Programming Languages and Compiler Design

Programming Language Semantics
Compiler Design Techniques

Yassine Lakhnech & Laurent Mounier

{lakhnech,mounier}@imag.fr
http://www-verimag.imag.fr/lakhnech
http://www-verimag.imag.fr/~mounier.

Master of Sciences in Informatics at Grenoble (MoSIG)
Grenoble Universités
(Université Joseph Fourier, Grenoble INP)
Code Optimization
Objective (of this chapter)

- give some indications on general optimization techniques:
  - data-flow analysis
  - register allocation
  - software pipelining
  - etc.
- describe the main data structures used:
  - control flow graph
  - 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
(e.g.: a real compiler, for a modern processor)
Objective of the optimization phase

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

Efficiency $\rightarrow$ several (antagonist) criteria

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

$\Rightarrow$ no optimal solution, no general algorithm

$\Rightarrow$ a bunch of optimization techniques:

- inter-dependant each others
- sometimes heuristic based
Two kinds of optimizations

Independant from the target machine

“source level” or “assembly level” pgm transformations:
- dead code elimination
- constant propagation, constant folding
- code motion
- common subexpressions elimination
- etc.

Dependant from the target machine

optimize the use of the hardware resources:
- machine instruction
- memory hierarchy (registers, cache, pipeline, etc.)
- etc.
Overview

1. Introduction
2. Some optimizations independant from the target machine
3. Some optimizations dependant from the target machine
Some optimizations independant from the target machine
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
Intraprocedural 3-address code (TAC)

“high-level” assembly code:

- binary logic and arithmetic operators
- use of temporary memory location \( t_i \)
- assignments to variables, temporary locations
- a label is assigned to each instruction
- conditional jumps \( \text{goto} \)

Examples:

- \( l: \ x := y \ op \ x \)
- \( l: \ x := \ op \ y \)
- \( l: \ x := y \)
- \( l: \ \text{goto} \ l' \)
- \( l: \ \text{if} \ x \ \text{oprel} \ y \ \text{goto} \ l' \)
Basic block (BB)

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 basic bloc is atomic

Partition of a 3-address code BBs:

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

$\Rightarrow$ a single traversal of the TAC
Control Flow Graph (CFG)

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

→ a graph \((V, E)\) such that:

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

\[
E = \{ (B_i, B_j) \mid \text{“last inst. of } B_i \text{ is a jump to 1st inst of } B_j \text{” } \lor \text{“1st inst of } B_j \text{ follows last inst of } B_i \text{ in the TAC”} \}
\]
Example

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
Optimizations performed on the CFG

Two levels:

Local optimizations:
- computed inside each BB
- BBs are transformed independent each others

Global optimizations:
- computed on the CFG
- transformation of the CFG:
  - code motion between BBs
  - transformation of BBs
  - modification of the CFG edges
Local optimizations

- algebraic simplification, strength reduction
  → replace costly computations by less expensive ones
- copy propagation
  → suppress useless variables
  (i.e., equal to another one, or equal to a constant)
- constant folding
  → perform operations between constants
- common subexpressions
  → suppress duplicate computations
  (already computed before)
- dead code elimination
  → suppress useless instructions
  (which do not influence pgm execution)
Example of local optimizations

Initial code:

\[
\begin{align*}
a & := x ** 2 \\
b & := 3 \\
c & := x \\
d & := c \times c \\
e & := b \times 2 \\
f & := a + d \\
g & := e \times f
\end{align*}
\]
Example of local optimizations

Algebraic simplification:

\[
\begin{align*}
a & := x \times 2 & a & := x \times x \\
b & := 3 & b & := 3 \\
c & := x & c & := x \\
d & := c \times c & d & := c \times c \\
e & := b \times 2 & e & := b \ll 1 \\
f & := a + d & f & := a + d \\
g & := e \times f & g & := e \times f
\end{align*}
\]
Example of local optimizations

Copies propagation:

\[
\begin{align*}
a & := x \times x & a & := x \times x \\
b & := 3 & b & := 3 \\
c & := x & c & := x \\
d & := c \times c & d & := x \times x \\
e & := b \ll 1 & e & := 3 \ll 1 \\
f & := a + d & f & := a + d \\
g & := e \times f & g & := e \times f
\end{align*}
\]
Example of local optimizations

Constant folding:

\[
\begin{align*}
a & := x \times x & a & := x \times x \\
b & := 3 & b & := 3 \\
c & := x & c & := x \\
d & := x \times x & d & := x \times x \\
e & := 3 \ll 1 & e & := 6 \\
f & := a + d & f & := a + d \\
g & := e \times f & g & := e \times f
\end{align*}
\]
Example of local optimizations

Elimination of common subexpressions:

\begin{align*}
a & := x \times x & b & := 3 \\
b & := 3 & c & := x \\
c & := x & d & := x \times x \\
d & := a & e & := 6 \\
e & := 6 & f & := a + d \\
f & := a + d & g & := e \times f \\
g & := e \times f
\end{align*}
Example of local optimizations

Copies propagation:

\[
\begin{align*}
a &:= x \ast x \\
b &:= 3 \\
c &:= x \\
d &:= a \\
e &:= 6 \\
f &:= a + d \\
g &:= e \ast f
\end{align*}
\]
Example of local optimizations

Dead code elimination (+ strength reduction):

\[
\begin{align*}
    a & := x \times x & a & := x \times x & a & := x \times x \\
    b & := 3 \\
    c & := x \\
    d & := a \\
    e & := 6 \\
    f & := a + a & f & := a + a & f & := a \ll 1 \\
    g & := 6 \times f & g & := 6 \times f & g & := 6 \times f
\end{align*}
\]
Local optimization: a more concrete example

Initial source program: addition of matrices

\[
\begin{align*}
\text{for} \ (i=0 \ ; \ i < 10 \ ; \ i++) \\
\text{for} \ (j=0 \ ; \ j < 10 \ ; \ j++) \\
\end{align*}
\]

Basic blocks:

- **B1**: \(i := 0\)
- **B2**: if \(i > 10\) goto B7
- **B3**: \(j := 0\)
- **B4**: if \(j > 10\) goto B6
- **B5**
- **B6**: \(i := i + 1\)
  
  goto B2
- **B7**: end
Control Flow Graph
Inital Block B5

B5:

\[
\begin{align*}
& t1 := 4 \times i \\
& t2 := 40 \times j \\
& t3 := t1 + t2 \\
& t4 := A[t3] \\
& t5 := 4 \times i \\
& t6 := 40 \times j \\
& t7 := t5 + t6 \\
& t8 := B[t7] \\
& t9 := t4 + t8 \\
& t10 := 4 \times i \\
& t11 := 40 \times j \\
& t12 := t10 + t11 \\
& S[t12] := t9 \\
& j := j + 1 \\
& \text{goto B4}
\end{align*}
\]
**Optimization of B5 (1/4)**

B5:  
\[
\begin{align*}
  t1 & := 4 \times i \\
  t2 & := 40 \times j \\
  t3 & := t1 + t2 \\
  t4 & := A[t3] \\
  t5 & := 4 \times i \\
  t6 & := 40 \times j \\
  t7 & := t5 + t6 \\
  t8 & := B[t7] \\
  t9 & := t4 + t8 \\
  t10 & := 4 \times i \\
  t11 & := 40 \times j \\
  t12 & := t10 + t11 \\
  S[t12] & := t9 \\
  j & := j + 1 \\
  \text{goto B4}
\end{align*}
\]

A same value is assigned to temporary locations t1, t5, t10
Optimization of B5 (2/4)

B5:

\[
\begin{align*}
  t1 & := 4 \times i \\
  t2 & := 40 \times j \\
  t3 & := t1 + t2 \\
  t4 & := A[t3] \\
  t6 & := 40 \times j \\
  t7 & := t1 + t6
\end{align*}
\]

\[
\begin{align*}
  t8 & := B[t7] \\
  t9 & := t4 + t8 \\
  t11 & := 40 \times j \\
  t12 & := t1 + t11 \\
  S[t12] & := t9 \\
  j & := j + 1
\end{align*}
\]

goto B4

A same value is assigned to temporary locations t2, t6, t11
Optimization of B5 (3/4)

B5:  \[ t_1 := 4 \times i \]
  \[ t_2 := 40 \times j \]
  \[ t_3 := t_1 + t_2 \]
  \[ t_4 := A[t_3] \]
  \[ t_7 := t_1 + t_2 \]

\[ t_8 := B[t_7] \]
\[ t_9 := t_4 + t_8 \]
\[ t_{12} := t_1 + t_2 \]
\[ S[t_{12}] := t_9 \]
\[ j := j + 1 \]
\[ \text{goto B4} \]

A same value is assigned to temporary locations \( t_3, t_7, t_{12} \)
Optimization of B5 (4/4): the final code obtained

B5:  
\[
\begin{align*}
\text{t1} & := 4 * i \\
\text{t2} & := 40 * j \\
\text{t3} & := \text{t1} + \text{t2} \\
\text{t4} & := A[\text{t3}] \\
\text{t8} & := B[\text{t3}] \\
\text{t9} & := \text{t4} + \text{t8} \\
S[\text{t3}] & := \text{t9} \\
\text{j} & := \text{j} + 1 \\
\text{goto B4}
\end{align*}
\]
Global optimizations
**Global optimization: the principle**

Typical examples of global optimizations:
- constant propagation through several basic blocks
- elimination of global redundancies
- code motion: move invariant computations outside loops
- dead code elimination

How to “extrapolate” local optimizations to the whole CFG?

1. associate (local) properties to entry/exit points of BBs
   (set of active variables, set of available expressions, etc.)

2. propagate them along CFG paths
   → enforce consistency w.r.t. the CFG structure

3. update each BB (and CFG edges) according to these global properties

⇒ a possible technique: data-flow analysis
Data-flow analysis

Static computation of data related properties of programs

• (local) properties $\varphi_i$ associated to some pgm locations $i$

• set of data-flow equations:
  → how $\varphi_i$ are transformed along pgm execution

Rks:
• forward vs backward propagation (depending on $\varphi_i$)
• cycles inside the control flow ⇒ fix-point equations!

• a solution of this equation system:
  → assigns “globally consistent” values to each $\varphi_i$
Rk: such a solution may not exist . . .

• decidability may require abstractions and/or approximations
Example: elimination of redundant computations

An expression $e$ is **redundant** at location $i$ iff
- it is computed at location $i$
- this expression is computed on every path going from the initial location to location $i$
  
  **Rk:** we consider here syntactic equality
- on each of these paths: operands of $e$ are not modified between the last computation of $e$ and location $i$

Optimization is performed as follows:

1. computation of available expressions (data-flow analysis)
2. $x := e$ is redundant at loc $i$ if $e$ is available at $i$
3. $x := e$ is replaced by $x := t$
   (where $t$ is a temp. memory containing the value of $e$)
Elimination of redundant computation: an example
Data-flow equations for available expressions (1/2)

For a basic block $b$, we note:

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

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

$F_b =$ transfer function of block $b$
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 each predecessor of $b$ in the CFG
  \[ In(b) = \bigcap_{b' \in Pre(b)} Out(b') \]

⇒ forward data-flow analysis along the CFG paths

Q: cycles inside the CFG ⇒ fix-points computations greatest $vd$ least solutions?
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 \((\text{lub}, \sqcup)\) is the smallest upper bound
- The great lower bound \((\text{glb}, \sqcap)\) is the largest lower bound
- \((E, \leq)\) is a lattice if every subset of \(E\) admits a \text{lub} and a \text{glb}.

- A function \(f : 2^E \rightarrow 2^E\) is monotonic if:
  \[
  \forall X, Y \subseteq E \quad X \leq Y \quad \Rightarrow \quad f(X) \leq f(Y)
  \]

- \(X = \{x_0, x_1, \ldots x_n, \ldots\} \subseteq E\) is an (increasing) chain if \(x_0 \leq x_1 \leq \ldots x_n \leq \ldots\)

- A function \(f : 2^E \rightarrow 2^E\) is (\(\sqcup\)-)continuous if \(\forall\) increasing chain \(X, f(\sqcup X) = \sqcup f(X)\)
Fix-point equation: solution?

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

\[
\text{lfp}(f) = \sqcup\{f^i(\perp) | i \in \mathbb{N}\} \quad \text{gfp}(f) = \sqcap\{f^i(\top) | i \in \mathbb{N}\}
\]
Back to the example

```plaintext
x := ...
a := ...
b := ...
y := y + 1
x := a + b
y := c
z := x + 1
v := a + b
r := a + b
end
```

\[ \text{In} = \{a + b\} \]

\[ \text{Out} = \{a + b\} \]

\[ \text{In} = 0 \]

\[ \text{Out} = 0 \]
Generalization

- Data-flow properties are expressed as finite sets associated to entry/exit points of basic blocks: $\text{In}(b)$, $\text{Out}(b)$

- For a forward analysis:
  - property is “false” ($\bot$) at entry of initial block
  - $\text{Out}(b) = F_b(\text{In}(b))$
  - $\text{In}(b)$ depends on $\text{Out}(b')$, where $b' \in \text{Pred}(b)$ ($\cap$ for “∀ paths”, $\cup$ for “∃ path”)

- For a backward analysis:
  - property is “false” ($\bot$) at exit of final block
  - $\text{In}(b) = F_b(\text{Out}(b))$
  - $\text{Out}(b)$ depends on $\text{In}(b')$, where $b' \in \text{Succ}(b)$
### Data-flow equations: forward analysis

<table>
<thead>
<tr>
<th>Forward analysis, least fix-point</th>
<th>[ \text{In}(b) = \begin{cases} \bot &amp; \text{if } b \text{ is initial} \ \bigsqcup_{b' \in \text{Pre}(b)} \text{Out}(b') &amp; \text{otherwise.} \end{cases} ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \text{Out}(b) = F_b(\text{In}(b)) ]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Forward analysis, greatest fix-point</th>
<th>[ \text{In}(b) = \begin{cases} \bot &amp; \text{if } b \text{ is initial} \ \bigcap_{b' \in \text{Pre}(b)} \text{Out}(b') &amp; \text{otherwise.} \end{cases} ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \text{Out}(b) = F_b(\text{In}(b)) ]</td>
<td></td>
</tr>
</tbody>
</table>
### Data-flow equations: backward analysis

<table>
<thead>
<tr>
<th>Backward analysis, least fix-point</th>
<th>[\text{Out}(b) = \begin{cases} \bot &amp; \text{if } b \text{ is final} \ \bigsqcup_{b' \in \text{Succ}(b')} \text{In}(b') &amp; \text{otherwise.} \end{cases}]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[\text{In}(b) = F_b(\text{Out}(b))]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Backward analysis, greatest fix-point</th>
<th>[\text{Out}(b) = \begin{cases} \bot &amp; \text{if } b \text{ is final} \ \bigsqcup_{b' \in \text{Succ}(b')} \text{In}(b') &amp; \text{otherwise.} \end{cases}]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[\text{In}(b) = F_b(\text{Out}(b))]</td>
</tr>
</tbody>
</table>
Active Variable

• A variable \( x \) is inactive at location \( i \) if it is not used in every CFG-path going from \( i \) to \( j \), where \( j \) is:
  • either a final instruction
  • or an assignement to \( x \).

• An instruction \( x := e \) at location \( i \) is useless if \( x \) is inactive at location \( i \).

⇒ useless instuctions can be removed . . .

Rk: used means

“in a right-hand side 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 assignement 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)

- \(\text{Out}(b) = \bigcup_{b' \in \text{Succ}(b)} \text{In}(b')\)
- \(\text{In}(b) = (\text{Out}(b) \setminus \text{Kill}(b)) \cup \text{Gen}(b)\)
- \(\text{Out}(b) = \emptyset\) if \(b\) is final.
Computation of functions *Gen* and *Kill*

Recursively defined on the syntax of a basic bloc *B*:

\[
B ::= \varepsilon \mid B ; x := a \mid B ; \text{if } b \text{ goto } l \mid B ; \text{goto } l
\]

<table>
<thead>
<tr>
<th>( Gen(B) )</th>
<th>= ( Gen_l(B, \emptyset) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Kill(B) )</td>
<td>= ( Kill_l(B, \emptyset) )</td>
</tr>
<tr>
<td>( Gen_l(B ; x := a, X) )</td>
<td>= ( Gen_l(B, X \setminus {x} \cup \text{Used}(a)) )</td>
</tr>
<tr>
<td>( Gen_l(B ; \text{if } b \text{ goto } l, X) )</td>
<td>= ( Gen_l(B, X \cup \text{Used}(b)) )</td>
</tr>
<tr>
<td>( Gen_l(B ; \text{goto } l, X) )</td>
<td>= ( Gen_l(B, X) )</td>
</tr>
<tr>
<td>( Gen_l(\varepsilon, X) )</td>
<td>= ( X )</td>
</tr>
</tbody>
</table>

| \( Kill_l(B ; x := a, X) \)                                 | = \( Kill_l(B, X \cup \{x\}) \) |
| \( Kill_l(B ; \text{if } b \text{ goto } l, X) \)          | = \( Kill_l(B, X) \)         |
| \( Kill_l(B ; \text{goto } l, X) \)                        | = \( Kill_l(B, X) \)         |
| \( Kill_l(\varepsilon, X) \)                               | = \( X \)                  |

\( \text{Used}(\varepsilon) \): set of variables appearing in expression \( \varepsilon \)
Removal of useless instructions

1. Compute the sets \( \text{In}(B) \) and \( \text{Out}(B) \) of active variables at entry and exit points of each blocks.

2. Let \( F : \text{Code} \times 2^{\text{Var}} \rightarrow \text{Code} \)
   
   \( F(b, X) \) is the code obtained when removing useless assignments inside \( b \), assuming that variables of \( X \) are active at the end of \( b \) execution.

   \[
   F(B; x := a, X) = \begin{cases} 
   F(B, X) & \text{if } x \notin X \\
   F(B, (X \setminus \{x\}) \cup \text{Used}(a)); x := a & \text{if } x \in X
   \end{cases}
   \]

   \[
   F(B; \text{if } b \text{ goto } l, X) = F(B, X \cup \text{Used}(b)); \text{if } b \text{ goto } l
   \]

   \[
   F(B; \text{goto } l, X) = F(B, X); \text{goto } l
   \]

   \[
   F(\epsilon, X) = \epsilon
   \]

3. Replace each block \( B \) by \( F(B, \text{Out}(B)) \).

Rk: this transformation may produce new inactive variables . . .
**Constant propagation**

**Example:**

B1

```
  i := 2
  j := i + 1
```

B2

```
  i := 3
  j := i
```

B3

```
  t := i + j
  z := j + j
```

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

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

• 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 $v_1 \neq v_2$ | $v_1 \sqcup v_1 = v_1$ |

Rk: relations $\leq$ is extended to functions $Var \to D$

$$f_1 \leq f_2 \text{ iff } \forall x. f_1(x) \leq f_2(x)$$
Constant propagation: data-flow equations

- property at location 1 is a function \( Var \rightarrow D \).
- Forward analysis:

\[
\begin{align*}
In(b) &= \begin{cases} 
\lambda x. \bot & \text{if } b \text{ is initial,} \\
\bigsqcup_{b' \in Pred(b)} Out(b') & \text{otherwise}
\end{cases} \\
Out(b) &= F_b(In(b))
\end{align*}
\]

Transfer function \( F_b \)?

A basic block = sequence of assignments

\[
b ::= \epsilon | x := e ; b
\]

\( F_b \) defined by syntactic induction:

\[
\begin{align*}
F_{x := e ; b}(f) &= F_b(f[x \mapsto f(e)]) \quad \text{(assuming variable initialization)} \\
F_{\epsilon}(f) &= f
\end{align*}
\]

Pgm transformation:

\[\forall \text{ block } b, f \in In(b), f(e) = v \Rightarrow x := e \text{ replaced by } x := v\]
**Exercise**

Constant propagation can be viewed as abstraction of the standard semantics where expressions values are interpreted in some other domain $D$

1. Write this abstract semantics for the `while` language in an operational style (relation $\rightarrow^*$)

2. Define a program transformation which removes useless computations (i.e., computations between constant operands)

3. Give the equations which express the correctness of this transformation
Another example of data-flow analysis

A computation of an expression $e$ can be anticipated at loc. $p$ iff:

- all paths from $p$ contains a location $p_i$ s.t. $e$ is computed at $p_i$
- $e$ operands are not modified between $p$ and $p_i$

Example:

```c
if (x>0)  
    x = i + j;
else
    repeat y = (i + j) * 2; x := x+1 ; until x>10
```

can be changed to

```c
tmp = i + j;  
if (x>0)  
    x = tmp;
else
    repeat y = tmp * 2; x := x+1 ; until x>10
```

Application: moving invariants outside loops
Interprocedural analysis

main()
{
    int i, j;
    void f()
    {
        int x, y;
        y = i + j; x = y;
    }
    i = 0;
    f();
    j = 1;
}

• a dedicated basic block \( B_{call} \) for the call instruction

\[ In(B_{call}) = In(B_{f_{in}}), \quad Out(B_{call}) = Out(B_{f_{out}}) \]

Rks:

• static binding is be assumed

• parameters ?

Exercice: Computation of active variables
Control-flow analysis

→ retrieve program control structures from the CFG ?
Application: loop identification

⇒ use of graph-theoretic notions:
  • dominator, dominance relation
  • strongly connected components

Rk1: most loops are easier to identify at syntactic level, but:
  • use of goto instruction still allowed in high-level languages
  • optimization performed on intermediate representations (e.g., CFG)

Rk2: other approaches can be used to identify loops . . .
Loop identification

Node $B_1$ is a dominator of $B_2 (B_2 \leq B_1)$ iff every path from the entry block to $B_2$ goes through $B_1$. $Dom(B) = \{B_i|B_i \leq B\}$.

An edge $(B_1, B_2)$ is a loop back edge iff $B_2 \leq B_1$

To find “natural loops”:
1. find a back edge $(B_1, B_2)$
2. find $Dom(B_2)$
3. find blocks $B_i \in Dom(B_2)$ s.t. there is a path from $B_i$ to $B_2$ not containing $B_1$. 
Some machine level optimization techniques
Register Allocation

Pb:

- expression operands are much efficiently accessed when lying in registers (instead of RAM)
- the “real” number of registers is finite (and usually small)

⇒ register allocation techniques:

- assigns a register to each operand (variable, temporary location)
- performs the memory exchange (LD, ST) when necessary
- optimality ?

Several existing techniques:

- optimal code generation for arithmetic expressions
- graph-coloring techniques (more general case)
- etc.
Code generation for arithmetic expressions: example

code generation for \((a+b) - (c - (d+e))\)
with 2 registers, and instruction format = \(OP \ Ri, \ Ri, \ X\) (where \(X=Ri\) or \(X=M[x]\))

Solution 1: one register needs to be saved

\[
\begin{align*}
&LD \ R0, \ M[a] \\
&ADD \ R0, \ R0, \ M[b] \\
&LD \ R1, \ M[d] \\
&ADD \ R1, \ R1, \ M[e] \\
&ST \ R1, \ M[t1] \quad \text{! register R1 needs to be saved ...} \\
&LD \ R1, \ M[c] \\
&SUB \ R1, \ R1, \ M[t1] \\
&SUB \ R0, \ R0, \ R1
\end{align*}
\]

Solution 2: no register to save

\[
\begin{align*}
&LD \ R0, \ M[c] \\
&LD \ R1, \ M[d] \\
&ADD \ R1, \ R1, \ M[e] \\
&SUB \ R0, \ R0, \ R1 \\
&LD \ R1, \ M[a] \\
&ADD, \ R1, \ R1, \ M[b] \\
&SUB, \ R1, \ R1, \ R0
\end{align*}
\]
**Code generation for arithmetic expressions: principle**

Evaluation of $e_1 \ op \ e_2$, assuming:

- $r$ registers are available, evaluation of $e_i$ requires $r_i$ registers
- Instruction format is “op reg, reg, ad” where “ad” is a register or a memory location

Several cases:

- $r_1 > r_2$:
  - after evaluation of $e_1$, $r_1 - 1$ registers available
  - $r_1 - 1 \geq r_2 \Rightarrow r_1 - 1$ registers are enough for $e_2$
  - $\Rightarrow r_1 - r$ register allocations are required

- $r_1 = r_2$:
  - after evaluation of $e_1$, $r_1 - 1$ registers available
  - $r_1 - 1 < r_2$, $\Rightarrow r_2 (= r_1)$ registers required for $e_2$
  - $\Rightarrow r_1 + 1 - r$ register allocations are required

- $r_1 < r_2$:
  - after evaluation of $e_1$, $r_1 - 1$ registers available
  - $r_1 - 1 < r_2$, $\Rightarrow r_2 (> r_1)$ registers required for $e_2$
  - $\Rightarrow r_2 + 1 - r$ register allocations are required
  - $r_2 - r$ allocations are enough if $e_2$ is evaluated first!
A two-phase algorithm

Step 1: each AST node is labeled with the number of registers required for its evaluation

\[ rNb : Aexp \rightarrow \mathbb{N} \] 

\( rNb(e) \) is the number of registers required to evaluate \( e \)

\[ rNb(e) = \begin{cases} 
1 & \text{if } e \text{ is a left leaf} \\
0 & \text{if } e \text{ is a right leaf} 
\end{cases} \]

\[ rNb(e_1 \text{ op } e_2) = \begin{cases} 
\max(rNb(e_1), rNb(e_2)) & \text{if } rNb(e_1) \neq rNb(e_2) \\
rNb(e_1) + 1 & \text{if } rNb(e_1) = rNb(e_2) 
\end{cases} \]

Step 2: “optimal” code generation using these labels (exercice)

→ for a binary node \( e_1 \text{ op } e_2 \):

• evaluate the more register demanding sub-expression first
• write the result in a register \( Ri \) (save one if necessary)
• evaluate the other sub-expression, write the result in a register \( Rj \)
• generate \( \text{OP, Ri, Ri, Rj} \)
A more general technique

1. Intermediate code is generated assuming \( \infty \) numbers of “symbolic” registers \( S_i \)

2. Assign a real register \( R_i \) to each symbolic register s.t.
   - if \( R_i \) is assigned to \( S_i \), \( R_j \) is assigned to \( S_j \)
   - then \( \text{Lifetime}(S_i) \cap \text{lifetime}(S_j) \neq \emptyset \Rightarrow R_i \neq R_j \)

where \( \text{Lifetime}(S_i) \): sequences of pgm location where \( S_i \) is active

How to ensure this condition?

Collision graph \( G_C \):
- Nodes denote lifetime symbolic registers: \( N_i = (S_i, \text{Lifetime}(S_i)) \)
- Edges are the set \( \{(S_1, L_1), (S_2, L_2) \mid L_1 \text{ and } L_2 \text{ overlap}\} \)

\( \Rightarrow \) register allocation with \( k \) real register = \( k \)-coloring problem of \( G_C \)

(i.e., assign a distinct colour to each pair of adjacent nodes)
Example 1

\[
\begin{align*}
S_1 & := e_1 \\
S_2 & := e_2 \\
\quad \ldots \quad & \\
\quad \ldots & S_2 \quad \ldots & S_2 \text{ used} \\
S_3 & := S_1 + S_2 & S_1 \text{ and } S_2 \text{ used} \\
\quad \ldots & \\
S_4 & := S_1 \times 5 & S_1 \text{ used} \\
\quad \ldots & S_4 \quad \ldots & S_4 \text{ used} \\
\quad \ldots & S_3 \quad \ldots & S_3 \text{ used}
\end{align*}
\]

Collision Graph:

\[
\begin{tikzpicture}
  \node[shape=circle,fill=green] (S1) at (0,0) {S1};
  \node[shape=circle,fill=blue] (S2) at (1,0) {S2};
  \node[shape=circle,fill=green] (S3) at (0,-1) {S3};
  \node[shape=circle,fill=blue] (S4) at (1,-1) {S4};
  \draw (S1) -- (S2);
  \draw (S3) -- (S4);
\end{tikzpicture}
\]

Can be colored with 2 colors \( \Rightarrow \) 2 real registers are enough.
**k-coloring in practice? (1)**

When \(k > 2\), this problem is NP-complete . . .

An efficient heuristic:

Repeat:

- if exists a node \(N\) of \(G_C\) such that \(\text{degree}(N) < k\)
  - \(N\) can receive a distinct colour from all its neighbours
    - remove \(N\) (and corresponding edges) from \(G_C\) and push it on a stack \(S\)
  - else (\(G_C\) is assumed to be non \(k\)-colourable)
    - choose a node \(N\) (1)
    - remove \(N\) from \(G_C\) (2)

until \(G_C\) is empty

While \(S\) is not empty

- pop a node from \(S\)
  - add it to \(G\), give it a colour not used by one of its neighbours

**Rk:** this algo may sometimes miss \(k\)-colorable graphs . . .
$k$-coloring in practice? (2)

What happens when there is no node of degree $< k$?

1. **choose** a node $N$ to remove:
   → high degree in $G_C$, not corresponding to an inner loop, etc.

2. **remove** node $N$:
   → save a register into memory before (*register spilling*)

Several attempts to improve this algorithm:

**node coalescing:**

\[ S_1 := S_2, \text{Lifetime}(S_1) \cap \text{Lifetime}(S_2) = \emptyset \]

⇒ nodes associated to $S_1$ and $S_2$ could be merged

**pb:** it increases the graph degree . . .

**lifetime splitting:**

long lifetime increases the graph degree

⇒ split it into several parts . . .

**pb:** where to split?
Instruction scheduling

Motivation: exploit the instruction parallelism provided in many target architectures (e.g., VLIW processors, instruction pipeline, etc.)

Pbs:
• possible data dependancies between consecutive instructions (e.g., \( x := 3 \); \( y := x+1 \))
• possible resource conflicts between consecutive instructions (ALU, co-processors, bus, etc.)
• consecutive instructions may require various execution cycles
• etc.

⇒ Main technique: change the initial instruction sequence (instruction scheduling)
• preserve the initial pgm semantics
• better exploit the hardware resources

Rks: “loop unrolling” and “expression tree reduction” may help . . .
Dependency Graph

Data dependencies:
→ execution order of 2 instructions should be preserved in the following situation:

Read After Write (RAW) : inst. 2 read a data written by inst. 1
Write After Read (WAR) : inst. 2 write a data read by inst. 1
Write After Write (WAW) : inst. 2 write a data written by inst. 1

Dependency graph $G_D$

• nodes = \{ instructions \}
• edges = \{(i_1, d, i_2) | there is a dependency $d$ from $i_1$ to $i_2$\}

Rk: if we consider a basic block, $G_D$ is a directed acyclic graph.

Any topological sort of $G_D$ leads to a valid result (w.r.t. pgm semantics).
This sort can be influenced by several factors:

• the resources used by the instruction (∃ a static reservation table)
• the number of cycles it requires (latency)
• etc.
Example

1. Draw the dependency graph $G_D$ associated to the following program

2. Give a topological sort of $G_D$

3. Rewrite this program with a “maximal” parallelism

1. $a := x+1$
2. $x := 2+y$
3. $y := z+1$
4. $t := a*b$
5. $v := a*c$
6. $v := 3+t$
Software pipelining (overview . . .)

Idea: exploit the parallelism between instructions of distinct loop iterations

\[
\begin{align*}
\text{for } k \text{ in } 1 \ldots N \text{ loop} \\
& r := T[k] ; \quad \text{– inst. A} \\
& x := x + r ; \quad \text{– inst. B} \\
& T[k] := x ; \quad \text{– inst. C}
\end{align*}
\]

end loop

Assumptions: 3 cycles per instruction, 1 cycle delay when no dependencies

- Initial exec. sequence: A(1), B(1), C(1), A(2), B(2), C(2), \ldots A(k), B(k), C(k)
  \Rightarrow 7 \text{ cycles / iteration}

- “Pipelined exec. sequence”: A(1), A(2), A(3), B(1), B(2), B(3), C(1), C(2), C(3), \ldots
  \Rightarrow 3 \text{ cycles / iteration !}

(real life) pbs:

- N not always divisible by the number of instruction in the loop body
  \[
  \begin{align*}
  \text{for } k \text{ in } 1 \text{ to } N-2 \text{ step 3 loop } & A(k) ; \ A(k+1) ; \ A(k+2) \ldots
  \end{align*}
  \]

- high latency instruction in the loop body
- possible overhead when k is not “large enough”
- . . .
Code Generation
Overview

1. Introduction

2. The “M” Machine

3. Code generation for basic while

4. Extension 1: blocks and procedures

5. Extension 2: some OO features
Main issues for code generation

- input: (well-typed) source pgm AST
- output: machine level code

Expected properties for the output:

- compliance with the target machine instruction set, architecture, memory access, OS, ...
- correctness of the generated code semantically equivalent to the source pgm
- optimality w.r.t. non-functional criteria execution time, memory size, energy consumption, ...
A pragmatic approach

```
AST
↓
intermediate code generation
↓
Intermediate Representation 1
↓
...
↓
Intermediate Representation n
↓
(final) code generation
↓
target machine code
```

optimization(s)

optimization(s)
Intermediate Representations

• Abstractions of a real target machine
  • generic code level instruction set
  • simple addressing modes
  • simple memory hierarchy

• Examples
  • a “stack machine”
  • a “register machine”
  • etc.

Rk: other intermediate representations are used in the optimization phases . . .
The “M” Machine

- Machine with (unlimited) registers Ri
  special registers: program counter PC, frame pointer FP, stack pointer SP, register R0 (contains always 0)
- Instructions, addresses, and integers take 4 bytes in memory
- Address of variable x is E – offx where:
  - E = address of the environment definition of x
  - offx = offset of x within this environment
    (staticaly computed, stored in the symbol table)
- Addressing modes:
  Ri, val (immediate), Ri +/- Rj, Ri +/- offset
- usual arithmetic instructions OPER: ADD, SUB, AND, etc.
- usual (conditional) branch instructions BRANCH: BA, BEQ, BGT, etc.
# Instruction Set

<table>
<thead>
<tr>
<th>instruction</th>
<th>informal semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPER Ri, Rj, Rk</td>
<td>Ri ← Rj oper Rk</td>
</tr>
<tr>
<td>OPER Ri, Rk, val</td>
<td>Ri ← Rj oper val</td>
</tr>
<tr>
<td>CMP Ri, Rj</td>
<td>Ri-Rj (set cond flags)</td>
</tr>
<tr>
<td>LD Ri, [adr]</td>
<td>Ri ← Mem[adr]</td>
</tr>
<tr>
<td>ST Ri, [adr]</td>
<td>Mem[adr] ← Ri</td>
</tr>
<tr>
<td>BRANCH label</td>
<td>if cond then PC ← label else PC ← PC + 4</td>
</tr>
<tr>
<td>CALL label</td>
<td>branch to the procedure labelled with label</td>
</tr>
<tr>
<td>RET</td>
<td>end of procedure</td>
</tr>
</tbody>
</table>
The while language

\[
p ::= d ; c \\
d ::= \text{var } x | d ; d \\
s ::= x := a | s ; s | \text{if } b \text{ then } s \text{ else } s | \text{while } b \text{ s} \\
a ::= n | x | a + a | a * a | ... \\
b ::= a = a | b \text{ and } b | \text{not } b | ...
\]

Rk: terms are well-typed
\[\rightarrow\] distinction between boolean and arithmetic expr.

Exo: Give the “M Machine” code for the following terms:
1. \[y := x+42 \ast (3+y)\]
2. \[\text{if (not } x=1 \text{) then } x := x+1 \]
   \[\text{else } x := x-1; y := x;\]
**Functions for Code Generation**

\[ \text{GCStm} : \text{Stm} \rightarrow \text{Code}^* \]
\[ \text{GCStm}(s) \text{ computes the code } C \text{ corresponding to statement } s. \]

\[ \text{GCAExp} : \text{Exp} \rightarrow \text{Code}^* \times \text{Reg} \]
\[ \text{GCAExp}(e) \text{ returns a pair } (C, i) \text{ where } C \text{ is the code allowing to 1. compute the value of } e, \text{ 2. store it in } R_i. \]

\[ \text{GCBExp} : \text{BExp} \times \text{Label} \times \text{Label} \rightarrow \text{Code}^* \]
\[ \text{GCBExp}(b, l_{\text{true}}, l_{\text{false}}) \text{ produces code } C \text{ allowing to compute the value of } b \text{ and branch to label } l_{\text{true}} \text{ when this value is “true” and to } l_{\text{false}} \text{ otherwise.} \]
Auxiliary functions

AllocRegister : → Reg
allocate a new register Ri

newLabel : → Labels
produce a new label

GetOffset : Var → N
returns the offset corresponding to the specified name

∥ denotes concatenation for Code sequences.
<table>
<thead>
<tr>
<th>GCStmt ( (x := e) )</th>
<th>( = )</th>
<th>Let ( (C, i) = \text{GCAExp}(e) ), ( k = \text{GetOffset}(x) )</th>
<th>( \text{in} )</th>
<th>( C \parallel ST\ Ri, [FP-k] )</th>
</tr>
</thead>
</table>

| GCStmt \( (c_1 ; c_2) \) | \( = \) | Let \( C_1 = \text{GCStmt}(c_1) \), \( C_2 = \text{GCStmt}(c_2) \) | \( \text{in} \) | \( C_1 \parallel C_2 \) |
\textit{GCStm (2)}

\[
\text{GCStm}\left(\text{while e c}\right) = \text{Let } \begin{align*}
\text{lb} &= \text{newLabel}(), \\
\text{ltrue} &= \text{newLabel}(), \\
\text{lfalse} &= \text{newLabel}() \\
\end{align*}
\text{in } \begin{align*}
\text{lb} : &\parallel \\
\text{GCBEexp}\left(\text{e, ltrue, lfalse}\right) &\parallel \\
\text{ltrue} : &\parallel \\
\text{GCStm}\left(\text{c}\right) &\parallel \\
\text{BA lb} &\parallel \\
\text{lfalse} : &
\end{align*}
\]
GCStm (3)

\[
\text{GCStm}(\text{if } e \text{ then } c_1 \text{ else } c_2) = \text{ Let } \begin{align*}
& \text{lnext = newLabel()} , \\
& \text{ltrue = newLabel()} , \\
& \text{lfalse = newLabel()} \\
\text{in } \begin{align*}
& \text{GCBExp}(e, \text{ltrue}, \text{lfalse}) || \\
& \text{ltrue:} \\
& \text{GCStm}(c_1) || \\
& \text{BA lnext ||} \\
& \text{lfalse:} || \\
& \text{GCStm}(c_2) || \\
& \text{lnext:}
\end{align*}
\]
<table>
<thead>
<tr>
<th>Expr</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCAExp(x)</td>
<td>Let ( i = \text{AllocRegister()} )</td>
</tr>
<tr>
<td></td>
<td>( k = \text{GetOffset}(x) )</td>
</tr>
<tr>
<td></td>
<td>( \text{in} ) ((\text{LD} \ Ri, [FP-k]), i))</td>
</tr>
<tr>
<td>GCAExp(n)</td>
<td>Let ( i = \text{AllocRegister()} )</td>
</tr>
<tr>
<td></td>
<td>( \text{in} ) ((\text{ADD} \ Ri, R0, n), i))</td>
</tr>
<tr>
<td>GCAExp(e₁ + e₂)</td>
<td>Let ((C₁, i₁) = \text{GCAExp}(e₁), )</td>
</tr>
<tr>
<td></td>
<td>((C₂, i₂) = \text{GCAExp}(e₂), )</td>
</tr>
<tr>
<td></td>
<td>( k = \text{AllocRegister()} )</td>
</tr>
<tr>
<td></td>
<td>( \text{in} ) ((C₁</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{GCBExp} (e_1 = e_2, l_{\text{true}}, l_{\text{false}}) )</td>
<td>Let ( (C_1, i_1) = \text{GCAExp}(e_1) ), ( (C_2, i_2) = \text{GCAExp}(e_2) ), in ( C_1 \parallel C_2 \parallel \text{CMP} ; R_i_1, R_i_2 ; \text{BEQ} ; l_{\text{true}} ; \text{BA} ; l_{\text{false}} )</td>
</tr>
<tr>
<td>( \text{GCBExp}(e_1 ; \text{et} ; e_2, l_{\text{true}}, l_{\text{false}}) )</td>
<td>Let ( l = \text{newLabel}() ), in ( \text{GCBExp}(e_1, l, l_{\text{false}}) \parallel l : | \text{GCBExp}(e_2, l_{\text{true}}, l_{\text{false}}) )</td>
</tr>
<tr>
<td>( \text{GCBExp}(\text{NOT} ; e, l_{\text{true}}, l_{\text{false}}) )</td>
<td>( \text{GCBExp}(e, l_{\text{false}}, l_{\text{true}}) )</td>
</tr>
</tbody>
</table>
Exercises

- code obtained for
  - \( y := x + 42 \times (3 + y) \)
  - if (not \( x = 1 \)) then \( x := x + 1 \)
    else \( x := x - 1 \); \( y := x \);
- add new statements (e.g, repeat)
- add new operators (e.g, \( b \ ? \ e_1 \ : \ e_2 \))
Extension 1: blocks
Blocks

Syntax

\[ S ::= \ldots | \textbf{begin} \ D_V ; \ S \textbf{end} \]
\[ D_V ::= \textbf{var} \ x | D_V ; D_V \]

Rk: variables are uninitialized and assumed to be of type \texttt{Int}

Problems raised for code generation

→ to preserve scoping rules:

- local variables should be \textit{visible} inside the block
- their \textit{lifetime} should be limited to block execution

Possible locations to store local variables

→ registers vs \textit{memory}
Storing local variables in memory - Example 1

begin
    var x ; var y ; var z ;
    ...
end

- a memory environment is associated to each declaration \( D_v \)
- register \( FP \) contains the address of the current environment
- (static) offsets are associated to each local variable
Storing local variables in memory - Example 2

begin
  var x ; var y ; <s1>
  begin
    var x ; var z ; <s2>
    end ;
    <s3>
  end
end

- entering/leaving a block → allocate/de-allocate a mem. env.
- nested block env. have to be linked together: “Ariane link”

⇒ a stack of memory environments . . . (≈ operational semantics)
Structure of the memory

1: global variables
2: execution stack, \( SP = \text{last occupied address} \)
3: heap (for dynamic allocation)
**Code generation for variable declarations**

\[
\text{SizeDecl: } D_V \rightarrow \mathbb{N}
\]

\[\text{SizeDecl}(d) \text{ computes the size of declarations } d\]

<table>
<thead>
<tr>
<th>\text{SizeDecl}(\text{var } x)</th>
<th>=</th>
<th>4 \text{ (} x \text{ of type } \text{Int})</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{SizeDecl}(d_1 ; d_2)</td>
<td>=</td>
<td>\text{Let } v_1 = \text{SizeDecl}(d_1), v_2 = \text{SizeDecl}(d_2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\text{in } v_1 + v_2</td>
</tr>
</tbody>
</table>

Yassine Lakhnech, Sémantique Start C3 C4 – p.88/115
\[ \text{GCStm}(\text{begin } d ; s ; \text{end}) = \text{Let } \text{size} = \text{SizeDecl}(d), \]
\[ C = \text{GCStm}(s) \]
\[ \text{in } \text{ADD, SP, SP, -4 } \parallel \]
\[ \text{ST FP, [SP]} \parallel \]
\[ \text{ADD FP, SP, 0 } \parallel \]
\[ \text{ADD SP, SP, size } \parallel \]
\[ C \parallel \]
\[ \text{ADD SP, FP, 0 } \parallel \]
\[ \text{LD FP, [SP]} \parallel \]
\[ \text{ADD SP, SP, 4 } \parallel \]
With the help of some auxiliary functions . . .

<table>
<thead>
<tr>
<th>prologue(size)</th>
<th>epilogue</th>
<th>push <code>register(Ri)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD SP, SP, -4</td>
<td>ADD SP, FP, 0</td>
<td>ADD SP, SP, -4</td>
</tr>
<tr>
<td>ST FP, [SP]</td>
<td>LD FP, [SP]</td>
<td>ST Ri, [SP]</td>
</tr>
<tr>
<td>ADD FP, SP, 0</td>
<td>ADD SP, SP, +4</td>
<td></td>
</tr>
<tr>
<td>ADD SP, SP, size</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{GCStm}(\text{begin } d; s; \text{end}) = \begin{cases} 
\text{size} = \text{SizeDecl}(d), \\
C = \text{GCStm}(s) \\
\text{in } \text{Prologue}(\text{size}) \ || \\
C \ || \\
\text{Epilogue} 
\end{cases}
\]
Access to variables from a block?

... begin
  var ...
  \texttt{x := ...}
end

What is the memory address of \texttt{x}?

- If \texttt{x} is a local variable (w.r.t the current block)
  \[\text{adr}(x) = FP + \text{GetOffset}(x)\]

- If \texttt{x} is a non local variable
  \[\text{it is defined in a “nesting” memory env. } E\]
  \[\text{adr}(x) = \text{adr}(E) + \text{GetOffset}(x)\]
  \text{adr}(E) can be accessed through the “Ariane link” . . .
Access to non local variables

The number $n$ of indirections to perform on the “Ariane link” depends on the “distance” between:

- the nesting level of the current block : $p$
- the nesting level of the target environment : $r$

More precisely:

- $r \leq p$
- $n = p - r$

$\Rightarrow n$ can be statically computed . . .
Example

begin
  var x ; /* env. E1, nesting level = 1 */
  begin
    var y ; /* env. E2, nesting level = 2 */
    begin
      var z ; /* env. E3, nesting level = 3 */
      x := y + z /* s, nesting level = 3 */
    end
  end
end

From statement s:
  • no indirection to access to z
  • 1 indirection to access to y
  • 2 indirections to access to x
Code generation for variable access

1. the nesting level $r$ of each identifier $x$ is computed during type-checking;
2. it is associated to each occurrence of $x$ in the AST (via the symbol table)
3. function GCStm keeps track of the current nesting level $p$
   (incremented/decremented at each block entry/exit)

$\text{adr}(x)$ is obtained by executing the following code:

- if $r = p$:
  $$\text{FP} + \text{GetOffset}(x)$$

- if $r < p$:
  $$\text{LD Ri, [FP]}
  \text{LD Ri, [Ri]} \quad (p - r - 1) \text{ times}
  \text{Ri} + \text{GetOffset}(x)$$
Example (ctn’d)

begin
    var x ; /* env. E1, nesting level = 1 */
    begin
        var y ; /* env. E2, nesting level = 2 */
        begin
            var z ; /* env. E3, nesting level = 3 */
            x := y + z /* s, nesting level = 3 */
        end
    end
end

LD  R1, [FP]  ! R1 = adr(E2)
LD R2, [R1 + offy]  ! R2 = y
LD R3, [FP + offz]  ! R3 = z
ADD R4, R2, R3  ! R4 = y+z
LD  R5, [FP]
LD  R5, [R5]  ! R5 = adr(E1)
ST R4, [R5 + offx]  ! x = y + z

Code generated for statement s
Extension 2: Procedures
Syntax

Procedure declarations:

\[ DP ::= \text{proc } p \left( FP_L \right) \text{ is } S ; \quad DP | \epsilon \]

\[ FP_L ::= x, FP_L | \epsilon \]

Statements:

\[ S ::= \cdots | \begin{align*} \text{begin } & DV ; DP ; \quad S \text{ end} | \text{call } p(EPL) \\ EP_L ::= & AExp, EP_L | \epsilon \end{align*} \]

\[ FP_L: \text{ formal parameters list} ; \quad EP_L: \text{ effective parameters list} \]

\[ \text{Rk: we assume here value-passing of integer parameters . . .} \]
Example

var z ;

proc p1 () is
  begin
    proc p2(x, y) is z := x + y ;
    z := 0 ;
    call p2(z+1, 3) ;
  end

proc p3 (x) is
  begin
    var z ;
    call p1() ; z := z+x ;
  end

call p3(42) ;
Main issues for code generation

Procedure $P$ is calling procedure $Q$ . . .

Before the call:

- set up the memory environment of $Q$
- evaluate and “transmit” the effective parameters
- switch to the memory environment of $Q$
- branch to first instruction of $Q$

During the call:

- access to local/non local procedures and variables
- access to parameter values

After the call:

- switch back to the memory environment of $P$
- resume execution to the $P$ instruction following the call
**Access to non-local variables**

```plaintext
proc main is
begin
  /* definition env. of p */
  var x ;
  proc p() is x:=3 ;
  proc q() is
    begin
      var x ;
      proc r() is call p() ;
      call r() ;
    end ;
    call q() ;
  end
end

**Static binding** ⇒ when p is executed:

- acces to the memory env. of main = definition environment of the callee, **static link**
- acces to the memory env. of r = memory environment of the caller, **dynamic link**
```
Information exchanged between callers and callees?

- parameter values
- return address
- address of the caller memory environment (dynamic link)
- address of the callee environment definition (static link)

This information should be stored in a memory zone:

- dynamically allocated
  (exact number of procedure calls cannot be foreseen at compile time)
- accessible from both parties
  (those address could be computed by the caller and the callee)

⇒

inside the execution stack, at well defined offsets w.r.t FP
A possible “protocol” between the two parties

Before the call, the caller:

• evaluates the effective parameters
• pushes their values
• pushes the static link of the callee
• pushes the return address, and branch to the callee’s 1st instruction

when it begins, the callee:

• pushes FP (dynamic link)
• assigns SP to FP (memory env. address)
• allocates its local variables on the stack

when it ends, the callee:

• de-allocates its local variables
• restores FP to caller’s memory env. (dynamic link)
• branch to the return address, and pops it from the stack

After the call, the caller

• de-allocates the static link and parameters
Organization of the execution stack

Addresses, from the callee:
- local variables: FP+d, d<0
- dynamic link: FP
- return address: FP+4
- static link: FP+8
- parameters: FP+d, d>=12

Addresses, from the caller:
- memory environment of the caller
- memory environment of the callee
- local variables
- parameters
- dynamic link
- return address
- static link
## Memory environment of the callee

<table>
<thead>
<tr>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Loc. var(n)</strong></td>
<td>(\leftarrow \text{SP}, \text{FP}-4\times n)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Loc. var(_1)</strong></td>
<td>(\leftarrow \text{FP})</td>
</tr>
<tr>
<td><strong>Dynamic link</strong></td>
<td>(\leftarrow \text{FP})</td>
</tr>
<tr>
<td><strong>Return address</strong></td>
<td>(\leftarrow \text{FP}+4)</td>
</tr>
<tr>
<td><strong>Static link</strong></td>
<td>(\leftarrow \text{FP}+8)</td>
</tr>
<tr>
<td><strong>Param(_n)</strong></td>
<td>(\leftarrow \text{FP}+12)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Param(_1)</strong></td>
<td>(\leftarrow \text{FP}+8+4\times n)</td>
</tr>
</tbody>
</table>
**Code generation for a procedure declaration**

**GCProc**: $D_P \rightarrow \text{Code}^*$

$\text{GCStm}(dp)$ computes the code $C$ corresponding to procedure declaration $dp$.

\[
\begin{align*}
\text{GCProc}(\text{proc } p \ (FP_L) \ \text{is } s \ \text{end}) &= \text{Let} \\
C &= \text{GCStm}(s) \\
in & \text{Prologue}(0) \ || \\
C & || \\
\text{Epilogue}
\end{align*}
\]

\[
\begin{align*}
\text{GCProc}(\text{proc } p \ (FP_L) \ \text{is begin } dv ; dp ; s \ \text{end}) &= \text{Let} \\
\text{size} &= \text{SizeDecl}(dv), \\
C &= \text{GCStm}(s) \\
in & \text{Prologue}(\text{size}) \ || \\
C & || \\
\text{Epilogue}
\end{align*}
\]

**Rk:** this function is applied to each procedure declaration
Prologue & Epilogue

Prologue (size):

push (FP)  ! dynamic link
ADD FP, SP, 0  ! FP := SP
ADD SP, SP, -size  ! loc. variables allocation

Epilogue:

ADD SP, FP, 0  ! SP := FP, loc. var. de-allocation
LD FP, [SP]  ! restore FP
ADD SP, SP, +4
RET  ! return to caller

RET:

LD PC, [SP]  // ADD SP, SP, +4
**Code Generation for a procedure call**

Four steps:
1. evaluate and push each effective parameter
2. push the static link of the callee
3. push the return address and branch to the callee
4. de-allocate the parameter zone

\[
\text{GCStmt}(\text{call } p \ (ep)) \ = \ \text{Let} \ (C, \ size) = \text{GCParam}(ep) \\text{ in} \\
\quad C \ || \\
\quad \text{Push} \ (\text{StaticLink}(p)) \ || \\
\quad \text{CALL} \ p \ || \\
\quad \text{ADD SP, SP, size+4}
\]

\[
\text{CALL} \ p: \\
\quad \text{ADD R1, PC, +4} \ // \text{Push (R1)} \ // \text{BA} \ p
\]

\textbf{Rk:} procedures code is \textit{flattened} . . .
Parameters evaluation

\[ \text{GCP}aram : EPL \rightarrow \text{Code}^* \times \mathbb{N} \]

\[ \text{GCStm}(ep) = (c, n) \quad \text{where} \quad c \text{ is the code to evaluate and “push” each effective parameter of } ep \text{ and } n \text{ is the size of pushed data.} \]

\begin{align*}
\text{GCP}aram(\varepsilon) & = (\varepsilon, 0) \\
\text{GCP}aram(a; ep) & = \text{Let} \\
& \quad (Ca, i) = \text{GCAexp}(a), \\
& \quad (C, \text{size}) = \text{GCP}aram(ep) \\
& \quad \text{in} \\
& \quad (Ca \ || \ \text{Push}(R_i) \ || \ C, 4 + \text{size})
\end{align*}
Static link and non local variable access?

- A global (unique) name is given to each identifier:

  ```
  proc Main is
    proc P1 (...) is
      ...
    proc Pn (...) is
      begin
        var x ...
      end
  \rightarrow x is named Main.P_1 \ldots P_n.x
  ```

- This notation induces a partial order:

  \[(Main.P_1 \ldots P_n \leq Main.P'_1 \ldots P'_n) \iff (n \leq n' \text{ and } \forall k \leq n.P_k = P'_k)\]

- For an identifier \(x = Main.P_1 \ldots P_n.x\),

  \(x^* = Main.P_1 \ldots P_n\) is the definition environment of \(x\)

- For any identifier \(x\) (variable or procedure), procedure \(P\) can access \(x\) iff \(x^* \leq P\).
Examples

• A variable $x$ declared in $P$ can be accessed from $P$ since $x^\bullet = P$ (hence $x^\bullet \leq P$).

• If $g$ and $x$ are declared in $f$, then $x$ can be accessed from $g$ since $x^\bullet = f$ and $f \leq g$.

• If $x$ and $f_1$ are declared in $Main$, $f_2$ is declared in $f_1$, then $x$ can be accessed from $f_2$ since $x^\bullet = Main$, $f_2 = Main.f_1.f_2$ ($x^\bullet \leq f_2$)

• If $p_1$ and $p_2$ are both declared in $Main$, $x$ is declared in $p_1$, then $x$ cannot be accessed from $p_2$, since $x^\bullet = Main.p_1$ and $Main.p_1 \not\leq Main.p_2$
Code Generation for accessing (non-) local identifiers

$d_x$: offset of $x$ (variables or parameters) in its definition environment ($x^\bullet$)

$P$: current procedure

<table>
<thead>
<tr>
<th>Condition</th>
<th>$x = \text{variable or parameter}$</th>
<th>$x = \text{procedure}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x^\bullet = P$</td>
<td>$\text{adr}(x) = \text{FP} + d_x$</td>
<td>$\text{SL}(x) = \text{FP}$</td>
</tr>
<tr>
<td>$x^\bullet &lt; P$</td>
<td>$\text{n-k-1 indirections}$</td>
<td>$\text{n-k-1 indirections}$</td>
</tr>
<tr>
<td>$x = M.P_1 \cdots P_k$</td>
<td>$\text{LD } R, [\text{FP}+8]$</td>
<td>$\text{LD } R, [\text{FP}+8]$</td>
</tr>
<tr>
<td>$P = M.P_1 \cdots P_k \cdots P_n$</td>
<td>$\text{LD } R, [R+8] \times (n-k-1)$</td>
<td>$\text{LD } R, [R+8] \times (n-k-1)$</td>
</tr>
<tr>
<td></td>
<td>$\text{adr}(x) = R + d_x$</td>
<td>$\text{SL}(x) = R$</td>
</tr>
</tbody>
</table>
Back to the 1st example

var z ;

proc p1 () is
    begin
        proc p2(x, y) is z := x + y ;
        z := 0 ;
        call p2(z+1, 3) ;
    end

proc p3 (x) is
    begin
        var z ;
        call p1() ; z := z+x ;
    end

call p3(42) ;

Exercice:

• give the execution stack when p2 is executed
• give the code for procedures p1 and p2
Exercice

Consider the following extensions

- functions
- other parameter modes (by reference, by result)
- dynamic binding for variables and procedures?
Procedures used as variables or parameters

```plaintext
var z1 ;
var p proc (int) ; /* p is a procedure variable */
proc p1 (x : int) is z1 := x ;
proc p2 (q : proc (int)) is call q(2) ;

proc q1 is
  begin
    var z1 ;
    proc q2 (y int) is z1 := x ;
    p := q2 ;
    call p ;
  end

p := p1 ;
call p ;
call p2 (p1) ;

Q: what code to produce for \texttt{p := ...} \? for \texttt{call p2(p1)} \? for \texttt{call p} \?
```
Information associated to a procedure at code level

\begin{align*}
p & := q2 \\
& \ldots \\
& \text{call } p
\end{align*}

To translate a procedure call, we need:

\begin{itemize}
  \item the address of its 1st instruction
  \item the address of its environment definition
\end{itemize}

\Rightarrow Variable p should store both information
\Rightarrow At code level, a procedure type is a \textbf{pair}
\hspace{1cm} (address of code, address of memory environment)

\textbf{Exercice:} code produced for the previous example ?