COMP 512
Rice University
Spring 2015

Algebraic Reassociation, Revisited

Moving Beyond Rank-Ordering Schemes

K. Cooper, J. Eckhardt, & K. Kennedy, "Redundancy Elimination Revisited", PACT 08, pages 12-21.

Citation numbers refer to entries in the EaC2e bibliography.

Background

Last Lecture

- Looked at Briggs' technique to reorder expressions for **PRE/LCM**
- Three step algorithm
  1. Compute a rank for each expression
  2. Forward substitute to flatten & broaden expression trees
  3. Sort operands, where allowed, into rank order
- Because **PRE/LCM** needs a specific name space, the authors then rebuilt the name space using **AWZ** before applying **PRE/LCM**.†

Expressions are reordered to improve loop-invariant code motion

⇒ Chosen order provides limited help with exposing redundancy or, for that matter, any other property of the code

† The name space that they built both encoded value identity so that provably equal values had the same name, and handled a subtle correctness issue in PRE.

Motivating Example

Consider value numbering applied to the following expression:

```
(+  (* x  (+ a c i h b))
    (* y  (+ a b i d c e f g a c i b))
    (* z  (+ f d e g d e f g)) )
```

Eckhardt named this example “Diabolic”

Imagine applying local value numbering (LVN) to Diabolic

- **LVN** would start hashing
  - ♦ Might use (+ a c), then (+ (+ a c) i), then (+ (+ (+ a c) i) h), then …
  - ♦ Might use (+ a c i h b)
  - ♦ Neither approach finds redundancies
- The expression must be rearranged to expose redundancies

Motivating Example

Consider value numbering applied to the following expression:

(+ (* x (+ a c i h b))
(* y (+ a b i d c e f g a c i b))
(* z (+ f d e g d e f g)))

Eckhardt named this example “Diabolic”

Careful examination reveals two major redundant subexpressions

- (+ a b c i) occurs three times
- (+ d e f g) occurs three times

We would like to rewrite the expression as

(+ (* x (+ (+ a b c i) h))
(* y (+ (+ a b c i) (+ a b c i) (+ d e f g)))
(* z (+ (+ d e f g) (+ d e f g))))

In this form, any competent technique should find the redundancies

Relating Briggs' Technique to "Diabolic"

Some parts of the Briggs-Cooper algorithm are still relevant

- Need to **reorder operands**
- **Forward substitution** to build large, flat, n-ary expressions exposes more opportunities to reorder expressions
- The rank & sort paradigm is too weak to do well on "Diabolic"

As a researcher, how do you attack this problem?

- Work lots of examples
  - ♦ Jason sent me an example every day or two for about a month
  - ♦ Some from practice, some devised to elicit difficult points
- I worked the examples and sent them back
- We tried to extract common principles behind the solutions

Classic progression of harder questions & more obscure answers

Back to Diabolic

Diabolic highlights the problems with rank-ordering schemes

```
(+  (* x  (+ a c i h b))
    (* y  (+ a b i d c e f g a c i b))
    (* z  (+ f d e g d e f g)) )
```

Can you devise a ranking scheme that groups (+ a b c i) and (+ d e f g) ?

- Canonical order based on name (or some other attribute)?
  - ♦ Commutativity in local value numbering
- Ranking based on placement of definition?
  - ♦ Briggs' approach for loop-invariant code motion
- We tried a fair number of rank-order schemes
- Each rank-order scheme that we devised had a bad counter example

Affinity

After careful thought, we arrived at the notion of "affinity"

- (a,b) have an affinity if they occur in the same term
  - ♦ Term is defined as an eligible operator and all of its operands in the flattened tree (after forward substitution & flattening)
- If distinct instances of (a,b) occur *k* times, we assign (a,b) an affinity of *k*

```
(+  (* x  (+ a c i h b))
    (* y  (+ a b i d c e f g a c i b))
    (* z  (+ f d e g d e f g)) )
```

Diabolic

> While the affinity matrix clearly captures some aspect of the property that we want, it is not obvious how to tease the information out of it.

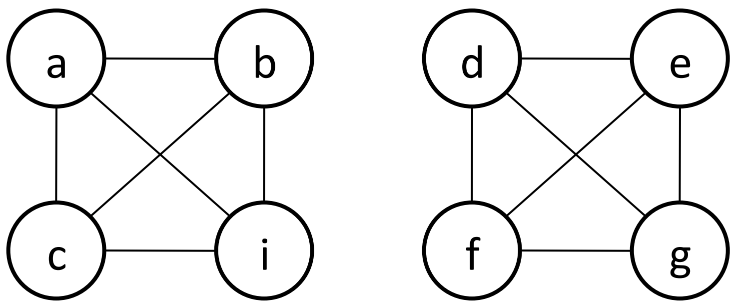
	a	b	c	d	e	f	g	h	i
a	–	3	3	1	1	1	1	1	3
b		–	3	1	1	1	1	1	3
c			–	1	1	1	1	1	3
d				–	3	3	3	1	1
e					–	3	3	1	1
f						–	3	1	1
g							–	1	1
h								–	1
i									–

Diabolic's Affinity Matrix

Affinity

We had the right information in the wrong form

- View the affinity matrix as defining a graph with weighted edges
- Edge weight of 2 or more defines a redundancy
- Build the graph, excluding trivial (non-redundant or weight one) edges



Affinity Graph, excluding trivial edges

The redundant subexpressions form cliques in the graph

- Maximal cliques yield the largest subexpressions
- Minimum edge weight in a clique indicates multiplicity in the code

Clique ≅ a subgraph where every pair of edges are adjacent.

Expressions with Nested Cliques

Expressions may have more complex structures

- Large terms with multiple occurrences
- Number of occurrences is more important than number of terms

Second Example

```
(+  (*  x  (+  a b c d a b))
    (*  y  (+  a b c d e f))
    (*  z  (+  a b g h)) )
```

It is easy to envision a scheme that requires an arithmetic tradeoff between expression size (# of operands) and multiplicity of the expression

- (+ a b c d) occurs twice
- (+ a b) occurs four times

There is an easier way to capture this tradeoff

Expressions with Nested Cliques

Second Example

(+ (* x (+ a b c d a b))

(* y (+ a b c d e f))

(* z (+ a b g h)))

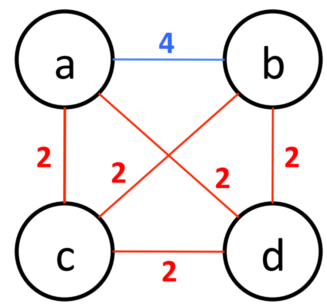
	a	b	c	d	e	f	g	h
a	–	4	2	2	1	1	1	1
b		–	2	2	1	1	1	1
c			–	2	1	1	1	1
d				–	1	1	1	1
e					–	1	1	1
f						–	1	1
g							–	1
h								–

Expressions with Nested Cliques

Second Example

(+ (* x (+ a b c d a b))
(* y (+ a b c d e f))
(* z (+ a b g h)))

	a	b	c	d	e	f	g	h
a	–	4	2	2	1	1	1	1
b		–	2	2	1	1	1	1
c			–	2	1	1	1	1
d				–	1	1	1	1
e					–	1	1	1
f						–	1	1
g							–	1
h								–



Affinity Graph, excluding trivial edges

If the implementation can find the nested clique, it will produce

(+ (* x (+ (+ (+ a b) c d) (+ a b)))
(* y (+ (+ (+ a b) c d) e f))
(* z (+ (+ a b) g h)))

The Algorithm (High-Level Sketch)

1. Build large subexpressions with forward substitution
2. Flatten those expressions to create large terms
3. Build the affinity graph for those terms, excluding trivial edges
4. From largest weight to two, find maximal cliques at each weight
5. Rewrite the code to place each clique in a distinct subterm

Maximal clique finding is **NP** Complete. Eckhardt used an **O**(n^2) greedy heuristic that did well, in practice. (*n* is # nodes.)

The Plan

Apply the algorithm

1. Build large subexpressions with forward substitution
2. Flatten those expressions to create large terms
3. Build the affinity graph for those terms, excluding trivial edges
4. From largest weight to two, find maximal cliques at each weight
5. Rewrite the code to place each clique in a distinct subterm

On Diabolic, this approach yields

```
(+  (* x  (+  (+ a b c i) h))
    (* y  (+  (+ a b c i) (+ a b c i) (+ d e f g)))
    (* z  (+  (+ d e f g) (+ d e f g))) )
```

We can perform this transformation as a prelude to value numbering

→ *Value numbering will "do the right thing"*

The Plan

Apply the algorithm

1. Build large subexpressions with forward substitution
2. Flatten those expressions to create large terms
3. Build the affinity graph for those terms, excluding trivial edges
4. From largest weight to two, find maximal cliques at each weight
5. Rewrite the code to place each clique in a distinct subterm

On our second example, this approach yields

```
(+  (*  x  (+  (+ (+  a b) c d) (+  a b)))
    (*  y  (+  (+ (+  a b) c d) e f))
    (*  z  (+  (+  a b) g h)) )
```

We can perform this transformation as a prelude to value numbering

→ *Value numbering will "do the right thing"*

Handling More Complex Subexpressions

Consider an expression such as:

(* a (cos b) c (cos b) a c)

(cos b) is a function call, but one with no side effects & therefore redundant

We want to find

(* (* a c (cos b)) (* a c (cos b)))

We need abstract names for the subexpressions

- (* a s_1 c s_2 a c), where s_i is a symbolic name
- Names should reflect values; same name implies same value
  - ♦ (cos b) = (cos b), so expression would be (* a s_1 c s_1 a c)

How can we construct these symbolic names?

- Classic answer is to use value numbering
- Leads to a circularity in the algorithm

Another example of Click's theory of combining optimizations?

Combining Value Numbering With Reassociation

To combine value numbering with reassociation, Eckhardt reasoned that they needed a common paradigm

- He reformulated **LVN** into a treewalk, rather than a linear sweep on the IR
  - ♦ Algorithm is reminiscent of **DAG** finding algorithm in Aho, Sethi, & Ullman
  - ♦ Visit the nodes in postorder & apply the **LVN** step at each operator
    - → *Postorder visits children before their parent*
- **LVN** step must recognize redundancy & assign value numbers
  - ♦ Use a hash table with keys from operation & its operands
  - ♦ Use arbitrary arity operations
  - ♦ Rewrite subtrees on the fly
- Finally, he reformulated it as a worklist algorithm
  - ♦ Algorithm makes one pass over the tree — no notion of a fixed point
  - ♦ I don't think that this aspect of the algorithm is fundamental
    - → *Could reformulate it in a deterministic and obvious order*

One More Point About LVN

For the "reassociation-enabled" LVN to work well, it needs large commutative expressions

- Need to perform forward substitution of expressions to their uses
  - ♦ Just as in [Briggs 94]
  - ♦ Incorporate Frailey's trick with unary minus
    - → ***x – y + z becomes x + (-y) + z***
- Need to flatten commutative operation trees from multiple operators to single operators
    - → ***x + (-y) + z becomes (+ x (- y) z)***
- Need a framework where forward substitution does not break subtle rules in the name space

Worklist Version of LVN

for each node, n
    if n is a leaf, add it to worklist
    if n has k children, set ready(n) to k

while (worklist is not empty)
    remove a node n with parent p from worklist

    if n is an leaf
        hash n and assign n a value number

    else
        hash n's operands
        construct a hash key from n & its operands' value numbers
        if key already has a value number v then
            mark n as equivalent to v and replace n with the node for v

    let p be n's parent
    decrement p's "ready counter"
    if p is ready, add it to worklist

This version of **LVN** retains its linear-time expected case behavior.

The worklist structure ensures that subexpressions are evaluated before parents — effectively, **RPO**

Adding Reassociation to LVN

for each node, n
  if n is a leaf, add it to worklist
  if n has k children, set ready(n) to k
deferred ← empty list

while (worklist is not empty)
  remove a node n with parent p from worklist

  if n is an leaf
    hash n and assign n a value number

  else
    hash n's operands
    construct a hash key from n & its operands' value numbers
    if key already has a value number v then
      mark n as equivalent to v and replace n with the node for v

  decrement p's "ready counter"
  if p is ready, add it to deferred

  if worklist is empty
    reorder the nodes on deferred
    worklist ← deferred
    deferred ← empty list

When all of *p*'s children have been processed, rearrange its operands before processing it

Combining Value Numbering with Reassociation

Algorithm was implemented in the Open 64 Compiler

- Implemented as part of *extended strength reduction*
  - ♦ Algorithm captures inter-iteration reuse under different names
  - ♦ Value numbering is the building block of that algorithm
- Experimental results on some performance-critical loop nests
  - ♦ Break out improvements due to different factors
  - ♦ Reassociation sometimes helps, sometimes does not
    - → *Cannot capture improvement when conditions are not present*
  - ♦ Reduces arithmetic operations (integer + and *)
  - ♦ Finds duplicate calls to intrinsic operations (cos(x))

Lessons

- "Rank & sort" works when the role of context is simple
- Complex context requires more complex choice of orders
- Can afford a more expensive technique than Briggs' preorder rank computation
- Will run into combinatorial explosions
  - ♦ Deal with them using effective heuristics

This slide is speculative and describes ideas on an open problem.

What About Orders for Other Transformations

Reassociation for strength reduction looks profitable

- Simple five point stencil

```
do i …
    do j …
        a(i,j) ← (a(i,j) + a(i-1,j) + a(i+1,j) + a(i,j-1) + a(i,j+1)) / 5
    end do
end do
```

Would like one or two reduced induction variables, not five

- Markstein, Markstein, & Zadeck suggest a sum of products form
  - ♦ Hope to reduce the product terms
  - ♦ Hope to implement the addition terms with address arithmetic
- Give induction variables large weights and redistribute to obtain 3 parts ?
  - ♦ Induction variable term, varying term, & constant term

On the readings page