Common Subexpression Elimination in a Lazy Functional Language

Olaf Chitil

RWTH Aachen

Common Subexpression Elimination

- Purpose: reduce run-time through avoiding repetition of same computation
- For imperative languages: standard
  - ▷ flow graph of basic blocks of 3-address code
  - ▷ data-flow analysis

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

- For strict functional languages: [Appel 92]
  - ▷ continuation passing style code
  - ▷ similar to imperative languages

Lazy Functional Languages: Advantages

- referential transparency $\Rightarrow$ simple identification of repeated computation
  - ▷ no destructive update
  - ▷ no side effect
- arbitrary expressions, comprising non-primitive functions and data structures

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

Lazy Functional Languages: Disadvantages

- no explicit control flow
  - are all occurrences of a repeated expression evaluated?
- no static space allocation
  - heap residency may be increased or decreased considerably

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

Our Approach

- simple implementation without complex flow analysis
  - ▷ conservative elimination of common subexpressions

- added to the Glasgow Haskell Compiler
  - ▷ test effectiveness on real-world Haskell programs
  - ▷ easy to extend with new transformations

The Intermediate Language Core

- 2^{nd} order λ-calculus + `let(rec)` + `case`
- arguments of functions must be variables or constants

```
app []       ys = ys
app (x:xs') ys = x : (app xs' ys)
```

```
letrec app = \xs -> \ys -> case xs of
                              []     -> ys
                              x:xs' -> let zs = app xs' ys in
                                           x:zs
```

- a `let` and only a `let` allocates a closure
- a `case` and only a `case` triggers evaluation

Transformation Rules

- ~~$e'[e,e] \quad \leadsto \quad$ `let` x `=` e `in` $e'[x,x]$~~
  - ▷ low chances that both e are evaluated
  - ▷ new closure with long lifetime allocated
- compare only with named, syntactically dominating expressions

1. `let` x `=` e `in` $e'[e] \quad \leadsto \quad$ `let` x `=` e `in` $e'[x]$

   ```
   let x = 6 * 7 in fib (6 * 7)
   ```

2. `case` e `of` $\{\ldots$ `;` x `->` $e'[e]$`;` $\ldots\} \quad \leadsto \quad$ `case` e `of` $\{\ldots$ `;` x `->` $e'[x]$`;` $\ldots\}$

   ```
   case 6 * 7 of x -> fib (6 * 7)
   ```

3. `case` e `of` $\{\ldots$`;` $C\ x_1 \ldots x_n$ `- >` $e'[e]$`;` $\ldots\}$
   $\leadsto \quad$ `case` e `of` $\{\ldots$`;` $C\ x_1 \ldots x_n$ `->` $e'[C\ x_1 \ldots x_n]$`;` $\ldots\}$

   ```
   case 6 * 7 of I# x# -> fib (6 * 7)
   ```

Are these rules very restrictive?

- arguments of functions are atomic $\Rightarrow$ most expressions are named

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

in Core:

```
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 sum l3 in
      s + p
```

Existing Scope Increasing Transformations

- float `let` from `let`

$$\begin{array}{l}\texttt{let}\ x\ \texttt{=}\ (\ \texttt{let}\ y\ \texttt{=}\ e_y\ \texttt{in}\ e_x\ )\ \texttt{in} \\ \quad e\end{array} \rightsquigarrow \begin{array}{l}\texttt{let}\ y\ \texttt{=}\ e_y\ \texttt{in} \\ \quad \texttt{let}\ x\ \texttt{=}\ e_x\ \texttt{in} \\ \qquad e\end{array}$$

- float `case` from `let`
- float `let` from `case`
- float `case` from `case`
- float `let` from abstraction (full laziness)

Example

```
sum [1..1000] + sum [-1000..1] + sum [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# s -> let l3 = [1..1000] in
                                    case (sum l3) of
                                     I# p -> case (s +# p) of
                                               q -> I# q
```

- need existing transformation rule

4. `let` x `=` y `in` e $\quad\leadsto\quad$ $e[y/x]$

Conditions for Application of the Transformation Rules

- CSE is inverse of inlining

- Run-time
  - introduction of indirections
  - only sharing of non-WHNF may save evaluation time
  - elimination of `let` right-hand-side leads to elimination of `let`

- Total heap allocation
  - decreased because of sharing and deleted `let`s

Avoidance of Space Leaks

- eliminating non-WHNFs may increase heap residency
- restrict to non recursive types without recursive type components [Santos 95]
- however, a variable with value 1000 may be represented by 1001 thunks for `0 + 1 + 1 + 1 + ... + 1`
- hence
  - ▷ eliminate WHNFs
  - ▷ eliminate non-WHNFs of save type only by rules 2 and 3
  - ▷ safe types = `Bool`, `Char`, `Int`, `Float`, ... (`Bool#`, `Char#`, ...)

Integration of Rule 4

- let x = y in e $\rightsquigarrow$ $e[y/x]$
  has to be integrated for identification of common subexpressions whose components are not eliminated

```
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
```

Measurements

- replace WHNFs that are `let` rhs and all non-WHNFs
- all optimisations + CSE / all optimisations

program	lines	elim.	time	total alloc.	max. residency	code
sum	2	2 (1)	-26 %	-33.4 %	+0.001 %	-0.3 %
hslib	10738	160 (19)	–	–	–	-3.4 %
compress	320	0	–	–	–	–
fulsom	1357	20	+0.3 %	-0 %	-0.01 %	+3.9 %
gamteb	718	15 (7)	+1.4 %	+0.02 %	-0.3 %	+3.6 %
grep	356	17	–	-3.1 %	–	+1.9 %
lift	2033	7	0 %	-0.09 %	-0.03 %	+1.6 %
pic	544	6	+0.5 %	+0.4 %	-0.01 %	+4.1 %
prolog	539	6	0 %	-0.3 %	+6 %	+0.4 %
reptile	1522	32	-1.7 %	-0.2 %	-0.02 %	+1.7 %
rsa	74	4	-1.7 %	-0.24 %	-2.5 %	+1.4 %

Conclusion

- importance of interaction with other transformations
  - ▷ subsumes other transformations (constructor reuse, case elimination)
  - ▷ but rule 4 has to be incorporated
- transformation needs to be improved
  - ▷ reasons for increased run-time / code size?
  - ▷ heuristics for avoidance of space leaks?
  - ▷ how many common subexpressions are named, syntactically dominating?
- CSE of limited importance for (lazy) functional languages
  - ▷ programmers avoid common subexpressions
  - ▷ compilers for functional language introduce few common subexpressions