Static Analysis

in the compilation process (an overview)

Laure Gonnord
http://laure.gonnord.org/pro/

Lille1 (USTL)/LIFL
Lille, France

Feb 2012

Context

Before actually generating code / allocate registers, it would be useful to get some information on

- Variables : value range, scope, lifespan, constants, ...
- Arrays : illicit accesses, alias discovery...
- Data Structures : memory leaks, null pointer dereferences...

▶ static analyses, of different kinds

Slides

Recycling de Cours de M2 compil, Ens Lyon
Biblio : Inspiration “dragon” et Nielson/Nielson.

1 Data Flow analysis
- Available expressions
- Live Variable analysis

2 Dans GCC

3 Toward a generalisation of these analysis

What for ?

Avoiding the computation of an (arithmetic) expression :

```
x:=a+b;
y:=a*b;
while(y>a+b) do
   a:=a+a;
   x:=a+b;
done
```

Some defs

Definition

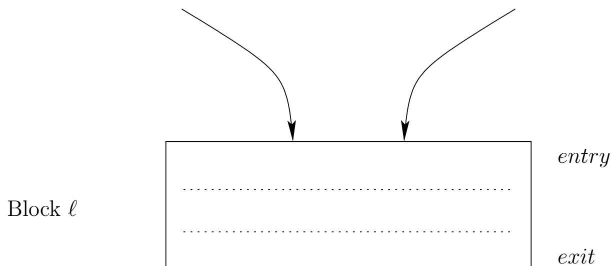
An expression is **killed** in a block if any of its variables is redefined in the block.

Definition

A **generated** expression is an expression evaluated in the block and none of its variables is killed in the block.

- Sets : $kill_{AE}(block)$ and $gen_{AE}(block)$

Data flow expressions

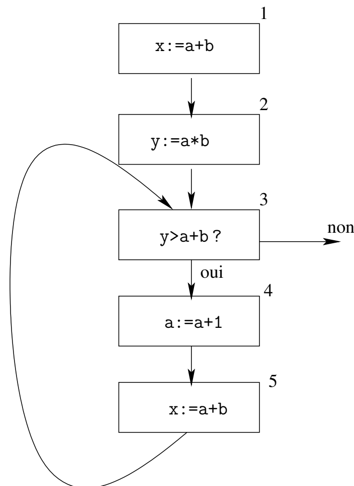


$$AE_{entry}(\ell) = \begin{cases} \emptyset & \text{if } \ell = init \\ \bigcap\{AE_{exit}(\ell') | (\ell', \ell) \in flow(G)\} \end{cases}$$

$$AE_{exit}(\ell) = \big(AE_{entry}(\ell) \backslash kill_{AE}(\ell)\big) \cup gen_{AE}(\ell)$$

On the example - equations

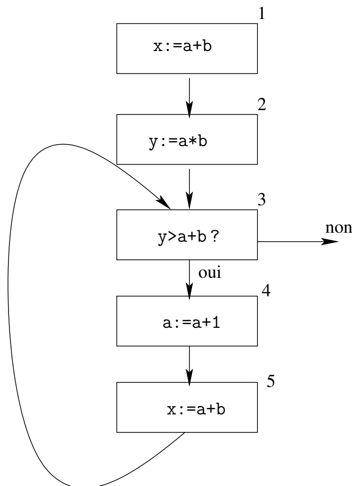
ℓ	$kill_{AE}(\ell)$	$gen_{AE}(\ell)$
1	$\emptyset$	{a+b}
2	$\emptyset$	{a*b}
3	$\emptyset$	{a+b}
4	{a+b, a*b, a+1}	$\emptyset$
5	$\emptyset$	{a+b}



On the example - final solution

ℓ	$AE_{entry}(\ell)$	$AE_{exit}(\ell)$
1	$\emptyset$	{a+b}
2	{a+b}	{a+b, a*b}
3	{a+b}	{a+b}
4	{a+b}	$\emptyset$
5	$\emptyset$	{a+b}

- a+b is available on entry to the loop, not a*b
- Improvement of code generation



Generated code

```
x:=a+b;
y:=a*b;
while(y>a+b) do
   a:=a+a;
   x:=a+b;
done
```

- cf avail.s

Another example : live ranges

```
x:=2;
y:=4;
x:=1;
if (y>x) then z:=y else z=y*y ;
x:=z;
```

Definition

A variable is **live** at the exit of a block if there exists a path from the block to a use of the variable that does not redefine the variable.

Problem : determine the set of variables that *may be* live after each control point.

Data flow expressions

Definition

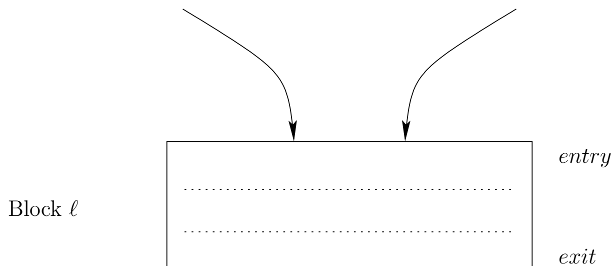
A variable that appears on the left hand side of an assignment is **killed** by the block. Tests do no kill variables.

Definition

A **generated** variable is a variable that appears in the block.

- Sets : $kill_{LV}(block)$ and $gen_{LV}(block)$

Data flow expressions



$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{if } \ell = final \\ \bigcup\{LV_{entry}(\ell') | (\ell', \ell) \in flow(G)\} \end{cases}$$

$$LV_{entry}(\ell) = \left(LV_{exit}(\ell) \backslash kill_{LV}(\ell)\right) \cup gen_{LV}(\ell)$$

Final result and use

Backward analysis and we want the smallest sets, here is the final result : (we assume all vars are dead at the end).

ℓ	$LV_{entry}(\ell)$	$LV_{exit}(\ell)$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{y\}$
3	$\{y\}$	$\{x, y\}$
4	$\{x, y\}$	$\{y\}$
5	$\{y\}$	$\{z\}$
5	$\{y\}$	$\{z\}$
5	$\{z\}$	$\emptyset$

- Use : Dead code elimination + register allocation. Note : can be improved by computing the use-defs paths. (see Nielson/Nielson/Hankin)

1 Data Flow analysis

2 Dans GCC

3 Toward a generalisation of these analysis

Slides

cours de Prog avancée, LG, polytech'lille 2011

Optimisations courantes de GCC

(machine indépendantes)
Faites par le compilateur :

- Variables vivantes
- Élimination de code mort
- Propagation de constantes
- Déroulement des boucles, inlining ...

Variables vivantes

Exemple :

```
x:=1;
y:=7;
if y < 20 then x:=78 else x:=42
```

- La première affectation est inutile : « x n'est pas vivante au test $y < 20$ ».

Propagation de constantes

Exemple :

```
x:=3;
[....pas d'affectation à  x...]
if z < x then ...
```

- Le test peut être réécrit en $z < 3$.

Les options -O2/-O3 de gcc

Ces options réalisent les optimisations suivantes (entre autres) :

- Inlining des petites fonctions
- Available expressions
- Transformations ”intelligentes” de boucles

Démo options de compilation - 1

(sources : Bogdan Pasca, ENS Lyon)
Variables, constantes, registres.

```
int main(int argc, char** argv)
{
    int i = 0, j = 0;
    j += 1;
    return j;
}
```

Démo (compiler avec `-S`) et comparaison.

- Sans option, que peut-on améliorer ?
- Avec -O1, -O2, -O3.
- En changeant l'expression ?

Démo options de compilation - 2

Optimisations de boucle

```
int main(int argc, char** argv)
{
   int i,j;
   j=0;
   for (i=0 ; i < 100; i++)
        j += 1;
   return j;
}
```

- Observons !
- Complexifier l'expression de la boucle (`j = i*j + 42` par ex).

Démo options de compilation - 3

Inlining

```
int fortytwo(void)
{
    return 42;
}
```

- Regarder le code généré pour un appel dans le main

1. Data Flow analysis
2. Dans GCC
3. Toward a generalisation of these analysis

Common points

- Computing growing sets from $\emptyset$ via *fixpoint iterations*. (or the dual)
- Sets of equations of the form (collecting semantics) :

$$\mathcal{S}(\ell) = \bigcup_{(\ell',\ell)\in E} f(\mathcal{S}(\ell'))$$

where f is computed w.r.t. the *program statements*

▶ $\mathcal{S}$ is an **abstract interpretation** of the program.

More analyses

Bitvectors, ...