Programming Languages and Compiler Design

Optimization Using Data-flow Analysis

Yliès Falcone, Jean-Claude Fernandez

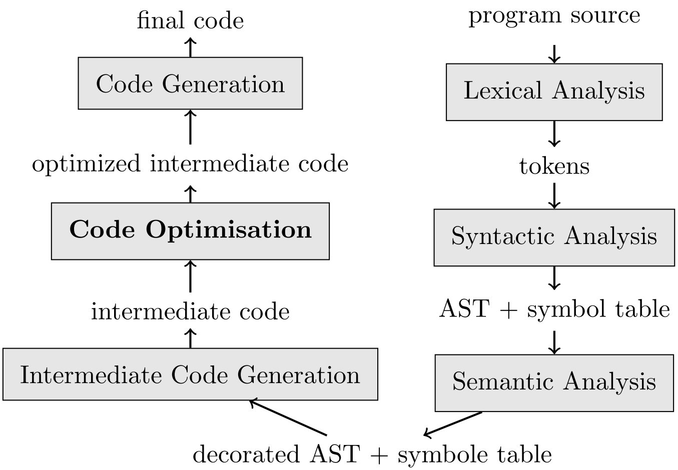
Master of Sciences in Informatics at Grenoble (MoSIG)
Univ. Grenoble Alpes
(Université Joseph Fourier, Grenoble INP)

Academic Year 2015 - 2016

Outline - Optimization Using Data-flow Analysis

Optimization

Where are we in the compiler steps?



Outline - Optimization Using Data-flow Analysis

Objectives of this chapter

Give some hints on general optimization techniques:

- **data-flow analysis**,
- register allocation,
- software pipelining,
- etc.

Describe the main data-structures used:

- Control Flow Graph (CFG),
- intermediate code (e.g., 3-address code),
- Static Single Assignment form (SSA),
- etc.

See some concrete examples.

But not a complete panorama of the whole optimization process.

Objective of the optimization phase

Improve the **efficiency** of the target code, while preserving the source semantics.

Several (antagonist) criteria:

- execution time,
- code size,
- used memory,
- energy consumption,
- etc.

$\Rightarrow$ no optimal solution, no general algorithm

A bunch of optimization techniques:

- dependent from each other,
- sometimes based on heuristics.

Two kinds of optimization techniques

Optimization independent from the target machine

- Objective: optimize the performance of the program.
- "source level" or "assembly level" pgm transformations.

Example (Optimization independent from the target machine)

- constant propagation, constant folding
- common sub-expressions elimination
- dead code elimination
- code motion

Optimization dependent from the target machine

- Objective: optimize the use of hardware resources.

Example (Optimization dependent from the target machine)

- machine instruction,
- memory hierarchy (registers, cache, pipeline, etc.).

Outline - Optimization Using Data-flow Analysis

Main principle

Input: initial intermediate code

Output: optimized intermediate code

Several steps:

1. generation of a control flow graph (CFG)
2. analysis of the CFG
3. transformation of the CFG
4. generation of the output code

Analysis and transformations

Analysis	**Transformation**
Available expressions common sub-expressions	Elimination of redundant computation
Active Variables	Elimination of useless code
Constant propagation	Replacing variables by their constant value
Induction Variable	Strength reduction
Loop Invariant	Moving the invariant code outside the loop
Dead-code elimination	Suppress useless instructions (which do not influence the execution)
Constant folding	Performing operations between constants
Copy propagation	Suppress useless variables (i.e., equal to another one or to a constant)
Algebraic simplification Strength reduction	Replace costly computations by less expensive ones

Intra-procedural 3-address code (TAC)

"High-level" assembly code:

- binary logic and arithmetic operators,
- use of temporary memory location `ti`,
- assignments to variables, temporary locations,
- a label can be assigned to an instruction,
- conditional jumps `goto`.

Example (3-address code)

- `l:  x := y op x`
- `l:  x := op y`
- `l42:  x := y`
- `l9:  goto l'`
- `l':  if x oprel y goto l''`

Basic block (BB)

Definition and how to compute them

Definition (Basic Block)

A maximal instruction sequence $S = i_1. \cdots i_n$ such that:

- S execution is never "broken" by a jump
  $\Rightarrow$ no `goto` instruction in $i_1 \cdots i_{n-1}$
- S execution cannot start somewhere in the middle
  $\Rightarrow$ no `label` in $i_2 \cdots i_n$

$\Rightarrow$ execution of a BB is "atomic".

Partitioning a 3-address code into BBs

1. computation of BB heads:
   1st inst., inst. target of a jump, inst. following a jump
2. computation of BB tails:
   last inst., inst. before a BB head

$\Rightarrow$ a single traversal of the TAC.

Control-Flow Graph (CFG)

A representation of how the execution may progress inside the TAC.

Definition (Control-Flow Graph)

A graph (V, E) such that:

$$V = \{B_i \mid B_i \text{ is a basic block}\}$$

$$\begin{aligned} E = \{(B_i, B_j) \mid & \\ & \text{"tail of } B_i \text{ is a jump to head of } B_j\text{"} \\ & \text{or} \\ & \text{"head of } B_j \text{ follows the tail of } B_i \text{ in the TAC"}\} \end{aligned}$$

Basic Block and Control-Flow Graph: example

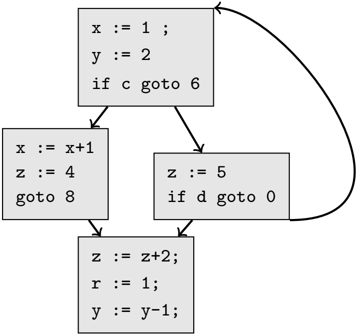
Example/Exercise

Give the Basic Blocks and CFG associated to the following TAC sequence:

```
0. x := 1
1. y := 2
2. if c goto 6
3. x := x+1
4. z := 4
5. goto 8
```

```
6. z := 5
7. if d goto 0
8. z := z+2
9. r := 1
10 y := y-1
```

- Heads of blocks: 0, 3, 6, 8.
- Tails of blocks: 2, 5, 7, 10.



Optimization techniques performed on the CFG

Two levels: local and global.

Local optimization techniques

- Computed inside each BB.
- BBs are transformed independently from each other.

Global optimizations techniques

- Computed on the CFG.
- Transformation of the CFG:
  - code motion between BBs,
  - transformation of BBs,
  - modification of the CFG edges.

Outline - Optimization Using Data-flow Analysis

Examples of local optimization techs

Initial code:

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

Examples of local optimization techs

Algebraic simplification:

```
a := x ** 2       a := x * x
b := 3            b := 3
c := x            c := x
d := c * c        d := c * c
e := b * 2        e := b << 1
f := a + d        f := a + d
g := e * f        g := e * f
```

Examples of local optimization techs

Copy propagation:

`a := x * x`	`a := x * x`
`b := 3`	`b := 3`
`c := x`	`c := x`
`d := c * c`	`d := x * x`
`e := b << 1`	`e := 3 << 1`
`f := a + d`	`f := a + d`
`g := e * f`	`g := e * f`

Examples of local optimization techs

Constant folding:

```
a := x * x          a := x * x
b := 3              b := 3
c := x              c := x
d := x * x          d := x * x
e := 3 << 1         e := 6
f := a + d          f := a + d
g := e * f          g := e * f
```

Examples of local optimization techs

Elimination of common sub-expressions:

`a := x * x`	`a := x * x`
`b := 3`	`b := 3`
`c := x`	`c := x`
`d := x * x`	`d := a`
`e := 6`	`e := 6`
`f := a + d`	`f := a + d`
`g := e * f`	`g := e * f`

Examples of local optimization techs

Copy propagation:

```
a := x * x        a := x * x
b := 3            b := 3
c := x            c := x
d := a            d := a
e := 6            e := 6
f := a + d        f := a + a
g := e * f        g := 6 * f
```

Example of local optimizations

Dead code elimination (+ strength reduction):

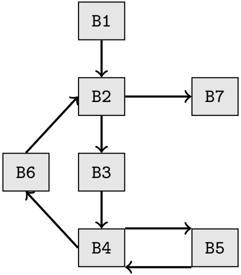
`a := x * x`	`a := x * x`	`a:= x * x`
`b := 3`		
`c := x`		
`d := a`		
`e := 6`		
`f := a + a`	`f := a + a`	`f := a << 1`
`g := 6 * f`	`g := 6 * f`	`g := 6 * f`

Local optimization: a more concrete example

Initial source program: addition of matrices

```
for (i=0 ; i < 10 ; i ++)
     for (j=0 ; j < 10 ; j++)
        S[i,j] := A[i,j] + B[i,j]
```

B1: $i := 0$
B2: if $i > 10$ goto B7
B3: $j := 0$
B4: if $j > 10$ goto B6
$\boxed{\text{B5}}$
B6: $i := i + 1$
goto B2
B7: end



Initial Block B5

```
B5:   t1 := 4 * i
      t2 := 40 * j
      t3 := t1 + t2
      t4 := A[t3]
      t5 := 4 * i
      t6 := 40 * j
      t7 := t5 + t6
      t8 := B[t7]
      t9 := t4 + t8
      t10:= 4 * i
      t11:= 40 * j
      t12:= t10 + t11
      S[t12] := t9
      j := j + 1
      goto B4
```

Optimization of B5 (1/4)

```
B5:  [t1 := 4 * i]        t8 := B[t7]
     t2 := 40 * j         t9 := t4 + t8
     t3 := t1 + t2        [t10:= 4 * i]
     t4 := A[t3]          t11:= 40 * j
     [t5 := 4 * i]        t12:= t10 + t11
     t6 := 40 * j         S[t12] := t9
     t7 := t5 + t6        j := j + 1
                          goto B4
```

The same value is assigned to temporary locations t1, t5, t10.

Optimization of B5 (2/4)

```
B5:  t1 := 4 * i
     t2 := 40 * j
     t3 := t1 + t2
     t4 := A[t3]
     t6 := 40 * j
     t7 := t1 + t6
     t8 := B[t7]
     t9 := t4 + t8
     t11:= 40 * j
     t12:= t1 + t11
     S[t12] := t9
     j := j + 1
     goto B4
```

A same value is assigned to temporary locations t2, t6, t11.

Optimization of B5 (3/4)

```
B5:  t1 := 4 * i              t8 := B[t7]
     t2 := 40 * j             t9 := t4 + t8
     t3 := t1 + t2            t12:= t1 + t2
     t4 := A[t3]              S[t12] := t9
     t7 := t1 + t2            j := j + 1
                              goto B4
```

A same value is assigned to temporary locations t3, t7, t12.

Optimization of B5 (4/4): the final code

```
B5:  t1 := 4 * i
     t2 := 40 * j
     t3 := t1 + t2
     t4 := A[t3]
     t8 := B[t3]
     t9 := t4 + t8
     S[t3] := t9
     j := j + 1
     goto B4
```

Outline - Optimization Using Data-flow Analysis

Global optimization techniques

Example (Global optimization techniques)

- constant propagation trough several basic blocks
- elimination of global redundancies
- code motion: move invariant computations outside loops
- dead code elimination

How to extend local optimization to the whole CFG?

1. Associate (local) properties to entry/exit points of BBs
   (e.g., set of active variables, set of available expressions, etc.)
2. Propagate them along CFG paths
   $\rightarrow$ enforce consistency w.r.t. the CFG structure
3. Update each BB (and CFG edges) according to these global properties.

$\Rightarrow$ a possible technique: data-flow analysis

Outline - Optimization Using Data-flow Analysis

Data-flow analysis

Static computation of data-related properties of programs.

Data-flow problem

- (local) Properties φ_i associated to some pgm locations i
- Set of data-flow equations:
  $\rightarrow$ how the φ_i's are transformed along pgm executions.
- Regarding propagation:
  - forward vs backward propagation (depending on φ_i)
  - the property can depend on its previous value on either all paths or at least one path
  - cycles inside the control flow $\Rightarrow$ fix-point equations!

Solving the equation system

- A solution of this equation system assigns "globaly consistent" values to each φ_i.
- Such a solution may not exist...
- Decidability may require abstractions and/or approximations.

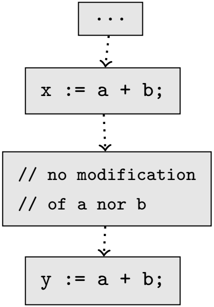
Outline - Optimization Using Data-flow Analysis

Computing available expressions

Getting intuition on examples

Let us consider expression `a + b`.

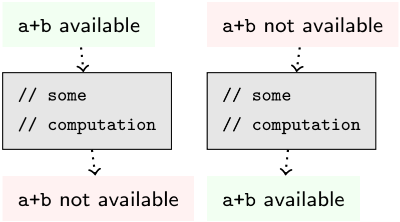
How to determine whether `a + b` is available, i.e., whether its value has been previously computed and does not need to be computed again.



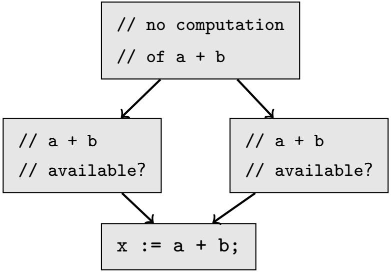
- Which computation of a+b is not needed?
- How does the information *"being previously computed"* propagate?

Computing available expressions

Getting intuition on examples



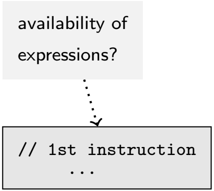
- What can make a+b not available?
- What can make a+b available?



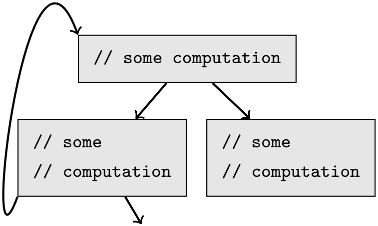
- What is needed for a+b to be available in the last block?
- How does the notion of availability depend on previous paths?

Computing available expressions

Getting intuition on examples



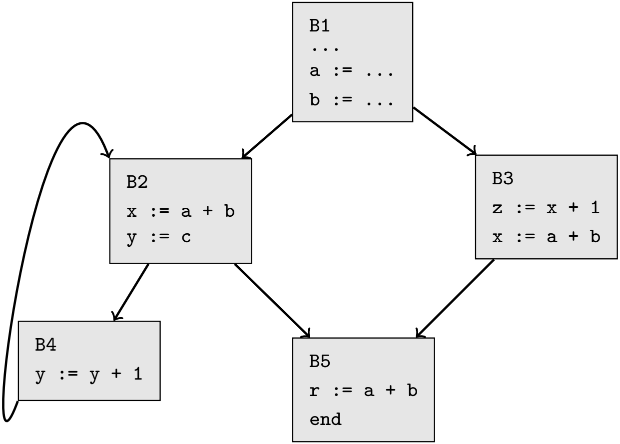
- What is available when the program starts?



- The CFG structure gives us relations between the availabilities at the entrances and exits of blocks.
- Availability at the entrance is obtained from the availability at the exits.
- How to deal with cycles?

Elimination of redundant computation with available expressions

Running example



Available expressions

We consider the set of expressions appearing in the program.

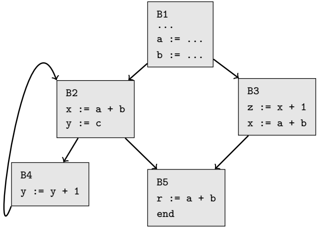
Definition (Available expression)

An expression e is available at location i iff

- it is computed at location i,
- this expression is computed on every path going from the initial location to location i,
- on each of these paths: operands of e are not modified between the last computation of e and location i

Remark We consider syntactic equality. □

Available expression



- `a + b` is available at the exit of B2 and B3, and at the entrance of B5.
- `y + 1` is not available at the exit of B4 .

Data-flow equations for available expressions (1/3)

For a basic block B, we note:

- $Kill(B)$: expressions made non available by B
  (because an operand of e is modified by B).
- $Gen(B)$: expressions made available by B
  (computed in B, operands not modified afterwards).
- $In(B)$: available expressions when entering B.
- $Out(B)$: available expressions when exiting B

$$Out(B) = (In(B) \setminus Kill(B)) \cup Gen(B) = F_b(In(B)),$$

where: F_B: transfer function of block B

Data-flow equations for available expressions (2/3)

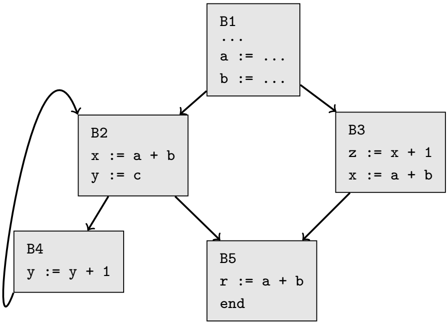
To define *Gen* and *Kill*, for a block B, we introduce local functions Gen_l and $Kill_l$:

$$
\begin{array}{|lcl|}
\hline
Gen(B) & = & Gen_l(B, \emptyset) \\
Kill(B) & = & Kill_l(B, \emptyset) \\
\hline
Gen_l(\texttt{x} := \texttt{a}; B, X) & = & Gen_l(B, X \setminus \{e' \mid x \in Used(e')\} \\
 & & \qquad\qquad \cup \{a \mid x \notin Used(a)\}) \\
Gen_l(\texttt{if b goto l}, X) & = & X \cup \{b\} \\
Gen_l(\texttt{goto l}, X) & = & X \\
Gen_l(\epsilon, X) & = & X \\
\hline
Kill_l(\texttt{x} := \texttt{a}; B, X) & = & Kill_l(B, X \cup \{e' \mid x \in Used(e')\} \\
Kill_l(\texttt{if b goto l}, X) & = & X \\
Kill_l(\texttt{goto l}, X) & = & X \\
Kill_l(\epsilon, X) & = & X \\
\hline
\end{array}
$$

Running example

Computing *Gen* and *Kill*

Gen and *Kill*



- $Gen(B1) = \emptyset$, $Kill(B1) = \{\texttt{a+b}\}$
- $Gen(B2) = \{\texttt{a+b}\}$, $Kill(B2) = \{\texttt{y+1}\}$
- $Gen(B3) = \{\texttt{a+b},\ \texttt{x+1}\}$, $Kill(B3) = \emptyset$
- $Gen(B4) = \emptyset$, $Kill(B4) = \{\texttt{y+1}\}$
- $Gen(B5) = \{\texttt{a+b}\}$, $Kill(B5) = \emptyset$

Data-flow equations for available expressions (2/3)

How to compute $In(b)$?

- if b is the initial block:
$$In(b) = \emptyset$$
- if b is not the initial block: An expression e is available at its entry point iff it is available at the exit point of all predecessor of b in the CFG.
$$In(b) = \bigcap_{b' \in Pre(b)} Out(b')$$

$\Rightarrow$ forward data-flow analysis along the CFG paths.

How to deal with cycles inside the CFG? fix-points computation!

We want to as much available expressions possible: greatest fix-point.

Using data-flow equations to compute available expressions

Initialisation

It is a forward analysis $\Rightarrow$ initialisation concerns the $In(b)$ sets.

- Initialise $In(B_{\mathrm{init}})$ to $\emptyset$: there is no available expression at the beginning of the program.
- Initialise $In(b)$, for $b \neq B_{\mathrm{init}}$ to the maximal element, i.e., to the set of all expressions.

Iteration until stabilisation

Iterate the following steps until stabilisation, i.e., when the $In(b)$ sets are the same as in the previous step. At stabilisation, one has found the (greatest) fix-point.

1. Compute $Out(b)$ sets using $Out(b) = (In(b) \setminus Kill(b)) \cup Gen(b)$.
2. Compute the (new) $In(b)$ sets using $In(b) = \bigcap_{b' \in Pre(b)} Out(b')$.

Suppressing redundant computations

We look at all computed available expressions in each block (*In*(*b*) sets).

Is the computation of an available expression redundant?

Let *e* be an available expression at the entry of a basic block *b*.
If the two following conditions are met:

- *e* appears in *b*, and
- none of the operand of *e* is assigned from the beginning of the block until the use of *e*

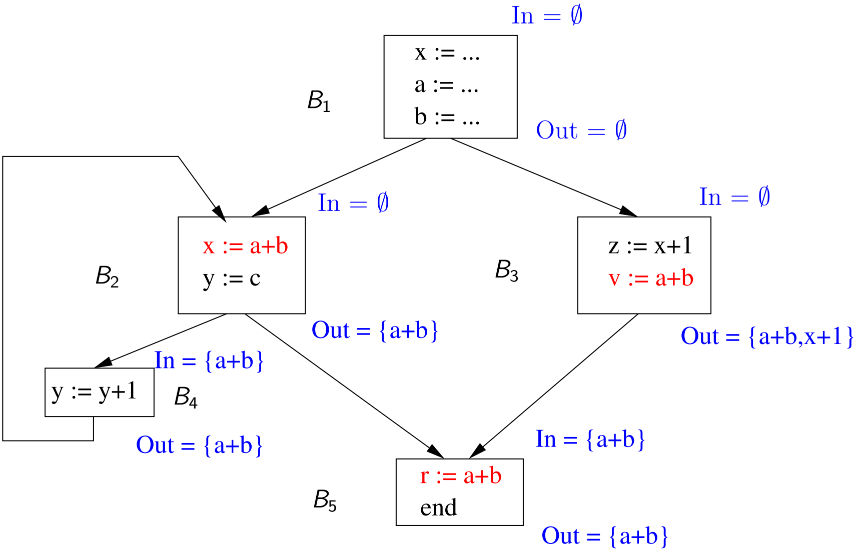
$\Rightarrow$ the computation of *e* is *redundant*.

Suppressing redundant computation

Let *e* be an expression which computation is redundant in a block *b*.

- Introduce a new variable, say *u*.
- Using a backward analysis from *b*, locate each occurence of `x := e`, and replace it with $\left\{ \begin{array}{l} \texttt{u := e;} \\ \texttt{x := u;} \end{array} \right.$
- Replace the occurence of *e* where its computation is redundant.

Back to the example of available expressions



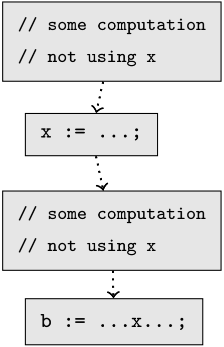
Outline - Optimization Using Data-flow Analysis

Computing active variables

Getting intuition on examples

Let us consider a variable `x` appearing in a program.

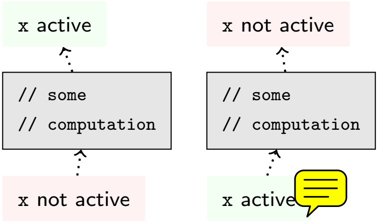
How to determine whether `x` is *active*, i.e., whether its value is needed for computation.



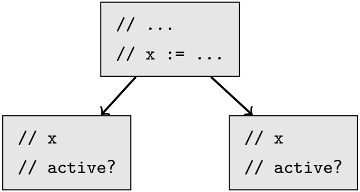
- Where is the value of `x` needed?
- How does the information *"being needed"* propagate?

Computing active variables

Getting intuition on examples



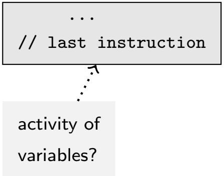
- What can make `x` not active?
- What can make `x` active?



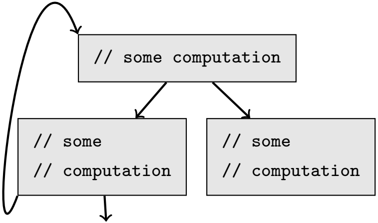
- What is needed for `x` to be active in the first block?
- How does the notion of activity depend on previous paths? (note, previous refers to a successor block)

Computing active variables

Getting intuition on examples



- What is active when the program terminates?



- The CFG structure gives us relations between the availabilities at the exits and entrances of blocks.
- Activity at exits is obtained from the activity at the entrance.
- How to deal with cycles?

Active Variables

Objective: remove useless instructions.

Definition (Active Variable)

A variable `x` is active at location `i` if it is *used* in at least one CFG-path going from `i` to `j`, where `j` is:

- either a final instruction, or
- an assignment to `x`.

Definition (Useless instructions)

An instruction `x := e` at location `i` is useless if `x` is inactive at location `i`.

Remark Used means "in the right-hand side of an assignment or in a branch condition". □

Data-flow analysis for inactive variables

We compute the set of active variables...

Local analysis

Gen(b) is the set of variables `x` s.t. `x` is used in block b, and, in this block, any assignment to `x` happens after the (first) use of `x`.

Kill(i) is the set of variables `x` assigned in block b.

Global analysis

Backward analysis, $\exists$ a CFG-path (least solution).

- $In(b) = (Out(b) \setminus Kill(b)) \cup Gen(b)$.
- $$\mathtt{Out}(b) = \begin{cases} \emptyset & \text{if } b \text{ is final,} \\ \bigcup_{b' \in Succ(b)} In(b') & \text{otherwise.} \end{cases}$$

Computation of functions *Gen* and *Kill*

Recursively defined on the syntax of a basic block B:

$$\texttt{B} ::= \varepsilon \mid \texttt{B ; x := a} \mid \texttt{B ; if b goto l} \mid \texttt{B ; goto l}$$

$$
\begin{array}{|lcl|}
\hline
Gen(B) & = & Gen_l(\texttt{B}, \emptyset) \\
Kill(B) & = & Kill_l(\texttt{B}, \emptyset) \\
\hline
Gen_l(\texttt{B ; x := a}, X) & = & Gen_l(\texttt{B}, X \setminus \{\texttt{x}\} \cup Used(\texttt{a})) \\
Gen_l(\texttt{B ; if b goto l}, X) & = & Gen_l(\texttt{B}, X \cup Used(\texttt{b})) \\
Gen_l(\texttt{B ; goto l}, X) & = & Gen_l(\texttt{B}, X) \\
Gen_l(\varepsilon, X) & = & X \\
\hline
Kill_l(\texttt{B ; x := a}, X) & = & Kill_l(\texttt{B}, X \cup \{\texttt{x}\}) \\
Kill_l(\texttt{B ; if b goto l}, X) & = & Kill_l(\texttt{B}, X) \\
Kill_l(\texttt{B ; goto l}, X) & = & Kill_l(\texttt{B}, X) \\
Kill_l(\varepsilon, X) & = & X \\
\hline
\end{array}
$$

$Used(\texttt{e})$: set of variables appearing in expression e.

Removal of useless instructions

1. Compute the sets $In(B)$ and $Out(B)$ of active variables at entry and exit points of each block.
2. Let $F : Code \times 2^{Var} \rightarrow Code$
   $F(B, X)$ is the code obtained when removing useless assignments inside B, assuming that variables of X are active at the end of B execution.

$$
\begin{array}{rcll}
F(\mathtt{B}\ ;\ \mathtt{x}\ := \mathtt{a}, X) & = & \left\{ \begin{array}{ll} F(B, X) & \text{if } x \notin X \\ F(B, (X \setminus \{x\}) \cup \mathit{Used}(a)); x := a & \text{if } x \in X \end{array} \right. \\
F(\mathtt{B}\ ;\ \mathtt{if\ b\ goto\ l}, X) & = & F(B, X \cup \mathit{Used}(b)); \mathtt{if\ b\ goto\ l} \\
F(\mathtt{B}\ ;\ \mathtt{goto\ l}, X) & = & F(B, X); \mathtt{goto\ l} \\
F(\epsilon, X) & = & \epsilon
\end{array}
$$

3. Replace each block B by $F(B, Out(B))$.

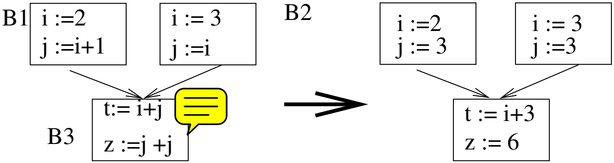
Remark This transformation may produce new inactive variables... $\square$

Outline - Optimization Using Data-flow Analysis

Constant propagation

A variable is constant at location `l` if its value at this location can be computed at compilation time.

Example (Constant propagation principle)



- At exit point of B1 and B2, `i` and `j` are constants.
- At entry point of B3, `i` is not constant, `j` is constant.

Constant propagation: the lattice

Intuitively, the property is an “abstraction” of the memory:

- Each variable takes its value in $D = \mathbb{N} \cup \{\top, \bot\}$, where:
  - $\top$ means “non constant value”
  - $\bot$ means “no information”

Defining the lattice:

- Partial order relation $\leq$:

  if $v \in D$ then $\bot \leq v$ and $v \leq \top$.

- The least upper bound $\sqcup$:

  for $x \in D$ and $v_1, v_2 \in \mathbb{N}$

$x \sqcup \top = \top$	$x \sqcup \bot = x$	$v_1 \sqcup v_2 = \top$ if $v_1 \neq v_2$	$v_1 \sqcup v_1 = v_1$

Remark Relation $\leq$ is extended to functions $Var \to D$

$f1 \leq f2$ iff $\forall x : f1(x) \leq f2(x)$. □

Constant propagation: data-flow equations

A basic block = sequence of assignments

$$\texttt{b} \quad ::= \quad \epsilon \mid \texttt{x:=e ; b}$$

- Property at location `l` is a function $Var \to D$.
- Forward analysis:

$$\begin{array}{rcl} In(b) & = & \begin{cases} \lambda x.\bot & \text{if } b \text{ is initial,} \\ \sqcup_{b' \in Pred(b)} Out(b') & \text{otherwise} \end{cases} \\ Out(b) & = & F_b(In(b)) \end{array}$$

Transfer function F_b by syntactic induction

$$\begin{array}{rcl} F_{\texttt{x:=e ; b}}(f) & = & F_{\texttt{b}}(f[x \mapsto f(e)]) \quad \text{(assuming variable initialization)} \\ F_\epsilon(f) & = & f \end{array}$$

Program transformation

$\forall$ block b, $f \in In(b), f(\texttt{e}) = v \Rightarrow$ `x:=e` replaced by `x:=v`

Remark We assume that variables are properly initialized. □

Outline - Optimization Using Data-flow Analysis

Recall data-flow analysis

Static computation of data-related properties of programs:

Data-flow problem

- (local) Properties φ_i associated to some pgm locations i
- Set of data-flow equations:
  $\rightarrow$ how φ_i are transformed along pgm execution
- Regarding propagation:
  - forward vs backward propagation (depending on φ_i)
  - cycles inside the control flow $\Rightarrow$ fix-point equations!

Solving the equation system

- A solution of this equation system assigns "globaly consistent" values to each φ_i.
- Such a solution may not exist...
- Decidability may require abstractions and/or approximations

Generalization

Data-flow properties are expressed as finite sets associated to entry/exit points of basic blocks: *In*(b), *Out*(b).

Forward analysis

- property is "false" ($\perp$) at entry of initial block
- $Out(b) = F_b(In(b))$
- *In*(b) depends on *Out*(b'), where $b' \in Pred(b)$
  ($\sqcap$ for "$\forall$ paths", $\sqcup$ for "$\exists$ path")

Backward analysis

- property is "false" ($\perp$) at exit of final block
- $In(\mathrm{b}) = F_b\ (Out(\mathrm{b}))$
- *Out*(b) depends on *In*(b'), where $b' \in Succ(b)$
  ($\sqcap$ for "$\forall$ paths", $\sqcup$ for "$\exists$ path")

Data-flow equations: forward analysis

Forward analysis, least fix-point	$In(b) = \begin{cases} \bot & \text{if } b \text{ is initial} \\ \bigsqcup_{b' \in Pre(b)} Out(b') & \text{otherwise.} \end{cases}$ $Out(b) = F_b(In(b))$
Forward analysis, greatest fix-point	$In(b) = \begin{cases} \bot & \text{if } b \text{ is initial} \\ \bigsqcap_{b' \in Pre(b)} Out(b') & \text{otherwise.} \end{cases}$ $Out(b) = F_b(In(b))$

Data-flow equations: backward analysis

Backward analysis, least fix-point	$Out(b) = \begin{cases} \bot & \text{if } b \text{ is final} \\ \bigsqcup_{b' \in Succ(b')} In(b') & \text{otherwise.} \end{cases}$ $In(b) = F_b(Out(b))$
Backward analysis, greatest fix-point	$Out(b) = \begin{cases} \bot & \text{if } b \text{ is final} \\ \bigsqcap_{b' \in Succ(b)} In(b') & \text{otherwise.} \end{cases}$ $In(b) = F_b(Out(b))$

Solving the data-flow equations (1/2)

Let $(E, \leq)$ a partial order.

- For $X \subseteq E, a \in E$:
  - a is an upper bound of X if $\forall x \in X : x \leq a$,
  - a is a lower bound of X if $\forall x \in X : a \leq x$.
- The least upper bound (*lub*, $\sqcup$) is the smallest upper bound.
- The greatest lower bound (*glb*, $\sqcap$) is the largest lower bound.
- $(E, \leq)$ is a lattice if any two elements of E admit a *lub* and a *glb*.
- A function $f : 2^E \to 2^E$ is increasing if:

$$\forall X, Y \subseteq E \quad X \leq Y \ \Rightarrow \ f(X) \leq f(Y)$$

- $X = \{x_0, x_1, \dots x_n, \dots\} \subseteq E$ is an (increasing) chain if $x_0 \leq x_1 \leq \dots x_n \leq \dots$
- A function $f : 2^E \to 2^E$ is ($\sqcup$-)continuous if $\forall$ increasing chain X, $f(\sqcup X) = \sqcup f(X)$

Solving the data-flow equations (2/2)

Fix-point equation: solution?

- properties are finite sets of expressions $\mathcal{E}$
- $(2^{\mathcal{E}}, \subseteq)$ is a complete lattice
  $\bot$: least element (aka minimum), $\top$: greatest element (aka maximum)
  $\sqcap$: greatest lower bound (aka supremum), $\sqcup$: least upper bound (aka infimum)
- data-flow equations are defined on monotonic and continuous operators ($\cup$, $\cap$) on $(2^{\mathcal{E}}, \subseteq)$.
- Kleene and Tarski theorems:
  - the set of solutions is a complete *lattice*
  - the greatest (resp. least) solution can be obtained by successive iterations w.r.t. the greatest (resp. least) element of $2^{\mathcal{E}}$

$$\mathrm{lfp}(f) = \sqcup\{f^i(\bot) \mid i \in \mathbb{N}\} \qquad \mathrm{gfp}(f) = \sqcap\{f^i(\top) \mid i \in \mathbb{N}\}$$

Outline - Optimization Using Data-flow Analysis

Summary

Code Optimization using data-flow analysis

- Objective: produce a semantically equivalent version of 3-address code that is *optimized*.
- Focused on optimizations independent from the target machine:
  - available expressions for the suppression of redundant computations,
  - active variables for useless assignments
  - constant propagation for replacing variables with their (fixed) values.