Common Subexpression Elimination in a Lazy Functional Language

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. To our knowledge it has not yet been applied to lazy functional programming languages, although there are several advantages. First, the referential transparency of these languages makes the identification of common subexpressions very simple. Second, 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 and measured its effectiveness on real-world programs.

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 ([ASU86]). 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 already 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 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 like 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 ([App92], 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 that hoists a continuation expression above another. The following simplified example from ([App92], Chapter 9) shows the transformation of the expression

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

It 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 those built from primitive operations that 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 runtime and only a minor positive effect on program size. However, Appel gives no explanation for this disappointing result.

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, 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.

This suggests 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 ([Pey92]). We suppose that this limits the ability of the GNU C-compiler gcc to find common subexpressions severely. 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 subexpression 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. [Pey87], 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 immediately be reclaimed 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 [San95] 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 3.5. However, if the lifetime of the original expressions overlap, then sharing compound values is even beneficial for space consumption.

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. To evaluate the usefulness of CSE for lazy functional languages in practise, we implemented it in the Glasgow Haskell compiler (GHC) and measured the effects of the transformation on real-world programs.

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 we discuss, how the problems mentioned above are (partially) overcome. Section 4 discusses the implementation of the transformation. Afterwards, Section 5 presents measurements of the effects of the transformation on several programs. We conclude in Section 6.

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 that other people can extend it with new optimising transformations ([Par94]). 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 2.04 which implements Haskell 1.4 ([GHC]).

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 thus many transformations. Core has a fixed operational semantics besides 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 runtime.

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 now-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 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 [PeySan97].

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

3 Development of the Transformation

We define CSE for Core by giving three term rewriting rules. To argue that these three rules suffice, we show how other transformations implemented in GHC transform a program into a form suitable for our transformation. Then we discuss the application conditions of our rules that assure that common subexpression elimination reduces costs. For lazy functional languages the costs of major interest are runtime, 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.

3.1 Transformation Rules

We decided on a simple implementation that performs no complicated analysis. Hence we only want to eliminate a common subexpression when this is safe, that is, it cannot increase costs. Although absolute safety is probably unattainable, the obvious, general transformation rule

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

certainly cannot be used. The two occurrences of the expression e may be very far apart and the chances that the value of both is 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. This closure also has a long lifetime, the maximum of the lifetimes of the two occurrences of e in the original program. If the whole expression is inside the body of a λ-abstraction, then this may lead to the allocation of an unbound number of closures. If the added `let` binding is global, then the closure will never be deallocated at all (see Section 4.3).

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

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

The language Core can also express another type of named, syntactically dominating expression: `case 6 * 7 of {x -> fib 6 * 7}`

Programmers do not write such code, but if an expression e' is strict in x then the expression `let` x `=` e `in` e' is transformed into `case` e `of` $\{x$ `->` $e'\}$ ([San95], Section 3.6). Hence we add the transformation rule

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

Later we learned that this strictness transformation is only applied after all other transformations (to avoid having to handle similar kinds of named expressions during inlining). However, GHC has a special strictness transformation that is executed earlier. If the type of e has only a single data constructor C, then `let` x `=` e `in` e' is transformed into `case` e `of` $\{Cx_1 \ldots x_n$ `->` $e'[Cx_1 \ldots x_n/x]\}$. Therefore we add the rule

$$\begin{aligned} &\texttt{case}\ e\ \texttt{of}\ \{\ldots\texttt{;}\ Cx_1\ldots x_n\ \texttt{->}\ e'[e]\texttt{;}\ldots\} \\ \rightsquigarrow\quad &\texttt{case}\ e\ \texttt{of}\ \{\ldots\texttt{;}\ Cx_1\ldots x_n\ \texttt{->}\ e'[Cx_1\ldots x_n]\texttt{;}\ldots\} \qquad (3)\end{aligned}$$

This rule is important for handling unboxed data types, which have exactly one data constructor (see Section 4.3).

The restriction to syntactically dominating expressions renders the transformation more similar to CSE in imperative languages. Standard CSE eliminates not all common subexpressions but only those that are already computed before on every execution path.

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.

3.2 Flattening of `let` and `case` expressions

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

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

In Core it looks as follows:[1]

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

```
let s =
  let s1 = let l1 = [1..1000] in sum l1 in
    let s2 = let l2 = [-1000..1] in sum l2 in
       s1 + s2
  in
    let p = let l3 = [1..1000] in prod l3 in
      s + p
```

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:

– float `let` from `let`. ([San95], Section 3.4.2)

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

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

– float `case` from `let`. ([San95], Section 3.5.3)

```
                                 case e of
let x = case e of                     alt_1 -> let x = e_1
          alt_1 -> e_1           in e
          ...               ⇝         ...
          alt_n -> e_n                alt_n -> let x = e_n
in e'                            in e
                                 in e'
```

This transformation requires e' to be strict in x.

– float `let` from `case`. ([San95], 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`. ([San95], Section 3.5.2)

```
                                        case e of
                                                  ⎛case e_1 of        ⎞
     ⎛case e of         ⎞                 alt_1 -> ⎜     alt'_1 -> e'_1 ⎟
case ⎜     alt_1 -> e_1 ⎟ of                      ⎜     ...            ⎟
     ⎜     ...          ⎟                         ⎝     alt'_n -> e'_m ⎠
     ⎝     alt_n -> e_n ⎠           ⇝         ...
   alt'_1 -> e'_1                                 ⎛case e_n of        ⎞
   ...                                    alt_n -> ⎜     alt'_1 -> e'_1 ⎟
   alt'_n -> e'_m                                 ⎜     ...            ⎟
                                                  ⎝     alt'_n -> e'_m ⎠
```

Join points are used to avoid code duplication (see Section 4.3)

Strictness analysis infers that our example 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 above leads to the following 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# s -> let l3 = [1..1000] in
                                   case (sum l3) of
                                    I# p -> case (s +# p) of
                                             q -> I# q
```

The data constructor `I#` is part of the unboxed representation of integers (see Section 4.3). CSE can easily remove the second occurrence of `sum [1..1000]` by applying rule 1 and 3.

Furthermore, GHC performs a full laziness transformation that, similar to the hoisting transformation implemented by Appel, may introduce new application possibilities for CSE.

3.3 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 transformed. 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 4, 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.

3.4 Application Conditions of the Transformation rules

The observant reader will notice that our first transformation rule 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 ([San95], Sections 5.1 and 3.4). For determining the conditions for applying CSE we just have to reverse the known arguments for inlining ([San95], Section 6; [PeySan97], Section 4).

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 runtime, because an additional indirection is introduced ([San95], Section 3.2.3). Eliminating a common non-WHNF saves execution time, if the value of both expressions is needed.

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

A special case is a common subexpression that is the right hand side of a `let` binding. Replacing this expression by a variable always leads to a complete removal of the `let` binding (see section 4) and is thus reduces runtime and code size.

Hence common subexpression elimination also subsumes the constructor reuse transformation applied by GHC ([San95], Section 3.2.3).

3.5 Avoidance of Space Leaks

Here we discuss the effect of CSE on space usage. Whereas the transformation always decreases total heap usage (not however that constructor applications often do not require the allocation of a closure at all), it may considerably influence heap residency.

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 ([San95], Sections 8.5.11 and 8.6.2), similar to his approach to the full laziness transformation. If the values of the expressions only takes a small, fixed amount of space, then the increased lifetime of the values on the heap should hardly matter.

Santos suggests to consider only types that are not recursive and do not contain recursive types as subcomponents. This restriction is however 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 (x, y) -> (x+1, y+1)

let z = f 1000 in
  case z of
    (x, y) -> case x+1 of
                x' -> e[z]
```

When e`[z]` is evaluated, y is represented by 1001 thunks denoting the unevaluated expression $0 + 1 + 1 + \ldots + 1$.

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

Still, an unevaluated expression, that is a thunk, may require an arbitrary amount of heap space, because its environment may refer to arbitrary large data structures.

Therefore a space leak can only safely be avoided by restricting the transformation to `case` expressions, that is, to rules 2 and 3, and to safe types. Nonetheless the transformation is more general than standard CSE, because the subexpressions of the eliminated expression may be of arbitrary type.

4 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.

4.1 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 sensible in order to gain a cumulative effect, e.g. 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
```

By eliminating the expression on the right hand side of a `let` binding the transformation may produce trivial `let` bindings of the form `let` x `=` y `in` e. The transformation eliminates such trivial `let` bindings by the additional rule:

$$\texttt{let}\ x\ \texttt{=}\ y\ \texttt{in}\ e \quad \leadsto \quad e[y/x] \tag{4}$$

This helps in finding more common subexpressions.

Note that we cannot reduce expressions of the form `case` x `of` $\{$ y `->` e $\}$ to $e[y/x]$, because in the first expression x is evaluated to WHNF whereas in the second it is not. This may change the termination behaviour.

When implementing the restriction to WHNFs and safe types we have to be careful to not to loose many good transformations, since e.g.

```
   let v = [1..1000] in sum v + (sum [-1000..1] + sum [1..1000])
 ⤳ let v = [1..1000] in let z = sum v in z + (sum [-1000..1] + z)
```

depends on `[1..1000]` being recognised as a common subexpression as well.

For this purpose the functions that transforms a subexpression does not only return the transformed subexpression but also a version of the subexpression in which all common subexpressions are eliminated. The latter version is memorised when passing `let` or `case` bindings during recursive decent and it is the basis for comparison of expressions.

Altogether this assures that the transformation is idempotent, that is after being applied once a second application has no effect.

4.2 Cost of the Transformation

The single recursive traversal of the core program takes linear time in the size of the program and lookup in the table happens in logarithmic time in its size. The comparison of two expressions modulo α-conversion by recursive decent can be regarded similarly. However, since in practise a comparison is nearly always decided after examination of the top of the two expressions, it can realistically be considered as constant. Hence our transformation takes roughly $\mathcal{O}(n \log n)$ time, with n being the size of the Core program. Compared to the rest of the compiler the time spend on the transformation is not noticeable in practise.

The current implementation of the transformation makes a complete copy of the Core program. This could be avoided for unchanged subexpressions to reduce garbage collection.

4.3 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 [PeyLau91] 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 save types. However, since the right-hand side of a binding that is of unboxed type has to be in head normal form, the transformation will only be applied to the scrutinee of a `case` expression.

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 save 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 [MTW95] 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.

Unfortunately our implementation of common subexpression evaluation even 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 2.04 of GHC does not yet make use of the use information for code generation or any program transformation.

Join Points. When explaining the operational semantics of Core in Section 2 we said that every `let` binding allocates a closure. This is not true. The back end of GHC performs a simple syntactic escape analysis to identify `let` bound variables whose evaluation is certain to take place before evaluation of the body of the `let` expression has terminated. In that case the `let` binding is implemented, not by allocating a closure, but by jumping to some common code whenever the bound variable is subsequently evaluated ([PeySan97], Section 5.1; [San95], Section 3.5.2).

Thus our analysis of the effects of CSE are called into question. On the one hand the introduction of a `let` binding of this kind does not increase total heap allocation or heap residency. On the other hand the replacement of a subexpression by a variable may turn a non-escaping `let` binding into a normal `let` binding.

Because we do expect the later to happen very rarely and escape information is only generated in the back end of the compiler, we ignored these effects on the effectiveness of the transformation.

Top Level Constants. Top level constants, so called constant applicative forms (CAFs), are never garbage collected by the runtime system of GHC. Hence replacing a subexpression of unsafe type by a top level variable almost certainly leads to a serious space leak. We did not distinguish between global and local bindings in our implementation but could easily do so.

Exported Top Level Bindings. Every Haskell module exports a set of identifiers. Trivial bindings of the form `let x = y in ...` with `x` being exported by the module may occur in Core code. In this case the transformation must not delete the trivial binding.

If the identifier `y` is not exported, then it is feasible to delete the binding and replace every occurrence of `y` by `x` in the whole module instead. However, because this case rarely occurs, we refrained from implementing this extension.

Cost Centres. Finally, programs that are compiled for profiling are annotated with cost centres. We suppressed them when giving the syntax of Core in Section 2. 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 [SanPey95] 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.

5 Measurement of the Effects of the Transformation

We measured a version of the transformation that uses all four reduction rules, but eliminates a WHNF only if it is the right hand side of a `let` binding and it is replaced by a variable. Eliminated expressions are not restricted to safe types.

We observed in Section 3 that the applicability of CSE is increased by various `let` and `case` floating transformations and by strictness analysis. Hence we apply CSE after all transformations that are normally turned on by the `-O2` option. Because transformation rule 3 opens new possibilities for optimisations, we repeat GHC's simplifier pass afterwards. As standard for comparison we use the same sequence of optimisations without CSE. The standard prelude was also compiled with the respective optimisations.

The objects of our comparison are the example of Section 3.2, the Haskell standard library hslib which contains the prelude, and nine programs from the Glasgow nofib test suite, version 1.4 [Par93]. 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 gives the number of recognised and eliminated common subexpressions as reported by our transformation. The second row gives the number of lines of each program. The recognised subexpressions are divided into WHNFs and non-WHNFs. Only those WHNFs that appear as right hand sides of `let` bindings are eliminated. Of the number of all recognised non-WHNFs the number of pure type applications is given in an additional row. The number of eliminated common subexpressions is the sum of the number of all non-WHNFs and those WHNFs that are right hand sides of `let` bindings. Finally the number of `let` bindings that are eliminated by rule 4 is given. The numbers behind a slash denote the number of common subexpressions that were recognised by the two `case` rules 2 and 3. All other common subexpressions were recognised by rule 1.

		recognised common subexpr.				eliminated	
		WHNFs		non-WHNFs			
program	lines	let rhs	others	all	type appl.	subexpr.	let bindings
example	2	0	0	1 / 1	0	4	4
hslib	10738	95	320	57 / 19	11	171	139
compress	320	0	0	0	0	0	0
fulsom	1357	16	133	12	8	28	19
gamteb	718	8	3	0/7	0	15	3
grep	356	7	91	17	7	24	22
lift	2033	4	23	31	28	35	30
pic	544	1	18	5	0	6	5
prolog	539	1	16	5	0	6	6
reptile	1522	10	21	70	54	86	32
rsa	74	0	0	4	0	4	4

Table 1. Recognised and eliminated common subexpressions

Table 2 shows the measured effects of the CSE on costs, that is runtime, the total amount of heap allocated, maximal heap residency, and code size. For the time we took the geometric means of five 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 mark costs that were not measured (hslib) or could not be measured because the runtime was too short.

program	time	total heap alloc.	max. heap residency	code size
example	**-26 %**	**-33.4 %**	+0.001 %	-0.3 %
hslib	–	–	–	-3.4 %
fulsom	+0.3 %	-0 %	-0.01 %	+3.9 %
gamteb	**+1.4 %**	+0.02 %	-0.3 %	+3.6 %
grep	–	**-3.1 %**	–	+1.9 %
lift	0 %	-0.09 %	-0.03 %	+1.6 %
pic	+0.5 %	+0.4 %	-0.01 %	+4.1 %
prolog	0 %	-0.3 %	**+6 %**	+0.4 %
reptile	**-1.7 %**	-0.2 %	-0.02 %	+1.7 %
rsa	**-1.7 %**	-0.24 %	**-2.5 %**	+1.4 %

Table 2. Effect on costs

The number of eliminated subexpressions is distributed very unevenly between the programs and has only a very weak relationship with the size of a program. However, all programs have disappointingly few common subexpressions with respect to their size. This limited applicability of CSE becomes even more apparent, when we subtract the number of recognised common subexpressions that are pure type applications from the number of eliminated subexpressions. All type information is dropped by GHC during code generation and hence the elimination of type applications has no influence on the generated code. Table 1 also shows that the `case` transformation rules 2 and 3 were hardly applied. Therefore we did not make measurements of a variation of the transformation that avoids "space leaks" as discussed in Section 3.5 by eliminating only expressions of save type by rules 2 and 3. It would hardly eliminate any common subexpression.

Table 2 proves that our example program from Section 3.2 profits considerably from the transformation. Both runtime and total heap allocation are reduced by the expected one third. However, the effect on the real-world programs of the nofib suite is not that advantageous. On the one hand, the runtime of `reptile` and `rsa` is reduced. In particular, `rsa` proves that only few eliminated subexpressions can have an effect. Even heap residency is reduced. Grep shows a reduction of total heap allocation. On the other hand, the transformation seems to have introduced a "space leak" into `prolog`. Furthermore, the runtime of `gamteb` increased. The latter proves that our transformation is not safe as intended and requires further analysis. The slight increase of total heap allocation of some programs can be explained by the effect of the transformation on the size of heap closures. The transformation cannot increase the number of closures allocated on the heap but may both increase and decrease the number of free variables of subexpressions and thus increase and decrease the size of heap closures. The slight increase of code size indicates that the mapping of Core expressions into code is not straightforward.

6 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.

We claim that CSE cannot be expected to be of equal importance for lazy functional languages as for imperative languages. 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. We suppose that the lack of common subexpressions is also the reason for Appel's disappointing results of CSE. A programming style that uses many abstract data types may 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.

Because of the limited number of common subexpressions we do not believe that it is worth to develop a transformation that employs complex analysis techniques that ascertain that repeated expressions are actually evaluated repeatedly and that avoid space leaks while retaining the shareability of complex data structures.

The current implementation of CSE is not yet fit to be used in practise. It produces a modest improvement of runtime and space behaviour of some programs, but it also has a negative effect in a few cases. To avoid "space leaks" a pragmatic compromise between safety and usability of the transformation could be to restrict eliminated subexpressions to safe types but to still use transformation rule 1. The applicability of the transformation rules could be increased by permitting the transformation of independent `let` and `case` expressions as discussed in Section 3.3 and by implementing cross-module CSE, that is enable the replacement of expressions by variables defined in imported modules. The conditions for the application of the transformation rules could make more use of the context of common subexpressions than only handling the right hand sides of `let` bindings specially. Finally the transformation could be used for reducing code size by eliminating large common WHNFs that were introduced by inlining but did not enable other transformations.

The development of CSE in Section 3 demonstrates the importance of close interaction of transformations. Several existing optimisations increase the opportunities for applying CSE. Because the effect of CSE depends highly on the operational semantics of Core it is unfortunately difficult to transfer the discussion of Section 3 to another implementation of a lazy functional programming language. In fact, Section 4.3 proves that a 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.

Finally we note that despite its long history CSE for imperative languages is still an active research area. The authors of [SKR90, KRS94] show that it is an instance of a more general transformation: code motion. Code motion moves every expression as far to the beginning of the program as possible to minimise the number of computations in the program. Thus it subsumes CSE, partial redundancy elimination, and loop invariant code motion. The latter is known as full laziness transformation in the context of functional languages. The authors define the notion of safeness: the transformation should not introduce the computation of a new value on any path. They also avoid any unnecessary code motion to minimise the lifetime of the temporary variables. The authors define an optimality criterion and give a complex algorithm that is safe and both computationally and lifetime optimal. The large number of expression floating transformations in GHC motivate the development of a similar general theory of code motion for imperative languages.

Acknowledgement

The author thanks all members of the program transformation group at Aachen for numerous discussions and Simon Peyton Jones for answering many questions about GHC.

References

[ASU86] A. Aho, R. Sethi, J. Ullman: *Compilers: Principles, Techniques and Tools*; Addison-Wesley, 1986.

[App92] Andrew W Appel: *Compiling with Continuations*; Cambridge University Press, 1992.

[GHC] *The Glasgow Haskell compiler*; `http://www.dcs.gla.ac.uk/fp/software/ghc/`.

[KRS94] Jens Knoop, Oliver Rüthing, and Bernhard Steffen: *Optimal Code Motion: Theory and Practice* ACM Transactions on Programming Languages and Systems, Vol. 16, No. 4, 1994, 1117–1155.

[MTW95] 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.

[Par93] Will Partain: *The nofib benchmark suite of Haskell programs*; part of the nofib distribution; `http://www.dcs.gla.ac.uk/fp/software/ghc/`.

[Par94] 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.

[Pey87] Simon L Peyton Jones: *The Implementation of Functional Programming Languages* Prentice-Hall, 1987.

[PeyLau91] 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.

[Pey92] Simon L Peyton Jones: *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine*; J. Functional Programming, **2** (2):127–202, 1992.

[PeySan97] Simon L Peyton Jones and André L M Santos: *A transformation-based optimiser for Haskell*; submitted to Science of Computer Programming, 1997.

[SanPey95] 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.

[San95] André L M Santos: *Compilation by transformation in non-strict functional languages*; PhD Thesis, University of Glasgow, July 1995.

[SKR90] Bernhard Steffen, Jens Knoop, Oliver Rüthing: *The Value Flow Graph: A program Representation for Optimal Program Transformations*; ESOP'90, LNCS 432, 1990, 389–405.