Optimization

(Jan 2010)

Example

Consider the following code fragment:

```
strcopy( const char* p, const q )
{
   unsigned int i = 0;
   while( p[i] )
   {
      q[i] = p[i];
      ++ i;
   }
}
```

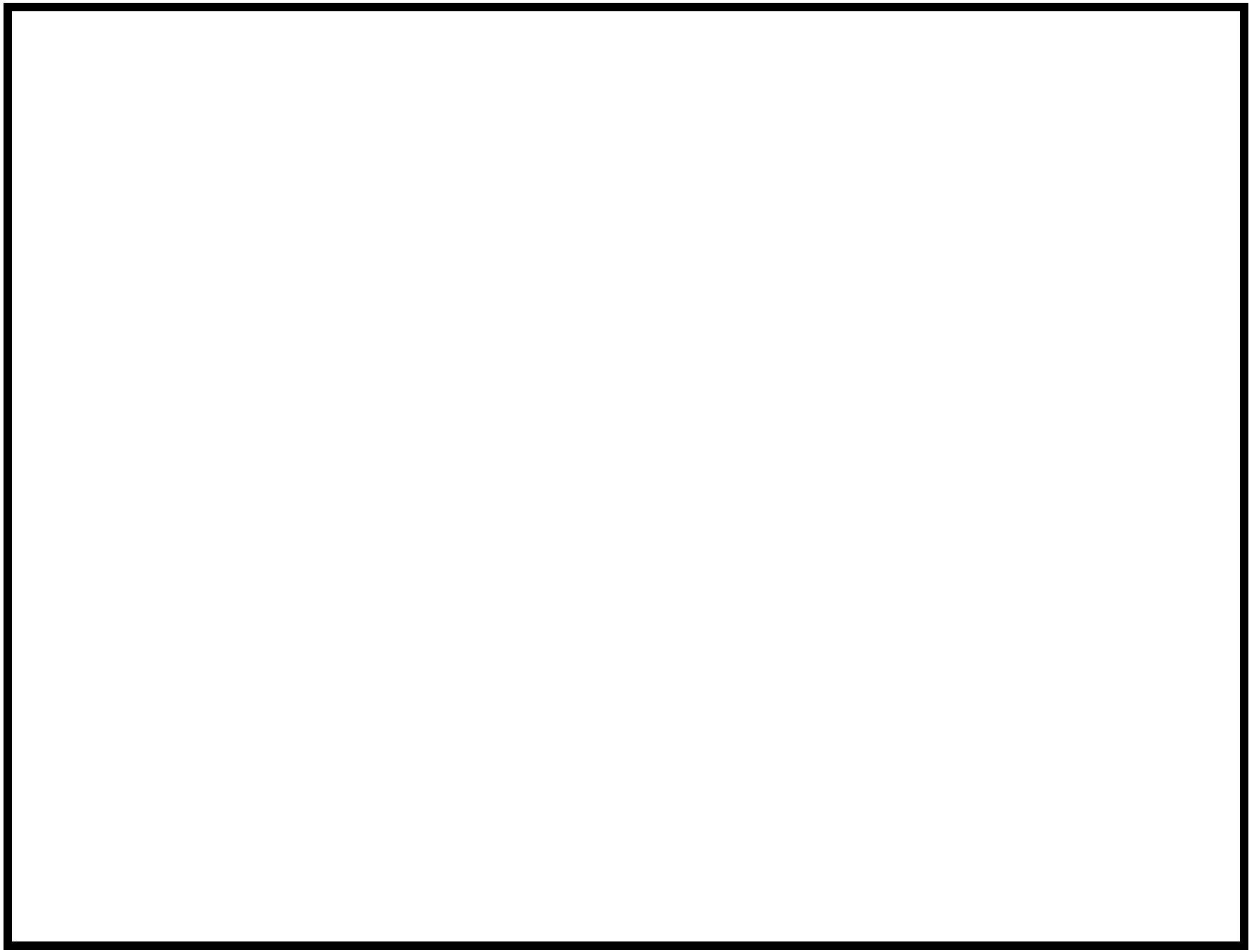
Optimization

```
strcpy( pointerto( const char ) p,
        pointerto( char ) q )
{
   create i array( sizeof::<unsigned int > ( ), char )
   init::< unsigned int > ( i, 0 );

loop:
   jumpfalse end
      equal::<char> (
         copy::<char> (
            add::< pointerto(char), unsigned int,
                   pointerto(char) >
               ( copy:: < pointerto(char) > ( q ),
                 copy:: < unsigned int > ( i )) ), 0 )
```

```
    assign< char > ( deref::<char> (
       add::< pointerto(char), unsigned int,
              pointerto(char) >
          ( copy:: < pointerto(char) > ( q ),
            copy:: < unsigned int > ( i )) ),
                     copy:: < char > (
       add::< pointerto(char), unsigned int,
              pointerto(char) >
          ( copy:: < pointerto(char) > ( q ),
            copy:: < unsigned int > ( i )) ) )

    assign< unsigned int > ( i,
       add::< unsigned int > ( copy::< unsigned int >,
                               1 ));
    goto loop;
end: }
```



Optimizations Applicable to Stringcopy

- **char** has size 1. This simplifies calculation of $p + i$ and $q + i$. One either has to detect this during generation of intermediate representation, or simplify later.
- $p[i]$ is calculated (and looked up) twice.
- If we would not be copying characters, but something with a size, we could reuse the multiplication.
- A very smart optimizer could remove the multiplications completely.
- Variable i can be put in a register.
- The label **end:** can be removed and replaced by the exit point. (Depends on machine)
- The whole procedure could be inlined.

Cost

Optimization means that one wants to reduce the cost of executing the program. There are several meaningful notions of cost:

1. Execution time of program.
2. Size of Program.
3. Energy Use of Program.

Nearly always, execution time is the most important goal.
(inlining, loop unfolding trade space for time.)

Redundant Expressions

Redundant expressions occur when expressions are recomputed.

Definition An expression E is redundant if it has already been computed on every path that leads to E.

Redundant Expressions (2)

$m := 2 \times y \times z;$

$n := 3 \times y \times z;$

$o := 2 \times y - z;$

can be replaced by

$k := 2 \times y;$

$m := k \times z;$

$n := 3 \times y \times z;$

$o := k - z.$

(Since $\times$ is left associative, $y \times z$ will not be automatically detected. One could also try detect that $3 \times y$ equals $k + y$.)

Redundant Expressions (3)

In the example

```
   unsigned int i = 0;
loop:
   if( *(p+i) == 0 ) goto end;

   *(q+i) = *(p+i);
   i = i + 1;
   goto loop;
end:
   return;
```

the second `*(p+i)` is redundant.

Redundant Expressions (4)

When is an expression redundant?

$a := *(p + i);$
$i := i + 1;$
$b := *(p + i)$ (obviously not)

$a := *(p + i);$
$*(p + i) := 44;$
$b := *(p + i);$ (obviously not, but one could reuse 44)

Redundant Expressions (5)

There exists a quite sophisticated field of automated theorem proving, but practical code is so big that only efficient (close to linear) algorithms have been used in practice: (but maybe this will change)

Instead, one builds a container of normalized available expressions. (Usually a hash map.)

Normalization

We will create a set of local variables $\mathcal{X}$ and a set of rewrite rules $\mathcal{R}$, which maps expressions to local variables. Initialize $\mathcal{X} := \{\ \}$.

First we give an algorithm for normalizing expressions. If necessary, the algorithm extends $\mathcal{X}$ and $\mathcal{R}$. The result NORM(E) is always an input variable, a variable in $\mathcal{X}$, or a constant.

- For an input variable $x \in \mathcal{X}$, $\quad$ NORM(x) $= x$.
- For a non-input variable v, find a rule $v \Rightarrow x$ in $\mathcal{R}$. If no such rule exists, then the variable is unitialized. Otherwise NORM(v) $= x$.
- For a constant, NORM(c) $= c$.

Normalization (2)

- For an expresssion $f(t_1, \ldots, t_n)$, first recursively compute $x_1 := \text{NORM}(t_1), \ldots, x_n := \text{NORM}(t_n)$.

  If there is a rule $f(x_1, \ldots, x_n) \Rightarrow x$ in $\mathcal{R}$, then $\text{NORM}(\ f(t_1, \ldots, t_n)\ ) = x$.

  Otherwise, invent a new x, add it to $\mathcal{X}$ and add the rule $f(x_1, \ldots, x_n) \Rightarrow x$ to $\mathcal{R}$. Now $\text{NORM}(\ f(t_1, \ldots, t_n)\ ) = x$.

$\mathcal{R}$ can be implemented very efficient with hashing or discrimination trees.

Normalization (3)

Using NORM, the normalization procedures processes the assignments. For each assignment $v := E$, do the following:

- Add a rule $v \Rightarrow \text{NORM}(E)$ to $\mathcal{R}$.

Normalization (4)

When the algorithm has processed all assignments, one can reconstruct the expressions for the output variables of the block. (These are the variables that are looked at later on a path that originates from the block)

The $x \in \mathcal{X}$ will become local variables.

There is a problem that variables have been renamed. We come to this later.

Normalization (5)

In practice, one should attempt to normalize expressions before analyzing:

- Replace $X + 0 \Rightarrow X, \quad 0 + X \Rightarrow X.$
- Replace $X \times 1 \Rightarrow X, \quad X \times 0 \Rightarrow 0$, etc.
- Sort long multiplications and additions. (For example, first numbers, next by index.) Try to evaluate as much as possible at compile time.

Normalization (6)

It remains to generate the simplification of the block. The simplification is a sequence of assignments, but without recomputations.

Let $v_1, \ldots, v_n$ be the output variables of the block. (The variables that are used on a path originating from the block.)

Replace each v_i by NORM(v_i) on every path originating from the block.

Normalization (7)

Put

$$X_d := \{\ \},$$

(the intermediate variables that have an assignment.)

$$X_n := \{\text{NORM}(v_i) \mid \text{NORM}(v_i) \text{ is not a constant or}$$

$$\text{input variable of the block } \},$$

(the intermediate variables that need to be defined.)

$$\text{SIMP} := (\ ),$$

(the simplification of the block.)

Normalization (8)

While $X_n \backslash X_d$ is not empty, select an x (with maximal weight) from $X_n \backslash X_d$, and call ASSIGN(x).

The procedure ASSIGN(x) recursively assigns the variables that are needed to obtain a definition of x. (It is assumed that x is has no assignment when ASSIGN(x) is called.)

- Lookup the rule of form ($f(x_1, \ldots, x_n) \Rightarrow x$) $\in \mathcal{R}$ that defines x.
- As long as one of the $x_1, \ldots, x_n$ that is not a constant, nor an input variable, does not occur in X_d, select the x_i with greatest weight among those. Call DEFINE(x_i).
- Append the assignment $x := f(x_1, \ldots, x_n)$; to SIMP.

  Put $X_d := X_d \cup \{x\}$.

Example

(x, y are input variables.)

$a := x + y;$

$b := x + 1 + y;$

$c := 17;$

$d := x + y + c;$

$e := x + z;$

(Later, a, b, d are used)

Static Single Assignment Form

- The normalization algorithm has problems when variables are reassigned:
  $a := (x + y + z)$;
  $a := a + a$;
  $b := (x + y + z)$;

  $\Rightarrow$ Rename variables in advance.

- It renames its output variables.

Renaming is problematic when paths merge.

Static Single Assignment Form

Definition: Let P be the code of a procedure. (We assume that this is the basic block of analysis.)

We call P in static single assignment form if each variable that occurs in P is either an input variable, or has exactly one point of assignment. (which is an initialization)

In order to merge variables, the Φ function is used. (It seems to mean 'phoney')

SSA

Consider the procedure:

Procedure fact(N)
$F := 1$;
loop :
if($N = 0$) goto end;
$F := F \times N$;
$N := N - 1$;
goto loop;
end :
return N

SSA

Its SSA is:

Procedure fact(N_1)
$F_1 := 1$;
loop : (This is a merging point)
$F_2 := \Phi(F_1, F_3)$;
$N_3 := \Phi(N_1, N_3)$;
if($N_3 = 0$) goto end;
$F_3 := F_2 \times N_2$;
$N_3 := N_2 - 1$;
goto loop;
end :
return N_3

SSA

SSA of string copy example:

```
   unsigned int i1 = 0;
loop:
   i2 := Phi( i1, i3 );

   if( *(p+i2) == 0 ) goto end;

   *(q+i2) = *(p+i2);
   i3 = i2 + 1;
   goto loop;
end:
   return;
```

Problems with Analysis

- Primitive types only: It is almost impossible to analyze objects.
- Pointer assignments are problematic.
- It will not 'take out' variables from arrays. (See next slide)
- Function calls are opaque.

Problematic code:

```
   // Remove capital vocals:


   unsigned int i = 0;
   if( p[i] == 0 ) goto end;
loop:
   if( p[i] == 'A' ) p[i] = 'a';
   if( p[i] == 'E' ) p[i] = 'e';
   if( p[i] == 'I' ) p[i] = 'i';
   if( p[i] == 'O' ) p[i] = 'o';
   goto loop;
end:
   return;
```

Optimizer will certainly see possible reuse of `(p+i)`, but what about `*(p+i)`?

Typical use of reference:

```
int x = 4;
{
   int unused;
   int& returnval = unused;
   {
      int& p = x;
      p = p + 1;
      returnval = p;
   }
   // Such code could originate from the
   // inlined translation of ( ++ x );
}
```

References that are initialized with a local variable can be substituted away.

Example where address is taken:

```
int x;

lots of computation involving x;

dosomesomethingwith(x);
   // x is reference variable.

more computation involving x;
```

In this case, variable `x` can be stored in memory before the procedure call, and retrieved after the procedure call. One must use a heuristic in order to decide if it is worth the effort.

Which variables can be analyzed?

Variables with a primitive type (boolean, int, unsigned, char, pointer, double, real), which in addition:

- are local to an expression. (Constructed by the compiler for the evaluation of the expression.)
- are local in a procedure. It must be guaranteed that no pointer/reference to the variable can exist, which can be read from or written to. (If the pointer is short-lived, one can shortly store the variable to memory.)

Which Variables can be Analyzed?

From variables that are accessed through pointer, reference or array, a local copy can be made if:

- the pointer, reference, array (with its index) are constant.
- the variable cannot be accessed in another way. (No other pointer/reference/array is accessed, there is no function call during the time the local copy exists.)

In general, as soon as pointers, references, arrays are involved, things get very very complicated. No satisfactory solution exists. (*C* has a `restrict` keyword.)

I will say something more about register allocation and code selection, when I understand a bit more about this topic. (Hopefully, next week)

Constant Analysis

Consider

```
for( unsigned int i = 0; i < n; ++ i )
{
   for( unsigned int j = 0; j < n; ++ i )
   {
      M[i][j] = 0.0;
   }
}
```

The address calculation $M + i$ can be reused in the inner loop.

Let $\mathcal{G}_1 \subseteq \mathcal{G}$ be a subset of the flow graph of a program.

The set of constants values $\mathcal{V}$ is a set of pairs (of variables and values). It is the smallest set satisfying the following conditions:

Definition

- If v has no assignment in $\mathcal{G}_1$, then $(v, v) \in \mathcal{V}$.
- If v is assigned an expression $f(v_1, \ldots, v_n)$ in $\mathcal{G}_1$, then first define $\overline{v}_1, \ldots, \overline{v}_n$ as follows: if $(v_i, x_i) \in \mathcal{V}$, then $\overline{v}_i = x_i$. Otherwise, $\overline{v}_i = v_i$. If the expression $f(\overline{v}_1, \ldots, \overline{v}_n)$ can be simplified into an expression e containing only variables in $\mathcal{V}$, then $(v, e) \in \mathcal{V}$.
- If, for a conditional statement, the condition evaluates to **f** or **t**, then the unreachable code can be removed from the flow graph.

A few simplification rules:

$0 \times A$	$\Rightarrow$	0
$t = t$	$\Rightarrow$	$\mathbf{t}$
c_1 op c_2	$\Rightarrow$	can be computed
		if c_1, c_2 are numerical.
$\mathbf{f}$ and A	$\Rightarrow$	$\mathbf{f}$
$\mathbf{t}$ or A	$\Rightarrow$	$\mathbf{t}$
neg $\mathbf{f}$	$\Rightarrow$	$\mathbf{t}$
neg $\mathbf{t}$	$\Rightarrow$	$\mathbf{f}$

AC operators (associative commutative) should be sorted with constant part before non-constant part, in order to improve the chance of partial evaluation.

Removal of Constants from Loops

One should run the constant construction algorithm for each strongly connected component of the flow graph.

Constant subexpressions can be computed at the entry points.