## Program proofs and compilation

Ando Saabas

#### Institute of Cybernetics, Tallinn

Joint work with Tarmo Uustalu, Tamara Rezk

Theory Days, Koke 2005

# OUTLINE

- Background and motivation
- Different optimizations and their effect on program proofs
- Extensions to optmization algorithms
- Conclusion and further work

## RECAP

- In case of mobile devices, post-issuance downloading of code is possible.
- Bytecode verification can guarantee the type and memory safety of the program.
- Besides obvious security guarantees, some guarantees about functional properties of the code might be needed.
- Typically, developers can use interactive verification tools to get some guarantees about functional and behavioral properties of a (source) program.
- How to bring these benefits to the code user?

## PROOF CARRYING CODE

- Proving programs is hard, but checking proofs is easy, so ship the proof with the code and have the user check it
- ...but who would want to prove functional properties of compiled code?

## TWO DIRECTIONS

- Automatic generation of certificates, based on properties of the high level code (Necula, Morrisett)
- Generation of certificates based on high level proofs.

#### COMPILING PROOFS

The code developer would ...

- Write a program A, annotate it with (Hoare style) specifications S, and build a proof P that A abides to S using some verification environment.
- Compile the program, its specification and the proof, obtaining a compiled program <u>A</u>, a (compiled) specification <u>S</u>, and a (compiled) proof <u>P</u>.

The code consumer...

- Generates the set of proof obligations from  $\underline{A}$  and  $\underline{S}$  using a weakest precondition calculus.
- Uses a simple and fast proof checker to check if the proof  $\underline{P}$  is valid.

How to compile proofs?

For a non-optimizing compiler...

**Theorem**. For all while programs c and assertions P, the weakest precondition of c is syntactically equal to the weakest precondition of its compiled counterpart C(c):  $wp_w(c, P) = wp_a(C(c), P)$ 

The compilation of proofs and specifications could be identity?

For a non-optimizing compiler...

**Theorem**. For all while programs c and assertions P, the weakest precondition of c is syntactically equal to the weakest precondition of its compiled counterpart C(c):  $wp_w(c, P) = wp_a(C(c), P)$ 

The compilation of proofs and specifications could be identity? No, because of compiler optimizations.

#### THREE TYPES OF OPTIMIZATIONS

1. Optimizations that do not break the equivalence between proof-obligations (eg optimizations reducing the number of load and store instructions)

load	a		load	a
load	а	$\Rightarrow$	dup	
plus			plus	

2. Optimizations which break the syntactic equivalence (eg dead code elimination)

if (true) then  $c_1$  else  $c_2 \implies c_1$ 

true  $\Rightarrow wp(c_1, \phi) \land \mathsf{false} \Rightarrow wp(c_2, \phi) \qquad wp(c_1, \phi)$ 

3. Optimizations which break loop annotations

while i < n {  

$$j = 4 * i$$
  
 $k = a + j$   
 $s = s + A[k]$   
 $i = i + 1$   
 $c = 4 * n + a$   
 $k = a + j$   
 $s = s + A[k]$   
 $k = k + 4$   
}

## THE THIRD CATEGORY

Optimizations based on dataflow analysis such as reaching definitions and available expressions analysis

- Common sub-expression elimination
- Constant folding
- Useless code elimination

Loop optimizations

- Strength reduction and induction variable change
- Code motion

How are the assertions to be changed?

## THE INTERMEDIATE LANGUAGE

$$e \quad ::= \quad x \mid n \mid e_1 \ \oplus \ e_2 \mid M[e]$$

$$c ::= goto L \mid x := e \mid M[e_1] := e_2 \mid c_1; c_2 \mid$$

if e then  $L_1$  else  $L_2 \mid$  assert  $\varphi$ 

COMMON SUB-EXPRESSION ELIMINATION

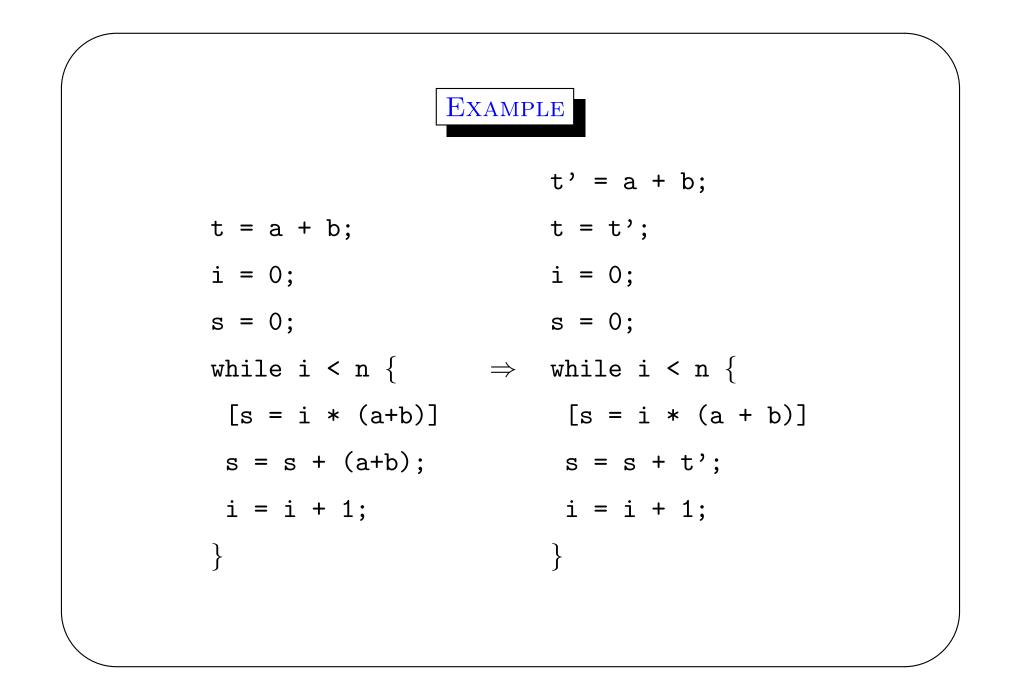
General idea: if an expression is calculated more than once, save it in a temporary variable to later reuse this result.

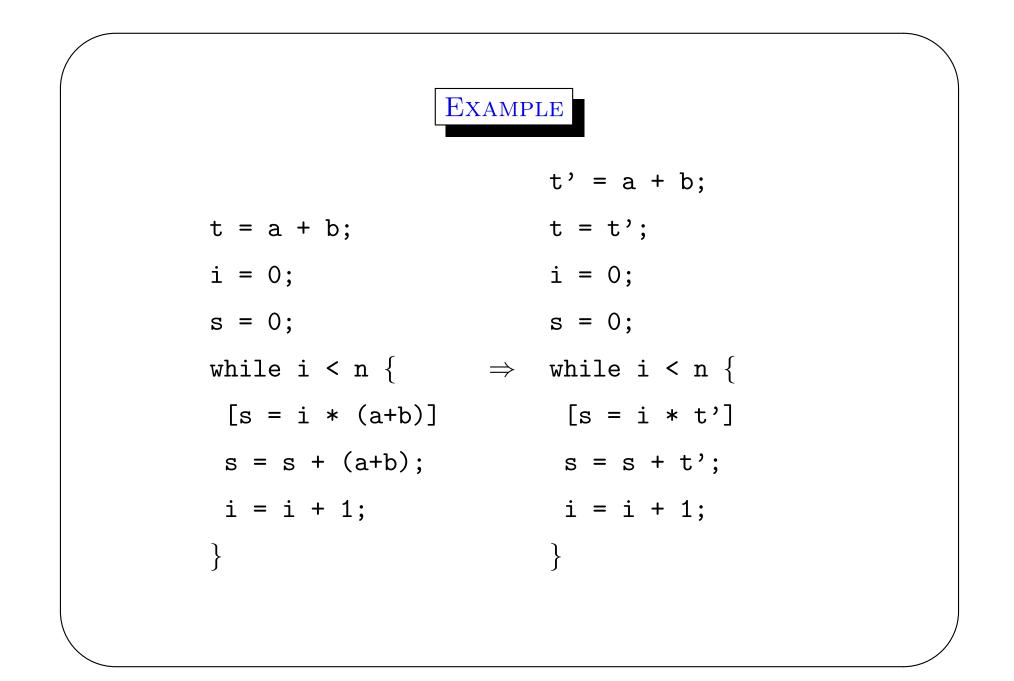
**Algorithm.** If there is a statement  $s: t = x \oplus y$  where  $x \oplus y$  is *available*, then compute *reaching expressions*, if find statements of the form  $n: v = x \oplus y$ , such that the path from n to s does not compute  $x \oplus y$  or define x or y. Choose a new temporary variable w, and rewrite n as

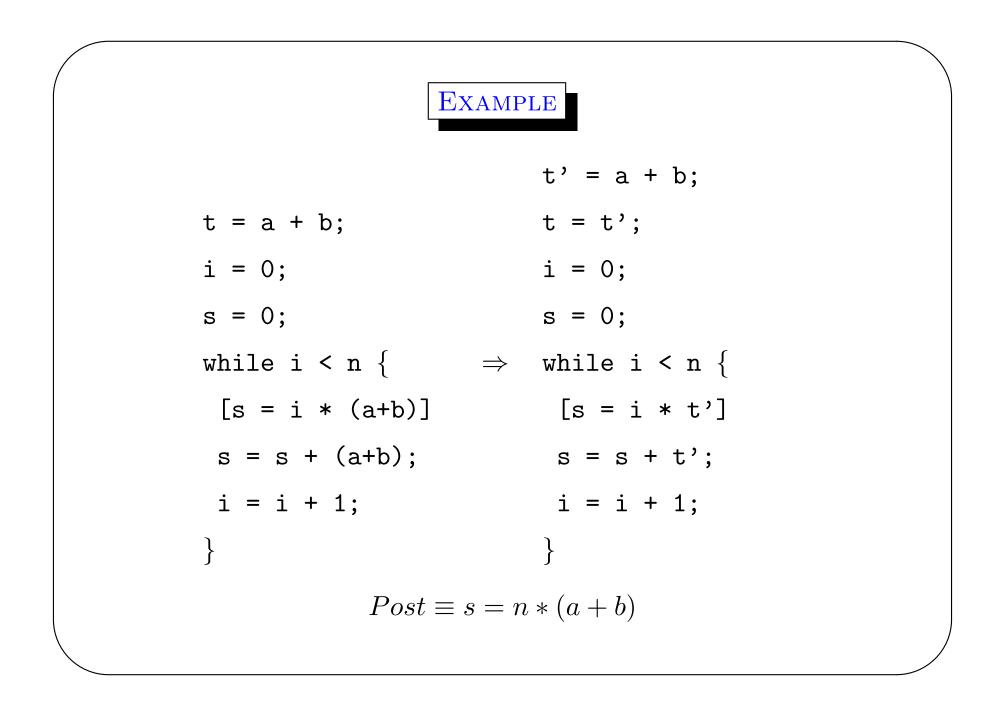
 $n : w = x \oplus y$ n' : v = w

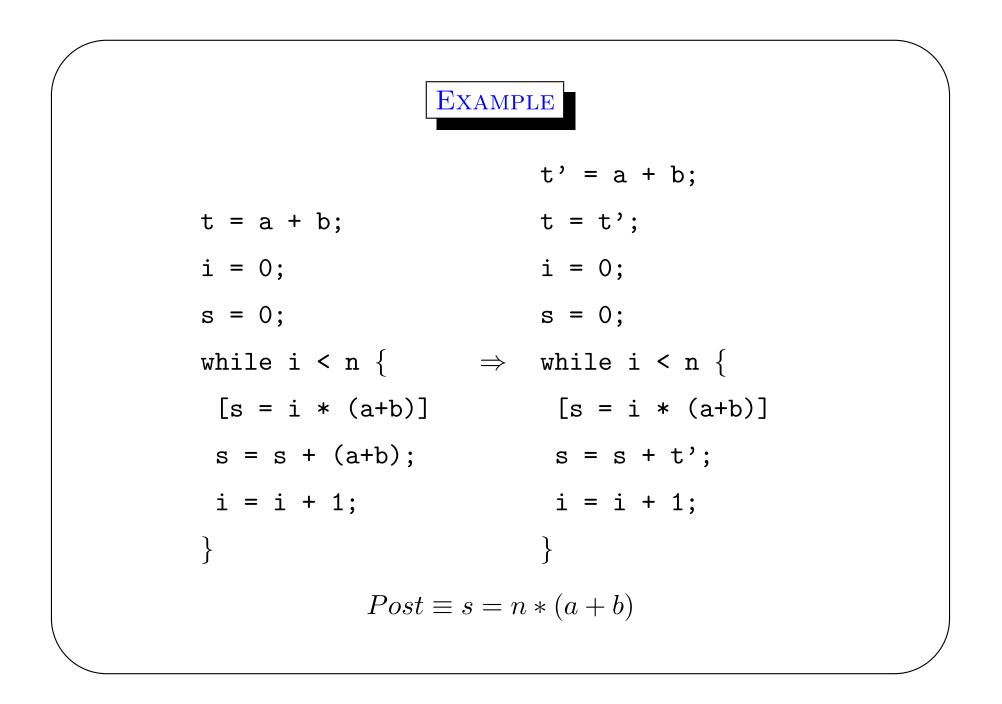
Finally, modify statement s to be

s:t=w



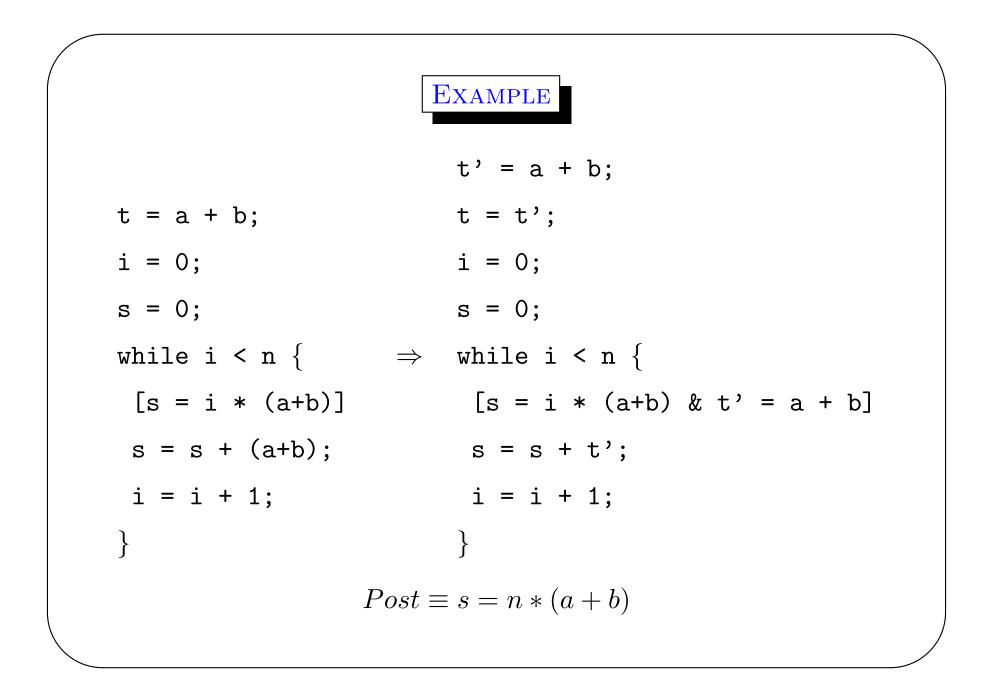






#### EXTENSION TO CSE OPTIMIZATION ALGORITHM

- 1. For each assert instruction, compute definitions that reach it
- 2. Compute reaching assertions, ie for each program point a set of asserts that may appear before it in the control flow
- 3. For to-be-optimized program points (n, s), find the set A of all asserts which n reaches
- 4. For all asserts  $\varphi$  in A which reach s, change the assert to  $(\varphi \wedge w = x \oplus y)$ , where w is the fresh variable and  $x \oplus y$  is the common sub-expression.



## CONSTANT PROPAGATION

- General idea: if the value of a variable is always constant, replace the variable with the constant.
- Based on reaching definitions analysis.
- Similar to CSE, the same algorithm applies.

EXAMPLE c = 5; i = 0; i = 0;s = 0;s = 0;while i < n {  $\Rightarrow$  while i < n { s = s + (i + c); s = s + (i + 5);i = i + 1; i = i + 1;} }

### USELESS CODE ELIMINATION

- General idea: assignments to variables which are not used later in the program can be removed.
- Based on liveness analysis (backward dataflow analysis)
- Does not pose a problem when asserts are considered as part of the language ie *use* variables

### STRENGTH REDUCTION

- General idea: replace multiplication with addition inside loops
- Based on induction variable detection

Algorithm: in a loop L, a variable i is an induction variable if it only changes by a given constant in each iteration of the loop  $(i = i \pm c)$ . A variable j is a derived induction variable, if the only definition of jin L is of the form j = a + i \* b. Strength reduction can be performed by introducing a new variable j', such that j' = j' + c \* b and j = j', and j' is initialized to a + i \* b

$$i = 0;$$
  

$$i = 0;$$
  

$$j' = 0;$$
  

$$s = 0;$$
  
while i < n {  

$$j = 4 * i;$$
  

$$s = s + M[j];$$
  

$$i = i + 1;$$
  

$$j' = j';$$
  

$$s = s + M[j];$$
  

$$i = i + 1;$$
  

$$j' = j' + 4;$$
  

$$j' = j' + 4;$$

$$I \equiv i \leq n \wedge s = \sum_{x=0}^{i-1} M[x*4]$$
$$Post \equiv s = \sum_{x=0}^{n-1} M[x*4]$$

Algorithm extension: find the set A of asserts in the loop. For each assert  $\varphi$  in A and derived induction variable definition j = a + i \* b, change the assert to  $(\varphi \wedge j' = a + i * b)$ .

i = 0;  
i = 0;  
j' = 0;  
s = 0;  
while i < n {  
j = 4 \* i;  
s = s + M[j];  
i = i + 1;  
}  
i = i + 1;  

$$i = j' = j';$$
  
s = s + M[j];  
j' = j' + 4;  
j' = j' + 4;  
j' = j' + 4;

;

$$I \equiv i \le n \land s = \sum_{x=0}^{i-1} M[x*4] \land j' = 4*i$$

#### INDUCTION VARIABLE CHANGE

i = 0;	i = 0;
j' = 0;	j' = 0;
s = 0;	s = 0;
while i < n $\{$	while j' < n * 4 $\{$
$s = s + M[j']; \Rightarrow$	s = s + M[j'];
i = i + 1;	j' = j' + 4;
j' = j' + 4;	}

Already taken care of in the strength reduction step (the relationship between induction variables is made explicit in the assertion).

## CODE MOTION

- General idea: lift computation out of the loop body when possible.
- Based on loop-invariant computation, ie finding statements
   t = a ⊕ b in a loop where the values of a and b are the same in each iteration of the loop.
- Similar to common sub-expression elimination

$$t = a + b;$$
  
i = 0;  
s = 0;  
while i < n {  
t = a + b;  
s = s + M[i + t];  
i = i + 1;  
}  
Post = s =  $\sum_{x=0}^{n-1} M[x + (a + b)]$ 

Algorithm extension: find the set A of asserts in the loop. For each assert  $\varphi$  in A and invariant definition  $t = a \oplus b$ , change the assert to  $(\varphi \wedge t = a \oplus b)$ .

### CONCLUSION AND FURTHER WORK.

- Assertion transformation not too complicated, so proof transformation seems feasible
- Improving the algorithms
- Implementation