Code Generation: Aho Johnson Algorithm

Amey Karkare

karkare@cse.iitk.ac.in

March 28, 2019

Aho-Johnson Algorithm

Characteristics of the Algorithm

- Considers expression trees.

Characteristics of the Algorithm

- Considers expression trees.
- The target machine model is general enough to generate code for a large class of machines.

Characteristics of the Algorithm

- Considers expression trees.
- The target machine model is general enough to generate code for a large class of machines.
- Represented as a tree, an instruction

Characteristics of the Algorithm

- Considers expression trees.
- The target machine model is general enough to generate code for a large class of machines.
- Represented as a tree, an instruction
  - can have a root of any arity.

Characteristics of the Algorithm

- Considers expression trees.
- The target machine model is general enough to generate code for a large class of machines.
- Represented as a tree, an instruction
  - can have a root of any arity.
  - can have as leaves registers or memory locations appearing in any order.

Characteristics of the Algorithm

- Considers expression trees.
- The target machine model is general enough to generate code for a large class of machines.
- Represented as a tree, an instruction
  - can have a root of any arity.
  - can have as leaves registers or memory locations appearing in any order.
  - can be of of any height

Characteristics of the Algorithm

- Considers expression trees.
- The target machine model is general enough to generate code for a large class of machines.
- Represented as a tree, an instruction
  - can have a root of any arity.
  - can have as leaves registers or memory locations appearing in any order.
  - can be of of any height
- Does not use algebraic properties of operators.

Characteristics of the Algorithm

- Considers expression trees.
- The target machine model is general enough to generate code for a large class of machines.
- Represented as a tree, an instruction
  - can have a root of any arity.
  - can have as leaves registers or memory locations appearing in any order.
  - can be of of any height
- Does not use algebraic properties of operators.
- Generates optimal code, where, once again, the cost measure is the number of instructions in the code.

Characteristics of the Algorithm

- Considers expression trees.
- The target machine model is general enough to generate code for a large class of machines.
- Represented as a tree, an instruction
  - can have a root of any arity.
  - can have as leaves registers or memory locations appearing in any order.
  - can be of of any height
- Does not use algebraic properties of operators.
- Generates optimal code, where, once again, the cost measure is the number of instructions in the code.
- Complexity is linear in the size of the expression tree.

Expression Trees Defined

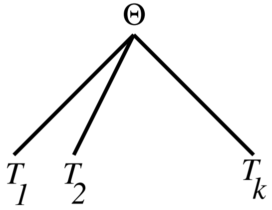
- Let Σ be a countable set of operands, and Θ be a finite set of operators. Then,

Expression Trees Defined

- Let Σ be a countable set of operands, and Θ be a finite set of operators. Then,
  1. A single vertex labeled by a name from Σ is an expression tree.

Expression Trees Defined

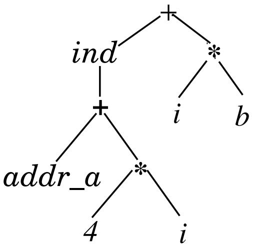
- Let Σ be a countable set of operands, and Θ be a finite set of operators. Then,
  1. A single vertex labeled by a name from Σ is an expression tree.
  2. If T_1, T_2, ..., T_k are expression trees whose leaves all have distinct labels and θ is a k-ary operator in Θ, then

     is an expression tree.

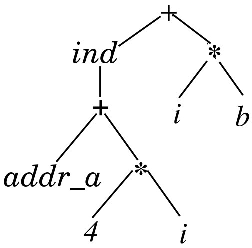
Example

- An example of an expression tree is



Example

- An example of an expression tree is



- **Notation:** If T is an expression tree, and S is a subtree of T, then T/S is the the tree obtained by replacing S in T by a single leaf labeled by a distinct name from Σ.

The Machine Model

1. The machine has *n* general purpose registers (no special registers).

The Machine Model

1. The machine has *n* general purpose registers (no special registers).
2. Countable sequence of memory locations.

The Machine Model

1. The machine has n general purpose registers (no special registers).
2. Countable sequence of memory locations.
3. Instructions are of the form:

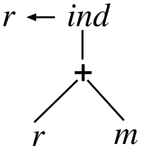
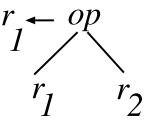
The Machine Model

1. The machine has n general purpose registers (no special registers).
2. Countable sequence of memory locations.
3. Instructions are of the form:
   a. $r \leftarrow E$, r is a register and E is an expression tree whose operators are from Θ and operands are registers, memory locations or constants. Further, *r should be one of the register names occurring (if any) in E.*

The Machine Model

1. The machine has n general purpose registers (no special registers).
2. Countable sequence of memory locations.
3. Instructions are of the form:
   a. $r \leftarrow E$, r is a register and E is an expression tree whose operators are from Θ and operands are registers, memory locations or constants. Further, *r should be one of the register names occurring (if any) in E.*
   b. $m \leftarrow r$, a store instruction.

Example Of A Machine

$r \leftarrow c$	**{MOV #c, r}**
$r \leftarrow m$	**{MOV m, r}**
$m \leftarrow r$	**{MOV r, m}**
	**{MOV m(r), r}**
	**{op $\mathbf{r_2}$, $\mathbf{r_1}$}**

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.
- The machine program below evaluates $a[i] + i * b$

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.
- The machine program below evaluates $a[i] + i * b$

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \dots I_q$.
- The machine program below evaluates $a[i] + i * b$

$$r_1 \leftarrow 4$$

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.
- The machine program below evaluates $a[i] + i * b$

$$r_1 \leftarrow 4$$
$$r_1 \leftarrow r_1 * i$$

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.
- The machine program below evaluates $a[i] + i * b$

$$r_1 \leftarrow 4$$
$$r_1 \leftarrow r_1 * i$$
$$r_2 \leftarrow addr\_a$$

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.
- The machine program below evaluates $a[i] + i * b$

$$r_1 \leftarrow 4$$
$$r_1 \leftarrow r_1 * i$$
$$r_2 \leftarrow addr\_a$$
$$r_2 \leftarrow r_2 + r_1$$

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.
- The machine program below evaluates $a[i] + i * b$

$$r_1 \leftarrow 4$$
$$r_1 \leftarrow r_1 * i$$
$$r_2 \leftarrow addr\_a$$
$$r_2 \leftarrow r_2 + r_1$$
$$r_2 \leftarrow ind(r_2)$$

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.
- The machine program below evaluates $a[i] + i * b$

$$r_1 \leftarrow 4$$
$$r_1 \leftarrow r_1 * i$$
$$r_2 \leftarrow addr\_a$$
$$r_2 \leftarrow r_2 + r_1$$
$$r_2 \leftarrow ind(r_2)$$
$$r_3 \leftarrow i$$

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.
- The machine program below evaluates $a[i] + i * b$

$$r_1 \leftarrow 4$$
$$r_1 \leftarrow r_1 * i$$
$$r_2 \leftarrow addr\_a$$
$$r_2 \leftarrow r_2 + r_1$$
$$r_2 \leftarrow ind(r_2)$$
$$r_3 \leftarrow i$$
$$r_3 \leftarrow r_3 * b$$

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.
- The machine program below evaluates $a[i] + i * b$

$$
\begin{aligned}
r_1 &\leftarrow 4 \\
r_1 &\leftarrow r_1 * i \\
r_2 &\leftarrow addr\_a \\
r_2 &\leftarrow r_2 + r_1 \\
r_2 &\leftarrow ind(r_2) \\
r_3 &\leftarrow i \\
r_3 &\leftarrow r_3 * b \\
r_2 &\leftarrow r_2 + r_3
\end{aligned}
$$

MACHINE PROGRAM

- A *machine program* consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.
- The machine program below evaluates $a[i] + i * b$

$$r_1 \leftarrow 4$$
$$r_1 \leftarrow r_1 * i$$
$$r_2 \leftarrow addr\_a$$
$$r_2 \leftarrow r_2 + r_1$$
$$r_2 \leftarrow ind(r_2)$$
$$r_3 \leftarrow i$$
$$r_3 \leftarrow r_3 * b$$
$$r_2 \leftarrow r_2 + r_3$$

VALUE OF A PROGRAM

- We need to define the value $v(P)$ computed by a program P.

VALUE OF A PROGRAM

- We need to define the value $v(P)$ computed by a program P.
    1. We want to specify what it means to say that a *program* P *computes an expression tree* T. This is when the value of the program $v(P)$ is the same as T.

VALUE OF A PROGRAM

- We need to define the value $v(P)$ computed by a program P.
  1. We want to specify what it means to say that a *program* P *computes an expression tree* T. This is when the value of the program $v(P)$ is the same as T.
  2. We also want to talk of *equivalence* of two programs P_1 and P_2. This is true when $v(P_1) = v(P_2)$.

VALUE OF A PROGRAM

- What is the *value of a program* $P = I_1, I_2, \ldots, I_q$?

VALUE OF A PROGRAM

- What is the *value of a program* $P = I_1, I_2, \ldots, I_q$?
- It is a tree, defined as follows:

VALUE OF A PROGRAM

- What is the *value of a program* $P = I_1, I_2, \ldots, I_q$?
- It is a tree, defined as follows:
- First define $v_t(z)$, the value of a memory location or register z after the execution of the instruction I_t.

VALUE OF A PROGRAM

- What is the *value of a program* $P = I_1, I_2, \ldots, I_q$?
- It is a tree, defined as follows:
- First define $v_t(z)$, the value of a memory location or register z after the execution of the instruction I_t.
  a. Initially $v_0(z)$ is z if z is a memory location, else it is undefined.

VALUE OF A PROGRAM

- What is the *value of a program* $P = I_1, I_2, \ldots, I_q$?
- It is a tree, defined as follows:
- First define $v_t(z)$, the value of a memory location or register z after the execution of the instruction I_t.
  a. Initially $v_0(z)$ is z if z is a memory location, else it is undefined.
  b. If I_t is $r \leftarrow E$, then $v_t(r)$ is the tree obtained by taking the tree representing E, and substituting for each leaf l the value of $v_{t-1}(l)$.

VALUE OF A PROGRAM

- What is the *value of a program* $P = I_1, I_2, \ldots, I_q$?
- It is a tree, defined as follows:
- First define $v_t(z)$, the value of a memory location or register z after the execution of the instruction I_t.
  a. Initially $v_0(z)$ is z if z is a memory location, else it is undefined.
  b. If I_t is $r \leftarrow E$, then $v_t(r)$ is the tree obtained by taking the tree representing E, and substituting for each leaf l the value of $v_{t-1}(l)$.
  c. If I_t is $m \leftarrow r$, then $v_t(m)$ is $v_{t-1}(r)$.

VALUE OF A PROGRAM

- What is the *value of a program* $P = I_1, I_2, \ldots, I_q$?
- It is a tree, defined as follows:
- First define $v_t(z)$, the value of a memory location or register z after the execution of the instruction I_t.
  a. Initially $v_0(z)$ is z if z is a memory location, else it is undefined.
  b. If I_t is $r \leftarrow E$, then $v_t(r)$ is the tree obtained by taking the tree representing E, and substituting for each leaf l the value of $v_{t-1}(l)$.
  c. If I_t is $m \leftarrow r$, then $v_t(m)$ is $v_{t-1}(r)$.
  d. Otherwise $v_t(z) = v_{t-1}(z)$.

VALUE OF A PROGRAM

- What is the *value of a program* $P = I_1, I_2, \ldots, I_q$?
- It is a tree, defined as follows:
- First define $v_t(z)$, the value of a memory location or register z after the execution of the instruction I_t.
  a. Initially $v_0(z)$ is z if z is a memory location, else it is undefined.
  b. If I_t is $r \leftarrow E$, then $v_t(r)$ is the tree obtained by taking the tree representing E, and substituting for each leaf l the value of $v_{t-1}(l)$.
  c. If I_t is $m \leftarrow r$, then $v_t(m)$ is $v_{t-1}(r)$.
  d. Otherwise $v_t(z) = v_{t-1}(z)$.
- If I_q is $z \leftarrow E$, then the value of P is $v_q(z)$.

EXAMPLE

- For the program:

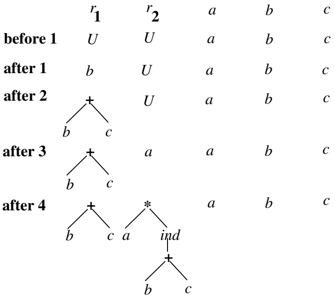
$$r_1 \leftarrow b$$
$$r_1 \leftarrow r_1 + c$$
$$r_2 \leftarrow a$$
$$r_2 \leftarrow r_2 * ind(r_1)$$

EXAMPLE

- For the program:

$$r_1 \leftarrow b$$
$$r_1 \leftarrow r_1 + c$$
$$r_2 \leftarrow a$$
$$r_2 \leftarrow r_2 * ind(r_1)$$

- the values of r_1, r_2, a, b and c at different time instants are:

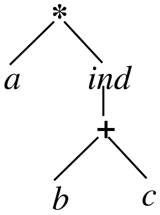


EXAMPLE

- For the program:

$$\begin{aligned} r_1 &\leftarrow b \\ r_1 &\leftarrow r_1 + c \\ r_2 &\leftarrow a \\ r_2 &\leftarrow r_2 * ind(r_1) \end{aligned}$$

- The values of of the program is



USELESS INSTRUCTIONS

- An instruction I_t in a program P is said to be *useless*, if the program P_1 formed by removing I_t from P is equivalent to P.

USELESS INSTRUCTIONS

- An instruction I_t in a program P is said to be *useless*, if the program P_1 formed by removing I_t from P is equivalent to P.
- NOTE: We shall assume that our programs do not have any useless instructions.

SCOPE OF INSTRUCTIONS

- The *scope of an instruction* I_t in a program $P = I_1 I_2 \ldots I_q$ is the sequence of instructions $I_{t+1}, \ldots, I_s$, where s is the largest index such that

SCOPE OF INSTRUCTIONS

- The *scope of an instruction* I_t in a program $P = I_1 I_2 \ldots I_q$ is the sequence of instructions $I_{t+1}, \ldots, I_s$, where s is the largest index such that
  a. The register or memory location defined by I_t is used by I_s, and

SCOPE OF INSTRUCTIONS

- The *scope of an instruction* I_t in a program $P = I_1 I_2 \ldots I_q$ is the sequence of instructions $I_{t+1}, \ldots, I_s$, where s is the largest index such that
  a. The register or memory location defined by I_t is used by I_s, and
  b. This register/memory location is not redefined by the instructions between I_t and I_s.

SCOPE OF INSTRUCTIONS

- The *scope of an instruction* I_t in a program $P = I_1 I_2 \ldots I_q$ is the sequence of instructions $I_{t+1}, \ldots, I_s$, where s is the largest index such that
  a. The register or memory location defined by I_t is used by I_s, and
  b. This register/memory location is not redefined by the instructions between I_t and I_s.
- The relation between I_s and I_t is expressed by saying that I_s is the *last use* of I_t, and is denoted by $s = U_p(t)$.

REARRANGABILITY OF PROGRAMS

- We shall show that each program can be rearranged to obtain an equivalent program (of the same length) in *strong normal form*.

REARRANGABILITY OF PROGRAMS

- We shall show that each program can be rearranged to obtain an equivalent program (of the same length) in *strong normal form*.
- Why is this result important? This is because our algorithm considers programs which are in strong normal form only. The above result assures us that by doing so, we shall not miss out an optimal solution.

REARRANGABILITY OF PROGRAMS

- We shall show that each program can be rearranged to obtain an equivalent program (of the same length) in *strong normal form*.
- Why is this result important? This is because our algorithm considers programs which are in strong normal form only. The above result assures us that by doing so, we shall not miss out an optimal solution.
- To show the above result, we shall have to consider the kinds of rearrangements which retain program equivalence.

Rearrangement Theorem

- Let $P = I_1, I_2, \ldots, I_q$ be a program which computes an expression tree.

Rearrangement Theorem

- Let $P = I_1, I_2, \ldots, I_q$ be a program which computes an expression tree.
- Let π be a permutation on $\{1 \ldots q\}$ with $\pi(q) = q$.

Rearrangement Theorem

- Let $P = I_1, I_2, \ldots, I_q$ be a program which computes an expression tree.
- Let π be a permutation on $\{1 \ldots q\}$ with $\pi(q) = q$.
- π induces a rearranged program $Q = J_1, J_2, \ldots, J_q$ with I_i in P becoming $J_{\pi(i)}$ in Q.

Rearrangement Theorem

- Let $P = I_1, I_2, \dots, I_q$ be a program which computes an expression tree.
- Let π be a permutation on $\{1 \dots q\}$ with $\pi(q) = q$.
- π induces a rearranged program $Q = J_1, J_2, \dots, J_q$ with I_i in P becoming $J_{\pi(i)}$ in Q.
- Then Q is equivalent to P if $\pi(U_P(t)) = U_Q(\pi(t))$.

Rearrangement Theorem: Notes

- The rearrangement theorem merely states that a rearrangement retains program equivalence, if any variable defined by an instruction in the original program is last used by the same instructions in both the original and rearranged program.

Rearrangement Theorem: Notes

- The rearrangement theorem merely states that a rearrangement retains program equivalence, if any variable defined by an instruction in the original program is last used by the same instructions in both the original and rearranged program.
- To see why the statement of the theorem is true, reason as follows.

Rearrangement Theorem: Notes

a. P is equivalent to Q, if the operands used by the last instruction I_q(also J_q) have the same value in P and Q.

Rearrangement Theorem: Notes

a. P is equivalent to Q, if the operands used by the last instruction I_q(also J_q) have the same value in P and Q.

b. Consider any operand in I_q, say z. By the rearrangement theorem, This must have been defined by the same instruction (though in different positions say I_t and $J_{\pi(t)}$) in P and Q. So z in I_q and J_q have the same value, if the operands used by I_t and $J_{\pi(t)}$ have the same value in P and Q.

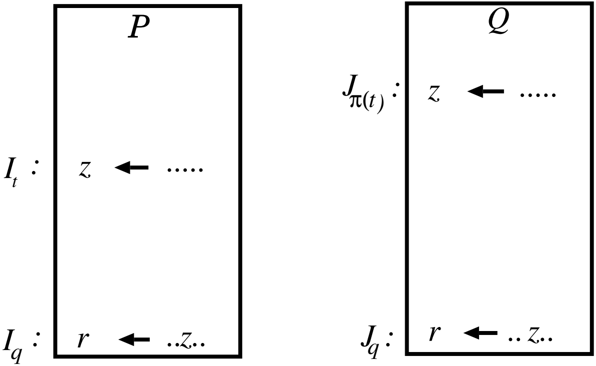
Rearrangement Theorem: Notes

a. P is equivalent to Q, if the operands used by the last instruction I_q(also J_q) have the same value in P and Q.

b. Consider any operand in I_q, say z. By the rearrangement theorem, This must have been defined by the same instruction (though in different positions say I_t and $J_{\pi(t)}$) in P and Q. So z in I_q and J_q have the same value, if the operands used by I_t and $J_{\pi(t)}$ have the same value in P and Q.

c. Repeat this argument, till you come across an instruction with all constants on the right hand side.

Rearrangement Theorem: Notes



WIDTH

- The *width* of a program is a measure of the minimum number of registers required to execute the program.

WIDTH

- The *width* of a program is a measure of the minimum number of registers required to execute the program.
- Formally, if P is a program, then the *width of an instruction* I_t is the number of distinct j, $1 \leq j \leq t$, with $U_P(j) > t$, and I_j not a store instruction.

WIDTH

- The *width* of a program is a measure of the minimum number of registers required to execute the program.
- Formally, if P is a program, then the *width of an instruction* I_t is the number of distinct j, $1 \leq j \leq t$, with $U_P(j) > t$, and I_j not a store instruction.

WIDTH

- The *width* of a program is a measure of the minimum number of registers required to execute the program.
- Formally, if P is a program, then the *width of an instruction* I_t is the number of distinct j, $1 \leq j \leq t$, with $U_P(j) > t$, and I_j not a store instruction.

$$
\begin{array}{lcl}
 & r_1 \leftarrow & \\
 & r_2 \leftarrow & \\
I_t : & & \text{Width} = 2 \\
 & \leftarrow r_1 & \\
 & \leftarrow r_2 &
\end{array}
$$

WIDTH

- The *width* of a program is a measure of the minimum number of registers required to execute the program.
- Formally, if P is a program, then the *width of an instruction* I_t is the number of distinct j, $1 \le j \le t$, with $U_P(j) > t$, and I_j not a store instruction.

$$
\begin{array}{lcl}
 & r_1 \leftarrow & \\
 & r_2 \leftarrow & \\
I_t : & & \text{Width} = 2 \\
 & \leftarrow r_1 & \\
 & \leftarrow r_2 &
\end{array}
$$

- The *width of a program* P is the maximum width over all instructions in P.

WIDTH

- A program of width w (but possibly using more than w registers) can be rearranged into an equivalent program using exactly w registers.

WIDTH

- A program of width w (but possibly using more than w registers) can be rearranged into an equivalent program using exactly w registers.
- EXAMPLE:

$r_1 \leftarrow a$	$r_1 \leftarrow a$
$r_2 \leftarrow b$	$r_2 \leftarrow b$
$r_1 \leftarrow r_1 + r_2$	$r_1 \leftarrow r_1 + r_2$
$r_3 \leftarrow c$	$r_2 \leftarrow c$
$r_3 \leftarrow r_3 + d$	$r_2 \leftarrow r_2 + d$
$r_1 \leftarrow r_1 * r_3$	$r_1 \leftarrow r_1 * r_2$

WIDTH

- A program of width w (but possibly using more than w registers) can be rearranged into an equivalent program using exactly w registers.
- EXAMPLE:

$$\begin{array}{ll} r_1 \leftarrow a & r_1 \leftarrow a \\ r_2 \leftarrow b & r_2 \leftarrow b \\ r_1 \leftarrow r_1 + r_2 & r_1 \leftarrow r_1 + r_2 \\ r_3 \leftarrow c & r_2 \leftarrow c \\ r_3 \leftarrow r_3 + d & r_2 \leftarrow r_2 + d \\ r_1 \leftarrow r_1 * r_3 & r_1 \leftarrow r_1 * r_2 \end{array}$$

- In the example above, the first program has width 2 but uses 3 registers. By suitable renaming, the number of registers in the second program has been brought down to 2.

LEMMA

Let P be a program of width w, and let R be a set of w distinct registers. Then, by renaming the registers used by P, we may construct an equivalent program P', with the same length as P, which uses only registers in R.

PROOF OUTLINE

1. The relabeling algorithm should be consistent, that is, when a variable which is defined is relabeled, its use should also be relabeled.

PROOF OUTLINE

1. The relabeling algorithm should be consistent, that is, when a variable which is defined is relabeled, its use should also be relabeled.
2. Assume that we are renaming the registers in the instructions in order starting from the first instruction. At which points will there be a question of a choice of registers?

PROOF OUTLINE

1. The relabeling algorithm should be consistent, that is, when a variable which is defined is relabeled, its use should also be relabeled.
2. Assume that we are renaming the registers in the instructions in order starting from the first instruction. At which points will there be a question of a choice of registers?
    a. There is no question of choice for the registers on the RHS of an instruction. These had been decided at the point of their definitions (consistent relabeling).

PROOF OUTLINE

1. The relabeling algorithm should be consistent, that is, when a variable which is defined is relabeled, its use should also be relabeled.
2. Assume that we are renaming the registers in the instructions in order starting from the first instruction. At which points will there be a question of a choice of registers?
    a. There is no question of choice for the registers on the RHS of an instruction. These had been decided at the point of their definitions (consistent relabeling).
    b. There is no question of choice for the register r in the instruction $r \leftarrow E$, where E has some register operands. r has to be one of the registers occurring in E.

PROOF OUTLINE

1. The relabeling algorithm should be consistent, that is, when a variable which is defined is relabeled, its use should also be relabeled.
2. Assume that we are renaming the registers in the instructions in order starting from the first instruction. At which points will there be a question of a choice of registers?
   a. There is no question of choice for the registers on the RHS of an instruction. These had been decided at the point of their definitions (consistent relabeling).
   b. There is no question of choice for the register r in the instruction $r \leftarrow E$, where E has some register operands. r has to be one of the registers occurring in E.
   c. The only instructions involving a choice of registers are instructions of the form $r \leftarrow E$, where E has no register operands.

PROOF OUTLINE

3. Since the width of P is w, the width of the instruction, just before $r \leftarrow E$ is at most $w - 1$. (Why?)

PROOF OUTLINE

3. Since the width of P is w, the width of the instruction, just before $r \leftarrow E$ is at most $w - 1$. (Why?)
4. Therefore a register can always be found for r in the rearranged program P'.

CONTIGUITY AND STRONG CONTIGUITY

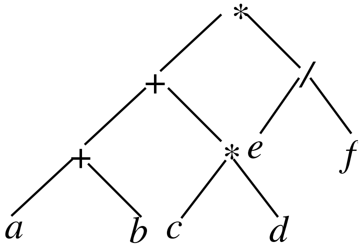
- Can one decrease the width of a program?

CONTIGUITY AND STRONG CONTIGUITY

- Can one decrease the width of a program?
- For *storeless programs*, there is an arrangement which has minimum width.

CONTIGUITY AND STRONG CONTIGUITY

- Can one decrease the width of a program?
- For *storeless programs*, there is an arrangement which has minimum width.
- EXAMPLE: All the three programs P_1, P_2, and P_3 compute the expression tree shown below:

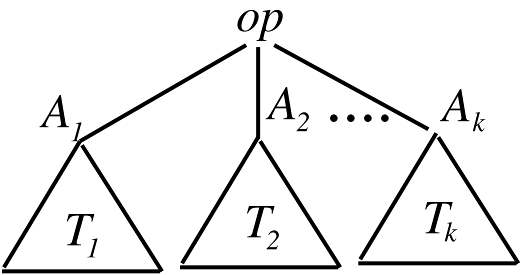


$\underline{P_1}$	$\underline{P_2}$	$\underline{P_3}$
$r_1 \leftarrow a$	$r_1 \leftarrow a$	$r_1 \leftarrow a$
$r_2 \leftarrow b$	$r_2 \leftarrow b$	$r_2 \leftarrow b$
$r_3 \leftarrow c$	$r_3 \leftarrow c$	$r_1 \leftarrow r_1 + r_2$
$r_4 \leftarrow d$	$r_4 \leftarrow d$	$r_2 \leftarrow c$
$r_5 \leftarrow e$	$r_1 \leftarrow r_1 + r_2$	$r_3 \leftarrow d$
$r_6 \leftarrow f$	$r_3 \leftarrow r_3 * r_4$	$r_2 \leftarrow r_2 * r_3$
$r_5 \leftarrow r_5 / r_6$	$r_1 \leftarrow r_1 + r_3$	$r_1 \leftarrow r_1 + r_2$
$r_3 \leftarrow r_3 * r_4$	$r_2 \leftarrow e$	$r_2 \leftarrow e$
$r_1 \leftarrow r_1 + r_2$	$r_3 \leftarrow f$	$r_3 \leftarrow f$
$r_1 \leftarrow r_1 + r_3$	$r_2 \leftarrow r_2 / r_3$	$r_2 \leftarrow r_2 / r_3$
$r_1 \leftarrow r_1 * r_5$	$r_1 \leftarrow r_1 * r_2$	$r_1 \leftarrow r_1 * r_2$

$\underline{P_1}$	$\underline{P_2}$	$\underline{P_3}$
$r_1 \leftarrow a$	$r_1 \leftarrow a$	$r_1 \leftarrow a$
$r_2 \leftarrow b$	$r_2 \leftarrow b$	$r_2 \leftarrow b$
$r_3 \leftarrow c$	$r_3 \leftarrow c$	$r_1 \leftarrow r_1 + r_2$
$r_4 \leftarrow d$	$r_4 \leftarrow d$	$r_2 \leftarrow c$
$r_5 \leftarrow e$	$r_1 \leftarrow r_1 + r_2$	$r_3 \leftarrow d$
$r_6 \leftarrow f$	$r_3 \leftarrow r_3 * r_4$	$r_2 \leftarrow r_2 * r_3$
$r_5 \leftarrow r_5 / r_6$	$r_1 \leftarrow r_1 + r_3$	$r_1 \leftarrow r_1 + r_2$
$r_3 \leftarrow r_3 * r_4$	$r_2 \leftarrow e$	$r_2 \leftarrow e$
$r_1 \leftarrow r_1 + r_2$	$r_3 \leftarrow f$	$r_3 \leftarrow f$
$r_1 \leftarrow r_1 + r_3$	$r_2 \leftarrow r_2 / r_3$	$r_2 \leftarrow r_2 / r_3$
$r_1 \leftarrow r_1 * r_5$	$r_1 \leftarrow r_1 * r_2$	$r_1 \leftarrow r_1 * r_2$

The program P_2 has a width less than P_1, whereas P_3 has the least width of all three programs. P_2 is a *contiguous* program whereas P_3 is a *strongly contiguous* program.

CONTIGUITY AND STRONG CONTIGUITY



CONTIGUITY AND STRONG CONTIGUITY

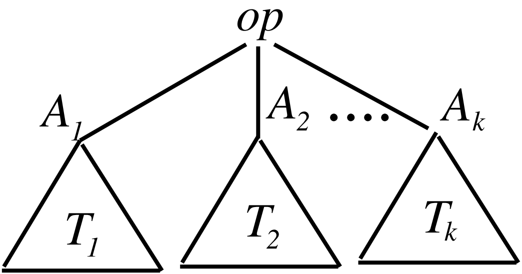
THEOREM: Let $P = I_1, I_2, \ldots, I_q$ be a program of width w with no stores. I_q uses k registers whose values at time $q-1$ are $A_1, \ldots, A_k$. Then there exists an equivalent program $Q = J_1, J_2, \ldots, J_q$, and a permutation π on $\{1, \ldots, k\}$ such that

i. Q has width at most w.

ii. Q can be written as $P_1 \ldots P_k J_q$ where $v(P_i) = A_\pi(i)$ for $1 \leq i \leq k$, and the width of P_i, by itself, is at most $w - i + 1$.

CONTIGUITY AND STRONG CONTIGUITY

Consider an evaluation of the expression tree:.



This tree can be evaluated in the order mentioned below:

CONTIGUOUS AND STRONG CONTIGUOUS EVALUATION

1. Q computes the entire subtree T_1 first using P_1. In the process all the w registers could be used.
2. After computing T_1 all registers except one are freed. Therefore T_2 is free to use $w - 1$ registers and its width is at most $w - 1$. T_2 is computed by P_2.
3. T_3 is similarly computed by P_3, whose width is $w - 2$.

Of course $A_1, \ldots, A_3$ need not necessarily be computed in this order. This is what brings the permutation π in the statement of the theorem. A program in the form $P_1 \ldots P_k J_q$ is said to be in

contiguous form. If each of the P_is is, in turn, contiguous, then the program is said to be in *strong contiguous form*. THEOREM: Every program *without stores* can be transformed into strongly