Alternative Methods for Implementing Explicit and Finding Implicit Sharing in embedded DSLs

Curtis D'Alves, Christopher Kumar Anand, Lucas Dutton, and Steven Gonder

McMaster University, 1280 Main St W Hamilton, Canada

Abstract. Detection of sharing is known challenge for implementers of embedded domain specific languages (DSLs). There are many solutions, each with their advantages and drawbacks. Many solutions are based on observable sharing, that requires either a monadic interface or use of unsafe referencing, e.g., Data.Reify. Monadic interfaces are consider unsuitable for domain experts, and the use of unsafe referencing leads to fragile software.
Kiselyov's methods for implicit and explicit sharing detection for finally tagless style DSLs is an elegant solution without having to resort unsafe observable sharing. However these methods are not applicable to all types of DSLs (including those generating hypergraphs). We will present alternative methods which handle these cases. The main difference comes from the use of a trie to perform hashconsing. Our method for implicit sharing essentially trades worst-case exponential growth in computation for increased memory footprint. To mitigate this issue, our method for explicit sharing reduces the memory footprint.

Keywords: DSL · sharing · common-subexpression elimination · Haskell.

1 Introduction

Kiselyov [5] presents a method for implementing eDSLs in finally tagless form that generate a directed acyclic graph (DAG) with sharing. However, as we will explain in sections 2.3 and 2.4, for DSL functions that return multiple outputs (e.g., tuples, lists, etc.), Kiselyov's method of detecting sharing may require computation exponential in the size of the program, and his method of explicitly declaring sharing is inapplicable.

In the toy example

```
class Exp repr where
  variable :: String -> repr Int
  constant :: String -> repr Int
  add :: repr Int -> repr Int -> repr Int
  novel :: (repr Int,repr Int) -> (repr Int,repr Int)
```

the function `novel` exhibits this problem. In our work, this translated into the inability to process large library functions.

In this paper, we review Kiselyov's methods, identifying the core issue, and present methods for implementing embedded DSLs with sharing that avoid unsafe referencing (i.e., `unsafePerformIO`) [4], maintain all the benefits of being embedded in the Haskell ecosystem and are computationally feasible. This means DSL functions are pure, type-safe and can return Haskell container types (i.e., tuples, lists, etc.) without breaking sharing.

2 Background: Detecting Sharing

Consider the naive DSL implemented as a Haskell data type:

```
data Exp
  = Add Exp Exp
  | Variable String
  | Constant Int
```

Expressions generate Abstract Syntax Trees (ASTs), but consider this example,

```
v0 = Variable "v0"
exp0 = Add v0 (Constant 0)
exp1 = Add exp0 exp0
```

in which the expression `exp0` is shared, and will therefore be stored once in memory. For large expressions with lots of sharing, this can make a substantial difference. However, one of the first things the developer will do is write a pretty printer. That recursive function will traverse the data structure as a tree, and pretty print `exp0` twice. This inefficiency is a problem for code generation, and naive traversal of the AST does the opposite of common-subexpression elimination performed by a good optimizing compiler. To avoid this, rather than representing the code as an AST, we should use a DAG, retaining all of the sharing in the original DSL code.

One way of maintaining sharing is by observable sharing (see Section 3 in [5]). In Haskell, this requires a monadic interface. Monads are useful, but don't match the expectations of domain experts [6].

2.1 Finally Tagless DSLs

It would be nice to make use of monadic state when we need it (i.e., for converting to a DAG) while hiding it behind a nice pure interface. The finally tagless approach [1] is popular for accomplishing this. In this approach, DSL expressions are built using type-class methods that wrap the DSL in a parameterized representation. For example, the previous data-type-based DSL could be written in finally tagless style as

```
class Exp repr where
  add :: repr Int -> repr Int -> repr Int
  variable :: String -> repr Int
  constant :: Int -> repr Int
```

We can then create different instances to implement different functionality. For example, we can implement a pretty printer for our AST as

```
newtype Pretty a = Pretty { unPretty :: String }

instance Exp Pretty where
  add x y = Pretty $ "("++unPretty x++") + ("++unPretty y++")"
  variable x = Pretty x
  constant x = Pretty $ show x
```

Finally tagless style provides extensible, user friendly DSLs.

2.2 Implicit Sharing via Hash-Consing

Kiselyov's method for detecting implicit sharing in finally tagless style uses hash-consing [5]. Hash-consing is based on a bijection of nodes and a set of identifiers, e.g., with interface

```
data BiMap a -- abstract
lookup_key :: Ord a => a -> BiMap a -> Maybe Int
lookup_val :: Int -> BiMap a -> a
insert :: Ord a => a -> BiMap a -> (Int,BiMap a)
empty :: BiMap a
```

An efficient implementation using hashing and linear probing is given by Thai in his Master's thesis [7].

Nodes need to be uniquely identifiable, and shouldn't be a recursive data type, such as

```
type NodeID = Int
data Node = NAdd NodeID NodeID
          | NVariable String
          | NConstant Int
```

The representation for the finally tagless instance is then a wrapper around a State monad that holds the DAG being constructed in its state and returns the current (top) `NodeID`:

```
newtype DAG = DAG (BiMap Node) deriving Show

newtype Graph a = Graph { unGraph :: State DAG NodeID }

instance Exp Graph where
  constant x = Graph (hashcons $ NConstant x)
  variable x = Graph (hashcons $ NVariable x)
  add e1 e2 = Graph (do
                     h1 <- unGraph e1
                     h2 <- unGraph e2
                     hashcons $ NAdd h1 h2)
```

The trick to uncovering sharing is in the `hashcons` function, which inserts a new node into the current DAG, but not before checking if it is already there.

```
hashcons :: Node -> State DAG NodeID
hashcons e = do
  DAG m <- get
  case lookup_key e m of
    Nothing -> let (k,m') = insert e m
               in put (DAG m') >> return k
    Just k -> return k
```

The technique is essentially that of hash-consing, popularized by its use in LISP compilers, but discovered by Ershov in 1958 [2]. Other works have explored the use of type-safe hash-consing in embedded DSLs, see [3].

2.3 Limitations of Hash-Consing

When we wrap our State monad in finally tagless style, we lose some expected sharing. In the following code, the use of the let causes the computation $x + y$ to only occur once

```
haskellSharing x y=
 let
   z = x + y
 in z + z
```

Implicit sharing via hash-consing prevents duplication in the resulting DAG, but unfortunately doesn't prevent redundant computation. Consider the following equivalent attempt at using Haskell's built-in sharing in the finally tagless DSL

```
dslSharing :: Exp Graph -> Exp Graph -> Exp Graph
dslSharing x y =
  let
    z = add x y
  in add z z
```

Knowing that `z` is a wrapper around a state monad, and recalling the implementation of `add` via hash-consing above, the values `h1` and `h2` are separately evaluated through the state monad, even if `e1` and `e2` are the same shared Haskell value. Hash-consing will prevent these redundancies from appearing in the resulting DAG, but in the process of discovering the sharing, the entire unshared AST will still be traversed.

Consider a chain of `add`s with sharing, for example

```
addChains :: Exp repr => Expr Int -> Expr Int
addChains x0 =
  let
```

```
    x1 = add x0 x0
    x2 = add x1 x1
    ...
  in xn
```

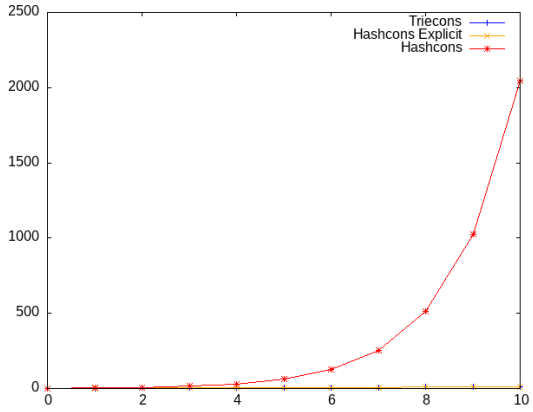

Fig. 1. Number of calls to `hashcons` plotted against the number of `add` operations performed. Hashcons is performed without explicit sharing and is clearly exponential, Triecons (without explicit sharing) and HashCons Explicit (with explicit sharing) overlap and are both linear

As shown in Fig. 1, this code will perform approximately 2^{n+1} `hashcons` operations, where n is the number of `add`s.

2.4 Explicit Sharing and Limitations

Kiselyov [5] recognized that the amount of computation with hash-consing "may take a long time for large programs," and proposed an ad-hoc solution, explicit sharing via a custom let construct

```
class ExpLet repr where
  let_ :: repr a -> (repr a -> repr b) -> repr b
instance ExpLet Graph where
  let_ e f = Graph (do x <- unGraph e
                       unGraph $ f (Graph (return x)))
```

which can be used to rewrite `addChains` as

```
addChains x =
  let_ x (\x0 ->
  let_ (add x0 x0)  (\x1 ->
  let_ (add x1 x1)  (\x2 ->
```

```
  ...
)))
```

This makes the code a bit clunky and adds an extra burden on the DSL writer, but it prevents unnecessary hash-consing in our example.

However the method does not work for DSL functions returning multiple outputs via tuples or container types like lists. Recall the definition

```
novel :: (repr Int,repr Int) -> (repr Int,repr Int)
```

The problem is that DAG generation requires splitting the state monad in two:

```
instance Exp Graph where
  ...
  novel e1 e2 = let
     g1 = Graph (do h1 <- unGraph e1
                    h2 <- unGraph e2
                    hashcons $ Novel1 h1 h2)
     g2 = Graph (do h1 <- unGraph e1
                    h2 <- unGraph e2
                    hashcons $ Novel2 h1 h2)
     in (g1,g2)
```

Each output it returns will now have to be individually evaluated, so a chain of DSL functions that output 2 or more values will suffer from the same exponential explosion of hashcons operations, and trying to adapt the let construct above, just creates another function function with the same problem (multiple outputs).

One solution to this issue is to integrate container types such as tuples and lists into the DSL language. However doing this eliminates the advantage of having an embedded language. Manipulating tuple values will be cumbersome, constantly requiring calls to custom implementations of `fst`,`snd` etc. And for lists you'll lose access to built-in Haskell list functionality.

3 Implicit Sharing Via Byte String ASTs

The heart of our problem is that whenever we need to sequence the state of the inputs for one of our DSL functions we want to first check if it's already been evaluated. But how do we do that without first evaluating it to gain access to its unique identifier. We need some way to uniquely identify it outside the monad.

Our proposed solution is to build a serialized AST using byte strings for each node along with our DAG. The byte string stays outside monad, while the DAG remains inside. We can do this efficiently by replacing the `BiMap` with a trie. In our toy example, we use the package `bytestring-trie`.

```
data Graph a = Graph { unGraph :: State DAG NodeID
                     , stringAST :: ByteString }
```

```
data DAG = DAG { unTrie :: Trie (Node,NodeID)
               , maxID  :: NodeID
               } deriving Show
```

This looks a bit different because the `BiMap` was a bijective relation between nodes and node ids, whereas the trie maps byte strings to pairs (node,node id). To get the DAG expressed as a relation, project out the values of the trie.

To prevent confusion, we rename the hash-consing function to `triecons`, as follows

```
triecons :: ByteString -> Node -> State DAG NodeID
triecons sAST node = do
  DAG trie maxID <- get
  case Trie.lookup sAST trie of
    Nothing -> let maxID' = maxID+1
                   trie' = Trie.insert sAST (node,maxID') trie
               in do put $ DAG trie' maxID'
                     return maxID'
    Just (_,nodeID) -> return nodeID
```

We use it to implement the DAG-building instance of the DSL, which looks a lot like the previous instance. The substantial differences are the `buildStringAST` calls which you can think of as pretty printing, but optimized for the trie, and the use of `seqArgs` (explained below):

```
instance Exp Graph where
  constant x = let
    node = NConstant x
    sAST = buildStringAST node []
    in Graph (triecons sAST $ NConstant x) sAST
  variable x = let
    node = NVariable x
    sAST = buildStringAST node []
    in Graph (triecons sAST $ NVariable x) sAST
  add e1 e2 = let
      sAST = buildStringAST "nadd" [e1,e2]
      sT = do ns <- seqArgs [e1,e2]
              case ns of
                [n1,n2] -> triecons sAST $ NAdd n1 n2
                _ -> error "black magic"
    in Graph sT sAST
```

The magic is in `seqArgs`. We only evaluate the inner state `sT` of each argument if we fail to look up its corresponding byte string AST in the Trie.

```
seqArgs :: [Graph a] -> State DAG [NodeID]
seqArgs inps =
  let
```

```
    seqArg (Graph sT sAST) =
      do DAG trie _ <- get
         case Trie.lookup sAST  trie of
           Nothing -> sT
           Just (_,nodeID) -> return nodeID
  in sequence $ map seqArg inps
```

This will prevent redundant hashconsing without the need for explicit sharing, but at the expense of storing redundant byte strings.

3.1 Memory Limitations

The byte string AST being built will itself suffer from lack of sharing. We're essentially trading extra computation for extra memory. In our same `addChains` example from Section 2.3, our method now has exponential scaling in memory instead of computation. This can be a good tradeoff, since memory is so plentiful in modern hardware, but still presents an issue.

4 Explicit Sharing Of ByteString ASTs

We propose another solution to this issue, taking inspiration again from the [5], we can introduce an explicit construct for specifying sharing. This time, the construct will substitute the current byte string for a more compact label. For safety purposes, we need to keep track of a table of these labels and their corresponding ASTs, to make sure we don't use the same label for different ASTs.

```
data DAG = DAG { dagTrie :: Trie (Node,NodeID)
               , dagSubMap :: Map ByteString ByteString
               , dagMaxID :: Int
               } deriving Show

data Graph a = Graph { unGraph :: State DAG NodeID
                     , unStringAST :: ByteString
                     , unSubT :: Maybe ByteString }

class Substitute repr where
  subT :: ByteString -> repr a -> repr a
instance Substitute Graph where
  subT s' (Graph g s _) = Graph g s' (Just s)

test x y = let
  z = subT "z" (add x y)
  in add z z
```

The `subT` operation substitutes the current byte string AST with a new one, and we define a new operation `subTInsert` to check if the label already exists in the cache map before inserting it.

```
seqArgs :: [Graph a] -> State DAG [NodeID]
seqArgs inps =
  let
    seqArg (Graph sT sAST mSubt) =
      do DAG trie _ _ <- get
         let sAST' = case mSubt of
                       Just s -> s
                       Nothing -> sAST
         case Trie.lookup sAST' trie of
           Nothing -> sT -- error "missing ast"
           Just (node,nodeID) ->
              do subTInsert mSubt sAST (node,nodeID)
                 return nodeID
  in sequence $ map seqArg inps

subTInsert :: Maybe ByteString -> ByteString
           -> (Node, NodeID) -> State DAG ()
subTInsert Nothing  _ _  = return ()
subTInsert (Just s) sAST nodeID =
  do DAG trie subtMap _ <- get
     case Map.lookup sAST subtMap of
        Just sAST' -> if sAST == sAST'
                      then return ()
                      else error "tried to resubT"
        Nothing -> let cMap' = Map.insert sAST s subtMap
                       trie' = Trie.insert sAST nodeID trie
                   in modify (\dag -> dag { dagTrie = trie'
                                          , dagSubMap = cMap' })
```

We need to make sure we don't attempt to insert the same substitution for two different ASTs. Unfortunately, if there is a collision there's no way to escape the State monad to prevent or modify the substitution. In the toy example, compilation crashes, but we could catch an exception instead. Either way it's up to the DSL writer to insure they don't reuse the same label.

5 BenchMarking

6 Conclusion

We have presented a method for constructing finally tagless style DSLs with sharing detection, that allow for multiple-output DSL functions. It also avoids the use of unsafe referencing as performed when doing observable sharing, c.f. [4].

The method has its drawbacks in terms of memory usage, but these can be mitigated by explicitly specifying sharing. This does present an extra burden on

the DSL writer to implement explicit sharing when necessary and ensure labels are not reused. Future work may investigate the use of a preprocessor or plugin to automate explicit sharing.

Acknowledgements We thank NSERC and IBM Canada for supporting this work.

References

1. Carette, J., Kiselyov, O., Shan, C.c.: Finally tagless, partially evaluated. In: Asian Symposium on Programming Languages and Systems. pp. 222–238. Springer (2007)
2. Ershov, A.P.: On programming of arithmetic operations. Communications of the ACM **1**(8), 3–6 (1958)
3. Filliâtre, J.C., Conchon, S.: Type-safe modular hash-consing. In: Proceedings of the 2006 Workshop on ML. pp. 12–19 (2006)
4. Gill, A.: Type-safe observable sharing in haskell. In: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell. pp. 117–128 (2009)
5. Kiselyov, O.: Implementing explicit and finding implicit sharing in embedded dsls. arXiv preprint arXiv:1109.0784 (2011)
6. O'Donnell, J.T.: Embedding a hardware description language in Template Haskell. In: Domain-Specific Program Generation, pp. 143–164. Springer (2004)
7. Thai, N.: Type-Safe Modeling for Optimization. Master's thesis (2021)