Implementing Explicit and Finding Implicit Sharing in Embedded DSLs

Oleg Kiselyov
oleg@okmij.org

Aliasing, or sharing, is prominent in many domains, denoting that two differently-named objects are in fact identical: a change in one object (memory cell, circuit terminal, disk block) is instantly reflected in the other. Languages for modelling such domains should let the programmer explicitly define the sharing among objects or expressions. A DSL compiler may find other identical expressions and share them, implicitly. Such common subexpression elimination is crucial to the efficient implementation of DSLs. Sharing is tricky in embedded DSL, since host aliasing may correspond to copying of the underlying objects rather than their sharing.

This tutorial summarizes discussions of implementing sharing in Haskell DSLs for automotive embedded systems and hardware description languages. The technique has since been used in a Haskell SAT solver and the DSL for music synthesis. We demonstrate the embedding in pure Haskell of a simple DSL with a language form for explicit sharing. The DSL also has implicit sharing, implemented via hash-consing. Explicit sharing greatly speeds up hash-consing. The seemingly imperative nature of hash-consing is hidden beneath a simple combinator language. The overall implementation remains pure functional and easy to reason about.

I think all DSLs suffer from the same problems: sharing and recursion. I've used wrappers for CSound, SuperCollider, MetaPost, they all have these problems. Henning Thielemann [19]

1 Introduction

We present implicit and explicit sharing in the original context of embedded domains-specific language (DSL) compilers. The sharing implementation techniques have since found other uses, e.g., writing SAT solvers [3]. Embedded compilers – typical for circuit description, embedded control systems or GPU programming DSLs – complement the familiar embeddings of a DSL in a host language as a library or an interpreter. For example, we may build a circuit model in Haskell using gate descriptions and combinators provided by the DSL library. We may test the circuit in Haskell by running the model on sample inputs. Eventually we have to produce a Verilog code or a Netlist to manufacture the circuit. Likewise, a control system DSL program should eventually be compiled into machine code and burned as firmware. An embedded compiler thus is a host language program that turns a DSL program into a (lower-level) code.

One of the important tasks of a compiler is the so-called "common subexpression elimination" – detecting subexpressions denoting the same computation and arranging for that computation to be performed once and the results shared. This optimization (significantly) improves both the running time and compactness of the code and is particularly important for hardware description and firmware compilers. As DSL implementers, it becomes our duty to detect duplicate subexpressions and have them shared. We call this detection implicit sharing, to contrast with the sharing explicitly declared by users. Imperative languages, where it matters whether an expression or its result are duplicated, have to provide a

Olivier Danvy, Chung-chieh Shan (Eds.): IFIP Working Conference on Domain-Specific Languages 2011 (DSL 2011).
EPTCS 66, 2011, pp. 210–225, doi:10.4204/EPTCS.66.11

form (some sort of a local variable introduction) for the programmer to declare the sharing of expression's result. In the present paper we limit ourselves to side-effect–free expressions such as arithmetic expressions or combinational circuits. Explicit sharing is still important: sharing declarations can significantly reduce the search space for common subexpressions and prevent exponential explosions. We shall cite several examples later. Sharing declarations also help human readers of the program, drawing their attention to the 'common' computations.

A common pitfall in implementing explicit sharing is attempting to use the **let** form of the host language to express sharing in the DSL code. §5 will show that although the **let** -form may speed up some DSL interpretations, it leads to code duplication rather than sharing in the compilation result.

Before we get to that discussion, we introduce our running example in §2, describing an unsuccessful attempt to detect sharing in the course of implementing real DSLs. Show-stopping was the expression comparison, which, in pure language is structural and requires the full traversal of expressions. In §3 we review the main approaches to speed-up the comparison, all relying on some sort of 'pointer' equality. Our method of expression comparison and sharing detection is presented in §4. The method is a pure veneer over hash-consing, alleviating the cost of comparison. Its main benefit is the facilitation of explicit sharing, described in §5.

The code accompanying the paper is available online at `http://okmij.org/ftp/tagless-final/sharing/`.

2 Detecting sharing: necessity and difficulty

Our running example is a show-stopping problem encountered by the implementer of an embedded DSL compiler for an embedded control system, posed in [12]. The example illustrates the need and the difficulty of common subexpression elimination. The problem was posed for arithmetic expressions, which are part of nearly every DSL. Typically arithmetic expressions are embedded in Haskell as the values of the datatype[1]

```
data Exp
  = Add Exp Exp
  | Variable String
  | Constant Int
  deriving (Eq, Ord, Show)
```

Here are the sample expressions in our DSL:

```
exp_a = Add (Constant 10) (Variable "i1")
exp_b = Add exp_a (Variable "i2")
```

The implementer wanted to compile these DSL expressions into C or the machine code, using the standard approach of finding common subexpressions and sharing them. To make the sharing explicit, the expression tree is converted into a directed acyclic graph (DAG). The graph is then topologically sorted and each subexpression is assigned a C operation or a machine instruction.

The first step, detecting identical subexpressions, was the most troublesome. The step is crucial since common subexpressions are abound, being easy to create. We show two real-life examples, to be used throughout the paper. The first example is the multiplication by a known integer constant. Our DSL does not have the multiplication operation (8-bit CPUs rarely have the needed instruction). Nevertheless,

[1] See the file `ExpI.hs` in the accompaniment for the complete code.

we can multiply a DSL expression by the known constant using repeated addition. Here is the standard efficient procedure based on the recursive subdivision:

```
mul : Int → Exp → Exp
mul 0 _ = Constant 0
mul 1 x = x
mul n x | n `mod` 2 == 0 = mul (n `div` 2) (Add x x)
mul n x = Add x (mul (n−1) x)
```

The result of the sample expression

```
exp_mul4 = mul 4 (Variable "i1")
```

shows two identical subexpressions of adding the variable i1 to itself:

```
Add (Add (Variable "i1") (Variable "i1")) (Add (Variable "i1") (Variable "i1"))
```

The result of mul 8 (Variable "i1") shows twice as much duplication.

The other running example, from the domain of hardware description, is sklansky by Naylor [15], with further credit to Sheeran and Axelsson. The example computes the running sum of given expressions; like the previous multiplication example, sklansky uses recursive subdivision to expose more parallelism and reduce latency:

```
sklansky : (a → a → a) → [a] → [a]
sklansky f [] = []
sklansky f [x] = [x]
sklansky f xs = left' ++ [f (last left') r | r ← right']
  where
    (left , right ) = splitAt (length xs `div` 2) xs
    left'  = sklansky f left
    right' = sklansky f right
```

The pretty-printed result of sklansky Add (map (Variable ∘show) [1..4]

```
["v1","(v1+v2)","((v1+v2)+v3)","((v1+v2)+(v3+v4))"]
```

demonstrates the triplication of the subexpression v1+v2. The duplication should be eliminated when we build the circuit.

In the process of converting expressions like exp_mul4 to a DAG, the implementer [12] had to compare subexpressions. It is there he encountered a problem: in a pure language, we may only compare datatype values structurally. General pointer comparison destroys referential transparency and parametricity. To check that the summands of the top-level addition in exp_mul4 are identical, we have to therefore traverse them completely. Such comparisons of two expression trees take more and more time with larger programs (say, as we multiply by bigger integers). "As these trees grow in size, the equality comparison in graph construction quickly becomes the bottleneck for DSL compilation. What's worse, the phase transition from tractable to intractable is very sharp. In one of my DSL programs, I made a seemingly small change, and compilation time went from milliseconds to not-in-a-million-years."[12]. His message, entitled "I love purity, but it's killing me" was a cry for help: he was about to give up on Haskell. He wondered how to dramatically speed-up comparisons, or find a better method of detecting common subexpressions and sharing them.

3 Pointer comparison

Sharing detection, specifically, fast identity comparison of expressions, is a common metaprogramming problem and has been investigated extensively. This section reviews the main approaches, which are all based on some sort of 'pointer' equality. They associate with an expression a unique datum, e.g., an integer, admitting efficient comparison. For instance, we may define the data type of labeled expressions, where each variant carries a unique integer label[2]:

```
type Lab = Int
data ExpL
   = AddL Lab ExpL ExpL
   | VariableL  Lab String
   | ConstantL Lab Int
   deriving Show
```

ExpL expressions are fast to compare, by comparing their labels. The full traversal of expressions is no longer needed:

```
instance Eq ExpL where
    e1 == e2 = label e1 == label e2
     where
     label (AddL p _ _)     = p
     label (ConstantL p _) = p
     label (VariableL  p _) = p
```

The first approach to building labeled expressions is manual labeling. For example, we construct our sample expressions as follows, taking great care to pick unique labels:

```
expL_a = AddL 3 (ConstantL 1 10) (VariableL 2 "i1")
expL_b = AddL 4 expL_a (VariableL 5 "i2")
```

Needless to say this approach is greatly error-prone, let alone tedious. When implementing the mul example we stumble on another complication: the manual threading of a counter to generate unique labels. Some sort of automation is direly needed.

A promising and increasingly popular way to hide and automate label assignment is Template Haskell (see the thesis [1] for the extensive discussion). A DSL implemented in Template Haskell quotations can hardly be called 'embedded' however. Another approach is the State monad hiding the counter used for the generation of unique labels:

```
type ExpM = State Lab ExpL

new_labelM : State  Lab Lab
new_labelM = do
  p ← get
  put (p+1)
  return p

run_expM : ExpM →ExpL
run_expM m = evalState m 0
```

[2]The complete code for this section is in the file `Ptrs.hs`.

The computation new_labelM, or 'gensym', yields a unique label; the function run_expM runs the monadic computation returning the produced labeled expression.

To hide the labeling of expression's constructors, we build expressions with 'constructor functions', often called 'smart constructors':

```
varM : String → ExpM
varM v   = new_labelM >>=\p →return $ VariableL p v

constM : Int → ExpM
constM x = new_labelM >>=\p →return $ ConstantL p x

addM : ExpL → ExpL → ExpM
addM x y = new_labelM >>=\p →return $ AddL p x y
```

The sample expression looks awful

```
expM_a = do
  xv ← constM 10
  yv ← varM "i1"
  addM xv yv
```

although appropriate combinators may improve the appearance. In fact, the multiplication example below is quite perspicuous: the labeling is well-hidden (compare with mul in §2). The occasional **return** and the 'call-by-value application' (=<<) betray the effectful computation:

```
mulM :Int → ExpL → ExpM
mulM 0 _ = constM 0
mulM 1 x = return x
mulM n x | n `mod` 2 == 0 = mulM (n `div` 2) =<<addM x x
mulM n x = addM x =<<mulM (n−1) x
```

The subexpression addM x x on the last-but-one line builds the addition expression from two identically named summands. The two occurrences of the Haskell variable x denote the same ExpL expression, with the same label. These identical summands are thus easily identifiable as shared. Indeed, the multiplication-by-4 computation

```
expM_mul4 = mulM 4 =<<varM "i1"
```

when run, yields the labeled expression

```
AddL 2
  (AddL 1 (VariableL 0 "i1") (VariableL  0 "i1"))
  (AddL 1 (VariableL 0 "i1") (VariableL  0 "i1"))
```

with the clearly seen sharing: just look at the labels.

Exercise 1 *Why we did not define addM as follows?*

```
addM : ExpM →ExpM →ExpM
addM x y = do
  xv ← x
  yv ← y
  p  ← new_labelM
  return $ AddL p xv yv
```

After all, it does let us write the sample expression expM_a concisely:

```
expM_a = addM (constM 10) (varM "i1")
```

Hint: implement mulM and see the result of expM_mul4.

The monadic approach may seem adequate – to us, DSL implementers. It does not however feel right to our users, domain experts. They are accustomed to writing and manipulating arithmetic expressions as familiar mathematical objects. Now they have to deal with effectful computations. O'Donnell, summarizing the dissatisfaction of hardware designers with such a monadic DSL, wrote: "A more severe problem is that the circuit specification is no longer a system of simultaneous equations, which can be manipulated formally just by 'substituting equals for equals'. Instead, the specification is now a sequence of computations that – when executed – will yield the desired circuit. It feels like writing an imperative program to draw a circuit, instead of defining the circuit directly."[17]. The hardware description DSL Lava has tried the monadic approach and abandoned it.

The most popular approach to detect sharing is a so-called 'observable sharing' [6], which hides the labeling further, to the point of breaking Haskell and using internal, unsafe operations of GHC. Fresh label generation is such a benign effect. The breaking of the referential transparency is hardly noticeable, one may argue [6, 16]. There are many variations of observable sharing [1, 6, 16]: some rely on IORef cells as labels (they can be compared efficiently as pointers), some use gensym. We demonstrate the latter, and define gensym, or new_label, as a supposedly pure, ordinary Haskell function:

```
{-# NOINLINE counter #-}
counter = unsafePerformIO (newIORef 0)
new_label : () → Int
new_label () = unsafePerformIO $ do
  p ← readIORef counter
  writeIORef counter (p+1)
  return p
```

The function is certainly not pure since each evaluation of new_label () (should) yield a new result.

Exercise 2 *Why do we need 'NOINLINE'?*

Smart constructors again hide the labeling. Now they have pure types, yielding DSL expressions ExpL themselves rather than expression computations ExpM:

```
varU : String → ExpL
varU v = VariableL (new_label ()) v

constU : Int → ExpL
constU x = ConstantL (new_label ()) x

addU : ExpL → ExpL → ExpL
addU x y = AddL (new_label ()) x y
```

The sample expressions look quite like those in §2, with no traces of labeling, with no leaking of implementation details into the domain-specific abstraction:

```
expU_a = addU (constU 10) (varU "i1")
expU_b = addU expU_a (varU "i2")
```

The multiplication example also looks just like the pure version in §2, defining multiplication with familiar mathematical equations.

```
mulU : Int → ExpL → ExpL
mulU 0 _ = constU 0
mulU 1 x = x
mulU n x | n `mod` 2 == 0 = mulU (n `div` 2) (addU x x)
mulU n x = addU x (mulU (n−1) x)
```

The result of the multiplication by four:

```
expU_mul4 = mulU 4 (varU "i1")
```

shows that identically-named expressions have identical labels and are hence clearly shared

```
AddL 0 (AddL 1 (VariableL 2 "i1") (VariableL 2 "i1"))
       (AddL 1 (VariableL 2 "i1") (VariableL 2 "i1"))
```

One look at the labels is enough to see the sharing.

Breaking the referential transparency and lying to the compiler about effects of our functions may come to haunt us however:

Exercise 3 *Why cannot we η-reduce the smart constructors as follows?*

```
constU = ConstantL (new_label ())
varU   = VariableL (new_label ())
addU   = AddL (new_label ())
```

Hint: try it.

Observable sharing will no longer be used in this paper.

4 Pure Hash-Consing

Hash-consing is a well-established technique to identify and share structurally equal data [10]. Although the name comes from Lisp, the technique has been first described prior to Lisp, in 1957 [9]. The technique relies on the global mutable hash table mapping structured values to integer hashes, which are quick to compare. Value constructors check the table to see if the equal value has been constructed already, returning the found hash. We describe hash-consing for a DSL embedded in pure, safe Haskell2010. The imperative details of hash-consing are hidden better in a final-tagless style of the DSL embedding, described next. The complete code for this section is in the file `ExpF.hs` in the accompanying code.

4.1 Tagless-final embedding

In the tagless-final approach [5], embedded DSL expressions are built with 'constructor functions' such as constant, variable , add rather than the data constructors Constant, Variable , Add that we have seen in §2. The constructor functions yield a representation for the DSL expression being built. The representation could be a string (for pretty-printing), an integer (for evaluator), etc. Since the same DSL expression may be concretely represented in several ways, the constructor functions are polymorphic, parameterized by the representation repr . In other words, the constructor functions are the members of the type class

```
class Exp repr where
  constant : Int → repr Int
  variable : String → repr Int
  add      : repr Int → repr Int → repr Int
```

The apparent difference from the datatype Exp of §2 is the lower case of the 'constructors'. We have parameterized the representation by expression's type, as common [5]. We did not have to, since the type so far has been the same, **Int**. The parameterization by the type will come handy once we add boolean expressions.

The sample expressions from §2 look almost the same:

```
exp_a = add (constant 10) (variable "i1")
exp_b = add exp_a (variable "i2")
```

differing only in the lower case of the 'constructors'.

The datatype Exp from §2 is one concrete (so-called 'initial') representation of the DSL expressions – one instantiation of repr :

```
newtype Expl t = Expl Exp

instance Exp Expl where
  constant = Expl ∘Constant
  variable = Expl ∘Variable
  add (Expl x) (Expl y) = Expl (Add x y)
```

Exercise 4 *Why do we need the wrapper Expl?*

Interpreting repr as Expl lets us pretty-print final-tagless expressions, thanks to the derived **Show** instance for the data type Exp:

```
test_shb = case exp_b of Expl e → e
-- Add (Add (Constant 10) (Variable "i1")) (Variable "i2")
```

The multiplication example is largely unchanged, modulo the lower-case of the constructors and the type signature:

```
mul : Exp repr ⇒ Int → repr Int → repr Int
mul 0 _ = constant 0
...
exp_mul4 = mul 4 (variable "i1")
```

The conversion to Expl took the form of an instance of the class Exp providing the interpretation for the expression primitives, as the values of the domain Expl. We may write other interpretations, for example, the evaluator, interpreting an expression as an element of the domain R

```
type REnv = [(String,Int)]
newtype R t = R{unR :REnv →t} -- A reader Monad
```

that is, an integer in the environment giving the values for the free variables occurring in the expression.

```
instance Exp R where
  constant x = R (\_ → x)
  variable x = R (\env →maybe (error $ "no_var:␣" ++ x) id $ lookup x env)
  add e1 e2  = R (\env →unR e1 env +unR e2 env)
```

The evaluator lets us test the multiplication-by-4 example:

```
test_val4  = unR exp_mul4 [("i1",5)]  -- 20
```

Exercise 5 *Add subtraction and negation to the language. Can we get by without changing the type class Exp, that is, without breaking the existing code?*

4.2 Detecting implicit sharing

Recall that our goal is to detect structurally equal subexpressions and share them, converting an expression tree into a DAG. The goal is closer if we construct an expression as a DAG to start with. We represent the DAG as a collection of Nodes identified by NodeIds, which link the nodes:

```
type NodeId = Int
data Node = NConst Int
          | NVar   String
          | NAdd   NodeId NodeId
            deriving (Eq,Ord,Show)
```

The Node data type resembles Exp from §2; however, Node is not a recursive data type and can be compared in constant time. The mapping between Nodes and NodeIds is realized through a BiMap 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
```

BiMap a establishes a bijection between the values of the type a and integers, with the operations to retrieve the value given its key, to find the key for the existing value, and to extend the bijection with a new association. The type a should at least permit equality comparison; in the present implementation, we require a to be a member of Ord. BiMaps can be pretty-printed. Our DAG thus is as follows:

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

Having settled on the DAG implementation we now describe its construction, which is easier bottom-up. As we construct a node for a subexpression, we check if the DAG already has the equal node. If so, we return its NodeId; otherwise, we add the node to the DAG. This procedure is nothing but hash-consing. In fact, it is quite close to Ershov's original description of hash-consing [9]; our DAG representation is also similar to his. The BiMap interface has exactly the right operations, to check for the presence of a node and to insert the node, allocating a new NodeId. Since the DAG is being modified as new nodes are built, the construction procedure is the State monad computation with the DAG as the state.

The bottom-up DAG construction maps well to computing a representation for a tagless-final expression, which is also evaluated bottom-up. The DAG construction can therefore be written as a tagless-final interpreter, an instance of the type class Exp. The interpreter maps a tagless-final expression to the concrete representation that is a NodeId in the current DAG:

```
newtype N t = N{unN :State DAG NodeId}

run_expN : N t → (NodeId, DAG)
run_expN (N m) = runState m (DAG empty)
```

The function run_expN runs the DAG-construction interpreter and returns the node, as a reference within a DAG. The construction algorithm is codified as follows

```
instance Exp N where
  constant x = N(hashcons $ NConst x)
  variable x = N(hashcons $ NVar x)
  add e1 e2  = N(do
                 h1 ← unN e1
                 h2 ← unN e2
                 hashcons $ NAdd h1 h2)
```

with the auxiliary hashcons doing the hash-consing, inserting the Node in the DAG if it has not been there already.

```
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
```

In §4.1 we have defined sample tagless-final expressions exp_mul4 and exp_mul8 for the multiplication by 4 and 8 and interpreted them in several ways, as an integer value and an expression tree. We now interpret the very same expressions as DAGs: run_expN exp_mul4 produces the result

```
(2, DAG BiMap[(0,NVar "i1"),(1,NAdd 0 0),(2,NAdd 1 1)])
```

whereas run_expN exp_mul8 gives

```
(3, DAG BiMap[(0,NVar "i1"),(1,NAdd 0 0),(2,NAdd 1 1),(3,NAdd 2 2)])
```

A DAG is printed as the list of (NodeId,Node) associations. The sharing of the left and right summands is patent,

The shown results are in fact netlists: a low-level representation of a circuit listing the gates and their connections, used in circuit manufacturing. Since our BiMap allocated monotonically increasing NodeIds, the resulting netlist comes out topologically sorted. Therefore, we can straightforwardly generate machine code after the standard register allocation.

Exercise 6 *The method add in the Exp N instance looks quite like the function addM in Exercise 1. The addM function was flawed. Why does add work?*

The imperative nature of hash-consing is well-hidden behind the pure final-tagless interface. We stress this point with the sklansky example, of computing the running sum of several expressions. The function sklansky was defined in §2 with the signature `sklansky : (a → a → a) → [a] → [a]`. In particular, sklansky Add applied to the list of four variables produced the following list of (pretty-printed) expressions

```
["v1","(v1+v2)","((v1+v2)+v3)","((v1+v2)+(v3+v4))"]
```

We will use the very same sklansky to build a 'DAG forest': the list of nodes sharing children within *the same* DAG. The result is emphatically not the list of isolated DAGs. Rather, we return the

list of NodeIds all referring to the same DAG structure, sharing the components not only within the same expression but also across independent expressions. The result might seem impossible yet is easily achievable: we build the list of DAG-constructing *computations* and then run them in **sequence**, with the same DAG state:

```
test_sklansky n = runState sk (DAG empty)
  where
  sk = sequence (map unN (sklansky add xs))
  xs = map (variable ∘show) [1..n]
```

Until the very end, the monadic nature of the DAG construction was hidden. We manipulated tagless-final expressions as pure, mapping and passing them around without any regard for their possible effects. Sharing will be detected nevertheless. The running sum for the list of four variables, test_sklansky 4, now reads

```
([0,2,4,7],
  DAG BiMap[
    (0, NVar "1"),(1,NVar "2"),(2,NAdd 0 1),
    (3, NVar "3"),(4,NAdd 2 3),
    (5, NVar "4"),(6,NAdd 3 5),
    (7, NAdd 2 6)])
```

The repeated expression v1+v2, represented by NodeId 2, is built only once and referenced at three places.

Exercise 7 *Our current implementation of BiMap relies on a pair of finite maps. We could have used a highly optimized hash table from the Haskell standard library. However, hash table operations are performed in the* **IO** *monad. Can we accommodate such mutable hash tables without leaking the* **IO** *monad, maintaining the form of the Exp N interpreter and the purity of tagless-final expressions?*

We have demonstrated the sharing detection technique that represents a DSL program as a DAG, eliminating multiple occurrences of common subexpressions. Alas, to find all these common subexpressions we have to examine the entire expression tree, which may take long time for large programs (large circuits). The next section describes this problem and its solution by explicit sharing.

5 Explicit sharing

This section motivates the extension of the DSL with a syntactic form to explicitly indicate expression sharing, and describes its implementation. This form lets the programmer state their view of a computation as 'common', to be executed once and its result shared. The programmer thus helps the DSL compiler as well as the human readers of the code.

The case for the sharing form as part of the pure, non-imperative DSL is compelling but subtle. At first blush, the host language **let** form seems sufficient – and it is, for some eDSL interpreters. The tagless-final DSL embedding helps clarify the subtlety. Our running example will be the familiar multiplication-by-4, written explicitly below:[3]

```
exp_mul4 =
 let x = variable "i1" in
```

[3] The complete code for this section is in the file `ExpLet.hs`.

```
let y = add x x in
add y y
```

The two occurrences of the variable y refer to the same tagless-final expression (namely, add x x). Such a representation of a repeated expression by a variable makes the code compact. One may also expect that in the internal representation of exp_mul4, the two arguments of add refer to the same run-time object.

Exercise 8 *Is there any guarantee, in the Haskell Report or the GHC documentation that two occurrences of the same variable refer to the common shared object rather than duplicated objects?*

Recall that add x x is a Haskell computation producing a particular representation of the DSL expression. What is shared in exp_mul4 is the computation rather than the representation. This sharing of computations, along with the memoization inherent in GHC, speeds up DSL interpretations. It appears therefore that the Haskell **let** is sufficient to express explicit sharing: it makes sharing easy to see in the code and it speeds up interpretations. The following two tagless-final interpreters show that the Haskell **let** may indeed speed up some interpretations, but not the others.

The first interpreter computes the size of an expression, in the number of its constructors:

```
newtype Size t = Size Int
instance Exp Size where
    constant _ = Size 1
    variable _ = Size 1
    add (Size x) (Size y) = Size (x+y+1)
```

The computed size of exp_mul4 is 7. When the computation add x x is first referred to from y, the computation will be performed and its result memoized. The second reference to the same computation via y will use the already determined result. The call-by-need evaluation strategy of GHC performs shared computations only once. Therefore, computing the size of even mul (2^30) (variable "i1") is instantaneous. Although that Haskell expression denotes a large DSL expression tree, the computation over the tree is compactly represented and is fast to perform.

The other interpreter prints a DSL expression

```
newtype Print t = Print (IO ())
instance Exp Print where
    constant = Print ∘ putStr ∘ show
    variable = Print ∘ putStr
    add (Print x) (Print y) = Print (x ≫ putStr "␣+␣" ≫ y)
```

Printing exp_mul4 gives i1 +i1 +i1 +i1 , with the subexpression i1 +i1 duplicated. The duplication is no surprise since our DSL has no sharing form and hence no way to indicate the sharing in the print-out. We stress that the printing of i1 +i1 was done two times. As before, the computation to print add x x was shared, and yet it has evidently been performed twice.

Exercise 9 *Why the Size t computations were memoized but Print t computations apparently were not? Is there is something special about* **IO***? What about other monads, such as State, implemented as pure functions?*

The Print interpreter will therefore take a long time to print mul (2^30) (variable "i1"), even if we redirect the output to /dev/**null**. Thus for some DSL interpreters including the DAG-constructing interpreter and almost any other DSL compiler, the running time will be at least proportional to the size of the

DSL expression rather to the size of the Haskell code to construct the expression. The compactly written Haskell code may represent exponentially large DSL expressions.

Compacting DSL expressions themselves requires a sharing form in the DSL itself. We call the form let_ and add it to our DSL by defining the type class ExpLet. (Tagless-final embedded DSLs are extended by introducing a new type class that describes the syntax of the new syntactic form. Such a change does not break the existing expressions written in the non-extended DSL.)

```
class ExpLet repr where
  let_ : repr a → (repr a → repr b) → repr b
```

The let_ of DSL, like the **let** of Haskell, expresses sharing through a local variable binding; multiple occurrences of the let_ -bound variable within the let_ body all refer to the same expression, the one to which the variable is bound. The DAG constructing interpretation of let_ , described below, will make clear that a let_ -bound variable refers to the result of the shared expression. The form let_ is a binding form, and is embedded in Haskell using the higher-order abstract syntax, with Haskell's λ-bound variable representing the DSL local variable. As an example of let_ , we re-write exp_mul4 indicating the sharing explicitly:

```
exp_mul4' =
 let_ (variable "i1") (\x →
 let_ (add x x)       (\y→
 add y y))
```

We tell the existing tagless-final interpreters how to deal with let_ . For example, the R interpreter treats let_ as the flipped application:

```
instance ExpLet R where
  let_ x f = f x
```

The evaluation of the sample expressions

```
val_mul4  = unR exp_mul4 [("i1",5)]  -- 20
val_mul4' = unR exp_mul4' [("i1",5)] -- 20
```

shows that exp_mul4 with and without explicit sharing evaluate to the same results. After all, sharing (of pure expressions) is an optimization and should not affect the results of DSL programs.

Exercise 10 *Extend the Size interpreter to account for explicit sharing (that is, write the instance ExpLet Size). The size of shared expressions should be counted only once.*

To 'see' the sharing, we need a **show**-like function, or an interpreter of tagless-final expressions as strings. Just strings will not suffice: to show sharing as let-expressions we need to generate local variable names. The domain of the show-interpretation is hence S t:

```
type LetVarCount = Int
newtype S t = S{unS :LetVarCount →String}
```

Exercise 11 *Write the tagless-final interpreter for S t, that is, the instances Exp S and ExpLet S.*

The S interpreter shows exp_mul4 as

```
i1 +i1 +i1 +i1
```

and exp_mul4' as

```
let v0 = i1 in let v1 = v0 +v0 in v1 +v1
```

We tell the DAG-constructing interpreter N how to handle explicit sharing. Recall that the expression let_ e (\x → body) states that multiple occurrences of the variable x in the body should refer to the same shared expression e. In the N interpreter, the meaning of a DSL expression is the DAG-constructing computation producing a NodeId. Sharing a computation across several places means performing the computation once and replicating its result. This principle is codified as follows

```
instance ExpLet N where
  let_ e f = N(do
                x ← unN e
                unN $ f (N (return x)))
```

The result of interpreting exp_mul4' as a DAG is identical to that of exp_mul4: the two expressions are indeed identical after the common subexpression elimination. In exp_mul4', sharing was explicitly declared; in exp_mul4 is had to be determined. The explicit sharing declaration makes the difference in the resources spent to get the results rather than in the results themselves.

Larger examples will show the difference in the resources. To obtain the examples, we re-write the mul generator to use the explicit sharing. The difference from mul is on the last-but-one line.

```
mul' : (ExpLet repr, Exp repr) ⇒ Int → repr Int → repr Int
mul' 0 _ = constant 0
mul' 1 x = x
mul' n x | n `mod` 2 == 0 = let_ x (\x' → mul' (n `div` 2) (add x' x'))
mul' n x = add x (mul' (n−1) x)
```

Exercise 12 *There is some sharing left to discover, isn't there? Modify mul' to explicitly declare all sharing.*

Without explicit sharing, running the DAG construction run_expN (mul n (variable "i")) in GHCi takes 0.09 secs for n equal to 2^{12}, and 0.20 secs for n equal to 2^{13}. With explicit sharing, running of the run_expN (mul' n (variable "i")) takes the same 0.01 secs for n equal to 2^{12}, or 2^{20} or even 2^{30}. The construction is so fast that its timing is lost in noise, even for the expression with $2^{31}-1$ constructors (most of which are fortunately shared).

The programmer does not have to explicitly declare all sharing. Some amount of sharing could be left implicit, for the DAG constructor to discover. The declared sharing may significantly speed up the detection of the implicit sharing. For example, the mul' code did not explicitly declared all sharing, as seen from the printout of (mul' 15 (variable "i")):

```
i +let v0 = i in v0 +v0 +
      let v1 = v0 +v0 in v1 +v1 +let v2 = v1 +v1 in v2 +v2
```

The DAG construction will find the undeclared sharing, producing the DAG

```
(6, DAG BiMap[
   (0, NVar "i"),
   (1, NAdd 0 0), (2,NAdd 1 1), (3,NAdd 2 2),
   (4, NAdd 2 3), (5,NAdd 1 4), (6,NAdd 0 5)])
```

For a large example, run_expN (mul (2^30−1) (variable "i")) finishes within the same 0.01 secs (although producing the twice as large DAG). The explicit sharing helps find the remaining implicit sharing.

Exercise 13 *How to re-write the sklansky example with the explicit sharing?*

6 Conclusions and further reading

We have demonstrated the sharing detection technique based on the tagless-final embedding that interprets a DSL program as a DAG, eliminating multiple occurrences of common subexpressions. We have argued for the extension of the DSL with the syntactic form let_ to declare sharing explicitly. The sharing declarations not only help the human readers of the code but also reliably, sometimes exponentially, speed up DSL interpreters. Implicit (not stated but detected) and explicit sharing play well together: the programmer does not have to identify all expressions to share; the declared sharing helps, often significantly, to detect the implicit one.

The technique, illustrated on arithmetic expressions, is immediately applicable to hardware description eDSLs. The technique has also been used in a SAT solver [3] and in the audio synthesizer mescaline [13].

The standard, thorough reference for compiling embedded DSLs is Elliott et al. [8]. Sections 4 and 8.1 of the paper discuss the detection and representation of sharing, with the particular attention to the placement of the target-code **let** -expressions to state the sharing in the target code. In the presence of loops and conditionals, the semantics-preserving **let** -insertion is quite non-trivial, as the paper discusses in detail. To transform an expression tree to a DAG, the paper relied on "non-declarative pointer manipulation", or so-called "observable sharing", which we illustrated in §3. Broadly, observable sharing denotes any use of unsafePerformIO for the detection of sharing [16]. Gill [11] has demonstrated that sharing at certain types is observable, in the **IO** monad. However, we have to resort to GHC-specific StableNames and accept their unreliability. §12 of [11] describes the advantages and many precautions of observable sharing.

Detecting sharing is crucial in hardware description languages, since the modeled circuits are general graphs rather than trees. The concise review of the long history of representing sharing in hardware description embedded DSLs is given in §2.4.1 of the thesis [1]. The thesis describes in more detail the approaches we have touched upon in §3.

Generating code with **let** -expressions to show sharing also has long history. The subtle aspects and the need for writing the generator in the continuation-passing (or monadic) styles or using control effects have been observed long time ago in partial evaluation community [2]. See [4, §3.1] for the detailed explanation of the problem specifically in the context of code generation.

Exercise 14 *Add recursion or iteration to our DSL. Do we need to extend the DSL with a new syntactic form (e.g., loop), or recursive definitions of Haskell will suffice?*

Exercise 15 *Add boolean expressions: true and false literals, conjunctions and disjunctions, integer comparison. Statically detect errors like taking the disjunction of integers.*

Exercise 16 *Add the conditional operator to the DSL. Should control-flow be taken into account when searching for common subexpressions and sharing them?*

Acknowledgement

I am indebted to Chung-chieh Shan for encouragement and many helpful discussions.

References

[1] Alfonso Acosta-Gómez (2007): *Hardware Synthesis in ForSyDe*. Master's thesis, Dept. of Microelectronics and Information Technology, Royal Institute of Technology, Stockholm, Sweden.

[2] Anders Bondorf (1992): *Improving Binding Times Without Explicit CPS-Conversion*. In Clinger [7], pp. 1–10.

[3] Denis Bueno (2009): *funsat-0.6.0: A modern DPLL-style SAT solver*. `http://hackage.haskell.org/package/funsat-0.6.0` `http://hackage.haskell.org/packages/archive/funsat/0.6.0/doc/html/Funsat-Circuit.html`.

[4] Jacques Carette & Oleg Kiselyov (2011): *Multi-stage Programming with Functors and Monads: Eliminating Abstraction Overhead from Generic Code*. *Science of Computer Programming* 76(5), pp. 349–375.

[5] Jacques Carette, Oleg Kiselyov & Chung-chieh Shan (2009): *Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages*. *Journal of Functional Programming* 19(5), pp. 509–543, doi:10.1017/S0956796809007205.

[6] Koen Claessen & David Sands (1999): *Observable Sharing for Functional Circuit Description*. In Thiagarajan & Yap [18], doi:10.1007/3-540-46674-6_7.

[7] William D. Clinger, editor (1992): *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*. *Lisp Pointers* V(1), ACM Press, New York.

[8] Conal Elliott, Sigbjorn Finne & Oege de Moor (2003): *Compiling Embedded Languages*. *Journal of Functional Programming* 13(3), pp. 455–481, doi:10.1017/S0956796802004574.

[9] A. P. Ershov (1958): *On programming of arithmetic operations*. *Communications of the ACM* 1(8), pp. 3–6, doi:10.1145/368892.368907.

[10] Jean-Christophe Filliâtre & Sylvain Conchon (2006): *Type-Safe Modular Hash-Consing*. In ML [14], pp. 12–19, doi:10.1145/1159876.1159880.

[11] Andy Gill (2009): *Type-safe observable sharing in Haskell*. In Weirich [20], pp. 117–128, doi:10.1145/1596638.1596653.

[12] Tom Hawkins (2008): *I love purity, but it's killing me.* `http://www.haskell.org/pipermail/haskell-cafe/2008-February/039339.html`.

[13] (2010): *Mescaline: a data-driven audio sequencer and synthesizer*. `http://mescaline.puesnada.es/` `http://mescaline.puesnada.es/doc/html/mescaline/src/Mescaline-Synth-Pattern-AST.html`.

[14] (2006): *2006 ACM SIGPLAN Workshop on ML*. ACM Press, New York.

[15] Matthew Naylor (2008): *Designing DSL with explicit sharing*. `http://www.haskell.org/pipermail/haskell-cafe/2008-February/039671.html`.

[16] Matthew Naylor (2008): *I love purity, but it's killing me.* `http://www.haskell.org/pipermail/haskell-cafe/2008-February/039347.html` `http://www.haskell.org/pipermail/haskell-cafe/2008-February/039449.html`.

[17] John T. O'Donnell (2003): *Embedding a Hardware Description Language in Template Haskell*. In Christian Lengauer, Don S. Batory, Charles Consel & Martin Odersky, editors: *Domain-Specific Program Generation*, *Lecture Notes in Computer Science* 3016, Springer, pp. 143–164, doi:10.1007/978-3-540-25935-0_9.

[18] P. S. Thiagarajan & Roland H. C. Yap, editors (1999): *Asian Computing Science Conference*. *Lecture Notes in Computer Science* 1742.

[19] Henning Thielemann (2008): *I love purity, but it's killing me.* `http://www.haskell.org/pipermail/haskell-cafe/2008-February/039343.html`.

[20] Stephanie Weirich, editor (2009): *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*. ACM Press, New York.