Common Subexpressions are Uncommon in Lazy Functional Languages

Olaf Chitil

Lehrstuhl für Informatik II, Aachen University of Technology, Germany
chitil@informatik.rwth-aachen.de
http://www-i2.informatik.RWTH-Aachen.de/~ chitil

Abstract. Common subexpression elimination is a well-known compiler optimisation that saves time by avoiding the repetition of the same computation. In lazy functional languages, referential transparency renders the identification of common subexpressions very simple. More common subexpressions can be recognised because they can be of arbitrary type whereas standard common subexpression elimination only shares primitive values. However, because lazy functional languages decouple program structure from data space allocation and control flow, analysing its effects and deciding under which conditions the elimination of a common subexpression is beneficial proves to be quite difficult. We developed and implemented the transformation for the language Haskell by extending the Glasgow Haskell compiler. On real-world programs the transformation showed nearly no effect. The reason is that common subexpressions whose elimination could speed up programs are uncommon in lazy functional languages.

1 Transformation of Different Language Classes

The purpose of common subexpression elimination (CSE) is to reduce the runtime of a program through avoiding the repetition of the same computation. The transformation statically identifies a repeated computation by locating multiple occurrences of the same expression. Repeated computations are eliminated by storing the result of evaluating the expression in a variable and accessing this variable instead of reevaluating the expression.

1.1 Imperative Languages

CSE is a well-known standard optimisation which is implemented in most compilers for imperative languages ([1]). The program to be optimised is represented as a flow graph whose nodes are basic blocks, that is sequences of 3-address instructions. An expression on the right hand side of an assignment is a common subexpression if it has been computed before and there is no assignment to any variable of the expression in between. For languages with pointers the latter condition is more complicated. Local elimination of all common subexpressions of a basic block is straightforward. Global elimination requires a data flow analysis, since an expression can only be eliminated, if it is computed on every path leading to its basic block.

It is important to note that some transformations are not feasible on source code level, because the required details are still hidden there. For example, Pascal only permits to access an array by an index, e.g. `a[i] := a[i]+1`. Assuming an array component requires 4 bytes this assignment is translated into the following 3-address code:

```
t1    := 4 * i;
t2    := a[t1]
t3    := t2 + 1;
t4    := 4 * i;
a[t4] := t3;
```

The programmer cannot avoid the repeated computation of `4 * i` which is eliminated by CSE.

Note that 3-address code only handles primitive data such as integers and floating point values and that the temporary variables `t1`, ..., `t4` are held in processor registers. The recomputation of complete arrays for example cannot be eliminated.

1.2 Strict Functional Languages

Appel implemented CSE in a compiler for the strict functional language ML ([2], Chapter 9). He uses continuation passing style as intermediate language on which all transformations operate. Whereas 3-address code consists of a sequence of instructions, continuation passing style code makes control flow explicit by nesting. Hence an expression is evaluated before another expression, if it syntactically dominates that expression. Only in case of syntactic domination common subexpressions can be eliminated. To increase the applicability of the transformation, an additional hoisting transformation is implemented which hoists a continuation expression above another. The following simplified example from ([2], Chapter 9) shows the transformation of

```
let f c = c (x+y) in f (x+y) k
```

This expression is written in continuation passing style as

```
FIX([(f, [c], PRIMOP(+, [VAR x, VAR y], [z], [
               APP(VAR c, [VAR z])]))],
   PRIMOP(+, [VAR x, VAR y], [w], [
    APP(VAR f, [VAR w, VAR k])]))
```

which is transformed by hoisting into

```
PRIMOP(+, [VAR x, VAR y], [w], [
  FIX([(f, [c], PRIMOP(+, [VAR x, VAR y], [z], [
                 APP(VAR c, [VAR z])]))],
    APP(VAR f, [VAR w, VAR k]))])
```

and by CSE into

```
PRIMOP(+, [VAR x, VAR y], [w], [
  FIX([(f, [c], APP(VAR c, [VAR w]))],
    APP(VAR f, [VAR w, VAR k]))])
```

Common subexpressions are restricted to expressions built from primitive operations which operate only on primitive types. We conclude that CSE for continuation passing style programs is very similar to CSE for imperative programs.

Appel reports that the transformation has no effect on run-time and only a minor positive effect on program size. However, Appel gives no explanation for this disappointing result. Subsequently developed ML-compilers still use CSE ([16]).

1.3 Lazy Functional Languages

To our knowledge, CSE has not yet been implemented in any compiler for lazy functional languages. As seen in the previous subsections, classic CSE is based on an explicit representation of control flow and data flow. In contrast, lazy functional languages decouple program structure from both control flow and data space allocation. That makes it hard, first, to ascertain that repeated expressions are evaluated repeatedly and, second, to predict the effect of the elimination of a common subexpression on space usage, a problem that we will discuss shortly.

These discrepancies suggest that CSE should be applied at a lower level in a compiler for a lazy functional language. The Glasgow Haskell compiler produces C programs. However, these C programs contain many indirect function calls via pointers ([12]). We suppose that these function calls severely limit the ability of the GNU C-compiler gcc to find common subexpressions. Unfortunately we are not able to verify this claim, because gcc does not provide an option for suppressing CSE.

On the other hand, there are several advantages of applying CSE directly to lazy functional programs.

First, lazy functional languages like Haskell are referentially transparent, that is, two identical expressions always denote the same value, independent of the time of evaluation. Hence the recognition of common subexpressions is easier to implement than for imperative languages or strict functional languages like Scheme and ML that have to take account of destructive updates and side effects. Thus even more common subexpressions may be recognised.

Second, CSE for lazy functional languages automatically recognises common subexpressions of arbitrary type. Therefore it is able to transform

```
sum [1..1000] + sum [-1000..1] + sum [1..1000]
```

into

```
let v = sum [1..1000] in v + sum [-1000..1] + v
```

CSE for imperative languages and as used by Appel can only eliminate an expression, if all its subexpressions including itself only handle primitive values.

The disadvantage of eliminating expressions of arbitrary type is that it can lead to a considerable increase in space requirements (cf. [10], Sections 14.7.2 and 23.4.2). Consider the transformation of the expression

```
sum [1..1000] + sum [-1000..1] + prod [1..1000]
```

into

```
let v = [1..1000] in sum v + sum [-1000..1] + prod v
```

The first expression creates three times a list of 1000 elements. The space of a list can be reclaimed immediately after the list is used by `sum` and `prod` respectively. Hence the amount of space required by one list suffices for the evaluation of the whole expression. In the second expression the space allocated for the list `[1..1000]` is not available when evaluating `sum [-1000..1]`, but can only be reclaimed after the evaluation of `prod v` has finished (assuming a left to right evaluation). In the case of such a "space leak" it could be cheaper to recompute the common expression. Santos shortly discusses CSE for the lazy functional language Haskell in [15] and points out this danger of "space leaks" He suggests restricting the type of common subexpression that are eliminated. We will follow up this idea in Section 4.2. However, if the lifetime of the original expressions overlap, then sharing compound values is even beneficial for space consumption.

To evaluate the usefulness of CSE for lazy functional languages in practise, we implemented it as an extension of the Glasgow Haskell compiler (GHC). In the next section we give a short introduction to GHC and present the simple lazy language Core on which our transformation operates. In Section 3 the transformation is developed in detail and in Section 4 we discuss, how the problems mentioned above are (partially) overcome. Because we do not want to loose the advantage of simplicity by performing a complex analysis, we have to find simple syntactic conditions under which the elimination of a common subexpression is beneficial. Section 5 discusses the implementation of the transformation. Afterwards, Section 6 presents measurements of the effects of the transformation on several real-world programs. In Section 7 we discuss the main result of this paper: CSE is not effective for our test programs because they have only few common subexpressions. This lack of common subexpressions is likely to be a characteristics of lazy functional languages. We conclude in Section 8.

2 The Glasgow Haskell Compiler and Core

We chose to implement CSE by extending GHC for the following reasons. First, GHC is heavily based on the "compilation by transformation" approach, that is, it consists of a front end which translates Haskell into a small lazy functional language named Core, a number of transformations which optimise Core programs, and a back end which translates Core into C. As much work as possible is done in the middle part. Furthermore, GHC has been designed with the goal

Program	$Prog \rightarrow$	$Bind_1; \ldots; Bind_n$	$n \geq 1$
Binding	$Bind \rightarrow$	$var = Expr$	Non-recursive
	$\mid$	`rec` $var_1 = Expr_1;$ $\ldots;$ $var_n = Expr_n;$	Recursive $n \geq 1$
Expression	$Expr \rightarrow$	$Expr\ Atom$	Application
	$\mid$	$Expr\ ty$	Type application
	$\mid$	$\lambda var_1 \ldots var_n$`->`$Expr$	Lambda abstraction
	$\mid$	$\Lambda tyvar_1 \ldots tyvar_n$`->`$Expr$	Type abstraction
	$\mid$	`case` $Expr$ `of` $\{Alts\}$	Case expression
	$\mid$	`let` $Bind$ `in` $Expr$	Local definition
	$\mid$	$con\ var_1 \ldots var_n$	Constructor $n \geq 0$
	$\mid$	$prim\ var_1 \ldots var_n$	Primitive op. $n \geq 0$
	$\mid$	$Atom$	
Atoms	$Atom \rightarrow$	var	Variable
	$\mid$	$Literal$	Unboxed object
Literals	$Literal \rightarrow$	$integer \mid float \mid \ldots$	
Alternatives	$Alts \rightarrow$	$Calt_1; \ldots; Calt_n; Default$	$n \geq 0$
	$\mid$	$Lalt_1; \ldots; Lalt_n; Default$	$n \geq 0$
Constr. alt.	$Calt \rightarrow$	$con\ var_1 \ldots var_n$`->`$Expr$	$n \geq 0$
Literal alt.	$Lalt \rightarrow$	$Literal$`->`$Expr$	
Default alt.	$Default \rightarrow$	`NoDefault`	
	$\mid$	var`->`$Expr$	

Fig. 1. Syntax of the Core language

that other people can extend it with new optimising transformations ([8], [3]). Second, it is one of the standard compilers for the lazy language Haskell. This permits us to test our transformation on real-world Haskell programs instead of toy programs in a toy language. The compiler itself is written in Haskell. We added our transformation to version 2.08 which implements Haskell 1.4 ([4], [9]).

The intermediate language of GHC, Core, is essentially the second-order λ-calculus augmented with `let`, `case`, data constructors, constants and primitive operations. The syntax of the language is given in Figure 1. To avoid always having to speak of global bindings and (local) `let` bindings, we refer to both kind of bindings as `let` bindings. The syntax does not include algebraic data type definitions, but data constructors are used in the patterns of case alternatives. Note that function arguments must be atoms to simplify the operational semantics of Core and many transformations. Core has a fixed operational semantics be-sides the usual denotational semantics to enable reasoning about the usefulness of a transformation. Hence we shortly describe the main characteristics of this operational semantics.

Type abstraction and application are only needed for the type system. No program code is generated for these constructs, because no types are passed at run-time.

The operational model of Core requires a garbage-collected heap. A heap object, also named closure, contains a data value, a function value, or is a thunk for suspended values. Like a function value, a thunk contains a pointer to its unevaluated code and an environment. The environment is the list of values of the free variables of the code. After a thunk has been evaluated it is overwritten by its freshly computed value. Thus lazy evaluation is implemented.

`let` bindings and only `let` bindings performs heap allocation. When a `let` binding is evaluated, a closure is allocated for the bound expression. If the bound expression is in weak head normal form (WHNF), a data value or function value is allocated, otherwise a thunk (trivial `let` bindings, i.e., `let x = y in ...`, are eliminated before code generation). Afterwards the body of the `let` is evaluated.

`case` expressions and only `case` expressions trigger evaluation. The evaluation of a case expression triggers the evaluation of the scrutinised expression to WHNF. The result is compared with the patterns of the alternatives and execution proceeds with the appropriate alternative.

A more detailed description of Core and the objectives of its design is given in [13].

3 Transformation Rules

We define CSE for Core by giving three transformation rules and three assisting rules. To argue that these rules suffice, we show how other transformations implemented in GHC transform a program into a form suitable for our transformation. In Section 4 we analyse in detail under which conditions our transformation rules are sure to reduce the execution costs of the transformed program and we restrict the application of the rules accordingly.

3.1 Dominating Expressions

The suggestive, general transformation rule

$$e'[e,e] \quad \rightsquigarrow \quad \texttt{let } x \texttt{ = } e \texttt{ in } e'[x,x]$$

certainly cannot be used, because we want only to eliminate a common subexpression when it is safe, that is, costs are not increased. The two occurrences of the expression e may be far apart and the chances that the value of both are needed during the evaluation of the program is low. Worse, a closure is always allocated for e in the transformed program, while this may not be the case for the original program. If the whole expression is inside the body of a λ-abstraction, then the transformation may lead to the allocation of an unbound number of closures. A closure for e also has a long lifetime: First, it is allocated before the value of e is needed. Then, all occurrences of x in e' reference the closure which can only be deallocated when it is no longer referenced. If the added `let` binding is global, then the closure will even never be deallocated at all.

We decided on a simple implementation that performs no complicated analysis. Hence, similar to Appel (cf. Section 1.2), we only look for common subexpressions when a named expression syntactically dominates another equal expression, that is, we use the transformation rule

$$\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e'[e] \quad \rightsquigarrow \quad \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e'[x] \tag{1}$$

The language Core can also express two other kinds of named, syntactically dominating expression, exemplified by the following expressions:

```
case tail xs of {ys -> tail xs}
case 6 * 7 of {I# x# -> fib (6 * 7)}
```

Whereas a programmer hardly writes such code, other program transformations may produce it. In fact, a strictness based transformation transforms `let` x `=` e `in` e' into `case` e `of` $\{C\ x_1 \ldots x_n$ `->` $e'[C\ x_1 \ldots x_n/x]\}$, if e' is strict in x and the type of x has only a single data constructor ([15], Section 3.6). All unboxed data types such as `Int#` have exactly one data constructor (see Section 5.2). Hence our transformation additionally applies the following two rules:

$$\texttt{case}\ e\ \texttt{of}\ \{\ldots\texttt{;}x\ \texttt{->}\ e'[e]\texttt{;}\ldots\} \rightsquigarrow \texttt{case}\ e\ \texttt{of}\ \{\ldots\texttt{;}x\ \texttt{->}\ e'[x]\texttt{;}\ldots\} \tag{2}$$

$$\begin{aligned} &\texttt{case}\ e\ \texttt{of}\ \{\ldots\texttt{;}\ C\ x_1 \ldots x_n\ \texttt{->}\ e'[e]\texttt{;}\ldots\} \\ \rightsquigarrow\quad &\texttt{case}\ e\ \texttt{of}\ \{\ldots\texttt{;}\ C\ x_1 \ldots x_n\ \texttt{->}\ e'[C\ x_1 \ldots x_n]\texttt{;}\ldots\} \end{aligned} \tag{3}$$

Our three rules do not perform an optimisation if the expression e is a variable. Furthermore, rule (3) cannot be applied in that case, because according to Core syntax a variable occurring as an argument of an application cannot be replaced by a constructor application. Sections 3.3 and 5.1 will show that if e is a variable the following reversed transformation rules are important for eliminating nested common subexpressions:

$$\texttt{let}\ x\ \texttt{=}\ y\ \texttt{in}\ e'[x] \quad \rightsquigarrow \quad \texttt{let}\ x\ \texttt{=}\ y\ \texttt{in}\ e'[y] \tag{1'}$$

$$\texttt{case}\ y\ \texttt{of}\ \{\ldots\texttt{;}x\ \texttt{->}\ e'[x]\texttt{;}\ldots\} \rightsquigarrow \texttt{case}\ y\ \texttt{of}\ \{\ldots\texttt{;}x\ \texttt{->}\ e'[y]\texttt{;}\ldots\} \tag{2'}$$

$$\begin{aligned} &\texttt{case}\ y\ \texttt{of}\ \{\ldots\texttt{;}\ C\ x_1 \ldots x_n\ \texttt{->}\ e'[C\ x_1 \ldots x_n]\texttt{;}\ldots\} \\ \rightsquigarrow\quad &\texttt{case}\ y\ \texttt{of}\ \{\ldots\texttt{;}\ C\ x_1 \ldots x_n\ \texttt{->}\ e'[y]\texttt{;}\ldots\} \end{aligned} \tag{3'}$$

Whereas it is still not guaranteed (but more probable) that the eliminated expression is computed twice in the original program, the transformation is safe in that the new program allocates no additional closures. The restriction to syntactically dominating expressions renders the transformation more similar to standard CSE in imperative languages. Standard CSE eliminates not all common subexpressions but only those that are already computed before on every execution path.

3.2 Flattening of `let` and `case` expressions

Considering only syntactically dominating expressions is not as severe a restriction as it seems. First of all, a Core program contains significantly more nested `let` expressions than a normal functional program, because Core requires the arguments of functions to be atoms. Regard the Haskell expression

```
sum [1..1000] + sum [-1000..1] + sum [1..1000]
```

In Core it looks as follows:[1]

```
let si =
  let s1 = let l1 = [1..1000] in sum l1 in
    let s2 = let l2 = [-1000..1] in sum l2 in
       s1 + s2
  in
    let s3 = let l3 = [1..1000] in sum l3 in
      si + s3
```

Our transformation rules cannot be applied to this program. However, GHC includes several transformations that flatten nested `let` and `case` expressions and thus bring a program into a form more suitable for our transformation. Note that in Core programs every bound variable is unique so that in the following transformation rules variable capture cannot arise.

– float `let` from `let`. ([15], Section 3.4.2)

$$\texttt{let } x = (\texttt{let } y = e_y \texttt{ in } e_x) \texttt{ in } e \rightsquigarrow \texttt{let } y = e_y \texttt{ in } (\texttt{let } x = e_x \texttt{ in } e)$$

This transformation is only applied, if e_y is in WHNF or the whole expression is strict in y.

– float `case` from `let`. ([15], Section 3.5.3)

$$\begin{array}{l} \texttt{let } x = \texttt{case } e \texttt{ of} \\ \qquad\qquad alt_1 \texttt{ -> } e_1 \\ \qquad\qquad \ldots \\ \qquad\qquad alt_n \texttt{ -> } e_n \\ \texttt{in } e' \end{array} \rightsquigarrow \begin{array}{l} \texttt{case } e \texttt{ of} \\ \qquad alt_1 \texttt{ -> let } x = e_1 \texttt{ in } e' \\ \qquad \ldots \\ \qquad alt_n \texttt{ -> let } x = e_n \texttt{ in } e' \end{array}$$

This transformation requires e' to be strict in x.

[1] The examples are simplified. In particular, the overloaded numbers of Haskell require the use of dictionaries.

– float `let` from `case`. ([15], Section 3.4.3)

$$\texttt{case (let } x \texttt{ = } e \texttt{ in } e'\texttt{) of}\ldots \;\rightsquigarrow\; \texttt{let } x \texttt{ = } e \texttt{ in (case } e' \texttt{ of } \ldots\texttt{)}$$

– float `case` from `case`. ([15], Section 3.5.2)

$$\begin{array}{l}
\texttt{case} \begin{pmatrix} \begin{array}{l} \texttt{case } e \texttt{ of} \\ \quad alt_1 \texttt{ -> } e_1 \\ \quad \ldots \\ \quad alt_n \texttt{ -> } e_n \end{array} \end{pmatrix} \texttt{of} \\
\quad alt'_1 \texttt{ -> } e'_1 \\
\quad \ldots \\
\quad alt'_n \texttt{ -> } e'_m
\end{array}
\;\rightsquigarrow\;
\begin{array}{l}
\texttt{case } e \texttt{ of} \\
\quad alt_1 \texttt{ -> } \begin{pmatrix} \begin{array}{l} \texttt{case } e_1 \texttt{ of} \\ \quad alt'_1 \texttt{ -> } e'_1 \\ \quad \ldots \\ \quad alt'_n \texttt{ -> } e'_m \end{array} \end{pmatrix} \\
\quad \ldots \\
\quad alt_n \texttt{ -> } \begin{pmatrix} \begin{array}{l} \texttt{case } e_n \texttt{ of} \\ \quad alt'_1 \texttt{ -> } e'_1 \\ \quad \ldots \\ \quad alt'_n \texttt{ -> } e'_m \end{array} \end{pmatrix}
\end{array}$$

Join points are used to avoid code duplication ([13], Section 5.1; [15], Section 3.5.2).

Furthermore, GHC performs a full laziness transformation ([15], Section 5.2) which lifts expressions from a λ-abstraction. The full laziness transformation may introduce new application possibilities for CSE, similar to the hoisting transformation implemented by Appel.

3.3 Application of the Transformation Rules: An Example

Consider our `sum` example of the previous subsection. Strictness analysis infers that the expression is strict in all subexpressions and thus all `let` bindings of numbers are transformed into `case` expressions. A subsequent application of the transformations listed in the previous subsection leads to the following program. The data constructor `I#` is part of the unboxed representation of integers (see Section 5.2).

```
let l1 = [1..1000] in
  case (sum l1) of
   I# s1 -> let l2 = [-1000..1] in
             case (sum l2) of
              I# s2 -> case (s1 +# s2) of
                        I# si -> let l3 = [1..1000] in
                                  case (sum l3) of
                                   I# s3 -> case (si +# s3) of
                                             s -> I# s
```

Applying rule (1) removes the second occurrence of `[1..1000]`:

```
let l1 = [1..1000] in
  case (sum l1) of
   I# s1 -> let l2 = [-1000..1] in
             case (sum l2) of
              I# s2 -> case (s1 +# s2) of
                        I# si -> let l3 = l1 in
                                  case (sum l3) of
                                   I# s3 -> case (si +# s3) of
                                             s -> I# s
```

Rule (1’) renames variable `l3` to `l1` in the body of the `let` binding so that rule (3) can be applied:

```
let l1 = [1..1000] in
  case (sum l1) of
   I# s1 -> let l2 = [-1000..1] in
             case (sum l2) of
              I# s2 -> case (s1 +# s2) of
                        I# si -> let l3 = l1 in
                                  case I# s1 of
                                   I# s3 -> case (si +# s3) of
                                             s -> I# s
```

The existing transformations of GHC finally yield the simplified program:

```
let l1 = [1..1000] in
  case (sum l1) of
   I# s1 -> let l2 = [-1000..1] in
             case (sum l2) of
              I# s2 -> case (s1 +# s2) of
                        I# si -> case (si +# s1) of
                                  s -> I# s
```

3.4 Swapping of independent `let` and `case` expressions

A disadvantage of the restriction to textually dominating expressions is that our transformation rules may not be applicable because of the accidental order of independent `let` or `case` expressions. The program

```
let y = 42 in (let x = 42 + 1 in f x y)
```

is transformed whereas

```
let x = 42 + 1 in (let y = 42 in f x y)
```

is not. Unfortunately the independence of several `let` expressions cannot be made explicit in the Core language; similarly for `case` expressions.

It is possible to extend our implementation of CSE, that we present in Section 5, to eliminate common subexpressions even in such cases. We did not yet do so, because we do not expect this case to occur very often.

4 Analysis of the Effects of the Transformation

Here we discuss application conditions for our rules which assure that common subexpression elimination reduces costs. For lazy functional languages the costs of major interest are run-time, total heap allocation, maximal heap residency, that is the maximal space required by life objects on the heap at one time, and size of the program code. Finally we take up Santos' idea of restricting the type of eliminated subexpressions to avoid space leaks.

The observant reader will notice that our transformation rule (1) is just the inverse of another well-known transformation: inlining. Inlining replaces occurrences of a `let`-bound variable by its defining expression to remove function-call overhead and to expose the defining expression to local context information and thus to increase the possibility of other transformations being applied.

Transformations that are inverses of each other occur quite often in GHC, it applies for example a let floating inward and a let floating outward transformation ([15], Sections 5.1 and 3.4). For determining the conditions for applying CSE we just have to reverse the known arguments for inlining ([15], Section 6; [13], Section 4).

4.1 Run-Time and Code Size

We have to distinguish two kinds of common subexpressions. If the expression concerned is a WHNF, that is, a variable, a literal, a constructor application, or a λ-abstraction, then CSE cannot save execution time, because the expression is already evaluated. Replacing the expression by a variable may even increase run-time, because an additional indirection is introduced ([15], Section 3.2.3). Eliminating a common non-WHNF saves execution time, if the value of at least two of its occurrences in the original program are needed.

A special case is a common subexpression that is the right hand side of a `let` binding. Replacing this expression by a variable yields a trivial binding of the form `let x = y in ...` which is later eliminated by an existing transformation. Thus run-time and code size are reduced.

In addition to decreasing run-time, CSE can also decrease program size. It should however be noted that eliminating small expressions like constructor applications has probably no effect on program size.

4.2 Avoidance of Space Leaks

Whereas the transformation may only decrease total heap usage, it may considerably influence heap residency both positively and negatively. The latter case is demonstrated in Section 1 by an example and has to be avoided.

Eliminating common subexpressions that are in WHNF does not increase heap residency, but we have seen that except for right hand sides of `let` bindings their elimination is not advantageous.

Santos suggests to eliminate only expressions of certain types ([15], Sections 8.5.11 and 8.6.2), similar to his approach to the full laziness transformation. If the values of the expressions only take a small, fixed amount of space, then the increased lifetime of the values on the heap should hardly matter.

Santos considers only types that are not recursive and do not contain recursive types as subcomponents. This restriction is not sufficient. A partially evaluated expression of a structured type may require an unbound amount of heap space, because it may contain an arbitrary number of (linked) thunks. The following example illustrates this.

```
f 0     = (0, 0)
f (n+1) = case (f n) of (l, m) -> (0, m+1)

let z = f 1000 in
  case z of
    (x, y) -> -> e[f 1000]
```

After `z` has been evaluated by the `case` expression, `y` is represented by 1001 thunks denoting the unevaluated expression $0 + 1 + 1 + \ldots + 1$. If neither `y` nor `z` occur in the body of the case expression this space can immediately be reclaimed by the garbage collector. However, if CSE replaces the second occurrence of `f 1000` by `z`, then this space cannot be reclaimed and thus heap residency is increased considerably.

Hence we see that a partially evaluated expression is certain to require only a small, fixed amount of space, if and only if its WHNF is already its normal form and it is not a function, whose environment may refer to arbitrary large data structures. Examples are all expressions of type `Bool`, `Char`, `Int`, and `Float`. We call such types *safe*.

To assure that a common expression is evaluated before sharing takes place, we apply rule (1) only if the `let` expression is strict in the `let`-bound variable. GHC applies a `let` to `case` transformation after all other transformations which guarantees that the `let`-bound expression is evaluated before the body of the `let` expression.

Eliminating only those non-WHNFs that are evaluated and of safe type assures that space leaks cannot occur. The transformation is still more general than standard CSE, because the subexpressions of the eliminated expression may be of arbitrary type.

5 Implementation of the Transformation

The restriction of the transformation to the elimination of subexpressions which are `let`- or `case`-bound in the same scope permits a single recursive traversal of the Core program. The named, dominating expressions are collected in a tree data structure with logarithmic lookup time. The comparison of two expressions modulo α-conversion by recursive decent is nearly always decided after examination of the top of the two expressions. Thus, compared to the rest of the compiler, the time spend on the transformation is not noticeable in practise.

The implementation is available from the author.

5.1 Recognition of Common Subexpressions

For each expression we first transform its subexpressions and then test, if the whole transformed expression occurred before. This order is necessary to gain a cumulative effect and it thus assures that the transformation is idempotent, that is, after being applied once a second application has no effect. For example we get

```
   let x = 3 in let y = 2+3 in 2+3+4
⇝ let x = 3 in let y = 2+x in y+4
```

while doing lookup first and then recursive transformation leads to

```
⇝ let x = 3 in let y = 2+x in 2+x+4
```

Section 4 showed that the transformation rules may only be applied under a given condition. Only common subexpressions that are evaluated non-WHNFs of safe type or that are WHNFs which occur as the right hand side of a `let` binding are eliminated.

This condition may restrict our transformation more than desirable. Consider the example of Section 3.3. The subexpression `[1..1000]` is not to be eliminated because it is not of safe type. In consequence rules (1') and (3) cannot be applied either. Thus the program is not optimised.

To solve this problem the function that transforms a subexpression returns two expressions. The first is the result of CSE under observance of the given condition, but the second is the subexpression with all common subexpressions eliminated. The latter version is memorised in the mentioned tree data structure and is thus the basis for comparison of expressions.

Hence in the example `l3` is eliminated although `[1..1000]` is not. The reader may also assure himself that the expression

```
sum [1..1000] + sum [-1000..1] + prod [1..1000]
```

is not modified by the transformation.

5.2 Considerations specific to the Glasgow Haskell Compiler

Unboxed Values. Implementations of non-strict languages like Haskell process so called boxed values, that is pointers into the heap that either point to a delayed computation or the actual value. In order to improve efficiency Core is also able to handle actual values directly, which are named unboxed values and are distinguished by their types. These unboxed values have been added carefully, so that — although they introduce explicit strictness into the otherwise non-strict language — they do not invalidate any program transformation, provided the produced code observes the following two restrictions: no polymorphic function is applied to an expression of unboxed type and every expression of unboxed type which appears as the argument of an application or as the right-hand side of a binding is in head normal form, that is, a literal, an application of an unboxed constructor, or a variable (see [11] for details).

Fortunately, the transformation handles unboxed data types correctly: the types of arguments of (polymorphic) functions are not changed and the transformation never turns an expression which is in head normal form into one that is not.

We can also add simple unboxed data types like `Bool#`, `Char#`, `Int#`, `Float#` and `Double#` to the list of safe types. However, since the right-hand side of a `let` binding that is of unboxed type has to be in head normal form, only rules (2) and (3) will benefit from this extension.

Uses Type System. Based on ideas from linear logic the type system of Core has been extended in version 2.01 by so called uses, which record when a value is used (accessed) at most once. This knowledge permits to avoid update of closures which are not accessed again, enables update in place of data structures whose old value is no longer needed, and may guide program optimisations, especially safe situations for inlining can be determined. Uses are attached to types. The use 1 of a type indicates that its values are used at most once, while the use ω indicates that the values of the type may be used any number of times. A program transformation has to produce Core programs that respect the use type system (see [6] for details).

The usage information is hardly useful for CSE. Only if a `let` or `case` bound variable has use 1, then common subexpression elimination very probably saves execution time iff after the transformation a repeated uses type inference yields use ω for the variable.

Our implementation of common subexpression evaluation violates the uses type system. Consider the following transformation:

```
let x = 1+2 in let y = 1+2 in x+y   ⤳   let x = 1+2 in x+x
```

In the left expression the type of the variable `x` may have use 1, and in the right expression it should have use ω. There does not seem to be any good solution to this problem. The transformation may either be restricted to variables of use ω, or the use of all variables used for subexpression elimination is set to ω, or the program has to be type checked again after the transformation. Currently this does not matter since version 2.08 of GHC does not yet make use of the use information for code generation or any program transformation.

Cost Centres. Finally, programs that are compiled for profiling are annotated with cost centres. Not respecting these annotations, that is, moving subexpressions from the scope of one cost centre to another, does not change the semantics of the program, but it invalidates the profiling measurements. Sansom and Peyton Jones suggest to curtail transformations to never move costs across a cost centre annotation. This means however, that observing a program (annotating it with cost centres) influences the behaviour to be observed! Alternatively, a subexpressions that is moved out of the scope of a cost centre can be annotated with its original cost centre. Nonetheless this usually moves a small cost to another cost centre and it evidently complicates every transformation (see [14] for details).

Our transformation eliminates subexpressions without caring about the scope of cost centres. It is not even clear how the cost of a common subexpression could be shared between cost centres. Cost centre annotations also limit the applicability of the transformation since terms which differ only by their annotation are regarded as different.

6 Measurement of the Effects of the Transformation

The optimisation option `-O2` of GHC turns on a long sequence of optimisations. We inserted our transformation three times into this sequence. Our transformation is invoked for the first time after the strictness analysis, because we observed in Sections 3 and 4 that the applicability of CSE is increased by various `let` and `case` floating transformations and by strictness analysis. As standard for comparison we use the same sequence of optimisations without CSE.

The objects of our comparison are the `sum` example of Section 3.2 and nine programs from the Glasgow nofib test suite, version 2.5 [7]. The latter are real applications, that is, applications that were not designed as benchmarks but to solve particular problems, for instance text compression (`compress`) and Monte Carlo photon transport (`gamteb`).

Table 1 shows the results of our measurements. The size of each program's source is given in number of lines. Afterwards the maximum of the number of common subexpressions that were recognised in each of the three passes of CSE is given. Obviously common subexpressions that are not eliminated are usually rediscovered in latter CSE passes. The subsequent number is the sum of common subexpressions eliminated by all three passes. Remember that all type information is dropped by GHC during code generation. To obtain meaningful numbers those common subexpressions which are just applications of variables to types are not counted.

The last four columns show the measured effects of CSE on costs, that is, run-time, the total amount of heap allocated, maximal heap residency, and code size. These numbers were obtained by using the profiling facilities of GHC. For all times we took the minimum of at least six runs. All measurements are given as differences in per cent between the numbers for the program compiled with CSE and those for the program compiled without. Horizontal lines signify that no meaningful numbers could be gained because the run-time was too short.

Both the numbers of recognised and of eliminated subexpressions have a very weak relationship with the size of the respective program. Most programs have few common subexpressions and even fewer which can be eliminated safely.

The measurements prove that our `sum` example profits considerably from the transformation. Both run-time and total heap allocation are reduced by the expected one third. However, the effect on the real-world programs of the nofib suite is disappointing. Only the run-time of `reptile` is slightly improved. Additionally the code size of all programs is slightly decreased.

To evaluate the effect of the condition for avoiding space leaks that was introduced in Section 4.2, Table 2 shows the measurements of an older version of our

program	lines	max. recogn.	elim.	time	total alloc.	max. residency	code
sum	2	3	1	**-30 %**	**-33.3 %**	0 %	-0.08 %
compress	320	3	0	0 %	0 %	0 %	0 %
fulsom	1357	93	22	-0.17 %	0 %	0 %	-0.55 %
gamteb	718	30	30	0 %	+0.02 %	-0.02 %	-0.08 %
grep	356	124	24	–	0 %	–	-0.59 %
lift	2033	97	5	–	0 %	0 %	-0.11 %
pic	544	29	0	0 %	0 %	0 %	0 %
prolog	539	31	5	–	-0.25 %	+0.5 %	-0.27 %
reptile	1522	115	31	**-0.8 %**	0 %	0 %	-0.34 %
rsa	74	7	0	0 %	0 %	0 %	0 %

Table 1. Eliminated common subexpressions and effect on costs

transformation which does not implement the condition. The increase of maximal heap residency of `prolog` exhibits the lack of the condition. Beware that the two tables are not directly comparable, because the old version did not implement rules (2') and (3') and the transformation was only applied once instead of three times. Nonetheless, the larger impact of the old transformation proves that the space safety condition does prohibit useful transformations. An analysis of the program `rsa` reveals that only two common subexpressions, `int2Integer# 2` and `int2Integer# 128`, are responsible for the decrease of run-time by 1.7 %. These expressions are not eliminated by the safe transformation because the strictness analyser could not infer the fact that they are demanded.

program	elim.	time	total alloc.	max. residency
sum	2	**-26 %**	**-33.4 %**	+0.001 %
compress	0	–	–	–
fulsom	20	+0.3 %	-0 %	-0.01 %
gamteb	15	**+1.4 %**	+0.02 %	-0.3 %
grep	17	–	**-3.1 %**	–
lift	7	0 %	-0.09 %	-0.03 %
pic	6	+0.5 %	+0.4 %	-0.01 %
prolog	6	0 %	-0.3 %	**+6 %**
reptile	32	**-1.7 %**	-0.2 %	-0.02 %
rsa	4	**-1.7 %**	-0.24 %	**-2.5 %**

Table 2. CSE without considerations for space leaks

7 Lack of Common Subexpressions

Unfortunately our transformation optimises only our demonstration example but none of our real-world programs. Obviously the elimination of a single common subexpressions cannot generally be expected to have an effect as large as in the `sum` example. The measurements suggest that the reason for the lack of speedup is that the real-world programs have few common subexpressions and even fewer which can be eliminated safely. We suppose that the lack of common subexpressions is also the cause of Appel's disappointing results of CSE.

Common subexpressions may either already exist in the source program or be introduced by the compiler. A programmer avoids repeated computations. The purpose of CSE for imperative languages is to eliminate repeated computations that are introduced by the compiler. The classical example is array indexing. However, first, arrays are seldom used in functional programs and, second, they are implemented by calling C-functions which are not reachable by transformations working on Core level. Our measurements suggest that GHC introduces few repeated computations. The only significant exception are overloaded numerical constants. In Haskell a constant `42` has to be replaced by `fromInteger 42` by the compiler. Nonetheless in most cases the existing specialisation and full-laziness transformation optimise these hidden common subexpressions sufficiently.

The idea comes up that the existence of a CSE phase may cause programmers to write their programs without worrying about repeated computations. A programming style that encourages the use of abstract data types may also lead to an increase of common subexpressions. Similar to the example of array indexing the limited interface of an abstract data type would prevent a programmer from avoiding duplicated computations himself. Furthermore, compiler phases could be simplified or improved if they were permitted to create common subexpressions. For example, in deforestation code duplication is a problem ([5]).

However, lazy functional languages decouple program structure from data space allocation and control flow. Consequently, as we discussed in Chapters 3 and 4, CSE has to be restrained by complex conditions to avoid a decrease of performance Hence CSE cannot be transparent, that is neither a functional programmer nor an implementor of an optimisation phase can easily assure that common subexpressions introduced by him will be eliminated.

8 Conclusion

In this paper we have developed a version of CSE for a lazy functional language, implemented it by adding it to GHC, and measured its effects on real-world programs.

Generally, the development of CSE in Section 3 demonstrates the importance of close interaction of transformations. Several existing optimisations increase the opportunities for applying CSE. Especially an improved strictness analysis would improve our transformation. Because the effect of CSE depends highly on the operational semantics of Core it is unfortunately difficult to transfer the discussion of Section 4 to another implementation of a lazy functional programming language. In fact, Section 5.2 proves that detailed knowledge of GHC specific properties like the underlying abstract machine (the STG-machine), unboxed data types, the use type system, cost centres, etc. is required for a real implementation.

The danger of creating space leaks are the same for CSE as for the full laziness transformation. We showed that the full laziness transformation implemented in GHC is not safe and gave a safe alas more restrictive transformation condition. Both transformations would profit from the development of a less restrictive safety condition.

Our implementation is fast and simple. Unfortunately it optimises only our demonstration example but none of our real-world programs. There is still room for improving the transformation. We already mentioned that possibly the space safety conditions could be partially lifted. More use of the context of common subexpressions could be made than only handling the right hand sides of `let` bindings specially. Cross-module CSE could enable the replacement of expressions by variables defined in imported modules. Finally, analysis techniques could be developed, which ascertain that repeated expressions are actually evaluated repeatedly.

However, the fundamental reason why our transformation is unsuccessful is that our test programs have only few common subexpressions. In Chapter 7 we gave good reasons for common subexpressions generally being uncommon in lazy functional languages. Note that the full laziness transformation is probably more effective, because expressions that can be lifted from a λ-abstraction are more difficult to spot and hence to avoid for a programmer than common subexpressions.

We claim that whereas CSE is an important optimisation for imperative languages it is not suitable for reducing the run-time of lazy functional programs. It remains to be investigated if the transformation could be used for reducing code size and thus also code generation time by eliminating large common WHNFs which were introduced by inlining but which did not enable other transformations.

Acknowledgement

I thank all members of the implementation of functional programming languages group at Aachen for numerous discussions and Simon Peyton Jones for answering many questions about GHC. I am grateful to one of the referees for suggesting the title of this paper.

References

1. A. Aho, R. Sethi, J. Ullman: *Compilers: Principles, Techniques and Tools*; Addison-Wesley, 1986.
2. Andrew W Appel: *Compiling with Continuations*; Cambridge University Press, 1992.
3. Olaf Chitil: *Adding an Optimisation Pass to the Glasgow Haskell Compiler*; unpublished report; `http://www-i2.informatik.rwth-aachen.de/~chitil/`, November 1997.
4. *The Glasgow Haskell compiler*; `http://www.dcs.gla.ac.uk/fp/software/ghc/`.

5. Simon D. Marlow: *Deforestation for Higher-Order Functional Programs*; PhD Thesis, University of Glasgow, September 1995.
6. Christian Mossin, David N. Turner, and Philip Wadler: *Once upon a type*; Technical Report TR-1995-8, University of Glasgow, 1995. Extended version of *Once upon a type* in 7'th International Conference on Functional Programming Languages and Computer Architecture, June 1995.
7. Will Partain: *The nofib benchmark suite of Haskell programs*; Functional Programming, Glasgow 1992, Workshops in Computing, Springer-Verlag, pp 195–202.
8. Will Partain: *How to add an optimisation pass to the Glasgow Haskell compiler (two months before version 0.23)*; part of the GHC 0.29 distribution, October 1994.
9. John Peterson and Kevin Hammond (eds.): *Report on the Programming Language Haskell, Version 1.4*; `http://haskell.org`, 1997.
10. Simon L. Peyton Jones: *The Implementation of Functional Programming Languages*; Prentice-Hall, 1987.
11. Simon L. Peyton Jones and John Launchbury: *Unboxed values as first class citizens in a non-strict functional language*; Conf. on Functional Programming Languages and Computer Architecture, 1991, pp 636–666.
12. Simon L. Peyton Jones: *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine*; J. Functional Programming, **2** (2):127–202, 1992.
13. Simon L. Peyton Jones and André L. M. Santos: *A transformation-based optimiser for Haskell*; submitted to Science of Computer Programming, 1997.
14. Patrick M. Sansom and Simon L. Peyton Jones: *Time and space profiling for non-strict, higher-order functional languages*; 22nd ACM Symposium on Principles of Programming Languages, January 1995.
15. André L. M. Santos: *Compilation by transformation in non-strict functional languages*; PhD Thesis, University of Glasgow, July 1995.
16. D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee: *TIL: A Type-Directed Optimizing Compiler for ML*; Sigplan Symposium on Programming Language Design and Implementation, 1996.