

Dataflow

Live variable analysis

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

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

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

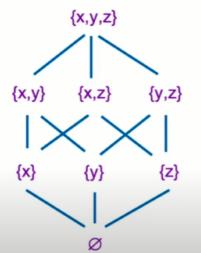
This rule models the fact that the set of live variables before the assignment is the same as the set after the assignment, except for the variable being written to and the variables that are needed to evaluate the right-hand-side expression.

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

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



Branch conditions and output statements are modelled as follows:

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

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

$$\text{var } X_1, \dots, X_n: \quad [v] = JOIN(v) \setminus \{X_1, \dots, X_n\}$$

$$[\text{exit}] = \emptyset$$

For all other nodes:

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

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

\cap	variables que estén vivas en la entrada del nodo IN[n]	variables que estén 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}
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 * (1 - 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 expressions

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

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

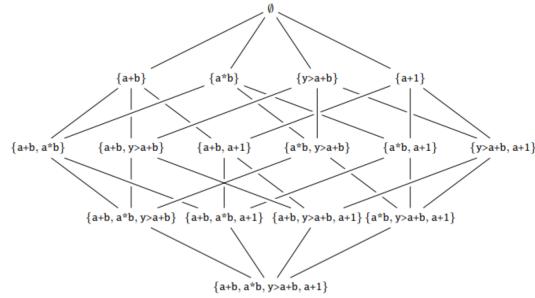
El reticulo usado es el conjunto de partes de todas las expresiones del programa con el orden definido por inclusión subconjunto inverso (superset \supseteq)

```
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

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

El elemento \emptyset 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:

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

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

$$\begin{aligned} exps(X) &= \emptyset \\ exps(I) &= \emptyset \quad \text{Integer} \\ exps(\text{Input}) &= \emptyset \\ exps(E_1 \text{ op } E_2) &= \{E_1 \text{ op } E_2\} \cup exps(E_1) \cup exps(E_2) \end{aligned}$$

For the example program, we generate the following constraints:

$$\begin{aligned} [entry] &= \emptyset \\ [\text{var } x, y, z, a, b] &= [entry] \\ [z=a+b] &= exps(a+b) \downarrow z \\ [y=a^*b] &= ([x=a+b] \cup exps(a^*b)) \downarrow y \\ [y>a+b] &= ([y=a^*b] \cap [x=a+b]) \cup exps(y>a+b) \\ [a=a+1] &= ([y>a+b] \cup exps(a+1)) \downarrow a \\ [x=a+b] &= ([a=a+1] \cup exps(a+b)) \downarrow x \\ [exit] &= [y>a+b] \end{aligned}$$

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

$$\begin{aligned} [entry] &= \emptyset \\ [\text{var } x, y, z, a, b] &= \emptyset \\ [z=a+b] &= \{a+b\} \\ [y=a^*b] &= \{a+b, a^*b\} \\ [y>a+b] &= \{a+b, y>a+b\} \\ [a=a+1] &= \emptyset \\ [x=a+b] &= \{a+b\} \\ [exit] &= \{a+b, y>a+b\} \end{aligned}$$

No expressions are available at entry nodes:

$$[entry] = \emptyset$$

Branch conditions and output statements accumulate more available expressions:

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

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 análisis 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:

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

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.

```
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;
```

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

Reaching definitions

El conjunto de definiciones que pueden llegar a un punto de un programa. Se define como un par \langle variable, nodo en el que se definió \rangle . Es un análisis 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:

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

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

$$JOIN(v) = \bigcup_{w \in pred(v)} \llbracket w \rrbracket$$

For assignments the constraint is:

$$X = E: \quad \llbracket v \rrbracket = JOIN(v) \downarrow X \cup \{X = E\}$$

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

$$\llbracket v \rrbracket = JOIN(v)$$

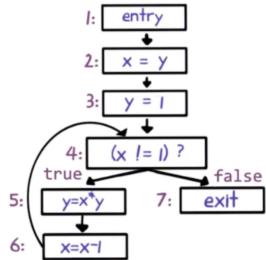
Nodo n	IN[n]	OUT[n]
1	-	\emptyset
2	\emptyset	$\{\langle x, 2 \rangle\}$
3	$\{\langle x, 2 \rangle\}$	$\{\langle x, 2 \rangle, \langle y, 3 \rangle\}$
4		
5		
6		
7		

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	$\{\langle x, 2 \rangle, \langle y, 3 \rangle\}$	$\{\langle x, 2 \rangle, \langle y, 3 \rangle\}$
5	$\{\langle x, 2 \rangle, \langle y, 3 \rangle\}$	$\{\langle x, 2 \rangle, \langle y, 5 \rangle\}$
6	$\{\langle x, 6 \rangle, \langle y, 5 \rangle\}$	$\{\langle x, 6 \rangle, \langle y, 5 \rangle\}$
7	$\{\langle x, 2 \rangle, \langle y, 3 \rangle\}$	-

Ejercicio 8

Sea el siguiente control-flow graph para una función:



n	In(n)	Out(n)
4	$\{\langle x, 2 \rangle, \langle y, 3 \rangle, \langle x, 6 \rangle, \langle y, 5 \rangle\}$	$\{\langle x, 2 \rangle, \langle y, 3 \rangle, \langle x, 6 \rangle, \langle y, 5 \rangle\}$
5	$\{\langle x, 2 \rangle, \langle y, 3 \rangle, \langle x, 6 \rangle, \langle y, 5 \rangle\}$	$\{\langle x, 2 \rangle, \langle x, 6 \rangle, \langle y, 5 \rangle\}$
6	$\{\langle x, 2 \rangle, \langle x, 6 \rangle, \langle y, 5 \rangle\}$	$\{\langle x, 6 \rangle, \langle y, 5 \rangle\}$
7	$\{\langle x, 2 \rangle, \langle y, 3 \rangle, \langle x, 6 \rangle, \langle y, 5 \rangle\}$	-

↙ retroalimentando con el for

Interprocedural

Call strings

Para contexto dado por call strings lo que se hace es modelar el contexto como los k últimos elementos del stack

Call node y function node

Ahora bien, el ejemplo donde $k=1$ solo tenemos un elemento del stack. Por lo que las restricciones del call-node (el de la entrada de la función):

The constraint rule for a function entry node v where $w \in pred(v)$ is a caller and $c' \in Context$ is a call context can then be written as follows.

$$s_w^{c'} \sqsubseteq \llbracket v \rrbracket(c) \text{ where } c = w$$

Informally, for any call context c' at the call node w , an abstract state $s_w^{c'}$ is built by evaluating the function arguments and propagated to call context c at the function entry node v .

$$\llbracket v \rrbracket(c) = \bigsqcup_{\substack{w \in pred(v) \wedge \\ c = w \wedge \\ c' \in Context}} s_w^{c'}$$

Quiere decir lo siguiente:

- $\llbracket v \rrbracket(c)$: Como $k=1$ entonces el contexto se define como el ultimo elemento del stack, es decir del nodo que está llamando a la función. Por eso se define $c = w$ para $w \in pred(v)$. Es decir, para el nodo v se define un contexto por cada nodo que lo llama.

Parameter passing at function calls is modeled in insensitive analysis, but now taking the call contexts a call node and v is the entry node of the function. The abstract state $s_w^{c'}$ defined by

$$s_w^{c'} = \begin{cases} \text{unreachable} \\ \perp[b_1 \mapsto eval(\llbracket w \rrbracket(c'), E_1^w), \dots, b_n \mapsto eval(\llbracket w \rrbracket(c'), E_n^w)] \end{cases}$$

- Dicho contexto se calcula con la union de todos los contextos del nodo w

Es decir, defino un contexto para v por cada nodo llamador (predecesor) y cada uno lo calculo con los contextos c' del nodo llamador

function entry node v . In this simple case where $k = 1$, the call node w is directly used as context c for the function entry node, but for larger values of k it is necessary to express how the call site is pushed onto the stack (represented by the call string from the call context).

After call node

Assume v is an after-call node that stores the return value in the variable X , and that v' is the associated call node and $w \in pred(v)$ is the function exit node. The constraint rule for v merges the abstract state from the v' and the return value from w , now taking the call contexts and reachability into account:

$$\llbracket v \rrbracket(c) = \begin{cases} \text{unreachable} & \text{if } \llbracket v' \rrbracket(c) = \text{unreachable} \vee \llbracket w \rrbracket(v') = \text{unreachable} \\ \llbracket v' \rrbracket(c)[X \mapsto \llbracket w \rrbracket(v')(result)] & \text{otherwise} \end{cases}$$

Notice that with this kind of context sensitivity, v' is both a call node and a call context, and the abstract value of **result** is obtained from the exit node w in call context v' .

El resultado es obtenido tomando el mismo contexto que el call node $\llbracket v \rrbracket(Interprocedural_analysis/c) = \llbracket v' \rrbracket(Interprocedural_analysis/c)[...]$ y agregando el resultado que se obtiene de la función usando el contexto del call node: $\llbracket w \rrbracket(Interprocedural_analysis/v')(result)$

functional

Se modela el contexto como los posibles estados que puede tomar las variables.

Ejemplo: Los 2 llamados a la función definen el contexto. En este caso con los valores 0 y + (para sign analysis) el resto unreachable

```

f(z) {
    return z*42;
}

main() {
    var x,y;
    x = f(0); // call 1
    y = f(87); // call 2
    return x + y;
}

```

$\begin{aligned} &[\perp[z \mapsto 0] \mapsto \perp[z \mapsto 0, \text{result} \mapsto 0], \\ &\perp[z \mapsto +] \mapsto \perp[z \mapsto +, \text{result} \mapsto +], \\ &\text{all other contexts} \mapsto \text{unreachable}] \end{aligned}$

Call node y function node

The constraint rule for an entry node v of a function $f(b_1, \dots, b_n)$ and a call $w \in \text{pred}(v)$ to the function is the same as in the call strings approach, except for the condition on c :

$$s_w^{c'} \sqsubseteq [v](c) \quad \text{where } c = s_w^{c'}$$

(The abstract state $s_w^{c'}$ is defined as in Section 8.3.) This rule shows that at the call w in context c' , the abstract state $s_w^{c'}$ is propagated to the function entry node v in a context that is identical to $s_w^{c'}$. This makes sense because contexts are abstract states with the functional approach.

Parameter passing at function calls is modeled in the same way as in context-insensitive analysis, but now taking the call contexts into account. Assume w is a call node and v is the entry node of the function $f(b_1, \dots, b_n)$ being called. The abstract state $s_w^{c'}$ defined by

$$s_w^{c'} = \begin{cases} \text{unreachable} & \text{if } [w](c') = \text{unreachable} \\ \perp[b_1 \mapsto \text{eval}([w](c'), E_1^w), \dots, b_n \mapsto \text{eval}([w](c'), E_n^w)] & \text{otherwise} \end{cases}$$

En palabras $S_w^{c'}$ es el estado formado a partir de los datos del nodo w en el contexto c' .

Dependiendo la elección del contexto, c' puede ser un call string (si $k=1$ sería simplemente un call node) o en el caso funcional sería un estado que define valores de las variables de entrada de la función a la que pertenece el nodo w .

Si con dicho contexto c' , w no es alcanzable entonces $[w](\text{Interprocedural analysis}/c') = \text{unreachable}$

After call node

Assume v is an after-call node that stores the return value in the variable X , and that v' is the associated call node and $w \in \text{pred}(v)$ is the function exit node. The constraint rule for v merges the abstract state from the v' and the return value from w , while taking the call contexts and reachability into account:

$$[v](c) = \begin{cases} \text{unreachable} & \text{if } [v'](c) = \text{unreachable} \vee [w](s_{v'}^{c'}) = \text{unreachable} \\ [v'](c)[X \mapsto [w](s_{v'}^{c'})(\text{result})] & \text{otherwise} \end{cases}$$

To find the relevant context for the function exit node, this rule builds the same abstract state as the one built at the call node.

Del mismo modo que en call string, el resultado se define en el mismo contexto del call node $[v](\text{Interprocedural analysis}/c) = [v'](\text{Interprocedural analysis}/c)[\dots]$ y usando el resultado de la función a partir de dicho contexto del call node (del estado formado por el contexto del call node en este caso): $[w](\text{Interprocedural analysis}/S_w^{c'})(\text{result})$

Control Flow Analysis

Objetivo: Analizar el flujo de llamados entre funciones. Es decir, modelar los posibles llamados a funciones en los distintos puntos del programa que utilicen funciones de primer orden (funciones como variables)

Para ello se buscara construir un Call graph que modele todos los posibles llamados a funciones. El reticulado que se usa es el del conjunto de partes de todos los nombres de funciones, por ejemplo $P(\text{main}, \text{foo}, \text{inc}, \text{dec}, \text{ide}), \subseteq$. Y las restricciones del dataflow se definen cómo:



CFA - Restricciones

Para cada nodo del AST introducimos una variable de restricción $\llbracket v \rrbracket$ definida de la siguiente manera:

- $f \in \llbracket f \rrbracket$
- $X = E: \llbracket E \rrbracket \subseteq \llbracket X \rrbracket$
- $E(E_1, \dots, E_n): f \in \llbracket E \rrbracket \Rightarrow$
 $\llbracket E_1 \rrbracket \subseteq \llbracket af_1 \rrbracket \wedge \dots \wedge \llbracket E_n \rrbracket \subseteq \llbracket af_n \rrbracket \wedge$
 $\llbracket \text{return } f \rrbracket \subseteq \llbracket E(E_1, \dots, E_n) \rrbracket \quad \forall f \text{ con argumentos } af_1, \dots, af_n$

La última restricción sirve para llamados a funciones que no sabemos a priori a que función hace referencia (info que vamos obteniendo por el dataflow). Por ejemplo si la función a llamar viene como input. Luego si f pertenece a los posibles funciones a llamar, se define las restricciones por los parametros de la función y el valor de retorno.

Si ya se sabe que función se va a llamar podemos usar la restricción incondicional:

We can optionally also introduce a simpler rule for the special case where the function being called is given directly by name (as in simple function calls before we added first-class functions to TIP). For a direct function call $f(E_1, \dots, E_n)$ where f is a function with arguments a_f^1, \dots, a_f^n and return expression E'_f , we have this (unconditional) constraint:

$$\llbracket E_1 \rrbracket \subseteq \llbracket a_f^1 \rrbracket \wedge \dots \wedge \llbracket E_n \rrbracket \subseteq \llbracket a_f^n \rrbracket \wedge \llbracket E'_f \rrbracket \subseteq \llbracket E(E_1, \dots, E_n) \rrbracket$$

Ejemplo:

CFA - Restricciones

```

inc(i) { return i+1; }
dec(j) { return j-1; }
ide(k) { return k; }

foo(n,f) {
    var r;
    if (n==0) { f=ide; }
    r = f(n);
    return r;
}

main() {
    var x,y;
    x = input;
    if (x>0) { y = foo(x,inc); } else { y = foo(x,dec); }
    return y;
}

```

```

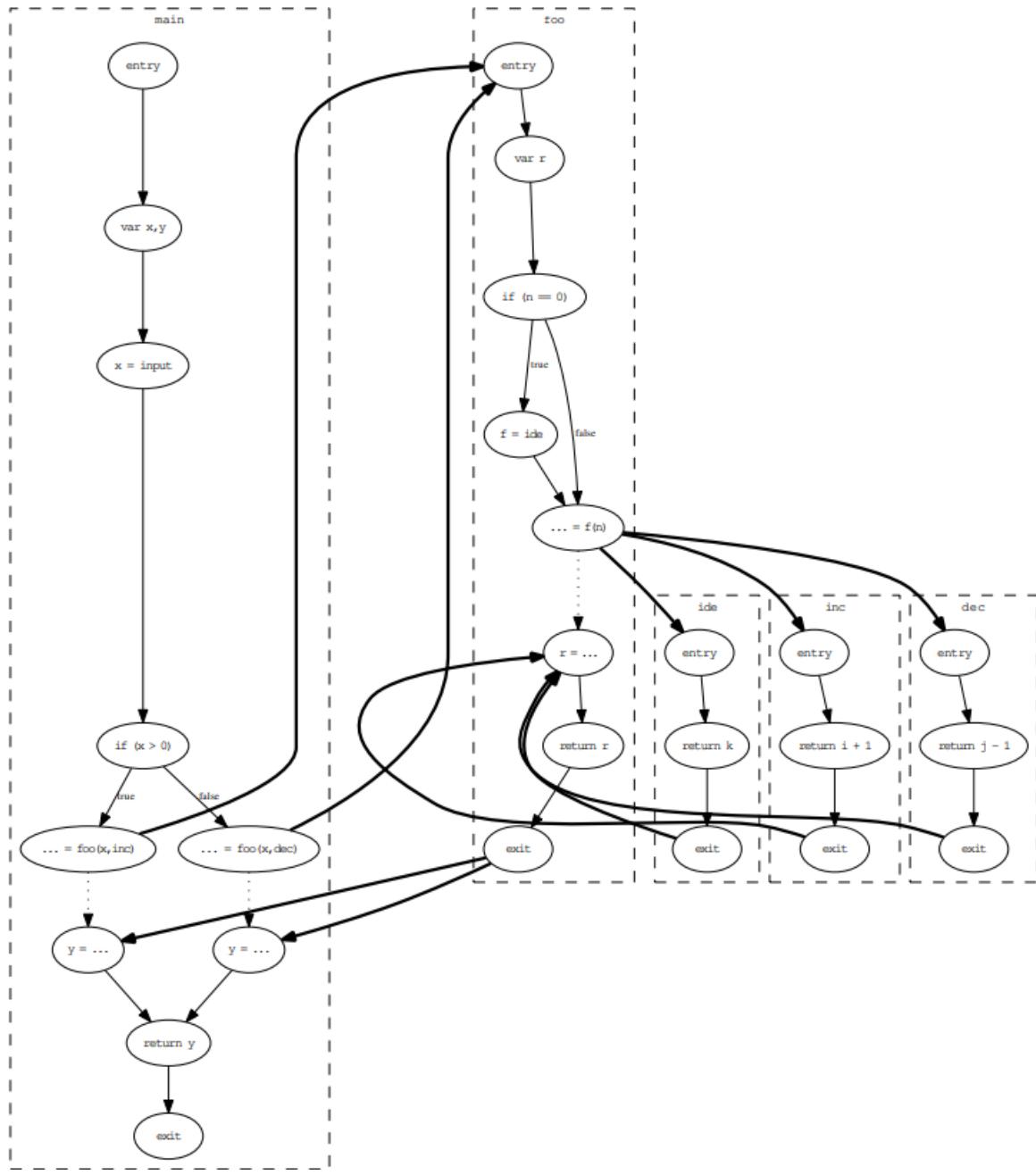
inc ∈ \llbracket inc \rrbracket
dec ∈ \llbracket dec \rrbracket
ide ∈ \llbracket ide \rrbracket
\llbracket ide \rrbracket ⊆ \llbracket f \rrbracket
\llbracket f(n) \rrbracket ⊆ \llbracket r \rrbracket
inc ∈ \llbracket f \rrbracket → \llbracket n \rrbracket ⊆ \llbracket i \rrbracket \wedge \llbracket i+1 \rrbracket ⊆ \llbracket f(n) \rrbracket
dec ∈ \llbracket f \rrbracket → \llbracket n \rrbracket ⊆ \llbracket j \rrbracket \wedge \llbracket j-1 \rrbracket ⊆ \llbracket f(n) \rrbracket
ide ∈ \llbracket f \rrbracket → \llbracket n \rrbracket ⊆ \llbracket k \rrbracket \wedge \llbracket k \rrbracket ⊆ \llbracket f(n) \rrbracket
\llbracket \text{input} \rrbracket ⊆ \llbracket x \rrbracket
\llbracket \text{foo}(x,inc) \rrbracket ⊆ \llbracket y \rrbracket
\llbracket \text{foo}(x,dec) \rrbracket ⊆ \llbracket y \rrbracket
foo ∈ \llbracket \text{foo} \rrbracket
\llbracket x \rrbracket ⊆ \llbracket n \rrbracket \wedge \llbracket \text{inc} \rrbracket ⊆ \llbracket f \rrbracket \wedge \llbracket y \rrbracket ⊆ \llbracket \text{foo}(x,inc) \rrbracket
\llbracket x \rrbracket ⊆ \llbracket n \rrbracket \wedge \llbracket \text{dec} \rrbracket ⊆ \llbracket f \rrbracket \wedge \llbracket y \rrbracket ⊆ \llbracket \text{foo}(x,dec) \rrbracket
main ∈ \llbracket \text{main} \rrbracket

```

```

\llbracket inc \rrbracket = \{inc\}
\llbracket dec \rrbracket = \{dec\}
\llbracket ide \rrbracket = \{ide\}
\llbracket f \rrbracket = \{inc, dec, ide\}
\llbracket \text{foo} \rrbracket = \{\text{foo}\}

```



Para POO

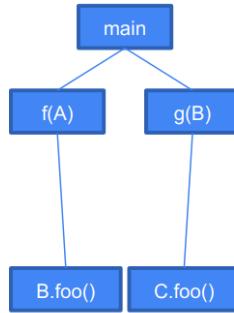
Objetivo: Analizar el flujo de llamado de métodos teniendo en cuenta la definición de clases. (polimorfismo y herencia)

Para el siguiente ejemplo la función *f* llama a *B.foo* por el parámetro que le pasan, y *g* crea un objeto de tipo *C* y llama a *C.foo*

- Un “mapa” para saber que métodos analizar
 - Fundamental en programas orientados a objetos

```

static void main() {
    B b1 = new B();
    A a1 = new A();
    f(b1);
    g(b1);
}
static void f(A a2) {
    a2.foo();
}
static void g(B b2) {
    B b3 = b2;
    b3 = new C();
    b3.foo();
}
class A {
    foo(){...}
}
class B extends A{
    foo() {...}
}
class C extends B{
    foo() {...}
}
class D extends B{
    foo() {...}
}
  
```

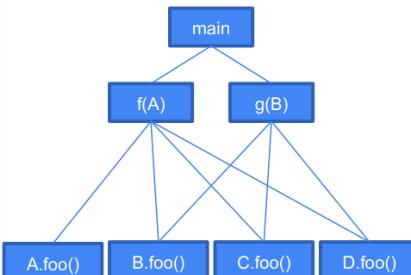


CHA recorre la estructura declarada de cada clase e infiere las posibles llamadas. Como *f* recibe un objeto de tipo A y este es superclase de B que a su vez es superclase de C y D, entonces se podría llamar a foo() de cualquiera de estas clases. Analogamente, en *g* se crea un objeto de tipo B que es superclase de C y D, por lo que puede llamar a foo() de dichas clases.

- Un “mapa” para saber que métodos analizar
 - Fundamental en programas orientados a objetos

```

static void main() {
    B b1 = new B();
    A a1 = new A();
    f(b1);
    g(b1);
}
static void f(A a2) {
    a2.foo();
}
static void g(B b2) {
    B b3 = b2;
    b3 = new C();
    b3.foo();
}
class A {
    foo(){...}
}
class B extends A{
    foo() {...}
}
class C extends B{
    foo() {...}
}
class D extends B{
    foo() {...}
}
  
```



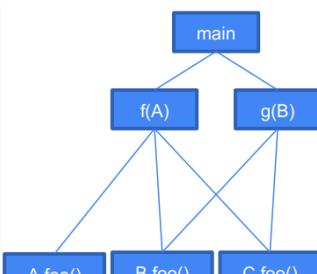
Calculado usando Class Hierarchy Analysis (CHA)

RTA recorre el programa buscando instancias de objetos y con ellos infiere los posibles llamados (además de usando los tipos de los objetos).

- Un “mapa” para saber que métodos analizar
 - Fundamental en programas orientados a objetos

```

static void main() {
    B b1 = new B();
    A a1 = new A();
    f(b1);
    g(b1);
}
static void f(A a2) {
    a2.foo();
}
static void g(B b2) {
    B b3 = b2;
    b3 = new C();
    b3.foo();
}
class A {
    foo(){...}
}
class B extends A{
    foo() {...}
}
class C extends B{
    foo() {...}
}
class D extends B{
    foo() {...}
}
  
```



Calculado usando Rapid Type Analysis (RTA)

Alias analysis

Objetivo: Hacer un análisis del código para modelar aliasing

aliasing: múltiples variables apuntan a la misma posición en memoria de forma directa o indirecta. (Para la interpretación de modelado: múltiples variables representan la misma locación mutable)

El análisis puede ser may alias (pueden cumplirse para algún camino) o must alias (se cumple para todo camino)

Como inferir aliasing

May alias vs Must alias

- **May:**

- Pares que pueden cumplirse para algún camino
 - (a,b), (a,c), (c,d), (a,d)

```
object a,b,c,d;
c = d
if(B)
    a = b;
else
    a = c;
```

- **Must:**

- Pares que deben cumplir para todo camino
 - (c,d)

- Dataflow: $Out(N) = Gen(N) + (In(N) - Kill(N))$;

- $a = b$
 - $Gen = \{ (a,x) \mid (b,x) \in In(N) \}$; $Kill = \{ (a,?) \}$
- $a = \text{new } A();$
 - $Gen = \{ \} ; Kill = \{ (a,?) \}$
- $a = b.f? a.f=b?$

Necesitamos recordar al objeto de tipo A?

- Necesitamos algún modelo de la memoria

Ignoramos los campos?

Con este modelado de generar tuplas no podemos modelar campos ($a = b.f; a.f = b$). Para ello necesitamos alguna forma de modelar la memoria (en particular el heap).

Points-to analysis

Objetivo: armar un análisis que nos diga a dónde apunta cada puntero

Un análisis de points-to es un may forward dataflow analysis que permite determinar a qué objetos puede apuntar una variable durante la ejecución de un programa.

Se simplifica a 4 operaciones

```
x = new C(); // new
x = y // copy
x = y.f // load
x.f = y // store
```

Tipos de análisis

- Sensibles a flujo: computan un análisis points-to para cada punto del programa (son caros)
- Insensibles al flujo: hacen un análisis points-to para todo el programa en general y todas las asignaciones son no destructivas. Hay 2 tipos
 - Andersen: basado en inclusión
 - Steengard: basado en unificación (no entra en el parcial)
- Sensibles a contexto: Pueden distinguir entre distintos llamados a métodos.

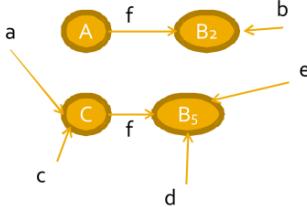
Points-to Graph (PTG)

Grafo donde tengo nodos por cada objeto (basicamente por cada new), nodos por cada puntero (o variable de tipo objeto, basicamente para trackear el aliasing), y ejes según las relaciones entre objetos, sus campos, y variables (punteros).

Para implementar el análisis utilizaremos un *points-to graph* (PTG) definido como $G = \langle N, E, L \rangle$ donde

- N es el conjunto de nodos del grafo, y representa los objetos creados por el programa. Cada nodo representa *todos* los objetos creados por una sentencia `new` del programa. Es decir, se creará un solo nodo incluso si el `new` se ejecuta dentro de un ciclo.
- E es el conjunto de aristas del grafo, y representa los objetos alcanzables desde cada objeto mediante campos. Una arista $e = (n_1, f, n_2)$ en el grafo indica que el nodo n_2 puede ser alcanzado desde el nodo n_1 mediante el campo f . Tener en cuenta que n_1 y n_2 representan ambos un conjunto de objetos.
- $L : Local \rightarrow P(N)$ es una función de etiquetado que asocia a cada variable local del programa un conjunto de nodos del grafo. Es decir, $L(x)$ es el conjunto de nodos a los que puede apuntar la variable x .

Gráficamente, un PTG = $\langle \{A, B_2, B_5, C\}, \{(A, f, B_2), (C, f, B_5)\}, [a \leftarrow \{C\}, b \leftarrow \{B_2\}, c \leftarrow \{C\}, d \leftarrow \{B_5\}, e \leftarrow \{B_5\}] \rangle$ luce de la siguiente forma:



El PTG debe ser utilizado como un reticulado por lo que hay que definir sus operaciones básicas:

- $\perp = (\{\}, \{\}, L)$: con $L(x) = \{\}$ para toda variable local.
- $G_1 \subseteq G_2$ si $L_1 \subseteq L_2$ ó $L_1 = L_2 \wedge N_1 \subseteq N_2$ ó $(L_1 = L_2 \wedge N_1 = N_2 \wedge E_1 \subseteq E_2)$.
- $G_1 \cup G_2 = \langle N_1 \cup N_2, E_1 \cup E_2, L_1 \cup L_2 \rangle$.

Las reglas Dataflow para el análisis son las siguientes:

Expresión	Efecto
$p : x = new A()$	$G' = G$ con $L'(x) = \{p\}$, donde p es el número de línea en el programa.
$x = y$	$G' = G$ con $L'(x) = L(y)$
$x = y, f$	$G' = G$ con $L'(x) = \{n_2 \mid (n_1, f, n_2) \in E \wedge n_1 \in L(y)\}$
$x, f = y$	$G' = G$ con $E' = E \cup \{(n_1, f, n_2) \mid n_1 \in L(x) \wedge n_2 \in L(y)\}$

Pueden asumir que no hay variables globales.

Resumen de las reglas en formato gráfico:

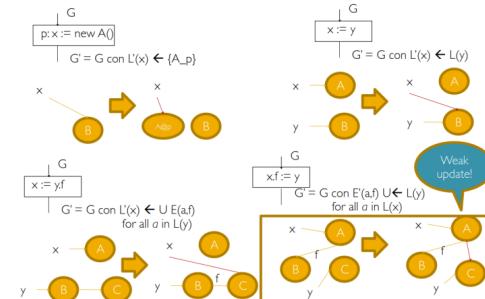
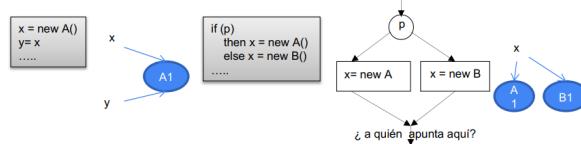


Figura 1: Weak update: No se sabe exactamente qué objeto va a ser actualizado; podría llegar por diferentes caminos.

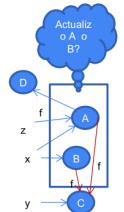
4ta operación dataflow

El weak update se justifica porque se tiene un análisis may. En el siguiente caso tenemos que x apunta a A y B . Luego si hacemos $x.f = y$ no sabemos si actualizar el campo f de A o B . Por eso ponemos ambas aristas.

- Si el points-to graph tiene ejes (a, A) y (a, B) una variable puede apuntar a A o B (MAY)
- Se puede mejorar con sensibilidad a caminos



- Strong update:
 - Cuando se sabe exactamente qué objeto está siendo escrito
 - Ejemplo: $x := y$
 - Se puede reemplazar la información de points-to de x
- Weak update:
 - No se sabe exactamente qué objeto va a ser actualizado.
 - Ejemplo: $x.f := y$
 - Durante el análisis puede no saberse a dónde podría apuntar x
- Pequeña mejora: si el grado de salida es 1 se puede aplicar un strong update



El PTG como reticulado calcula un PTG por cada punto del programa empezando con el grafo vacío y siguiendo la regla de inclusión definida en las operaciones del reticulado (imagen).

Abstracciones del heap

Podemos modelar el heap como un solo nodo, un nodo por cada allocation site (cada new). Este ultimo soluciona el problema de tener un new dentro de un bucle, en tal caso tendríamos que modelar que cada iteración es un objeto nuevo.

```
...
while(i < k) {
    A a = new A();
    b[i] = a;
}
...
...
```

Un nodo por allocation site

```

class L {
    N first;
    A data;
    L(int k) {
        0: N n = new NO;
        1: first = n;
        2: N cur = first;
        3: for(int i=1; i<k; i++) {
        4:     N n2 = new NO;
        5:     cur.next = n2;
        6:     cur = cur.next;
        }
    }
    void m0(L l) {
        0: m1();
        1: l.m1();
    }
    void m1(L l2) {
        0: N cur = first;
        1: while(cur!=l2) {
        2:     A a = new AO;
        3:     cur.data = a;
        4:     cur = cur.next;
        }
    }
    void main() {
        0: L l1 = new L(4);
        1: L l2 = new L(1);
        2: l1.m0(l2);
    }
}

```

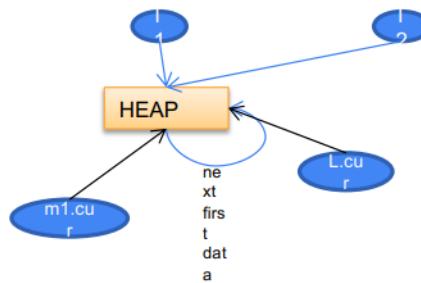
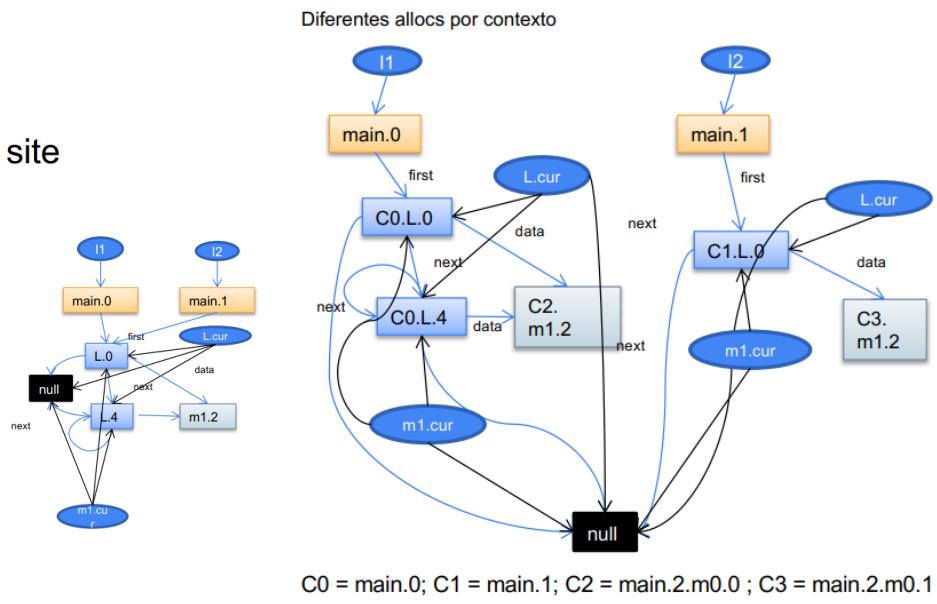


Figure 1: alt text

Andersen

Se basa en inclusiones: $p = q, pt(q) \subseteq pt(p)$. Es insensible a contexto y flujo (analiza todo el programa de una). Análisis may.

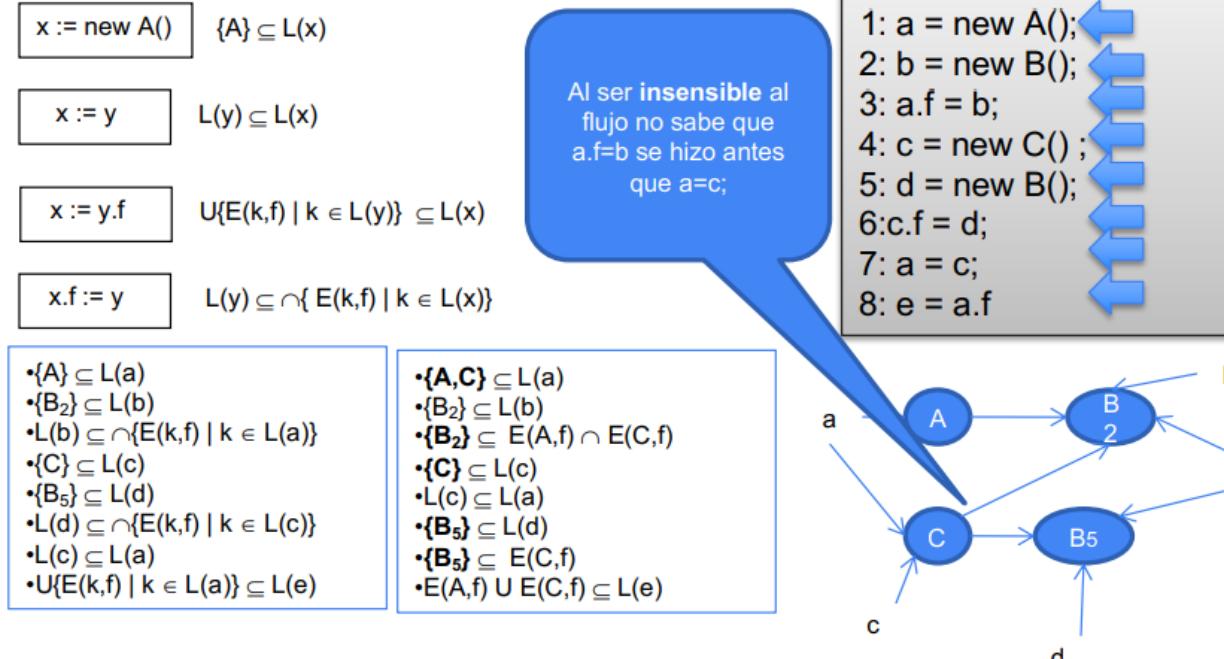
El algoritmo consiste en recorrer el código calculando las restricciones y luego resolverlas con la operación de inclusión hasta llegar a un punto fijo.

```

X = alloc P:   alloc-i ∈ [X]
X1 = &X2:  X2 ∈ [X1]
X1 = X2:  [X2] ⊆ [X1]
X1 = *X2: c ∈ [X2] ⇒ [c] ⊆ [X1] for each c ∈ Cell
*X1 = X2: c ∈ [X1] ⇒ [X2] ⊆ [c] for each c ∈ Cell

```

● Usando inclusiones de restricciones



En palabras, cada instrucción define restricciones a cumplir sobre los objetos del programa:

$x = \text{new}(A)$: A pertenece a los objetos apuntados por x

$x = y$: el conjunto de objetos apuntados por x deben tener a todos los objetos apuntados por y . (No se borran aristas, en la imagen justo se tapa pero la arista $a \rightarrow A$ no se borró).

$x = y.f$: los objetos apuntados por x deben tener a todos los objetos alcanzables por y a través de f

$x.f = y$: todos los objetos apuntados por y tienen que pertenecer a la intersección del conjunto de objetos formado por los objetos alcanzables por x a través de f

Verification

Dada una especificación S de un programa P , proveer una demostración rigurosa de que cualquier ejecución de P cumple con S

Dada una especificación S y un programa P generar un conjunto de fórmulas lógicas que, de ser derivables en base a un conjunto de axiomas y reglas de inferencia, implican que P cumple con S . Los predicados pueden ser demostrados de manera manual, con demostración semiautomática o con demostradores automáticos de fórmulas. No poder demostrar la fórmula no implica necesariamente que el programa contenga un error. Es decir, el verificador puede devolver proven (true o false) o unknown.

Ejemplo de un contrato

```

//@ requires x > 0
//@ ensures result * result == x
float raizCuadrada(int x) {
    return Math.sqrt(x);
}
  
```

Para esto necesitamos traducir el programa y el contrato a una lógica en común. Una posibilidad es representar la semántica del programa con axiomas. El programa es un teorema del conjuntos de axiomas y reglas de inferencia

Hoare

Triplas y lógica

$\{A\} \text{ código } \{B\}$: Si el programa comienza con estado A y la ejecución del código termina, entonces el estado final cumple B. (Notar correctitud parcial: SI el código termina entonces ..., total se obtiene probando que termina).

```
float hypo(float leg_a, float leg_b) {
    // @requires: leg_a > 0
    // @requires : leg_b > 0
    // @ensures: result > 0
    return Math.sqrt(leg_a**2 + leg_b**2)
}
```

$\{\text{leg_a} > 0 \& \text{leg_b} > 0\}$
 $\text{ret math.sqrt}(\text{leg_a}^{**2} + \text{leg_b}^{**2})$
 $\{\text{result} > 0\}$

provee un conjunto de reglas deductivas para probar la correctitud de programas respecto a la estructura de sentencias de programas imperativos. La lógica de Hoare modela cómo cada operación modifica el estado del sistema en término de predicados lógicos.

Lenguaje y reglas

```
skip      // nada
x:=e      // asignación
s1;s2    // secuencia

if (cond) {s1} else {s2}    // condicional
while (cond) {s}           // iteración
```

- Backward (bw): sirve para calcular la Weakest Precondition (para $\{P\} S \{Q\}$ si $\forall P' (P' \Rightarrow P)$ entonces P es la WP) > una tripla es válida si la precondición es más fuerte que la WP ($P \Rightarrow WP$)
- Forward (fw): sirve para calcular la Strongest Postcondition (para $\{P\} S \{Q\}$ si $\forall Q' (Q \Rightarrow Q')$ entonces Q es la SP) > una tripla es válida si la postcondición es más débil que la SP ($SP \Rightarrow Q$) (Es más difícil de computar)

$\{P\} \text{ skip } \{P\}$

$\{A\} s1 \{C\} \{C\} s2 \{B\}$

$\{A\} s1; s2 \{B\}$

$\{A \&& \text{cond}\} s1 \{B\} \quad \{A \&& \neg \text{cond}\} s2 \{B\}$

$\{A\} \text{ if(cond) } \{s1\} \text{ else } \{s2\} \{B\}$

$\frac{}{\{[b/x]A\} x := b \{A\}}$ (Bw)

$\{A \&& \text{cond}\} \text{ body } \{A\} \quad (A \&& \neg \text{cond}) \Rightarrow B$

$\frac{}{\{A\} x := b \{\exists x' | A[x'/x] \&& x == b[x'/x]\}}$ (Fw)

$\{A\} \text{ while(cond) } \{\text{body}\} \{B\}$

$\frac{}{A[b/x] \rightarrow \text{reemplazar } x \text{ en } A \text{ por } b}$

Regla forward:

$\{A\} x := E \{\exists x' | A[x'/x] \&& x == E[x'/x]\}$

- Intuición: x' es el valor anterior de x . ($\text{old}(x)$)

$\{x >= 3\} x := x+2 \{\exists x' | (x >= 3)[x'/x] \&& x == (x+2)[x'/x]\}$

$\{x >= 3\} x := x+2 \{\exists x' | x' >= 3 \&& x == x'+2\}$

$\{x >= 3\} x := x+2 \{\exists x' | x' >= 3 \&& x-2 == x'\}$

$\{x >= 3\} x := x+2 \{x-2 >= 3\}$

$\{x >= 3\} x := x+2 \{x >= 5\}$

Regla backward:

$\{B[x \rightarrow E]\} x := E \{B\}$

- Intuición: Para que B valga luego de la asignación, la precondición debe tomar en cuenta la relación previa entre x y E

Ejemplo:

$\{?\} x := 4 \{x == 4\}$
 $\{x == 4[x \rightarrow 4]\} x := 4 \{x == 4\}$
 $\{4 == 4\} x := 4 \{x == 4\}$
 $\{\text{true}\} x := 4 \{x == 4\}$

Otro ejemplo:

$\{?\} x := x+2 \{x >= 5\}$
 $\{x >= 5[x \rightarrow x+2]\}$
 $x := x+2 \{x >= 5\}$
 $\{x+2 >= 5\} x := x+2 \{x >= 5\}$
 $\{x >= 3\} x := x+2 \{x >= 5\}$

Verification Condition

Dado un contrato M ($pre_M, post_M$), calcular una formula lógica que permita inferir la correctitud del programa. (Si la fórmula es verdadera, el programa cumple el contrato)

- Forward: Dado $\{pre_M\} S \{post_M\}$ calculo $SP(S, pre_M)$ y verifico que $SP(S, pre_M) \implies \{post_M\}$, es decir que la postcondición más fuerte que puedo armar con la pre_M sea más fuerte que lo que pide la $post_M$ (post del contrato)
- Backward: Dado $\{pre_M\} S \{post_M\}$ calculo $WP(S, post_M)$ y verifico que $\{pre_M\} \implies WP(S, post_M)$, es decir que la precondition del contrato sea más restrictiva que la WP

Calcular WP

- $WP(\text{skip}, B) =_{\text{def}} B$
 - $WP(x := E, B) =_{\text{def}} B[x \rightarrow E]$
 - $WP(s_1; s_2, B) =_{\text{def}} WP(s_1, WP(s_2, B))$
 - $WP(\text{if}(E) \{s_1\} \text{else}\{s_2\}, B) =_{\text{def}}$
 $E \Rightarrow WP(s_1, B) \&$
 $!E \Rightarrow WP(s_2, B)$
- Los ciclos!**
- $WP_k(\text{while}(E) \{S\}, B) = \bigvee_{i=0}^k H_i(D)$
 - $H_0(\dots) = \text{def}(E) \wedge \neg E \wedge B,$
 - $H_1(\dots) =_{\text{def}} \text{def}(E) \wedge E \wedge wp(S, \neg E \wedge B))$
 $= \text{def}(E) \wedge E \wedge wp(S, H_0(B)).$
 -
 - $H_{k+1}(Q) \equiv \text{def}(E) \wedge E \wedge wp(S, H_k(B))$ para $k \geq 0.$
 - Calcularla de forma precisa puede ser imposible en general...

Ejemplo

- $WP(\text{skip}, B) =_{\text{def}} B$
- $WP(x := E, B) =_{\text{def}} B[x \rightarrow E]$
- $WP(s_1; s_2, B) =_{\text{def}} WP(s_1, WP(s_2, B))$

- $WP(\text{if}(E) \{s_1\} \text{else}\{s_2\}, B) =_{\text{def}}$
 $E \Rightarrow WP(s_1, B) \&$
 $!E \Rightarrow WP(s_2, B)$

```
bool P(bool a, bool b)
requires true
ensures c==a || b
{
    if (a)
        c=true
    else
        c=b
}
```

$WP(\text{if}(a \dots, c==a||b) =$
 $a \Rightarrow WP(c=true, c==a||b) \&$
 $!a \Rightarrow WP(c=b, c==a||b)$
 $= (a \Rightarrow \underline{\text{true}}==a||b) \& (!a \Rightarrow b==a||b)$

Conjetura lógica a probar:
 $\text{preM} \Rightarrow WP(P, c==a||b)$
 $\text{true} \Rightarrow (a \Rightarrow \underline{\text{true}}==a||b) \& (!a \Rightarrow b==a||b) \checkmark$

Solucion ciclos: loop unrolling Es una solución unsafe porque puede dar una precondition más débil que la WP

Loop unrolling

```
public m(int n)
 $\text{requires } x > 0 \ \&\& \ n > 0;$ 
 $\text{ensures } x \geq n$ 
{
    for(int i=0; i < n; i++)
    {
        x=x+1;
    }
}
```

```
public m'(int n)
 $\text{requires } x > 0 \ \&\& \ n > 0;$ 
 $\text{ensures } x \geq n$ 
{
    if(0 < n) {
        x=x+1;
        if(1 < n) {
            x=x+1
            if (2 < n) assume false; // aceptar programa
            else skip
            else skip
        }
        else skip
    }
}
```

- La $WP(m, x \geq n)$ real es $x \geq 0$
- Con 2 unrolls, $WP(m', x \geq n) = (n \leq 2 \Rightarrow x \geq 0)$
 - Vale que $x > 0 \ \&\& \ n > 0 \Rightarrow (n \leq 2 \Rightarrow x \geq 0)$
 - Pero unrolling produjo algo más débil...
 - Hint: Que pasa si la pre es $n = 3$ y $x = -1$?

por lo tanto se cumple la VC.1 y daria que el programa con el contrato es correcto

En este ejemplo $pre_M \implies WP'$ y $WP \implies WP'$ por ser más débil, pero pre_M no implica la WP . Dando erroneamente que el programa con la especificación es correcto.

Solucion ciclos: invariantes de ciclo No garantiza terminación

```
{A}
Init;
{I}
while (Cond) {
    {I & Cond}
    S;
    {I}
}
{I & !Cond => B}
{B}
```

Probar

- {A} Init {Inv}
- {Cond && Inv} S {Inv}
- !(Cond && Inv) => B

Entonces

- {A} while (Cond) {S} {B}

- First consider *partial correctness*
 - The loop may not terminate, but if it does, the postcondition will hold
- {P} while B do S {Q}
 - Find an invariant Inv such that:
 - $P \Rightarrow Inv$
 - The invariant is initially true
 - $\{ Inv \ \&\& \ B \} S \{ Inv \}$
 - Each execution of the loop preserves the invariant
 - $(Inv \ \&\& \ \neg B) \Rightarrow Q$
 - The invariant and the loop exit condition imply the postcondition

Extensión de lenguaje

```
WP(assume E, B) == (E => B)
WP(assert E, B) == (E && B)
// para todo valor de x vale B
WP(havoc x, B) == \forall x. B
```

$$\frac{\{\phi \wedge B\} \ p \ \{\phi\}}{\{\phi\} \ \text{while } B \text{ do } p \text{ od } \{\phi \wedge \neg B\}}$$

While (I) B do S end ==
assert I
havoc S
assume I
if (B) then
 S
 assert I
 assume false
endif

Comprueba que el invariante se mantienen al principio
Olvidar la información de las variables afectadas, usar solo la información proporcionada por el invariante
Chequear que el invariante se conserva después de ejecutar el cuerpo del bucle

Tratamiento de llamadas

[x/j] a x asigne j

```
void foo() {
...
j = -4
h = m(j);
...
}
```

Genera en el método

```
//@ requires true;
//@ ensures \retvalue == |x|;

public void m(int x) {
    if(x<0) return -x;
    return x;
}
```

Y en el llamador...

```
public void m(int x) {
    //@ assume true;
    if(x<0) return -x;
    return x;
    //@ \retvalue == |x|;
}
```

2 Opciones:

- Inlining
- Usar el contrato

```
...
j = -4
//@ assert true[x/j];
h = m(j);
//@ assume \retvalue==|x|[x/j][retvalue/h];
...
```

Genera en el método

```
//@ requires P;
//@ ensures Q;
//@ modifies M // lo que se modifica
public void m() {
    ...
}

//@ assert P[params/args];
v = m(args); (havoc M)
//@ assume Q[params/args][result/v];
```

Y en el llamador...

```
...
```

Invariante de representación

Es un predicado que deben cumplir todas las instancias de la clase. Al llamar a un método el invariante vale antes y despues (durante la ejecución puede romperse siempre y cuando se recomponga para el final).

```
class Racional
{
    private int numerador;
    private int denominador;
    //@ invariant mcd(numerador, denominador)==1 && denominador>0;

    //@ requires d>0;
    public Racional(int n, int d){
        int mcd = Math.gcd(n,d);
        numerador = n/mcd;
        denominador = d/mcd;
    }
}
```

- Los invariantes deben valer a la entrada y salida del método
 - A veces esto es muy restrictivo
- Helper methods:
 - Tienen que ser privados
 - No requieren que valga el invariante
 - Ni establecer el invariante

```
/* @helper */ void reconstruir()
{
    int mcd = Math.gcd(numerador,
                       denominador);
    numerador = numerador/mcd;
    denominador = denominador/mcd;
}
```

```
class Racional
{
    private int numerador;
    private int denominador;
    invariant mcd(numerador, denominador)== 1
    && denominador>0;

    public void suma(Racional r){
        numerador = numerador*r.denominador
        + r.numerador * denominador;
        denominador = denominador*r.denominador;
        reconstruir();
    }
}
```

- **Axioma 1.** $wp(x := E, Q) \equiv \text{def}(E) \wedge_L Q_E^x$.
- **Axioma 2.** $wp(\text{skip}, Q) \equiv Q$.
- **Axioma 3.** $wp(S1; S2, Q) \equiv wp(S1, wp(S2, Q))$.
- **Axioma 4.** $wp(\text{if } B \text{ then } S1 \text{ else } S2 \text{ endif}, Q) \equiv$

$$\text{def}(B) \wedge_L ((B \wedge wp(S1, Q)) \vee (\neg B \wedge wp(S2, Q)))$$
- **Observación:** $wp(b[i] := E, Q) \equiv wp(b := \text{setAt}(b, i, E), Q)$

$$\begin{aligned}
& wp(b[i] := E, Q) \\
& \equiv wp(b := \text{setAt}(b, i, E), Q) \\
& \equiv \text{def}(\text{setAt}(b, i, E)) \wedge_L Q_{\text{setAt}(b, i, E)}^b \\
& \equiv ((\text{def}(b) \wedge \text{def}(i)) \wedge_L 0 \leq i < |b|) \wedge \text{def}(E) \wedge_L Q_{\text{setAt}(b, i, E)}^b
\end{aligned}$$

Dados $0 \leq i, j < |b|$, recordemos que:

$$\text{setAt}(b, i, E)[j] = \begin{cases} E & \text{si } i = j \\ b[j] & \text{si } i \neq j \end{cases}$$