Live variable analysis

En cada punto del programa ver qué variables están vivas: si hay un "camino" que la utiliza sin redefinirla. Es un analisis backward (usa info de los sucesores para inferir propiedades del nodo) y may (ante información que se pisa, redondea para arriba en el reticulado)

$$JOIN(v) = \bigcup_{w \in succ(v)} [\![w]\!]$$

$X = E$: $\quad [\![v]\!] = JOIN(v) \setminus \{X\} \cup vars(E)$

This rule models the fact that the set of live variables before the assignment is the same as the set after the assignment, except for the variable being written to and the variables that are needed to evaluate the right-hand-side expression.

Branch conditions and output statements are modelled as follows:

`if (E):` / `while (E):` / `output E:` $\quad [\![v]\!] = JOIN(v) \cup vars(E)$

where $vars(E)$ denotes the set of variables occurring in E. For variable declarations and exit nodes:

`var` $X_1, \ldots, X_n$: $\quad [\![v]\!] = JOIN(v) \setminus \{X_1, \ldots, X_n\}$

$$[\![exit]\!] = \emptyset$$

For all other nodes:

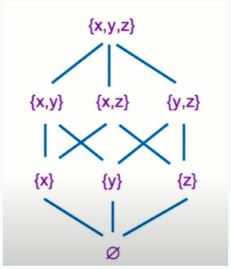
$$[\![v]\!] = JOIN(v)$$

El reticulado que se usa es el de la función partes del conjunto de variables de la función con el orden parcial definido por el operador subconjunto. Ejemplo:

```
var x,y,z;
x = input;
while (x>1) {
  y = x/2;
  if (y>3) x = x-y;
  z = x-4;
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```

the lattice modeling abstract states is thus:[4]

$$State = (\mathcal{P}(\{\mathtt{x},\mathtt{y},\mathtt{z}\}), \subseteq)$$



Todos los IN[n] y OUT[n] empiezan con el conjunto vacío, y aplicando el algoritmo de iteración caotica, se obtienen los valores

n	variables que esten vivas en la entrada del nodo IN[n]	variables que esten vivas al salir del nodo OUT[n]
0	—	{pid, j}
1	{pid, j}	{pid, j, i}
2	{pid, j, i}	{pid, k, j, i}
3	{pid, k, j, i}	{pid, k, j, t1, i}
4	{pid, k, j, t1, i}	{pid, k, j, t2}
5	{pid, k, j, t2}	{pid, k, j, t3}
6	{pid, k, j, t3}	{pid, k, j, t3, t4}
7	{pid, k, j, t3, t4}	{pid, k, j}
8	{pid, k, j}	{pid, k, h}
9	{pid, k, h}	{pid, k, h}
10	{pid, k, h}	{pid, k, h}
11	{pid, k, h}	{answer, pid, k }
12	{answer, pid, k }	{t5, k}
13	{t5, k}	∅
14	∅	—

Ejercicio 10

Sea el siguiente programa, donde `MASK`, `IA`, `IQ`, `IR`, `IM` y `AM` son constantes.

```
float foo(int pid) {
1:  int i, j, h;
2:  i = pid ^ MASK;
3:  int k = i / IQ;
4:  h = IA * (i - k * IQ) - IR * k;
5:  h = j ^ MASK;
6:  if (h < 0)
7:    h = h + IM;
8:  float answer = AM * h;
9:  return answer * pid / k;
}
```

a) Construir su control-flow graph.
b) Computar el análisis Live Variables.

Available expresions

Ver por cada punto de programa qué expresiones ya estan computadas y no fueron modificadas. Se computa las expresiones dadas por todos los caminos hacia ese punto del programa

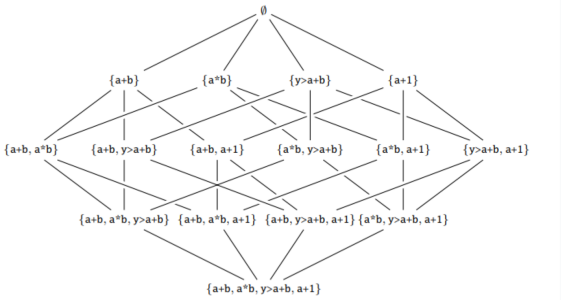
Por lo tanto es un analisis forward must, toma la información de las expresiones de los predecesores de un nodo (punto de un programa) y calcula las expresiones disponibles

El reticulo usado es el conjunto de partes de todas las expresiones del programa con el orden definido por inclusión subconjunto inverso (superset $\supseteq$)

```
var x,y,z,a,b;
z = a+b;
y = a*b;
while (y > a+b) {
  a = a+1;
  x = a+b;
}
```

we have four different nontrivial expressions, so our lattice for abstract states is

$$State = (\mathcal{P}(\{\texttt{a+b}, \texttt{a*b}, \texttt{y>a+b}, \texttt{a+1}\}), \supseteq)$$



El elemento ∅ es el menos preciso e indica que ninguna expresión está disponible. Descendiendo en el reticulo aumentamos precisión

Next we define the dataflow constraints. The intuition is that an expression is available at a node v if it is available from all incoming edges or is computed by v, unless its value is destroyed by an assignment statement.

The *JOIN* function uses $\cap$ (because the lattice order is now $\supseteq$) and *pred* (because availability of expressions depends on information from the past):

$$JOIN(v) = \bigcap_{w \in pred(v)} [\![w]\!]$$

Assignments are modeled as follows:

$$X = E: \qquad [\![v]\!] = (JOIN(v) \cup exps(E)) \downarrow X$$

Here, the function $\downarrow X$ removes all expressions that contain the variable X, and *exps* collects all nontrivial expressions:

$$exps(X) = \emptyset$$
$$exps(\underline{I}) = \emptyset \quad \texttt{Integer}$$
$$exps(\texttt{input}) = \emptyset$$
$$exps(E_1 \; op \; E_2) = \{E_1 \; op \; E_2\} \cup exps(E_1) \cup exps(E_2)$$

For the example program, we generate the following constraints:

$$[\![entry]\!] = \emptyset$$
$$[\![\texttt{var x,y,z,a,b;}]\!] = [\![entry]\!]$$
$$[\![\texttt{z=a+b;}]\!] = exps(\texttt{a+b}) \downarrow \texttt{z}$$
$$[\![\texttt{y=a*b;}]\!] = ([\![\texttt{z=a+b;}]\!] \cup exps(\texttt{a*b})) \downarrow \texttt{y}$$
$$[\![\texttt{y>a+b;}]\!] = ([\![\texttt{y=a*b;}]\!] \cap [\![\texttt{x=a+b;}]\!]) \cup exps(\texttt{y>a+b})$$
$$[\![\texttt{a=a+1;}]\!] = ([\![\texttt{y>a+b;}]\!] \cup exps(\texttt{a+1})) \downarrow \texttt{a}$$
$$[\![\texttt{x=a+b;}]\!] = ([\![\texttt{a=a+1;}]\!] \cup exps(\texttt{a+b})) \downarrow \texttt{x}$$
$$[\![exit]\!] = [\![\texttt{y>a+b;}]\!]$$

Using one of our fixed-point algorithms, we obtain the minimal solution:

$$[\![entry]\!] = \emptyset$$
$$[\![\texttt{var x,y,z,a,b;}]\!] = \emptyset$$
$$[\![\texttt{z=a+b;}]\!] = \{\texttt{a+b}\}$$
$$[\![\texttt{y=a*b;}]\!] = \{\texttt{a+b}, \texttt{a*b}\}$$
$$[\![\texttt{y>a+b;}]\!] = \{\texttt{a+b}, \texttt{y>a+b}\}$$
$$[\![\texttt{a=a+1;}]\!] = \emptyset$$
$$[\![\texttt{x=a+b;}]\!] = \{\texttt{a+b}\}$$
$$[\![exit]\!] = \{\texttt{a+b}, \texttt{y>a+b}\}$$

No expressions are available at entry nodes:

$$[\![entry]\!] = \emptyset$$

Branch conditions and output statements accumulate more available expressions:

$$\left.\begin{array}{l}\texttt{if } (E):\\ \texttt{while } (E):\\ \texttt{output } E:\end{array}\right\} \quad [\![v]\!] = JOIN(v) \cup exps(E)$$

For all other kinds of nodes, the collected sets of expressions are simply propagated from the predecessors:

$$[\![v]\!] = JOIN(v)$$

Again, the right-hand sides of all constraints are monotone functions.

Very busy expresions

Es un analisis backward must. Busca las expresiones que van a ser utilizadas nuevamente antes de redefinirlas en dicho punto de programa.

An expression is *very busy* if it will definitely be evaluated again before its value changes. To approximate this property, we can use the same lattice and auxiliary functions as for available expressions analysis. For every CFG node v the variable $[\![v]\!]$ denotes the set of expressions that at the program point before the node definitely are busy.

An expression is very busy if it is evaluated in the current node or will be evaluated in all future executions unless an assignment changes its value. For this reason, the *JOIN* is defined by

$$JOIN(v) = \bigcap_{w \in succ(v)} [\![w]\!]$$

and assignments are modeled using the following constraint rule:

$$X = E: \qquad [\![v]\!] = JOIN(v) \downarrow X \; \cup exps(E)$$

No expressions are very busy at exit nodes:

$$[\![exit]\!] = \emptyset$$

The rules for the remaining nodes, include branch conditions and output statements, are the same as for available expressions analysis.

En el siguiente programa la expresion $a * b$ es very busy en el loop

```
var x,a,b;
x = input;
a = x-1;
b = x-2;
while (x>0) {
  output a*b-x;
  x = x-1;
}
output a*b;
```

Reaching definitions

El conjunto de definiciones que pueden llegar a un punto de un programa. Se define como un par < variable, nodo en el que se definió >. Es un analisis forward may.

The *reaching definitions* for a given program point are those assignments that may have defined the current values of variables. For this analysis we need a powerset lattice of all assignments (represented as CFG nodes) occurring in the program. For the example program from before:

```
var x,y,z;
x = input;
while (x>1) {
  y = x/2;
  if (y>3) x = x-y;
  z = x-4;
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```

the lattice modeling abstract states becomes:

$$State = (\mathcal{P}(\{\texttt{x=input}, \texttt{y=x/2}, \texttt{x=x-y}, \texttt{z=x-4}, \texttt{x=x/2}, \texttt{z=z-1}\}), \subseteq)$$

For every CFG node v the variable $[\![v]\!]$ denotes the set of assignments that may define values of variables at the program point after the node. We define

$$JOIN(v) = \bigcup_{w \in pred(v)} [\![w]\!]$$

For assignments the constraint is:

$$X = E: \quad [\![v]\!] = JOIN(v) \downarrow X \cup \{X = E\}$$

where this time the $\downarrow X$ function removes all assignments to the variable X. For all other nodes we define:

$$[\![v]\!] = JOIN(v)$$

Nodo n	IN[n]	OUT[n]
1	-	$\emptyset$
2	$\emptyset$	$\{\langle x, 2\rangle\}$
3	$\{\langle x, 2\rangle\}$	$\{\langle x, 2\rangle, \langle y, 3\rangle\}$
4		
5		
6		
7		

Como es un condicional no se escriben ninguna definición

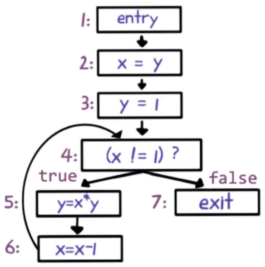
Y al tener varias entradas vamos a hacer la operación union entre todas

En este caso entre el nodo 3 y 6

n	In(n)	Out(n)
⋮		
4	{<x,2>. <y,3>}	{<x,2>. <y,3>}
5	{<x,2>. <y,3>}	{<x,2>. <y,5>}
6	{<x,6>. <y,5>}	{<x,6>. <y,5>}
7	{<x,2>. <y,3>}	-

Ejercicio 8

Sea el siguiente control-flow graph para una función:



retroalimentando con el for

n	In(n)	Out(n)
⋮		
4	{<x,2>. <y,3>, <x,6>. <y,5>}	{<x,2>. <y,3>, <x,6>. <y,5>}
5	{<x,2>. <y,3>, <x,6>. <y,5>}	{<x,2>, <x,6>. <y,5>}
6	{<x,2>, <x,6>. <y,5>}	{<x,6>. <y,5>}
7	{<x,2>. <y,3>, <x,6>. <y,5>}	-