in the diagram in Fig. 2, the component labeled "Plugin DSL" is new, and the topic of this paper. This DSL operates at two levels. The first, outer level is concerned with specifying HERMIT's behavior in the context of the GHC optimization pipeline. That is, when and where transformations are applied. It is introduced in Section 3. The second level is concerned with manipulating Core itself. For this, we use HERMIT's existing strategic rewriting facility provided by KURE.

### 2.3 KURE

KURE is a strongly-typed strategic rewriting DSL [Gill, 2009, Sculthorpe et al., 2013c]. HERMIT transformations are implemented as KURE transformations parameterized over a specific context and monad, targeting a sum type which contains the mutually recursive types that make up the Core language. The two principal types for HERMIT transformations are `TranslateH a b`, which transforms a value of type `a` into one of type `b`, and `RewriteH a`, which is equivalent to `TranslateH a a`.

HERMIT comes with an existing set of KURE transformations implementing common rewrites, such as $\beta$-reduction and let floating. It also implements a set of congruence combinators, which allow for concise pattern matching on, and transformation of, syntactic constructs. As an example, the following transformation returns a list of dead let bindings.

```
deadBinders :: TranslateH CoreExpr [Var]
deadBinders = letT bindVarsT freeVarsT (\\)

bindVarsT :: TranslateH CoreBind [Var]
freeVarsT :: TranslateH CoreExpr [Var]
(\\) :: Eq a => [a] -> [a] -> [a]
```

The `letT` congruence combinator pattern matches on a let(rec) binding expression. Applying `deadBinders` to a non-let expression will cause the translation to fail. The `bindVarsT` transformation is applied to the binding group, returning a list of bound variables. The `freeVarsT` transformation is applied to the let body, returning a list of free variables. These two lists are combined using Haskell's list-difference function, removing all the free variables from the list of bound variables and returning the result.

## 3 Controlling the Optimizer

To date, HERMIT has primarily been developed as an *interactive* tool for performing transformations on Core. As such, HERMIT runs at the front of the optimization pipeline (as the very first Core-to-Core pass), where the Core representation of a program is more easily related to the source program.

The goal of this section is to develop HERMIT's capabilities to express transformations that can be run in an automated manner. Much work can be saved by allowing existing GHC passes to do their work, only invoking a HERMIT transformation at the right moment. As a practical first step, HERMIT was modified to run both before and after each Core-to-Core pass. Each time a HERMIT pass is invoked, it determines whether to perform any transformations.

In order to specify what HERMIT should do each time it runs, we developed a deeply embedded monadic DSL. A single top-level function called `optimize` lifts a computation expressed using this DSL into a custom GHC plugin. As a first example, the existing HERMIT plugin, which runs the interactive HERMIT shell, is specified using:

```
plugin :: Plugin
plugin = optimize $ \ options -> phase 0 $ interactive [] options
```

Briefly, `optimize` takes a callback function which accepts per-module command-line options specified when HERMIT is invoked and returns a monadic description of the desired plugin behavior. The plugin above runs the interactive shell in phase zero, which is at the front of the pipeline. The type of `optimize` is:

```
optimize :: ([CommandLineOption] -> OM ()) -> Plugin
```

`OM` is a normalized deep embedding of a monadic computation by way of the `operational` package [Apfelmus, 2010a,b]. `operational` uses the monad laws to normalize the structure of the computation, resulting in a sequence of primitive operations terminated by a single `return`. Writing an interpreter for the language consists of defining an appropriate action for each primitive, and `operational` guarantees that the resulting monad obeys the monad laws. The primitives for `OM` are specified by the `OInst` generalized abstract data-type.

```
data OInst :: * -> * where
    RR     :: RewriteH Core                     -> OInst ()
    Query  :: TranslateH Core a                 -> OInst a
    Shell  :: [External] -> [CommandLineOption] -> OInst ()
    Guard  :: (PhaseInfo -> Bool) -> OM ()      -> OInst ()
    Focus  :: TranslateH Core Path -> OM ()     -> OInst ()
```

Primitives can either be rewrites or translations, a call to the interactive shell, a guard which uses a predicate to determine whether to run an inner computation, or a means applying an inner computation at a specific point in the AST. A current limitation is that calls to the shell never return, so subsequent actions are silently ignored. This limitation can be lifted with a significant redesign of the interactive shell, which remains future work.

Making the interactive shell a primitive in our DSL is the key feature that allows users to extend HERMIT with new primitive transformations without recompiling HERMIT itself. To add a new transformation, one provides a list of `External`s to the `interactive` function. The `External` type is defined by HERMIT, and is used to associate a transformation with a name and metadata such as help text and tags. The `interactive` function is the user-facing wrapper around the `Shell` primitive.

```
myRewrite :: RewriteH Core
myRewrite = ...

cmds :: [External]
cmds = [ external "my-rewrite" myRewrite [ "help text for my rewrite" ] ]

plugin :: Plugin
plugin = optimize $ \ options -> phase 0 $ interactive cmds options
```

In addition to the small examples in this section, this facility has been used to develop a custom optimization for the Scrap Your Boilerplate [Lämmel and Peyton Jones, 2003, 2004] generics library [Adams et al., 2013]. It will also be used in Section 5 to develop a custom Stream Fusion optimization.

The `Guard` primitive enables the ability to write HERMIT plugins which operate in multiple phases of the optimization pipeline. It accepts a predicate which can examine information about the current phase in order to signal whether the inner computation should be run. Note that the return type of `Guard` is always `()`, since the possible failure of the guard condition means there can be no return value.

**Primitives**

| | |
|---|---|
| **run** | Apply a KURE Rewrite. |
| **query** | Apply a KURE Translation, returning the result. |
| **interactive** | Start the HERMIT interactive shell. |
| **display** | Dump the currently focused AST using HERMIT's pretty printer. |

**Guards**

| | |
|---|---|
| **guard** | Use a predicate on current phase information to enable/disable actions in certain phases. |
| **phase** | Only run action in given phase. |
| **after** | Run action directly after named phase. |
| **before** | Run action directly before named phase. |
| **allPhases** | Run action in all phases (default). |

**Focusing**

| | |
|---|---|
| **at** | Use a path-generating translation to focus actions in the Core AST. |
| **when** | Apply an action to all places in the AST where a given translation succeeds. |

Fig. 3: Plugin pass domain-specific language.

A summary of the capabilities of this DSL is in Fig. 3. Each GHC pass is named using an enumeration type, allowing transformations to be scheduled before or after a named pass. As a small example, the following plugin runs directly after GHC's constructor specialization pass, displaying the definitions of functions whose names have been specified on the command line.

```
plugin :: Plugin
plugin = optimize $ \ fns -> after SpecConstr $
    forM_ fns $ \ fn -> at (considerName fn) display
```

Plugins implemented with `optimize` can be packaged as normal Cabal packages which depend on the HERMIT library. The HERMIT driver program has been modified to support loading these custom plugins. (Note that, as before, the HERMIT driver simply calls GHC with appropriate arguments. It is merely provided for convenience.) Here is an example call to HERMIT using a custom plugin.

```
hermit Sum.hs -opt=HERMIT.Optimization.SYB +Main mapIntM
```

This invokes HERMIT on Sum.hs, targeting the `Main` module, using the `plugin` found in the `HERMIT.Optimization.SYB` module. Values after the target module (`+Main`) are passed as command-line options to the callback given to `optimize`. In this case, they are interpreted as the names of functions to target with the optimization. Note that multiple target modules may be specified.

## 4 Rewriting Function Calls

It is often the case that one wishes to rewrite function calls. HERMIT provides a traversal strategy called `anyCallR` to *locate* function calls in a topdown manner.

It offers no support, however, for pattern matching and deconstructing them in the spirit of congruence combinators such as `letT`, described in Section 2.3. In the course of this work, we discovered a pleasing set of combinators for this task, which we describe in this section.

The motivating example is a rewrite which unfolds a definition. While inlining simply replaces a variable occurence with its definition, unfolding attempts to substitute the arguments of the application into the body of the inlined function. The inline rewrite only needs to pattern match on a single variable occurence, which it does with the `varT` congruence combinator. The resulting variable is then found in HERMIT's context, which maps binders to their right-hand sides. HERMIT's inlining rewrite is called `inlineR`.

An unfold rewrite must pattern match on both a single variable and a tree of application nodes. The former case occurs when unfolding functions with no arguments. When encountering a tree of applications, the left-most deepest leaf should be inlined (as application is left associative), then the tree should be maximally $\beta$-reduced. Assuming the existence of the $\beta$-reducing rewrite, the unfold rewrite can be defined as:

```
unfoldR :: RewriteH CoreExpr
unfoldR = leftmostDeepest >>> betaReduceStarR
    where leftmostDeepest :: RewriteH CoreExpr
          leftmostDeepest = inlineR <+ appAllR leftmostDeepest idR
```

This definition first inlines the left-most leaf of the application tree, then fully $\beta$-reduce the result. To inline the left-most leaf, we first try `inlineR` on the current node, in case it is a variable. If `inlineR` succeeds, we are done. If it fails, we try pattern matching on an application node with `appAllR`. If the node is not an application, this will fail, causing the entire `unfoldR` rewrite to fail. If it is an application node, we recursively apply `leftmostDeepest` to the left child and the identity rewrite `idR` to the right child. The only way for `leftmostDeepest` to succeed is if a left child can eventually be inlined. When this occurs, `betaReduceStarR` is called on the *transformed* application tree at location where `unfoldR` was originally called, giving it access to the arguments necessary for $\beta$-reduction.

The definition of `unfoldR` is pleasingly concise; a testament to the power of strategic programming. However, it is also rather indescriminate about *what* it unfolds. In practice, we likely want to only unfold a call to a certain function, or all calls with certain properties. HERMIT provides a function for inlining a specific variable:

```
inlineNameR :: Name -> RewriteH CoreExpr
```

It is tempting to create the following additional unfold rewrite, putting `inlineNameR` to use:

```
unfoldNameR :: Name -> RewriteH CoreExpr
unfoldNameR nm = leftmostDeepest >>> betaReduceStarR
    where leftmostDeepest :: RewriteH CoreExpr
          leftmostDeepest = inlineNameR nm <+ appAllR leftmostDeepest idR
```

However, this seems like unnecessary code duplication. We need a means of pattern matching on function calls before calling `unfoldR`. To do so, we define the following primitive, which deconstructs applications using the GHC `collectArgs` function. The first component of the returned pair is the function being called, and the second component is a list of argument expressions.

```
callT :: TranslateH CoreExpr (CoreExpr, [CoreExpr])
callT = contextfreeT $ \ expr ->
    case expr of
        Var {} -> return (expr, [])
        App {} -> return (collectArgs expr)
        _      -> fail "not an application or variable occurence."
```

Using `callT`, we can now build translations which only deconstruct certain function calls. Note the use of `guardMsg`, which accepts as arguments a boolean and a string. If the boolean is `False`, the entire translation fails with the message contained in the string.

```
callPredT :: (Id -> [CoreExpr] -> Bool)
          -> TranslateH CoreExpr (CoreExpr, [CoreExpr])
callPredT p = do
    call@(Var i, args) <- callT
    guardMsg (p i args) "predicate failed."
    return call

callNameT :: Name -> TranslateH CoreExpr (CoreExpr, [CoreExpr])
callNameT nm = callPredT (const . compareNameToId nm)

callSaturatedT :: TranslateH CoreExpr (CoreExpr, [CoreExpr])
callSaturatedT = callPredT (\ i args -> idArity i == length args)
```

We can combine these transformations with `unfoldR` to obtain a variety of unfold rewrites.

```
unfoldPredR :: (Id -> [CoreExpr] -> Bool) -> RewriteH CoreExpr
unfoldPredR p = callPredT p >>= \ _ -> unfoldR

unfoldNameR :: Name -> RewriteH CoreExpr
unfoldNameR nm = callNameT nm >>= \ _ -> unfoldR

specializeR :: RewriteH CoreExpr
specializeR = unfoldPredR (const (all isTyCoArg))
```

In summary, two primitive rewrites (`unfoldR` and `callT`) form the basis of a rich set of combinators for inspecting and transforming function calls. Combined with HERMIT's existing traversal combinator `anyCallR`, they allow for the concise specification of unfolding transformations. Fig. 4 lists some of the newly available transformations.

| | |
|---|---|
| **unfoldR** | Unfold a function call. |
| **unfoldPredR** | Unfold a function call which passes a predicate test. |
| **unfoldNameR** | Unfold a call to a named function. |
| **unfoldAnyR** | Unfold a call to a function present in a list of names. |
| **unfoldInlinableR** | Unfold a call if GHC considers the function 'inlinable' |
| **unfoldSaturatedR** | Unfold a completely saturated call. |
| **specializeR** | Specialize an application to its type and coercion arguments. |

Fig. 4: New Unfolding Transformations

## 5 Case Study: `concatMap`

In functional languages, it is often natural to implement sequence-processing pipelines by gluing together reusable combinators such as `foldr` and `zip` which encapsulate a specific recursion pattern. These combinators communicate their results to the next function in the pipeline by means of intermediate data structures. Naïvely compiled, these intermediate structures adversely affect performance as they must be allocated and subsequently garbage collected.

Fusion is the process of transforming these combinator pipelines in order to eliminate as many intermediate structures as possible. Intuitively, rather than allow each combinator to transform the entire sequence in turn, the resulting code processes sequence elements in an assembly-line fashion. In many cases, after fusion, no sequence structure need be allocated at all.

Stream Fusion [Coutts et al., 2007] is known as a short-cut fusion system. An excellent and thorough overview of Stream Fusion and rival systems, such as GHC's foldr/build, can be found in Coutts [2010]. The foldr/build system fuses `concatMap` well, but cannot fuse combinators which consume more sequences than they produce (or vice versa), such as `zip` and `unzip`. An alternative system, known as unfoldr/destroy (alternatively unfoldr/unbuild), exists which can properly fuse `zip`, but cannot fuse `filter`.

Stream Fusion, on the other hand, excels at fusing `zip` and `filter`, but cannot fuse `concatMap`. As `concatMap` is the combinator underlying nested list comprehensions, this is a major drawback to the system. Stream Fusion relies on a conversion to the eponymous `Stream` representation type, with a rewrite rule to eliminate redundant conversions. To see why `concatMap` cannot be fused requires understanding this type.

```
data Stream a = forall s. Stream (s -> Step a s) s
data Step a s = Done | Skip s | Yield a s
```

A stream is a pair of generator function and state. The key to the problem is that the state of a stream is existentially quantified. Consider the type of Stream Fusion's version of `concatMap`.

```
concatMapS :: (a -> Stream b) -> Stream a -> Stream b
```

The state of the inner stream is existentially quantified, meaning an entirely new state and generator function could be returned each time the function is given a value of the outer stream. To draw a parallel to loops, `concatMapS` has the ability to express this computation.

```
for (int i = 1; i <= 10; i++)
    switch i {
        case 1: for (int j = 1; j < 10; j++) { ... }
        case 2: for (int k = 100; k > 0; k = k/2) { ... }
        ...
    }
```

There is obviously no hope of fusing the case where each value of the outer sequence selects a different inner sequence. A less powerful combinator, sometimes called `flattenS` [Höner zu Siederdissen, 2012], can be fused, however.

```
flattenS :: (s -> Step b s) -> (a -> s) -> Stream a -> Stream b
```

`flattenS` makes explicit that the state and generator function are statically known, regardless of the value present in the outer stream. The disadvantage is that `flattenS` is more difficult to use. Whereas the rest of the Stream Fusion system hides the complexity of state and generator functions from the programmer, providing familiar sequence combinators, `flattenS` requires one to think in terms of generator functions and state. A call to `concatMapS` with a complicated inner stream pipeline can make use of existing stream combinators, while `flattenS` requires the programmer to write a hand-fused, potentially complex generator function.

*Proposed Solution* In his dissertation, Coutts [2010] proposes the following transformation for optimizing common uses of `concatMapS` by transforming them into calls to `flattenS`.

$$\forall\ g\ s\ strm.\ concatMapS\ (\lambda x \rightarrow Stream\ g\ s)\ strm \equiv flattenS\ g\ (\lambda x \rightarrow s)\ strm$$

This transformation is subject to certain restrictions. First, `x` cannot be free in `g`, as it would become unbound in the resulting code; a side condition which is inexpressible in GHC RULES rewrites. Second, the function passed to `concatMapS` must return the same inner stream regardless of the value of `x`. For instance, the following call to `concatMapS` could not be transformed with this rule.

```
concatMapS (\x -> case odd x of
                    True  -> Stream g1 s1
                    False -> Stream g2 s2) strm
```

The fact that one cannot pattern match against case expressions in RULES pragmas means that even with a scheme for combining generator functions and states in a suitable way, the rewrite cannot be expressed. Expressing this transformation and applying it automatically during compilation is the topic of the remainder of this paper.

### 5.1 Implementing `concatMap` Optimization

Expressing the proposed transformation rule as a HERMIT primitive is straightforward. First, we use `callNameT` from Section 4 to pattern match on a call to

`concatMapS`, extracting its arguments. It is important to remember that universally quantified types in Core are passed as explicit arguments, so they must be included in the pattern match. We then use an auxiliary transformation that we name `exposeInnerStreamT` to extract relevant information from the function argument to `concatMapS`. After checking the side condition that `x` is not free in the generator function, we use this information, the identifier for `flattenS`, and some GHC functions to build the resulting expression.

```
concatMapSR :: RewriteH CoreExpr
concatMapSR = do
    (_, [aTy, bTy, f, outerStream]) <- callNameT (TH.mkName "concatMapS")
    (x, gen, st) <- applyInContextT exposeInnerStreamT f
    fvs <- applyInContextT freeVarsT gen
    guardMsg (x `notElem` fvs) "x would become unbound in generator."
    flattenSid <- findIdT $ TH.mkName "flattenS"
    return $ mkCoreApps (varToCoreExpr flattenSid)
                        [ Type (exprType st), bTy, aTy
                        , gen, Lam x st, outerStream]
```

`exposeInnerStreamT` uses the `lamT` congruence combinator to pattern match on an explicit lambda, examining its body. If the body is an application of the `Stream` constructor, it extracts and returns the relevant information.

```
exposeInnerStreamT
    :: TranslateH CoreExpr ( CoreBndr   -- the 'x' in 'concatMap (\x -> ...)'
                           , CoreExpr   -- inner stream stepper function
                           , CoreExpr ) -- inner stream state
exposeInnerStreamT =
    lamT (callDataConNameT $ TH.mkName "Stream")
         (\ x (_dc, _univTys, [_sTy, gen, st]) -> (x, gen, st))
```

### 5.2 The Stream Within

The transformation in the previous section is subject to a number of limitations. As previously mentioned, the generator for the inner stream cannot depend on the value of the outer stream. Additionally, `exposeInnerStreamT` requires a very specific syntactic form for the function it examines. In this section, we improve our transformation to lift these restrictions. The entire transformation, after the these modifications, can be found in Fig. 5.

**Non-Constant Inner Streams** The biggest limitation is the free variable check on the generator function. For any interesting use of `concatMapS`, this will fail. Consider the following call, where the generator for the inner `enumFromToS` will necessarily depend on `x` in order to know when to stop generating additional values.

```
concatMapS (\x -> enumFromToS 1 x) (enumFromToS 1 100)
```

```
concatMapSR :: RewriteH CoreExpr
concatMapSR = do
    (_, [aTy, bTy, f, outerStream]) <- callNameT (TH.mkName "concatMapS")

    (x, gen@(Lam s _), st) <- applyInContextT exposeInnerStreamT f

    flattenSid <- findIdT $ TH.mkName "flattenS"
    fixStepid <- findIdT $ TH.mkName "fixStep"

    let st' = mkCoreTup [varToCoreExpr x, st]
    stId <- constT $ newIdH "st" (exprType st')
    wild <- constT $ cloneVarH ("wild_"++) stId

    let fixApp = mkCoreApps (varToCoreExpr fixStepid)
                            [ aTy, bTy, Type $ exprType st
                            , varToCoreExpr x, mkCoreApp gen (varToCoreExpr s) ]
        genFn = mkCoreLams [stId] $
                    mkSmallTupleCase [x,s] fixApp wild (varToCoreExpr stId)

    return $ mkCoreApps (varToCoreExpr flattenSid)
                        [ Type (exprType st'), bTy, aTy
                        , genFn, Lam x st', outerStream ]

exposeInnerStreamT
    :: TranslateH CoreExpr ( CoreBndr   -- the 'x' in 'concatMap (\x -> ...)'
                           , CoreExpr   -- inner stream stepper function
                           , CoreExpr ) -- inner stream state
exposeInnerStreamT =
   (lamR exposeStreamConstructor >>>
    lamT (callDataConNameT $ TH.mkName "Stream")
         (\ x (_dc, _univTys, [_sTy, gen, st]) -> (x, gen, st)))
    <+ (unfoldR >>> exposeInnerStreamT)

exposeStreamConstructor :: RewriteH CoreExpr
exposeStreamConstructor = tryR $ extractR $ repeatR $
    onetdR (promoteExprR $ rules ["stream/unstream", "unstream/stream"]
                           <+ letUnfloat <+ letElim <+ caseUnfloat)
     <+ simplifyR <+ promoteExprR unfoldR
```

Fig. 5: The final `concatMapS` transformation.

To lift this restriction, we stash `x` in the state. This involves changing the state type of the inner stream, which makes generating code for the resulting call to `flattenS` significantly more complicated. In essence, we tuple the value of the outer stream (`x`) with the original inner stream state. Then we build a new generator function using a new binder of the appropriate type. This generator cases on the new state to expose the original state, which it passes to the original generator. It then places `x` back into the resulting state using a helper function called `fixStep`.

```
fixStep :: a -> Step b s -> Step b (a,s)
fixStep _ Done        = Done
fixStep a (Skip s)    = Skip (a,s)
fixStep a (Yield b s) = Yield b (a,s)
```

The definition of `fixStep` is required to be in scope in the *target* program. This may seem curious, but currently we find building arbitrary Core expressions to be error-prone and time-consuming. It is easier to build a simple function application than to modify arbitrary code to properly tuple the state.

The resulting transformation can dispense with the free variable check, as now `x` is bound by scrutinizing the new state with a case expression.

**Non-Explicit λs** The `exposeInnerStreamT` transformation is limited to matching explicit lambda expressions. If the function passed to `concatMapS` is a partially applied function, there will be no explicit lambda manifest. Consider:

```
concatMapS (enumFromToS 1) (enumFromToS 1 100)
```

We could attempt to $\eta$-expand the function, but find it better to attempt to unfold instead, since we eventually also need an explicit `Stream` constructor.

**Unfloating** The final challenge is to get the `Stream` constructor to the head of the body of the lambda expression. Often, there are let bindings wrapping the constructor. We can push these into the arguments of the constructor (unfloat them). Unfloating into arguments necessarily duplicates let bindings. This loss of sharing could result in duplicated computation. In practice, it appears rare that a let binding is used in both the generator and the state (the two arguments to the `Stream` constructor), so at least one will usually be eliminated by dead code removal. However, we would like to examine this consequence in future work.

More interestingly, it turns out we can unfloat *case* expressions into the arguments of the Stream constructor. Normally this sort of transformation affects termination properties of the program, as case expressions perform evaluation in Core. In this case, our transformation is discarding the `Stream` constructor of the inner stream anyway, so termination behavior is preserved.

Unfloating case expressions effectively merges multiple streams (one from each case alternative) into a single stream with a more complicated generator function and state. While unfloating case expressions enables the overall `concatMapS` transformation, more examples need to be studied to determine how this affects the resulting Core. Preliminary examination shows that all `Step` constructors are fused away for simple examples.

If the pipeline for the inner stream involved multiple stream combinators, there may be residual `stream`/`unstream` pairs to eliminate. This can also happen if a pair of `stream` and `unstream` were previously separated by let bindings or a case statement which have now been unfloated.

These transformations are implemented by `exposeStreamConstructor`, which can be thought of as exposing a view of the function body that is more useful to `exposeInnerStreamT`.

### 5.3 The Plugin

The overall structure of the optimization is simple. We repeatedly apply GHC RULES pragmas, lifted into HERMIT, to replace normal sequence processing combinators with their stream counterparts; fuse pairs of `stream` and `unstream`; and simplify the expression with HERMIT's `bash` command, which is akin to GHC's simplifier. Next we apply our custom `concatMapS` transformation. Finally, we inline all Stream Fusion combinators and simplify, handing the result back to GHC for further optimization.

```
plugin :: Plugin
plugin = optimize $ \ opts -> phase 0 $ run $ tryR $
    repeatR (anybuR (rules [ "stream/unstream", "unstream/stream"
                           , ... list of Stream Fusion rules ... ])
             <+ bash)
    >>> tryR (anybuR concatMapSR)
    >>> repeatR (anyCallR
                   (unfoldAnyR [ "fixStep", "flattenS"
                               , ... list of combinators ... ]))
    >>> bash
```

## 6 Conclusion and Future Work

In this paper, we introduced a monadic DSL for controlling when and where transformations are applied to GHC's intermediate representation for Haskell programs. We also present a principled set of strategic-programming combinators for transforming function calls. Putting these two contributions to use, we implemented a custom GHC plugin for optimizing the `concatMap` combinator. This plugin was then extended to increase optimization opportunities.

The plugin DSL, though currently small, allows optimization algorithms to be expressed naturally and concisely. It remains to be seen how it will fare in the presence of the ability to *rearrange* other optimization passes, which is something we desire for HERMIT in the future.

The strategic programming combinators for function calls are lightweight and effective. As the `specializeR` example in Section 4 demonstrates, they enable terse encodings of complex transformations.

The experience of building a custom optimization for `concatMap` was illuminating. It revealed several aspects of HERMIT that are still painful, including debugging large transformations and building well-typed Core expressions. One insidious bug, where the first two type arguments to the application of `flattenS` built by the `concatMapSR` rewrite were reversed, kept one author occupied for hours. A safer means for constructing expressions is clearly needed.

Our next step is to apply our Stream Fusion optimization to large programs to measure its effectiveness in the wild. An enticing candidate is the `ADPfusion` library, which goes to elaborate lengths to make using the `flattenS` combinator on nested multi-dimensional vectors less painful [Höner zu Siederdissen, 2012]. It would be interesting to replace the `flattenS` machinery with more natural

calls to `concatMap`, optimize, then compare the results, both by examining the Core and by benchmarking.

So far, we are only interpreting the monadic DSL, but the deep embedding permits future opportunities to optimize the computation, or compile it [Sculthorpe et al., 2013a]. One could, for instance, compile a computation of type `OM a` into a KURE translation of type `TranslateH Core a`.

**Acknowledgements**

We thank Nicolas Frisby and Neil Sculthorpe for feedback on drafts of this paper. This material is based upon work supported by the National Science Foundation under Grant No. 1117569.

## References

M. Adams, A. Farmer, and J. P. Magalhes. Optimizing syb is easy! URL `http://www.ittc.ku.edu/csdl/fpg/files/Adams-13-OSIE.pdf`. Submitted to the International Conference on Functional Programming, 2013.

H. Apfelmus. The Operational monad tutorial. *The Monad.Reader*, 15:37–55, 2010a.

H. Apfelmus, 2010b. URL `http://hackage.haskell.org/package/operational`.

D. Coutts. *Stream Fusion: Practical Shortcut Fusion for Coinductive Sequence Types*. PhD thesis, University of Oxford, 2010.

D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: from lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 315–326, Freiburg, Germany, 2007. ACM.

A. Farmer, A. Gill, E. Komp, and N. Sculthorpe. The HERMIT in the machine: A plugin for the interactive transformation of GHC core language programs. In *2012 ACM SIGPLAN Haskell Symposium*, pages 1–12, New York, 2012. ACM.

GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.6.2*, 2013. URL `http://www.haskell.org/ghc`.

A. Gill. A Haskell hosted DSL for writing transformation systems. In *Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*, DSL '09, pages 285–309. Springer-Verlag, July 2009.

C. Höner zu Siederdissen. Sneaking around concatmap: efficient combinators for dynamic programming. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pages 215–226, Copenhagen, Denmark, 2012. ACM.

M. P. Jones. Dictionary-free overloading by partial evaluation. *Lisp Symb. Comput.*, 8(3):229–248, Sept. 1995. ISSN 0892-4635.

S. L. P. Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 636–666. Springer-Verlag New York, Inc., 1991.

R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Types in Languages Design and Implementation*, pages 26–37. ACM, 2003.

R. Lämmel and S. Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *International Conference on Functional Programming*, pages 244–255. ACM, 2004.

J. P. Magalhães, S. Holdermans, J. Jeuring, and A. Löh. Optimizing generics is easy! In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 33–42. ACM, 2010.

S. Peyton Jones. Call-pattern specialisation for haskell programs. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, pages 327–337. ACM, 2007.

S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4&5):393–433, 2002.

S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Haskell Workshop 2001*, pages 203–233, 2001.

B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

N. Sculthorpe, J. Bracker, G. Giorgidze, and A. Gill. The constrained-monad problem. URL `http://www.ittc.ku.edu/csdl/fpg/files/Sculthorpe-13-ConstrainedMonad.pdf`. Submitted to the International Conference on Functional Programming, 2013a.

N. Sculthorpe, A. Farmer, and A. Gill. The HERMIT in the tree: Mechanizing program transformations in the GHC core language. In *Proceedings of the 24th Symposium on Implementation and Application of Functional Languages*, 2013b. URL `http://www.ittc.ku.edu/csdl/fpg/files/Sculthorpe-13-HERMITinTree.pdf`.

N. Sculthorpe, N. Frisby, and A. Gill. KURE: A Haskell-embedded strategic programming language with custom closed universes. URL `http://www.ittc.ku.edu/csdl/fpg/files/Sculthorpe-13-KURE.pdf`. Submitted to the Journal of Functional Programming, 2013c.

M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *Types in Language Design and Implementaion*, pages 53–66. ACM, 2007.

B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *Types in Language Design and Implementation*, pages 53–66. ACM, 2012.