11 High level optimisation 2

Most writers follow the division of optimsations at this level into three kinds:

- local optimisations within basic blocks of the flow graph;
- loop oriented optimisations;
- global optimisations apart form loops.

To begin with we can examine local optimisations.

Local optimisation

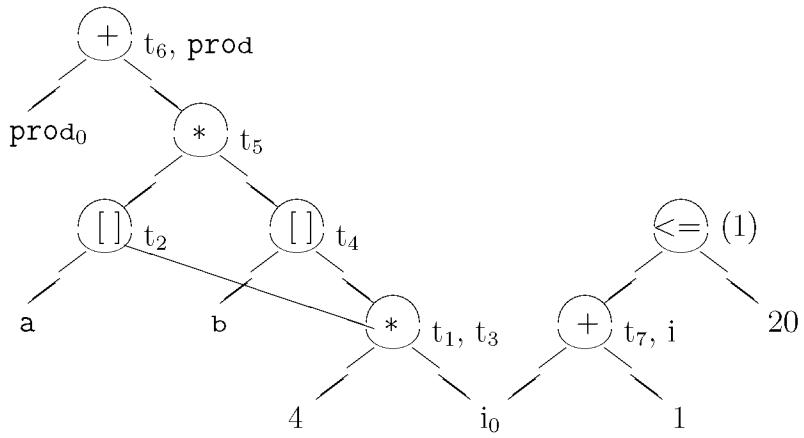
Some local optimisations are performed before generating tuples. This is particularly true of language dependent optimisations and constant folding. Language dependent optimisations often centre on removal of unnecessary runtime checking and improvement of storage allocation depending on range information on values. Scoping rules, such as Ada's restriction of the controlled variable to being local to a `for-loop`, can also allow more efficient storage allocation. This sort of improvement is too specific to particular languages to be dealt with here. Instead we will concentrate on language independent optimisations. These are based on dependency analysis in expression evaluation and on algebraic equivalence of replacement expressions. Directed Acyclic Graph (DAG) analysis of expressions

The identification of local dependencies within basic blocks uses the construction of a DAG for the code. This is similar in many ways to the examination of dependencies within the annotated parse tree, discussed under semantic analysis. It is also effectively the same as a triples based representation.

Consider the block B2 from our example, but with a very na"ive code generation scheme.

```
(1) T1 := 4 * i
(2) T2 := a[T1]
(3) T3 := 4 * i
(4) T4 := b[T3]
(5) T5 := T2 * T4
(6) T6 := prod + T5
(7) prod := T6
(8) T7 := i + 1
(9) i := T7
(10) if i <= 20 goto (1)
```

The corresponding DAG is as follows:



To construct a DAG the following steps are performed, assuming that each node is represented by a datastructure capable of holding left and right children links, a label for that node and a list of associated identifiers and constants. In a leaf the label is an identifier or constant; in an interior node it is an operator. The set of all identifiers currently associated with a node is maintained. The label of a leaf is one form of association, membership of the list of an interior node is another.

A function $NODE(IDENTIFIER)$ is assumed, which returns the most recently created node associated with $IDENTIFIER$ or none.

Three cases are identified for three address statements:

(a) `A := B` *op* `C`

(b) `A :=` *op* `B`

(c) `A := B`

`if I <= 20 goto (1)` and similar relational operators are treated as case (i) with undefined `A`.

1. If $NODE$(`B`) undefined create a leaf labelled `B`. Set $NODE$(`B`) to return this. For case (i) repeat this for `C`.

2. As appropriate:

   In case (i)

   > check for a node labelled *op* with **left child** = $NODE$(`B`), **right child** = $NODE$(`C`). If not create such a node. Let n be the node identified or created.

   In case (ii)

   > check for a node with **lone child** = B. If not create one. Let n be that node.

   In case (iii)

   > let n be $NODE$(`B`).

3. Add `A` to the list of identifiers attached to node n. Delete `A` from the list for $NODE$(`A`). Set $NODE$(`A`) to n.

Using this process the following information can be found:

(a) Common sub-expressions are found during step 2. These can be combined in many cases, by assigning their value to a temporary, rather than re-calculating each time.

(b) Identifiers and values used in the block are identified, as the label of any leaf node generated during the construction of the DAG.

(c) Statements generating values which persist at the end of the block are identified by noting those nodes n, where $NODE$(`A`) is still n at the end of the block.

Evaluation of the nodes of the DAG can be in any order which is a topological sort. This restricts evaluation of a node until any children which are interior nodes have been evaluated.

When a node is evaluated its value is assigned to one of its attached identifiers, preferably one whose value is still live at the end of the block. This is restricted by the need not to overwrite any value held in such a variable which is still live, i.e. needed within the evaluation of nodes remaining or still live at the end of the block. This reduces the number of assignments needed, especially those of the form `A := B`, since such assignments are only necessary to identifiers required outside the block.

Problems with dynamic memory locations

Arrays and pointers both allow identifiers to be specified where the actual location depends on runtime evaluations of expressions.Thus:

```
X := A[I]
A[J] := Y
Z := A[I]
```

might appear to contain the common sub-expression `A -[ I ]` in the first and third statements, but at runtime `J` might might equal `I`, making the following re-arrangement unsafe:

```
X := A[I]
Z := X
A[J] := Y
```

This is handled by killing all nodes labelled `=[]`, whose left argument is the array (say `A`), plus or minus a constant (including zero), when we process an assignment to `A`, if one of their descendants is `addr(A)`. This requires some additional information to keep track of this.

Pointers are similar, but more extensive in their effects. In the statement `*p := w`, `p` could map onto any location and so all nodes must be killed when a pointer is found. In fact, data flow analysis can be used to restrict the set of nodes needing to be killed, by finding the subset of all identifiers that `p` could refer to.

Procedure or function calls must also be regarded as killing all nodes, unless global data flow analysis can reduce the set of candidates. This is because the called procedure may set any value as a side effect, unless we can identify the restrictions on variables accessible by this procedure.

Optimisations local to blocks

Within a single block of a flow-graph many useful optimisations can be performed, some of which use information gathered during DAG construction. These optimisations are in the class of *structure preserving* transformations, since they do not change the expected order of execution of code significantly.

Common sub-expression elimination

The processing of DAGs reveals common sub-expressions. Where it is safe to do so, given the problems associated with dynamic memory access, these may be reduced to one evaluation and the result of each may be kept in a single location used by all subsequent references. At the end of the block it may be necessary to copy this value into other variables, if these are *live* at the end of the block. We will see how to determine this later.

Copy propagation

If there is a statement of the form `f := g`, copy propagation replaces uses of `f` by `g` wherever possible. Copy statements of this form often arise in elininating common sub-expressions by standard algorithms.

Thus the block:

```
x := t3
a[t2] := t5
a[t4] := x
goto B2
```

will become

```
x := t3
a[t2] := t5
a[t4] := t3
goto B2
```

In itself this is no improvement, but it may help us remove the assignment to `x` if all references to x with this value are eliminated. Again we will need data flow analysis to show such cases.

Algebraic equivalences

The other main form of local optimisation centres on understanding how to re-write sequences within expressions so as to maintain their mathematical meaning, but allowing more efficient implementation. Such optimisations are normally grouped under the title *algebraic identities.* The first of these is very commonly used and is usually called *strength reduction.* The other group of optimisations are more difficult to use generally.

Strength reduction

Strength reduction involves replacing certain occurences of costly runtime operations with equivalent, less costly operations. To make full use of strength reduction, the cost of actual machine instructions must be compared. For floating point multiplication, repeated addition

may be substituted. This depends on the relative efficiencies of the operations. However, some common strength reductions can safely be made at the stage of high level optimisation.

A trivial example is the replacement of multiplication by 2 with left shifting one place (only valid for positive or unsigned integers and multiplication by powers of 2). This can also work for division in the same restricted cases. For multiplication by non-powers of 2 repeated addition may be substituted in some cases. A very clever compiler might combine shifting and addition, so that, for instance, multiplication by 5 becomes left shift 2 places followed by addition of the original number. This sort of optimisation might be best left until final code generation or peephole optimisation.

More complex examples are scaling of array indices (effectively multiplying by a power of 2) and exponentiation being replaced by multiplication, e.g. `X ↑2` becomes `X * X`.

Sometimes repeated application of strength reduction may be possible, but this may really involve constant folding.

Other algebraic identities

One obvious use of algebraic properties is to identify commutative operators, such as addition and multiplication. These allow the identification of common sub-expressions to be extended, so that left and right can be reversed and still considered equivalent in step 2 of DAG construction.

Further benefits arise where strength reduction or rewriting of comparisons as subtraction followed by a test yield common expressions.

Associative properties can also be used, e.g.

```
A := B + C
E := C + D + B
```

could benefit from applying the associative and commutative properties of addition to recognise the common sub-expression `B + C`.