Intermediate Code Optimization using Static Analysis

1 Introduction

Static analysis of a program consists in, first determining information on the behavior of a program without executing it, and then using this information to either *check* that the program satisfies some property or optimize it. Optimization criteria include resource consumption (execution time, memory and energy consumption).

We present some data-flow analysis defined on an intermediate-representation that is independent of the source language and the target machine. This representation is structured as a control-flow graph (CFG) which nodes are basic blocks and edges denote the control-flow.

Information/optimization is performed as follows:

1. Generation of an equation system.
2. Solving equations.
3. Program transformation.

1.1 Preliminary definitions

A basic block `B` is a sequence of assignments that can end (or not) by a branching. The algorithm which is partitioning some 3-address code into basic blocks determines heads of blocks (i.e., first instruction, instruction reached by a branching instruction, or any instruction following a branching). The control-flow graph is a graph s.t. the set of nodes consists of the set of basic blocks augmented with a starting and an ending blocks. An edge links two nodes together if the control can move from a node to the other. Given a CFG, and a basic block B, we define the set of predecessors of B in the CFG as the set of basic blocks B' s.t. there is an edge in the CFG from B' to B. The set of predecessors of B is noted $\mathbf{pre}(B)$. We also define the set of successors of B in the CFG as the set of basic blocks B' s.t. there is an edge in the CFG from B to B'. The set of successors of B is noted $\mathbf{succ}(B)$.

1.2 Generation of an equation system

We want to compute some properties at each execution point of the program. To do so, we determine a local property for each basic block.

To each basic block B we associate two variables $\texttt{In}(B)$ and $\texttt{Out}(B)$. Then, we link these variables through equations in the following way:

- When information propagates in the same direction as control-flow, the variable $\texttt{In}(B)$ is expressed in terms of the variables $\texttt{Out}(B')$, where B' are the predecessors of B.
- When information propagates in the reverse direction of control-flow, the variable $\texttt{Out}(B)$ is expressed in terms of the variables $\texttt{In}(B')$, where the B' are the successors of B.
- Transfer functions link, for each basic block B, the variables $\texttt{In}(B)$ and $\texttt{Out}(B)$. The transfer function of basic block B is noted F_B.

1.3 Solving an equation system

Lattice. Properties are expressed in a lattice. Let us recall that a lattice (resp. complete lattice) $(E, \leq, \bigsqcup, \bigsqcap, \bot, \top)$ is a set E endowed with an ordering relation $\leq$, a least element, noted $\bot$, and a greatest element, noted $\top$, such that every sub-set (resp. finite sub-set) $X \subseteq E$ has an upper bound $\bigsqcup X$ and a lower bound $\bigsqcap X$, which belongs to this set.

Solving equations. A solution of the equation system is a fix point. We are interested in either the least fix point (noted LFP) which characterizes the existence of a location where the property is verified, or to a greatest fix point (noted GFP), which characterizes the fact that the property holds at all locations.

Forward analysis - LFP	$\begin{array}{rcl} \texttt{In}(B) & = & \begin{cases} \bot & \text{if } B \text{ is initial} \\ \bigsqcup\limits_{B' \in \mathbf{pre}(B)} \texttt{Out}(B') & \text{otherwise} \end{cases} \\ \texttt{Out}(B) & = & F_B(\texttt{In}(B)) \end{array}$
Forward analysis - GFP	$\begin{array}{rcl} \texttt{In}(B) & = & \begin{cases} \bot & \text{if } B \text{ is initial} \\ \bigsqcap_{B' \in \mathbf{pre}(B)} \texttt{Out}(B') & \text{otherwise.} \end{cases} \\ \texttt{Out}(B) & = & F_B(\texttt{In}(B)) \end{array}$
Backward analysis - LFP	$\begin{array}{rcl} \texttt{Out}(B) & = & \begin{cases} \bot & \text{if } B \text{ is final} \\ \bigsqcup\limits_{B' \in \mathbf{succ}(B')} \texttt{In}(B') & \text{otherwise.} \end{cases} \\ \texttt{In}(B) & = & F_B(\texttt{Out}(B)) \end{array}$
Backward Analysis - GFP	$\begin{array}{rcl} \texttt{Out}(B) & = & \begin{cases} \bot & \text{if } B \text{ is final} \\ \bigsqcap_{B' \in \mathbf{succ}(B)} \texttt{In}(B') & \text{otherwise.} \end{cases} \\ \texttt{In}(B) & = & F_B(\texttt{Out}(B)) \end{array}$

1.3.1 Solving an equation system in a lattice (Kleene/Tarski's theorem)

Let $(E, \leq, \bigsqcup, \bigsqcap, \bot, \top)$ be a complete lattice. Given an equation system

$$X_j = F_j(X_1, ..., X_n) \text{ for } j \in [1, n]$$

there exists a smallest (resp. greatest) solution if the F_j are $\sqcup$ (resp. $\sqcap$) continuous. In this case, the least solution is the limit of the series:

$$\begin{array}{rcl} X_j^0 & = & (\bot, \cdots, \bot) \\ X_j^{i+1} & = & F_j(X_1^i, ..., X_n^i) \end{array}$$

(resp. the greatest solution is the limit of the series:

$$\begin{array}{rcl} X_j^0 & = & (\top, \cdots, \top) \\ X_j^{i+1} & = & F_j(X_1^i, ..., X_n^i)). \end{array}$$

In the remainder, we shall say that a variable x is used (resp. defined) if it appears in the right-hand side (resp. left-hand) side of an assignment.

1.4 Transforming programs

Program transformation consists in using the information computed previously in order to modify a given program. This is done in such a way that the initial and resulting programs are semantically equivalent.

2 Available Expressions

An expression `x+y` is said to be *available* at some point p of the CFG if all paths from the initial block to p contain `x+y` and the operands `x` and `y` are not modified after the last occurrence of the expression.

2.1 Equations

The lattice is the power-set of the set of expressions appearing in the program, ordered according to the inclusion relation, and such that the greater bound operation is the set-union, and the lower bounder operation is the set-intersection. At a basic block level, the availability property is expressed using functions `Gen` and `Kill`. The transfer function is expressed according to `Gen` and `Kill`.

$$\begin{array}{lcl} \texttt{In}(B) & = & \bigcap_{B' \in \mathbf{pre}(B)} \texttt{Out}(B') \\ \texttt{Out}(B) & = & (\texttt{In}(B) \setminus \texttt{Kill}(B)) \cup \texttt{Gen}(B) \end{array}$$

Local properties are defined as follows:

- $\texttt{Kill}(B)$ is the set of suppressed expressions in basic block B. An expression `x+y` appearing in a basic block is suppressed if its last occurrence is followed by a definition of one of its two operands. An expression not appearing in a basic block is suppressed if one its two operands is modified.
- $\texttt{Gen}(B)$ is the set of generated expressions in basic block B. An expression is generated if it appears in a on the right-hand side of an assignment of the basic block and if its last occurrence is not followed by a definition of its operands.

2.2 Computing local properties

We shall determine, for each basic block, what is the created and suppressed information. To define `Gen` and `Kill`, we introduce local functions $\texttt{Gen}_l$ and $\texttt{Kill}_l$:

$\texttt{Gen}(B)$	$=$	$\texttt{Gen}_1(B, \emptyset)$
$\texttt{Kill}(B)$	$=$	$\texttt{Kill}_1(B, \emptyset)$
$\texttt{Gen}_1(\texttt{x} := \texttt{a}; B, X)$	$=$	$\texttt{Gen}_1(B, X \setminus \{e' \mid x \in \texttt{FreeVars}(e')\} \cup \{a \mid x \notin \texttt{FreeVars}(a)\})$
$\texttt{Gen}_1(\texttt{if b goto l}, X)$	$=$	$X \cup \{b\}$
$\texttt{Gen}_1(\texttt{goto l}, X)$	$=$	X
$\texttt{Gen}_1(\epsilon, X)$	$=$	X
$\texttt{Kill}_1(\texttt{x} := \texttt{a}; B, X)$	$=$	$\texttt{Kill}_1(B, X \cup \{e' \mid x \in \texttt{FreeVars}(e')\} \cup \{a \mid x \in \texttt{FreeVars}(a)\})$
$\texttt{Kill}_1(\texttt{if b goto l}, X)$	$=$	X
$\texttt{Kill}_1(\texttt{goto l}, X)$	$=$	X
$\texttt{Kill}_1(\epsilon, X)$	$=$	X

In other words, if block B is a sequence of n assignments $\texttt{x}_\texttt{i} := \texttt{a}_\texttt{i}$ (followed by a branching or not) :

$$\begin{array}{rcl} & & \cdots \\ 1 & : & \texttt{x}_1 := \texttt{a}_1 \\ & & \cdots \\ i & : & \texttt{x}_\texttt{i} := \texttt{a}_\texttt{i} \\ & & \cdots \\ n & : & \texttt{x}_\texttt{n} := \texttt{a}_\texttt{n} \end{array}$$

- for function `Gen`, we build the series $X_0 = \emptyset, X_1, \cdots, X_{n+1}$:

$X_0 = \emptyset$

1 : $\mathtt{x_1} := \mathtt{a_1}$

$X_1 = \begin{cases} \emptyset & \text{if } \mathtt{x_1} \in \mathtt{FreeVars}(\mathtt{a_1}) \\ \{a_1\} & \text{otherwise} \end{cases}$

X_i

i : $\mathtt{x_i} := \mathtt{a_i}$

$X_{i+1} = \begin{cases} X_i \setminus \{e' \mid \mathtt{x_i} \in \mathtt{FreeVars}(e')\} & \text{if } \mathtt{x_i} \in \mathtt{FreeVars}(\mathtt{a_i}) \\ X_i \setminus \{e' \mid x_i \in \mathtt{FreeVars}(e')\} \cup \{a_i\} & \text{otherwise} \end{cases}$

n : $\mathtt{x_n} := \mathtt{a_n}$

`Gen` is then X_{n+1} if this series of assignments is not followed by a conditional branching or $X_{n+1} \cup \{b\}$ if it is followed by a branching depending on condition b.

- $\mathtt{Kill} = \bigcup_{i=1}^{n} \{e \mid x_i \in \mathtt{FreeVars}(e)\}$

Remark. The previous analysis can be adapted so as to compute the largest constant sub-expressions.

2.3 Solving equations

We want to determine, for each basic block, the value of $\mathtt{In}(B)$ and $\mathtt{Out}(B)$. For this purpose:

- we determine the local properties $\mathtt{Gen}(B)$ and $\mathtt{Kill}(B)$ for each basic block B,
- we initialize $\mathtt{In}(B)$ to the set of all expressions for each block B, different from the initial block; for the initial block B_0, we initialize it to the empty set,
- we iterate the computations until stabilization:
  1. compute $\mathtt{Out}(B)$,
  2. compute $\mathtt{In}(B)$.

We can also solve the following equation system:

$$\mathtt{In}(B) \quad = \quad \bigcap_{B' \in \mathbf{pre}(B)} \{(\mathtt{In}(B') \setminus \mathtt{Kill}(B')) \cup \mathtt{Gen}(B')\}$$

or this one:

$$\mathtt{Out}(B) \quad = \quad \Big(\bigcap_{B' \in \mathbf{pre}(B)} \mathtt{Out}(B') \setminus \mathtt{Kill}(B) \Big) \cup \mathtt{Gen}(B)$$

2.4 Application: Suppression of redundant computations

For each available expression e at the entry of a basic block, if this expression exists in this block and if the operands are not modified between the beginning of the block and the position of the expression, then it is a redundant expression. In order to suppress it, we create a new variable u, and thanks to a backward analysis, we locate instructions of the form $x := e$, that we replace by $\begin{cases} u := e \\ x := u \end{cases}$ and we replace u by the computation of the expression, in the block where this computation is redundant.

3 Active Variables

- A variable `x` is *inactive* at a location `i` of the program if every path going from `i` to a location `j` does not contain any use of this variable; `j` being either the exit block or a block containing an assignment to `x`, which is not preceded by any use of `x`.
- An assignment `d:x := e` is *useless* if `x` is inactive at the end of the block and is not used in the block after the instruction.

3.1 Equations

The lattice is the power-set of the set of variables appearing in the program, ordered by the inclusion relation, such that the greater bound is the set-union, the lower bound operation is the set-intersection.

A simple method to determine whether a variable is active consists in performing an analysis of active variables for which we give the equations:

$$\begin{array}{lcl} \texttt{Out}(B) & = & \bigcup_{B' \in \mathbf{succ}(B)} \texttt{In}(B') \\ \texttt{In}(B) & = & (\texttt{Out}(B) \setminus \texttt{Kill}(B)) \cup \texttt{Gen}(B) \end{array}$$

$\texttt{Gen}(B)$ is the set of variables `x` such that: `x` is used in block B, and in this block B, any potential assignment to `x` is located after the first use of `x`.

$\texttt{Kill}(B)$ is the set of variables `x` assigned in block B.

3.2 Computing local properties

Similarly to available expressions, we shall determine local functions $\texttt{Gen}_l$ and $\texttt{Kill}_l$:

$\texttt{Gen}(B)$	$=$	$\texttt{Gen}_\texttt{l}(B, \emptyset)$
$\texttt{Kill}(B)$	$=$	$\texttt{Kill}_\texttt{l}(B, \emptyset)$
$\texttt{Gen}_\texttt{l}(B; \texttt{x} := \texttt{a}, X)$	$=$	$\texttt{Gen}_\texttt{l}(B, X \setminus \{\texttt{x}\} \cup \texttt{FreeVars}(a))$
$\texttt{Gen}_\texttt{l}(B; \texttt{if b goto l}, X)$	$=$	$\texttt{Gen}_\texttt{l}(B, X \cup \texttt{FreeVars}(b))$
$\texttt{Gen}_\texttt{l}(b; \texttt{goto l}, X)$	$=$	$\texttt{Gen}_\texttt{l}(B, X)$
$\texttt{Gen}_\texttt{l}(\epsilon, X)$	$=$	X
$\texttt{Kill}_\texttt{l}(B; \texttt{x} := \texttt{a}, X)$	$=$	$\texttt{Kill}_\texttt{l}(B, X \cup \{x\})$
$\texttt{Kill}_\texttt{l}(B; \texttt{if b goto l}, X)$	$=$	$\texttt{Kill}_\texttt{l}(B, X)$
$\texttt{Kill}_\texttt{l}(B; \texttt{goto l}, X)$	$=$	$\texttt{Kill}_\texttt{l}(B, X)$
$\texttt{Kill}_\texttt{l}(\epsilon, X)$	$=$	X

As before, if the block B is a series of n assignments, followed or not by a branching,

- for the function `Gen`, we compute the series $X_0 = \texttt{Gen}, X_1, X_{n+1}$, with

$$\begin{array}{lcl} X_{n+1} & = & \begin{cases} \emptyset & \text{if the sequence is not followed by a conditional branching} \\ \{b\} & \text{if the sequence if followed by a conditional branching depending on } b \end{cases} \\ X_i & = & (X_{i+1} \setminus \{x_i\}) \cup \texttt{FreeVars}(a_i) \end{array}$$

- $\texttt{Kill}(B) = \{x_1, \ldots, x_n\}$

3.3 Computing equations

Similarly to available expressions, we want to determine, for each basic block, the value of $\texttt{In}(B)$ and $\texttt{Out}(B)$. To do so,

- we determine local properties $\texttt{Gen}(B)$ and $\texttt{Kill}(B)$ for each basic block B,
- we initialize $\texttt{Out}(B)$ to the empty set $\emptyset$,
- we iterate the following computations until stabilization :
  1. compute $\texttt{In}(B)$,
  2. compute $\texttt{Out}(B)$.

3.4 Application: Suppression of useless instructions

Useless instructions are assignments such that the left-hand side is inactive. We propose the following transformation:

- computation of active variables for each basic block,
- for each basic block B, we perform the following local transformation:

We initialize $X = \texttt{Out}(B)$

$$
\begin{array}{lcl}
F(B; \texttt{if b goto l}, X) & = & F(B, X \cup \texttt{FreeVars}(b)); \texttt{if b goto l} \\
F(B; \texttt{goto l}, X) & = & F(B, X); \texttt{goto l} \\
F(B; \texttt{x} := \texttt{a}, X) & = & \begin{cases} F(B, X) & \text{if } \texttt{x} \notin X \\ F(B, X \cup \texttt{FreeVars}(a)); x := a & \text{if } \texttt{x} \in X \end{cases} \\
F(\epsilon, X) & = & \epsilon
\end{array}
$$

4 Constant Propagation

Each variable takes its values in a domain extended with two values : $\top$ and $\bot$. Value $\top$ indicates the non-constant information. Value $\bot$ indicates the absence of information. We aim at defining a lattice on the extended domain $D \cup \{\top, \bot\}$. The ordering relation is defined as follows: if $v \in D$ then $\bot < v$ and $v < \top$.

The least upper bound $\sqcup$: for $x \in D \cup \{\top, \bot\}$ and $v_1, v_2 \in D$

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

Let $\mathcal{V}$ be the set of variables appearing in a program. We suppose, for the sake of simplicity, that they all take their value in the same domain D. We consider the lattice $[\mathcal{V} \to D \cup \{\top, \bot\}]$ where the ordering relation comes from the one of $D \cup \{\top, \bot\}$: $f \leq q$ if and only if $\forall x \cdot f(x) \leq g(x)$. The least upper bound is defined as follows: $(f \sqcup g)(x) = f(x) \sqcup g(x)$. The least element is the function that is defined nowhere, denoted $f_\bot$. Moreover, the functions of $[\mathcal{V} \mapsto D \cup \{\top, \bot\}]$ are extended to the expressions of the program using the variables of $\mathcal{V}$, in the natural way.

4.1 Equations

$$
\begin{array}{lcl}
\texttt{In}(B) & = & \begin{cases} f_\bot & \text{if } B \text{ is initial,} \\ \bigsqcup_{B' \in \mathbf{pre}(B)} \texttt{Out}(B') & \text{otherwise} \end{cases} \\
\texttt{Out}(B) & = & F(B, \texttt{In}(B))
\end{array}
$$

The transfer function is defined as follows, by induction on the sequence of assignments in a basic block.

$$
\begin{array}{lcl}
F(\texttt{x:=a ; B}, f) & = & F(\texttt{B}, f[x \mapsto f(a)]) \\
F(\epsilon, f) & = & f
\end{array}
$$

Remark: For this analysis to be sensible, we need to suppose that variables are correctly initialized.

4.2 Program Transformation

In each basic block B, let us consider $f \in \texttt{In}(B)$, we replace `x := a` by `x := v` if $v = f(a) \neq \top$.

5 Visible Definitions

A definition `x := e` is said to be *visible* from a location `p` of the program, if there exists a path starting immediately after this definition ending in location p such that `x` is not redefined. The lattice is the power-set of the set of variable definitions appearing in the program and ordered using the inclusion relation, such that the greater bound operation is the set-union, and the lower bound operation is the set-intersection.

Each instruction receives a unique label; this label represents the program counter (instructions are numbered from 1 to the number of instructions). Thus, `x := e` becomes `I: x := e` (normally, instructions are numbered in the 3-address code by an integer modeling the control point).

$$\begin{array}{lcl} \texttt{In}(b) & = & \bigcup_{b' \in \mathbf{pre}(b)} \texttt{Out}(b') \\ \texttt{Out}(b) & = & (\texttt{In}(b) \setminus \texttt{Kill}(b)) \cup \texttt{Gen}(b) \end{array}$$

Formally, $\texttt{Gen}(b)$ and $\texttt{Kill}(b)$ can be defined in the following way: we consider the function `TheDef`(x) which, to a variable `x` associates all definitions of `x`, of the form `I: x := e`.

$$\boxed{\begin{array}{lcl} \texttt{Gen}_{\texttt{aux}}(I : x := e; b, X) & = & \texttt{Gen}_{\texttt{aux}}(b, X \setminus \texttt{TheDef}(x) \cup \{I\}) \\ \texttt{Kill}_{\texttt{aux}}(I : x := e; b, X) & = & \texttt{TheDef}(x) \setminus \{I\} \end{array}}$$

6 Very Busy Expressions

An expression `x+y` is said to be *anticipable, or very busy* at some location `p` of a program if all paths starting at `p` contain an evaluation of `x+y` such that its operands are not re-defined.

Similarly to available expression, the lattice is the power-set of the set of expressions:

The equations are the following:

$$\begin{array}{lcl} \texttt{Out}(B) & = & \bigcap_{B' \in \mathbf{succ}(B)} \texttt{In}(B') \\ \texttt{In}(B) & = & (\texttt{Out}(B) \setminus \texttt{Kill}(B)) \cup \texttt{Gen}(B) \end{array}$$

Local properties are defined as follows:

- $\texttt{Kill}(B)$ is the set of expressions that are suppressed in the basic block. An expression `x+y` appears in a basic block is suppressed if its first occurrence is preceded by a definition of one of its operands. An expression not occurring in a basic block is suppressed if one of its operands is modified.
- $\texttt{Gen}(b)$ is the set of expressions generated in the basic block. An expression is generated it appears in a basic block and if its first occurrence is not preceded by any definition of its operands.

We solve the equation by initializing the $\texttt{Out}(b)$ to the set of all expressions except for the final block B for which the initialization of $\texttt{Out}(B)$ is $\emptyset$.

6.1 Application: code moving, promoting expressions

We consider blocks B such that

- the expression `e` belongs to $\texttt{Out}(b)$
- dominates the set of all blocks,

- each B' contains an evaluation of `e` such that the operands of `e` are not modified between the beginning of B' and the first occurrence of `e`.

The we add at the end of B, the assignment `t` := `e` (or just before if the last instruction is a branching, t being a new temporary variable), then we replace the use of `e` in B' by t.