CAL Global Optimizer

Contributors: Greg McClement
Last revised: August 24, 2007

Contents

1 Overview

This document describes the optimizations that the CAL global optimizer written in CAL performs. The optimizations are designed to be as atomic as possible in order to eliminate redundancy and help ensure that the transformations are logically sound.

1.1 Document layout

The document contains a list of all the transformations performed by the optimizer. Many of the transformation are quick simple but when used in succession can produce complex transformations of expressions.

1.2 Call for feedback

This document aims to be as complete and easy-to-use as possible. Any feedback you might have to help improve it would be very welcome. Please send any comments, suggestions, or questions to the CAL Language Discussion forum on Google Groups (http://groups.google.com/group/cal_language).

2 Notation

This section describes the notation used in the rest of the document. Knowledge of CAL syntax is assumed and so is not documented here.

Assume, `e` is a CAL expression, `v` is a CAL variable and `x` is a CAL expression then `e[x/v]` means replace all occurrences of the variable `v` in `e` with `x`. Variables in e are renamed to avoid name capture where appropriate.

3 Transformations

This section describes each of the transformations that the optimizer applies. The description of each transformation is start on a new page. The description includes the following sections, transformations, example, implementation and references.

The transformation section provides a general schema of how the optimizer will modify an expression in order to perform the optimization. The example section provides at least on example of the transformation works. The implementation section contains the name of the function implementing the transformation. The references section lists the documents that the transformation was derived from.

3.1 Inlining

This transformation replaces a symbol in a given expression with the expression that defines the symbol. There are a large number of considerations that are made before performing the inlining. Expressions cannot always be inlined because this can reduce program efficiency or change program correctness because the expression might have side effect. All of the constraints are not described here. For further information refer to the implementation.

3.1.1 Transformation

Input

let v = ev in e

Output

let v = ev in e[ev/v]

This transformation is not applied for all input expressions that match the input form. There are a number of constraints some of which are:

1. ev is a literal or completely calculated at compile time.
2. ev is not recursive.
3. v is used only once in e or v is a lambda expression. There is a special case for 'case' expressions where v can occur at most once in each case alternative and still be considered to occur only once in the expression.
4. v is never inlined inside a lambda expression unless v is a lambda expression.

3.1.2 Example

Input

```
let x = 1 in x + 1
```

Output

```
let x = 1 in 1 + 1
```


3.1.3 Implementation

transform_3_2_2

3.1.4 Reference

[1]

3.2 *Case float from App*

The purpose of this transformation is to change the code in order to allow other transformations to apply. For this transformation, an case expression is applied to an argument. The argument is then embedded in the case alternatives. This provides a change that the case alternative expressions can be optimized.

3.2.1 Transformation

Input

(case ec of alt1 -> e1; alt2 -> e2; ...) v

Output

case ec of alt1 -> e1 v; alt2 -> e2 v; ...

3.2.2 Example

Input

```
(case x of
1 -> add;
2 -> subtract;) operand
```

Output

```
case x of
1 -> add operand;
2 -> subtract operand;
```

3.2.3 Implementation

transform_3_5_1

3.2.4 Reference

[1]

3.3 Embed the outer case into the inner case

This transformation takes alternatives from an outer case expression and embeds them in an inner one. Before the optimization a return value can be constructed in the inner case and then immediately analyzed by the other case expression. After the optimization, there is opportunity to eliminate the construction of the object in the inner case expression after the application of other transformations.

3.3.1 Transformation

Input

```
case
        (
        case innerExpr of
        innerAlt1 -> innerAltExpr1;
        innerAlt2 -> innerAltExpr2;
        …
        )
of
        outerAlt1 -> outAltExpr1;
        outerAlt2 -> outAltExpr2:
        …
```

Output

```
case innerExpr of
innerAlt1 ->
        case innerAltExpr1 of
        outerAlt1 -> outerAltExpr1;
        outerAlt2 -> outerAltExpr2;
        …
innerAlt2 ->
        case innerAltExpr2 of
        outerAlt1 -> outerAltExpr1;
        outerAlt2 -> outerAltExpr2;
        …
…
```

3.3.2 Example

Input


```
test7 start end =
    case
        case start > end of
            True -> [];
            False -> [start];
        of
    [] -> True;
    listHead : listTail -> False;
    ;
```

Output

```
test7 start end =
    case start > end of
    True ->
      case [] of
      [] -> True;
      listHead : listTail -> False;
    False ->
      case [start] of
      [] -> True;
      listHead : listTail -> False;
```

3.3.3 Implementation

transform_3_5_2

3.3.4 Reference

[1]

3.4 Simplify Case

This function exists to deal with trivial expressions that are made by the optimizer during simplification.

3.4.1 Transformation

Input

case expr of True -> True; False -> False;

Output

Expr

3.4.2 Example

Input

```
case x of True -> True; False -> False
```

Output

```
x
```

3.4.3 Implementation

transform_simplifyCase

3.5 *Lift Lets*

Various transformations for removing the let definitions out of the way so other transformations can happen.

3.5.1 Transformation

Case 1

Input

(e1 (let v=d in e2))

Output

(let v=d in (e1 e2)) as long as v is not free in e1.

Case 2

Input

((let v=d in e1) e2)

Output

(let v=d in (e1 e2))

3.5.2 Examples

Case 1

Input

```
negate (let r = expr1 in (r + r)
```

Output

```
let r = expr1 in negate (r+r)
```

Case 2

Input

```
(let v = t+t; in add v) 23
```

Output

```
let
      v = t + t;
in
      add v 23;
```

3.5.3 Implementation

transform_3_4_2

3.5.4 Example

3.6 Redundant Cases

This transformation removes a switch when the result of the switch is already known from the context.

3.6.1 Transformation

Input

```
case expr of
DC1 … ->
        case expr of
        DC1 … -> expr1
        …
…
```

Output

```
case expr of
DC1 … -> expr1;
…
```

3.6.2 Example

Input

```
test74 v =
    case v of
    Test74_Data_Value1 {} ->
        case v of
        Test74_Data_Value1 {} ->
            True;
        ;
    ;
```

Output

```
answer74 :: Test74_Data -> Boolean;
answer74 v =
    case v of
    Test74_Data_Value1 {} ->
        True;
    ;
```


3.6.3 Implementation

transform_redundantCases

3.7 Evaluate Case

In certain cases the data constructor of a switch expression is already know. This allows the optimizer to select the alternative without needing the actually evaluate the expression. This one has too many facets to demonstrate the general pattern so there will be only a number of examples.

3.7.1 Examples

Example 1

Input

```
data Test22_Whatever =
    Test22_Thing1 age::!Int |
    Test22_Thing2 age::Int;

test22 x =
    case Test22_Thing1 x of
    Test22_Thing1 {age} -> age;
    Test22_Thing2 _ -> 22;
    ;
```

Output

```
answer22 :: Int -> Int;
answer22 x = x;
```

Example 2

Input

```
test79 :: Data_Test75 -> Boolean;
test79 value =
    case value of
    Value1 {} ->
        case value of
        Value1 {arg1} -> arg1 == 7;
        ;
    ;
```

Output

```
answer79 :: Data_Test75 -> Boolean;
answer79 value =
    case value of
    Value1 {arg1} -> arg1 == 7;
```



```
;
```

3.7.2 Implementation

transform_evaluateCase

3.8 *Let Floating From Case Scrutinee*

Lift let definitions higher up to open the expression up for other optimizations.

3.8.1 Transformation

Input

case (let v = ev in e) of alts

Output

let v = ev in (case e of alts) where v is not free in alts.

3.8.2 Example

Input

```
case (let r = t+t; in D1 r) of
D1 {} -> True;
```

Output

```
let
      r = t + t
in
      case D1 r of
      D1 {} -> True;
```

3.8.3 Implementation

transform_letFloatingFromCaseScrutinee

3.8.4 References

[1]

3.9 Evaluate Record Selection

This transformation evaluates record expressions for the case that the data constructure for the expression is known. This case arise through other transformations performed by the optimized.

3.9.1 Tranformation

Input

(DC expr1 expr2 … exprK).DC.argI

Output

exprI

3.9.2 Example

Input

```
(Just x).Just.value
```

Output

```
x
```

3.9.3 Implementation

transform_evaluateRecordSelection

3.10 Evaluate Arithmetic Expressions

This transformations evaluates arithmetic expression with the operands are know constants. Expressions involving add, subtract, multiple, divide and all the comparison operators can be evaluated. The transformation also applied some identities involving zero such as "x+0" or "x*0".

3.10.1 Transformation

Evaluates arithmetic expressions with constants such as "2+3", "5*7", or "7 < 5". This can also handle identities involving zero such as "x + 0", "x * 0" or "0 / x".

3.10.2 Examples

Example One

Input

```
2+3
```

Output

```
5
```

Example Two

Input

```
x + 0
```

Output

```
x
```

3.10.3 Implementation

transform_evaluateArithmeticExpressions

3.11 Canonize Seq's

The seq'ed expression can come appear in a number of structurally different forms. This transformation puts all canonizes the structure to make pattern matching easier for the rest of the optimizer.

3.11.1 Transformation

Case 1

Input

((seq x y) z)

Output

(seq x (y z))

Case 2

Input

(seq (seq x y) z)

Output

(seq x (seq y z))

3.11.2 Example

Input

```
x `seq` y `seq` x
```

Output

```
(seq x (seq y x))
```

3.11.3 Implementation

transform_canonizeSeq

3.12 Simplify Seq's

Various transformation to remove seq's when they are determined to be not necessary from the context.

3.12.1 Transformations

x `seq` x => x

(\x1 -> (\x2 -> ... (prelude.seq x1 y)) => (\!x1 -> (\x2 -> ... y)) iff the intervening lambda vars are all not strict

(x `seq` case x of ...) => (case x of ...).

(seq x (seq y ((f x) y))) where f :: !a -> !b -> ... => (f x) y

y `seq` z => z iff y is known to be WHNF

(\!x -> case x of ...) => (\x -> case x of ...)

3.12.2 Examples

Example One

Input

```
expressionType `seq` (case expressionType of  App {} -> True; _ -
> False);
```

Output

```
case expressionType of  App {} -> True; _ -> False;
```

Example Two

Input

```
case z of
DC1 {} ->
      z `seq` helper x;
```


Output

```
case z of
DC1 {} -> helper x;
```

3.12.3 Implementation

transform_simplifySeq

3.13 Beta Reduction

This transformation replaces a lambda variable with the value of the lambda variable and removes the argument from the lambda expression.

Transformation

Input

(\v -> e) x

Output

e [x/v]

3.13.1 Example

Input

```
(\x -> x + 1) 2
```

Output

```
(2+1)
```

3.13.2 Implementation

transform_3_1

3.13.3 Reference

[1]

3.14 Dead Code Removal

This transformation removes the definition of a let variable with the variable is not referenced in the expression anymore.

3.14.1 Transformation

Input

let v = ev in e   (where v is not found free in e)

Output

e

3.14.2 Example

Input

```
let x = x + 1 in 1 + 2 + c
```

Output

```
1 + 2 + c
```

3.14.3 Implementation

transform_3_2_1

3.14.4 Reference

[1]

3.15 Fusion

Fusion is the process of creating a new function that is based on an expression involving two other functions. Definitions of the other functions are combined in the new function definition. The new function no longer makes references to the original two functions. This transformation is applied here only for the case of recursive functions. This can be thought of as a special case of inlining.

3.15.1 Transformation

Input

    (f1 e1 e2 … ek (f2 ek+1 ek+2 … ek+m) ek+m+1 … ek+m+n

Output

    let
        f1$f2 = eBody''
    in
        f1$f2 e1 e2 … ek ek+1 ek+2 … ek+m ek+m+1 … ek+m+n

The optimizer looks for patterns as shown in the input pattern. If found the optimizer makes an expression of the form

    eBody: f1 a1 a2 … ak (f2 ak+1 ak+2 … ak+m) ak+m+1 … ak+m+n

a<n> is a newly created variable name. The expression eBody is then optimized. For all occurrences of the pattern eBody in eBody' a call to f1$f2 is used instead. This produces the expression eBody''. If there are no calls to f1 or f2 in the expression eBody'' then the fusion is successful and the specified output is used as the new expression otherwise there is no change to the expression and the output equals the input.

Notes

1. f1 and f2 are a recursive functions.
2. a stands for argument that is a single variable name.
3. e stands for expression.
4. f stands for functor.

3.15.2 Example

Input

andList (map id [True, False])

1. Make an generalized expression

```
andList (map a1 a2).
```

2. Optimize the expression to produce

```
  case a1 of
    []     -> True;
    listHead : listTail -> f listHead  && andList (map f
listTail);
```

3. Replace andList(map f listTail) with recursive call

```
    case a1 of
    []     -> True;
    listHead : listTail -> f listHead  && andList$map f
listTail;
```

3.15.3 Implementation

performFusion

3.16 Specialization

This optimization involves creating special versions of functions where the variables are replaced with expression from the caller. This is used when the function is recursive and arguments are passed unchanged during the recursive call.

3.16.1 Transform

Input

Definition:

let
f a1 a2 … ak = … (f a1 a2′ … ai′ … ak) …
in
… (f e1 e2 … ei … ek)

Output

f′ a2 …ai = … f a2′ … ai′ [e1/a1]…[ek/ak]

f′ e2 … ei

3.16.2 Example

Input

```
andList$Map f list =
          case list of
          []     -> True;
          listHead : listTail -> f listHead  && andList$map f
      listTail;

andList$Map id list
```

Output:

```
andList$Map' list
          case list of
          []     -> True;
          listHead : listTail -> id listHead  && andList$map'
      listTail;
```


At this point other optimization can be performed such as transforming (id listHead) to (listHead).

3.16.3 Implementation

transform_specialization

4 References

[1] Santos [Sept 1995], "Compilation by transformation in non-strict functional languages.", PhD thesis, Department of Computing Science, Glasgow University.

http://citeseer.ist.psu.edu/cache/papers/cs/3175/http:zSzzSzwww.di.ufpe.brzSz~almszSzpszSzthesis.pdf/santos95compilation.pdf

[2] Santos [Jan 1996], "Compiling Haskell by program transformation: a report from the trenches", Department of Computing Science, Glasgow University.

http://citeseer.ist.psu.edu/cache/papers/cs/968/http:zSzzSzresearch.microsoft.comzSz~simonpjzSzPaperszSzcomp-by-trans.pdf/peytonjones96compiling.pdf

