UNIVERSIDAD NACIONAL DE ROSARIO
FACULTAD DE CIENCIAS EXACTAS, INGENIERÍA Y AGRIMENSURA
LICENCIATURA EN CIENCIAS DE LA COMPUTACIÓN

La forma SSA en el testing de programas con arreglos

Tesina presentada por Luis M. Peñaranda
para optar por el título de Licenciado en Ciencias de la Computación

Director: Maximiliano Cristiá

Febrero de 2006

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

1. INTRODUCCIÓN

El testing es una práctica casi imprescindible en la construcción de programas, especialmente de gran tamaño. Esta disciplina consiste en desarrollar métodos que aseguren que una pieza de software se desempeñe de acuerdo a lo esperado. Encontramos dos grandes clases de testing: *estructural* y *funcional*. En la primera clase, donde se puede contar este trabajo, se utiliza la estructura del programa (código) para evaluar el comportamiento del mismo. En la segunda no interesa la implementación específica del programa, sino que se considera, por ejemplo, su especificación formal.

En el testing estructural suele usarse la información de flujo de control y de datos del programa. Esta información se presenta en forma de árbol. Se definen los *criterios* de testing de acuerdo a la forma de *cubrir* el árbol: un conjunto de datos de entrada hará que se recorra un determinado camino; un criterio especifica un conjunto de caminos que seben ser cubiertos por diversos conjuntos de datos de entrada. Un completo resumen de técnicas de verificación, validación y testing de programas puede encontrarse en la introducción de [5], en [6], en [13] o en cualquier libro de ingeniería de software.

El análisis discutido en este trabajo propone ejecutar el programa con diversos datos de entrada, para verificar que la salida sea la correcta. En [12] se plantea realizar un análisis estático del código y así determinar conjuntos de prueba que satisfagan ciertos criterios. El lenguaje de programación usado en dicho trabajo considera sólo variables escalares, sin tener en cuenta otras estructuras de datos. En [5] se propone usar una técnica de compiladores optimizantes (la forma *SSA*) para realizar estos análisis, que no dejan de ser estáticos. Luego se presentan diversos criterios de cubrimiento que usan SSA, análogos a los ya existentes. En el presente, se propone atacar el problema de los arreglos por medio de la técnica precedente, ensayando nuevos análisis a los datos del programa.

El plan del trabajo es el siguiente. En la sección 2 se presentará el lenguaje usado a lo largo del trabajo, un subconjunto muy limitado del lenguaje COBOL. En la sección 3 se comentarán algunas nociones sobre la forma *SSA*, luego de haber presentado conceptos de flujo de control y de flujo de datos. La sección 4 es la parte fundamental del trabajo, en ella se discutirán diversos criterios existentes para seleccionar conjuntos de prueba y se plantearán otros

necesarios para programas con arreglos. En la sección 5 se discutirá la implementación de la forma SSA y la construcción de una herramienta destinada a asistir el proceso de testing.

2. LENGUAJE DE PROGRAMACIÓN USADO

En el presente trabajo se usará un lenguaje de programación basado en COBOL. El lector se preguntará por qué se ha elegido este lenguaje, que no se caracteriza por su claridad sintáctica. La razón es muy simple: este es un proyecto de testing, no de programación y el COBOL es uno de los lenguajes más difíciles de depurar. Además, existe infinidad de programas de uso industrial escritos en este lenguaje. De todos modos, el pequeño subconjunto elegido no llega a captar la estrambótica estructura de un programa COBOL. Básicamente, el lenguaje es idéntico al utilizado en [5], con el agregado de arreglos.

Variables

Un identificador de variable puede ser cualquier palabra (*word*) no reservada de COBOL. Esto es, una secuencia de 1 a 30 carácteres del conjunto formado por las letras mayúsculas y minúsculas, dígitos y el signo - (menos). Una palabra debe tener al menos una letra y no puede empezar ni terminar con el signo menos. Es importante notar que COBOL no distingue mayúsculas de minúsculas.

Los detalles de declaración de variables no son relevantes por el momento, ya que en este punto interesa el análisis de los algoritmos. Este tema se discutirá más adelante, cuando se haga foco en la implementación. Para más información, puede consultarse [14] o cualquier manual de COBOL.

Arreglos

Los arreglos de tamaño `n` tienen elementos con subíndices entre `0` y `n-1`. El elemento i-ésimo del arreglo `A` se nota `A(i)`. En caso de arreglos de múltiples dimensiones, la notación es `A(j0, .., jm)`.

Sentencia de entrada: `ACCEPT x1, .., xn.`

Donde `x1, .., xn` son variables definidas por la sentencia, a las cuales se les asigna un valor obtenido del entorno de entrada.

Sentencia de salida: `DISPLAY x1, .., xn.`

Donde `x1, .., xn` son variables cuyos valores son enviados al entorno de salida.

Sentencia de asignación: `COMPUTE y = f(x1, .., xn).`

Donde `y` es una variable que es definida con el valor devuelto por la función `f(x1, .., xn)` que toma como argumento los valores de las variables `x1, .., xn`.

Sentencia de transferencia: `MOVE x TO y.`

Donde `x` es una variable o un valor e `y` una variable. Esta sentencia copia el valor de `x` a `y`.

Etiqueta: `etiq.`

Donde `etiq` es una secuencia de carácteres alfanuméricos cuyo primer carácter no es un número. Esta sentencia no tiene ningún efecto y al llegar el control del programa a la misma se continuará con la ejecución del programa en la sentencia siguiente.

Sentencia de salto: `GO TO etiq.`

Donde `etiq` es una secuencia de carácteres alfanuméricos cuyo primer carácter no es un número y aparece en alguna otra parte del programa a donde se pasará el control al ser ejecutada esta sentencia.

Sentencia condicional: `IF p(x1, .., xn) sent1 ELSE sent2.`

Donde `p(x1, .., xn)` es un predicado que utiliza los valores de las variables `x1, .., xn`. `sent1` y `sent2` son dos sentencias cualesquiera excepto etiquetas. De acuerdo al valor del predicado, verdadero o falso, se ejecutará la sentencia `sent1` o `sent2`, respectivamente. Si `sent1` o `sent2` son sentencias de salto, el control del programa pasa al destino respectivo; de lo contrario, la ejecución del programa continúa por la siguiente sentencia.

Sentencia de fin: `STOP RUN.`

Una vez que el control del programa llega a una sentencia de este tipo, causa la terminación de la ejecución del programa.

Un programa es una secuencia de sentencias. El programa será válido o legal si contiene una sentencia de fin y cada uno de los destinos de los saltos también está dentro del programa.

```
MAIN.
ACCEPT A(0).
MOVE 1 TO A(1).

PRINCIPIOWHILE.
IF A(0)>0
        GO TO PASO.
ELSE
        GO TO FIN.

PASO.
COMPUTE A(1) = A(1) * A(0).
COMPUTE A(0) = A(0) - 1.
GO TO PRINCIPIOWHILE.

FIN.
DISPLAY A(1).
STOP RUN.
```

Fig. 2.1: Programa que calcula el factorial de un número

La figura 2.1 muestra un programa de ejemplo que calcula el factorial de un número ingresado por el usuario. El uso de arreglos es simplemente demostrativo, no es necesario en este caso. El funcionamiento de este programa es trivial. Primero, se guarda el valor ingresado por el usuario en `A(0)` y se guarda `1` en `A(1)`, que va a contener el resultado parcial del cálculo. Luego se repite, hasta que `A(0)` valga `0`, el bloque que comienza con el identificador `PASO`, es decir, se multiplica el valor actual del cálculo por `A(0)`, que es luego decrementado. Al terminar de ciclar, se imprime el valor del factorial. Se observa que el programa es válido, porque finaliza con `STOP RUN.` y existen todas las etiquetas referidas (`PRINCIPIOWHILE`, `PASO` y `FIN`).

3. ANÁLISIS DEL PROGRAMA

La forma de trabajar de un compilador es, en esencia, transformar sucesivamente el programa de entrada hasta obtener la representación deseada (generalmente, lenguaje de máquina). Para realizar estas transformaciones, analiza en cada paso la información que dispone, obtenida en los pasos anteriores. La metodología de testing propuesta debe realizar análisis al programa, algunos de ellos semejantes a los que se encuentran en un compilador. En este capítulo, se desarrollarán algunos conceptos derivados de la teoría de compiladores útiles para el testing.

3.1. Flujo de control

El análisis del flujo de control de un programa consiste en construir un grafo dirigido que represente las sentencias del programa como nodos y los saltos como lados. Las sentencias consecutivas se agrupan en bloques básicos cuando no hay saltos ni etiquetas, es decir, un bloque básico está formado por sentencias que se ejecutan todas, secuencialmente, siempre en el mismo orden.

Se formalizará ahora el concepto. Un grafo de flujo de control es un grafo dirigido $G = (V, E)$, siendo V el conjunto de nodos y E el conjunto de lados. Cada nodo representa un bloque básico. De este modo, las sentencias condicionales son consideradas como tres sentencias: el predicado y las sentencias de destino del mismo. Un lado $v_1 \rightarrow v_2$ pertenece al grafo si y sólo si se cumple una de las condiciones siguientes:

- v_1 es un nodo (bloque básico) cuya última sentencia es un salto a una etiqueta `E` y v_2 es un nodo cuya primer sentencia es la etiqueta `E`.
- v_1 es un nodo cuya última sentencia es condicional y v_2 es un nodo cuya primer sentencia es uno de sus destinos posibles.
- existe un nodo n que termina con una sentencia condicional, v_1 es uno de sus destinos posibles y v_2 es la sentencia siguiente a la condicional que representa n.

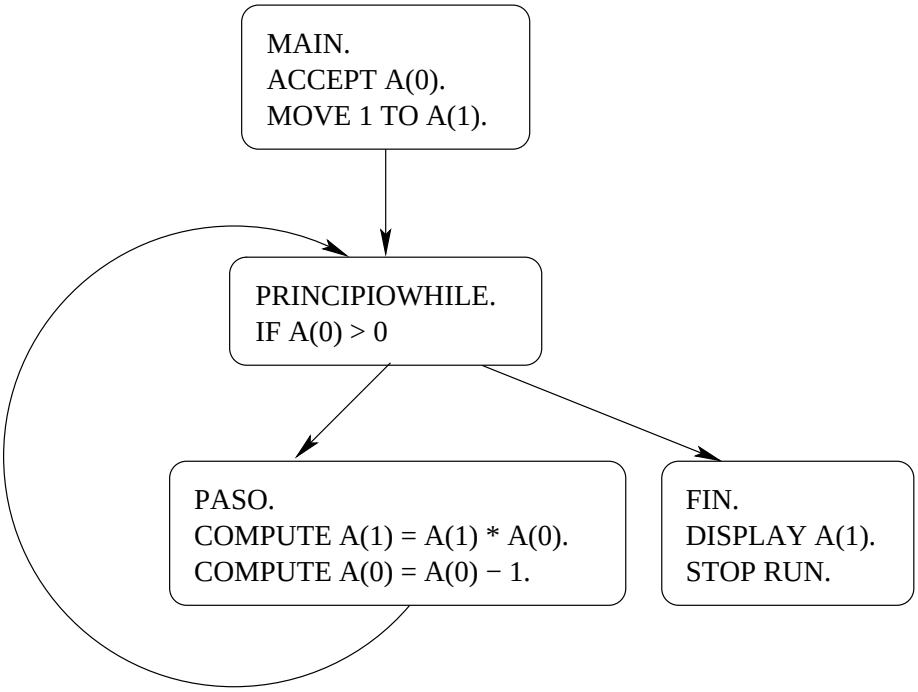


Fig. 3.1: Grafo de flujo de control del programa de la figura 2.1

La figura 3.1 muestra el grafo de flujo de control correspondiente al programa de la figura 2.1.

Debe notarse que, en caso de una sentencia condicional, las acciones a tomar no siempre son saltos. De todos modos, en el grafo de flujo de control, desde un nodo correspondiente a una sentencia condicional siempre habrá dos lados. Cuando la acción a tomar en una sentencia condicional es un salto, el bloque básico representado por el nodo al que se dirige el lado correspondiente comenzará con una etiqueta; de lo contrario, comenzará con cualquier otra sentencia.

Se llamará caso de prueba a un conjunto de valores de entrada que se le dan al programa para verificar su salida. Un criterio de cubrimiento es una propiedad que debe cumplir el conjunto de casos de prueba elegido para que se considere suficiente para ejercitar el programa. El análisis del flujo de control es utilizado para determinar criterios de cubrimiento. Esta técnica es apropiada para detectar errores en programas escritos en lenguajes que tienen saltos incondicionales, como el `GO TO` de COBOL, que hacen que el grafo de flujo de control sea más complejo y, por ende, los programas son más propensos a contener errores.

3.2. Flujo de datos

El análisis del flujo de datos consiste en recorrer el grafo de flujo de control para obtener información sobre las relaciones entre variables y cómo estas afectan la ejecución del programa.

Las técnicas de testing propuestas en [12] y [5] están basadas en el flujo de datos. Se discutirán a continuación los conceptos puntuales de dichos trabajos.

Cada ocurrencia de una variable se clasifica en *def* (definición), *c-uso* (uso computacional) o *p-uso* (uso en predicado). La distinción de la clase de uso tiene que ver estrictamente con el tipo de análisis que se realizará más adelante.

En el lenguaje propuesto, cada sentencia definirá o usará diversas variables de acuerdo a la siguiente tabla:

Sentencia	*def*	*c-uso*	*p-uso*
`COMPUTE y = f(x1, .., xn).`	`y`	`x1` a `xn`	
`MOVE x TO y.` (`x` es una variable)	`y`	`x`	
`MOVE` *valor* `TO y.`	`y`		
`ACCEPT x1, .., xn.`	`x1` a `xn`		
`DISPLAY x1, .., xn.`		`x1` a `xn`	
`IF p(x1, .., xn) sent1 ELSE sent2.`			`x1` a `xn`

En el último caso, `sent1` y `sent2` se consideran sentencias independientes, porque en el grafo serán representadas por otros nodos.

Cada nodo tiene asociados los *def*s de las variables que define y los *c-uso*s de las variables que usa para cómputos. Sin embargo, los nodos no tienen asociados *p-uso*s. Para entender la razón de esto, nótese que los *p-uso*s se encuentran sólo en sentencias condicionales. Están, entonces, siempre al final de un bloque básico y, por esta razón, obligan a agregar en el grafo dos lados con origen en el nodo actual. Esta propiedad hace que convenga asociar cada *p-uso* a lados y no a nodos.

Con las nociones precedentes, pueden ahora presentarse diversas formas de analizar los datos de un programa.

3.3. Definición-uso

Las cadenas definición-uso y uso-definición son usadas comúnmente por los compiladores para diversos análisis de datos, no necesariamente para optimizaciones. Se definirá aquí el grafo definición-uso como se hace en [12].

Cada ocurrencia de una variable se clasifica según su uso (definición, predicado o cómputo). Como lo que interesa es representar el flujo de datos entre nodos, cualquier definición que se usa solamente en el nodo donde ocurre es de poca importancia y será llamada *local*. En caso contrario, será *global* De esta forma se clasificarán los *def*s, *p-uso*s y *c-uso*s.

Se define un *camino* como una secuencia $(n_1, \ldots, n_k)$ de nodos del grafo de flujo de datos tal que existe el lado $n_i \rightarrow n_{i+1}$, para todo i entre 1 y $k-1$.

Sea x una variable del programa. Entonces:

- el camino $(i, n_1, \ldots, n_m, j)$, $m \geq 0$, que no tiene *def*s de x en los nodos $n_1, \ldots, n_m$, es un *camino libre de definición con respecto a x desde el nodo i hasta el nodo j.*

- el camino $(i, n_1, \ldots, n_m, j, k)$, $m \geq 0$, que no tiene *def*s de x en los nodos $n_1, \ldots, n_m, j$, es un *camino libre de definición con respecto a x desde el nodo i hasta el lado (j, k).*

- un lado (i, j) es un *camino libre de definición con respecto a x desde el nodo i al lado (i, j).*

- un *def* de la variable x en el nodo i es *global* si y sólo si es la última *def* de x en el bloque básico asociado al nodo i y hay un camino libre de definición con respecto a x desde i hasta un nodo que contiene un *c-uso* global de x o bien hasta un lado que contiene un *p-uso* de x. Es decir, un *def* global define una variable que será usada fuera del nodo donde se define.

- una *def* de x en el nodo i que no es global se dice *local* si y sólo si hay un *c-uso* local de x en el nodo i que sigue a esta *def* y no hay otra *def* de x entre la *def* anterior y el *c-uso*.

El grafo definición-uso se crea a partir del grafo de flujo de datos. Se asocia cada nodo i con dos conjuntos, correspondientes a los *def*s y a los *c-uso*s, y cada lado (i, j) con un conjunto, correspondiente a los *p-uso*s. Llamaremos a estos conjuntos *def*(i), *c-uso*(i) y *p-uso*(i, j), respectivamente.

Ahora se definen los conjuntos necesarios para enunciar más adelante los criterios. Sea i un nodo cualquiera y x una variable tal que $x \in$ *def*(i). Entonces:

- $dcu(x, i)$ es el conjunto de todos los nodos j tales que $x \in$ *c-uso*(j) y para los cuales hay un camino libre de definición con respecto a x desde i hasta j.

- $dpu(x, i)$ es el conjunto de todos los lados (j, k) tales que $x \in p\text{-}uso(j, k)$ y para los cuales hay un camino libre de definición con respecto a x desde i hasta (j, k).

Resumiendo, $dcu(x, n)$ es el conjunto de nodos donde se utiliza el valor de x definido en n; $dpu(x, n)$ es el conjunto de lados que tengan como inicio un nodo con una sentencia condicional que utilice el valor de x definido en n. Estos dos conjuntos se utilizan en [12] para definir criterios de cubrimiento en el programa.

3.4. La forma SSA

Muchos análisis del flujo de datos necesitan encontrar los lugares de uso de cada variable definida o los lugares de definición de las variables de una expresión. La cadena definición-uso es una estructura de datos que se encarga de esto. Para cada sentencia, el compilador mantiene una lista de punteros a todos los usos de las variables que se definen y otra a todas las definiciones de las variables usadas.

Una mejora en la idea es la forma de asignaciones estáticas únicas (static single-assignment, SSA). Es una representación intermedia en la cual cada variable tiene sólo una definición en el código del programa. Esta definición puede estar en un bucle, que hace que se ejecute varias veces: esta es la razón del nombre *estática*.

Podemos enumerar las razones de la utilidad de la forma SSA:

- los análisis del flujo de datos pueden hacerse más simples
- si una variable tiene n usos y m definiciones ($n + m$ instrucciones en el código del programa), la representación de cadenas definición-uso ocupará un espacio y tardará un tiempo proporcional a $n \times m$. En la práctica, la forma SSA ocupa un espacio lineal en el tamaño del programa original.
- los usos y definiciones de las variables en la forma SSA están relacionados directamente con la estructura de dominación del grafo de flujo de control, haciendo que algunos algoritmos se simplifiquen considerablemente.
- usos no relacionados de la misma variable en un programa resultan diferentes variables en la forma SSA, por lo que se eliminan relaciones innecesarias.

Pasar un programa a la forma SSA no es complicado. Dentro de un bloque básico, cada instrucción puede definir una nueva variable en lugar de redefinir una antigua. Cada uso de la variable se transforma en un uso de la nueva variable definida. Así, la variable x se transformará en x_1 en la primera definición, en x_2 en la segunda y así sucesivamente.

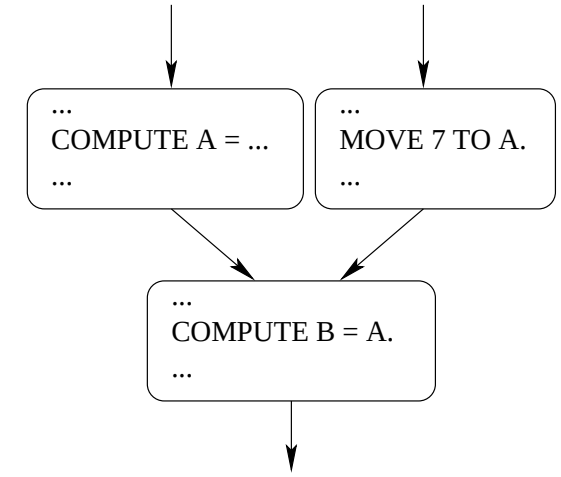
El problema se plantea cuando dos caminos de control se unen, como lo ilustra la figura 3.2(a). Al reemplazar un uso de una variable (`A` en la figura 3.2(a)), puede ocurrir que existan varios caminos posibles donde puede haberse realizado la definición de dicha variable. Se recurre en este caso a la inserción de una función mágica, llamada ϕ-función. Este es el caso de la figura 3.2(b): se necesita una ϕ-función para el cálculo de `B`. Esta función simplemente devuelve `A_1` o `A_2`, dependiendo del camino tomado por la ejecución del programa. El problema de su implementación será atacado más adelante.

La forma SSA permite realizar análisis complejos a costo relativamente bajo. Entre ellos, podemos mencionar:

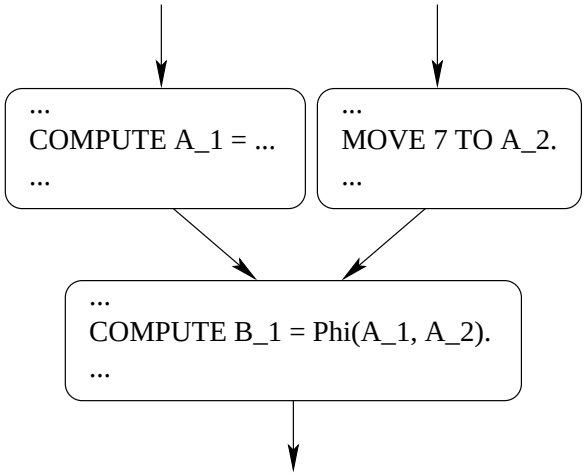
Eliminación de código muerto La estructura de datos de SSA hace este análisis particularmente rápido y fácil. Una variable está *viva* en su lugar de definición si y sólo si su lista de usos no es vacía (porque no puede haber otra definición de la misma variable y la definición de cada variable *domina* sus usos). Más adelante se estudiará la definición de dominancia.

Propagación de constantes Cuando hay una sentencia de la forma $x \leftarrow c$, donde c es una constante, puede reemplazarse en ella todo uso de x por un uso de c. Luego, cualquier aparición de la ϕ-función $\phi(c_1, .., c_n)$, donde todos los c_i son iguales a una constante c, puede ser reemplazada por c.

Eliminación agresiva de código muerto Para este tipo de optimización se construye el grafo de dependencias de control, a partir del grafo de flujo de control. Un nodo y es control-dependiente de x cuando desde x hay lados hacia u y v, desde u hay un camino a *exit* que no pasa por y y desde v todo camino a *exit* pasa por y (*exit* es el nodo de salida de un grafo de flujo de control; si hay varios, $e_1, .., e_n$, suponemos un nuevo nodo *exit* y lados $e_i \rightarrow exit$). Una vez construido el grafo de dependencias de control, el método es sencillo: de la misma forma que la propagación de constantes en condicionales asume que un bloque no es alcanzable hasta que haya evidencia de ello, la eliminación agresiva de código muerto asume que una sentencia está muerta hasta que se encuentra evidencia de que ésta contribuye al eventual resultado del



(a)



(b)

Fig. 3.2: Conversión a la forma SSA

programa. Esto se hace marcando como vivas las sentencias de entrada/salida, las que guardan datos en memoria o producen cualquier efecto lateral, definen alguna variable que se usa en otra sentencia y, si es un condicional, sobre el cual alguna otra sentencia viva es control-dependiente; después se eliminan las sentencias no marcadas.

En la sección 5.1 se discutirá la implementación de la forma SSA, presentando antes algunos conceptos necesarios para llevar a cabo esta tarea.

4. CRITERIOS

En esta sección se explicarán distintos criterios de cubrimiento existentes. Después, se comentarán sus análogos para programas en la forma SSA. Se plantearán luego algunos problemas encontrados aplicando los criterios precedentes a programas con arreglos, se introducirán otros criterios con el fin de solucionar dichos problemas y se propondrá una metodología de verificación de programas con arreglos.

4.1. Criterios basados en el flujo de datos

Los criterios que se presentarán en esta sección son los propuestos en [12]. Sea G un grafo de definición-uso y P un conjunto de caminos completos en G. Entonces:

- P satisface el criterio *todos-los-nodos* si todo nodo de G está incluido en algún camino de P.
- P satisface el criterio *todos-los-lados* si todo lado de G está incluido en algún camino de P.
- P satisface el criterio *todas-las-definiciones* si por cada nodo n de G y cada variable x definida en n, P contiene un camino libre de definición con respecto a x desde n hasta un nodo de $dcu(x, n)$ o hasta un lado de $dpu(x, n)$.
- P satisface el criterio *todos-los-p-usos* si por cada nodo n de G y cada variable x definida en n, P contiene un camino libre de definición con respecto a x desde n hasta todos los lados de $dpu(x, n)$.
- P satisface el criterio *todos-los-c-usos/algunos-p-usos* si por cada nodo n de G y cada variable x definida en n, P contiene algún camino libre de definición con respecto a x desde n hasta todos los nodos del conjunto $dcu(x, n)$ cuando $dcu(x, n) \neq \varnothing$ o hasta algún lado del conjunto $dpu(x, n)$ cuando $dcu(x, n) = \varnothing$.

- P satisface el criterio *todos-los-p-usos/algunos-c-usos* si por cada nodo n de G y cada variable x definida en n, P contiene algún camino libre de definición con respecto a x desde n hasta todos los lados del conjunto $dpu(x, n)$ cuando $dpu(x, n) \neq \varnothing$ o hasta algún lado del conjunto $dcu(x, n)$ cuando $dpu(x, n) = \varnothing$.

- P satisface el criterio *todos-los-usos* si por cada nodo n de G y cada variable x definida en n, P contiene algún camino libre de definición con respecto a x desde n hasta todos los nodos del conjunto $dcu(x, n)$ y hasta todos los lados del conjunto $dpu(x, n)$.

- P satisface el criterio *todas-las-definiciones-usos* si por cada nodo n de G y cada variable x definida en n, P contiene todos los caminos libres de bucles y libres de definición con respecto a x desde n hasta todos los nodos del conjunto $dcu(x, n)$ y hasta todos los lados del conjunto $dpu(x, n)$. (Decimos que un camino $n_1, \ldots, n_k$ es libre de bucles si y sólo si $n_i \neq n_j$, para todo $i \neq j$.)

- P satisface el criterio *todos-los-caminos* si contiene todos los caminos completos de G. Nótese que G puede tener infinitos caminos completos, dada la presencia de bucles.

4.2. Criterios basados en el flujo de datos con SSA

Se explicarán los criterios presentados en [5]. Primero se definen algunos conceptos.

- Se llamará ϕ-*vars*(x_i) al conjunto de todas las variables x_k tales que exista una asignación de la forma $x_k = \phi(\ldots)$, donde x_i o algún elemento de ϕ-*vars*(x_i) aparezca como argumento de la función ϕ.

- En la forma SSA se dirá que un nodo tiene un *c-uso-ssa* de una variable x_i si y sólo si ese nodo tiene un uso de la variable x_i, o de alguna de las variables de ϕ-*vars*(x_i), y el uso no ocurre en una sentencia condicional.

- Un nodo tendrá un *p-uso-ssa* de una variable x_i si y sólo si ese nodo tiene un uso de la variable x_i, o de alguna de las variables de ϕ-*vars*(x_i), y el uso ocurre en el predicado de una sentencia condicional.[1]

[1] Debe tenerse en cuenta que los *p-usos* están asociados a lados en [12] y aquí están asociados a nodos. Esta es la razón por la cual se pide, en los criterios que aparecen más adelante, que los caminos lleguen hasta los *sucesores* de los nodos donde hay *p-usos*.

Sea S un grafo, resultante de obtener la forma SSA de un programa, y P un conjunto de caminos completos de S. Entonces:

- P satisface el criterio *todos-los-nodos(ssa)* si todo nodo de S está contenido en algún camino de P.
- P satisface el criterio *todos-los-lados(ssa)* si todo lado de S está contenido en algún camino de P.
- P satisface el criterio *todas-las-definiciones(ssa)* si por cada nodo n de S y por cada variable x_i definida en n (que no corresponda a la definición de una ϕ-función) existe un camino en P desde n hasta un nodo que contenga un *c-uso-ssa* o un *p-uso-ssa* de x_i.
- P satisface el criterio *todos-los-p-usos(ssa)* si por cada nodo n de S y por cada variable x_i definida en n (que no corresponda a la definición de una ϕ-función) existen en P caminos desde n hasta todos los sucesores de todos los nodos que contengan un *p-uso-ssa* de x_i.
- P satisface el criterio *todos-los-c-usos/algunos-p-usos(ssa)* si por cada nodo n de S y por cada variable x_i definida en n (que no corresponda a la definición de una ϕ-función) existen en P caminos desde n hasta todos los nodos en los que haya un *c-uso-ssa* de x_i. En caso de no haber ningún nodo que contenga un *c-uso-ssa* de x_i en S, P debe contener un camino desde n hasta cada uno de los sucesores de un nodo que contenga un *p-uso-ssa* de x_i.
- P satisface el criterio *todos-los-p-usos/algunos-c-usos(ssa)* si por cada nodo n de S y por cada variable x_i definida en n (que no corresponda a la definición de una ϕ-función) existen en P caminos desde n hasta todos los sucesores de todos los nodos que contengan un *p-uso-ssa* de x_i. En caso de no haber ningún ningún nodo que contenga un *p-uso-ssa* de x_i en S, P debe contener un camino desde n hasta un nodo que contenga un *c-uso-ssa* de x_i.
- P satisface el criterio *todos-los-usos(ssa)* si por cada nodo n de S y por cada variable x_i definida en n (que no corresponda a la definición de una ϕ-función) existen en P caminos desde n hasta todos los nodos en los que haya un *c-uso-ssa* de x_i y hasta todos los sucesores de todos los nodos que contengan un *p-uso-ssa* de x_i.
- P satisface el criterio *todas-las-definiciones-usos(ssa)* si por cada nodo n de S y por cada variable x_i definida en n (que no corresponda a

la definición de una ϕ-función) P contiene todos los caminos libres de bucles desde n hasta todos los nodos en los que haya un *c-uso-ssa* de x_i y hasta todos los sucesores de todos los nodos que contengan un *p-uso-ssa* de x_i.

- P satisface el criterio *todos-los-caminos(ssa)* si P contiene todos los caminos del grafo S.

Nótese que, en estas definiciones, se hace innecesario el uso de *caminos libres de definición* y sólo se hace referencia a *caminos*, dada la naturaleza de asignación única de la forma SSA.

Después de definir los criterios precedentes, en [5] se demuestra la equivalencia de los mismos con los de [12]. Posteriormente se prueba que no es necesario el uso de los conjuntos *dcu* y *dpu*, mostrando así la ventaja de la forma SSA a la hora de implementar.

4.3. Criterios propuestos para programas con arreglos

La primera aproximación a programas con arreglos usando los criterios de las secciones precedentes es tomar cada arreglo como una variable. De ese modo, se estaría agrupando a todos los miembros del arreglo dentro de una variable y se crearían así asociaciones definición-uso innecesarias. Por este motivo, se descartó esta idea.

Otro enfoque alternativo es pensar cada miembro del arreglo como una variable diferente. Por ejemplo, el miembro i-ésimo de un arreglo, `A(i)`, sería considerado una variable `A_i`. Cuando el subíndice es una constante esta técnica crea las dependencias esperadas, pero cuando esto no ocurre se genera un nuevo problema. Se deberá decidir a qué variable se hace referencia; como esto no siempre es posible se deberán agregar, adoptando una postura conservadora, todas las dependencias posibles. Igual que antes, se generan dependencias superfluas y esta aproximación queda descartada.

Se adoptará en este trabajo una visión más cercana al último enfoque, pero la principal característica es el tratamiento de variables escalares y arreglos en forma diferente.

En primer lugar, y para evitar complicaciones con las próximas definiciones, se estudiará la naturaleza de los *p-uso*s. Obsérvese que, cuando un bloque básico termina con un condicional, existen dos saltos posibles. Cuál de ellos se seguirá, depende del valor de ciertas variables. Cuando se construye el grafo definición-uso, estas variables están asociadas con los dos lados que representan los saltos posibles, haciendo que cada uno de los lados tenga *p-uso*s de las mismas variables. Por lo tanto, no se comete ningún error al asociar

*p-uso*s de una variable a nodos, su aparición no será ambigua: significará que los lados que parten de él representando posibles saltos contienen *p-uso*s de la variable. (Debe tenerse en cuenta la consideración al pie de la página 19; esta es la razón por la cual se usan sucesores de estos nodos.)

Debe resolverse ahora qué clase de uso es la aparición de una variable en un subíndice. No es un *p-uso*, porque no se usa el valor de la variable en un predicado. Tampoco es un *c-uso*, porque el valor de la variable no se usa en un cómputo. Entonces, análogamente a los *c-uso*s y *p-uso*s, se definirá un nuevo tipo de uso.

Definición 1. *Una sentencia contendrá un* i-uso *de la variable x cuando aparezca en ella la variable x en el subíndice de un arreglo.*

El grafo de definición-uso de [12] tendrá ahora asociados a cada nodo los conjuntos *c-uso*, *def* e *i-uso*, y asociado a cada lado el conjunto *p-uso*. Debe notarse que un *i-uso* puede aparecer dentro de un *c-uso* o de un *p-uso* de un arreglo; esto significa que puede estar en un nodo o en un lado y es la razón por la cual se hizo la observación más arriba de la aparición de un *p-uso* en un nodo. También debe tenerse en cuenta que pueden aparecer en una subindización cero, uno o más *i-uso*s de variables; por ejemplo, la aparición de `A[4]` no contiene *i-uso*s de ninguna variable, `A[x]` contiene un *i-uso* de la variable `x` y `A[3*x+y/z]` contiene *i-uso*s de las variables `x`, `y` y `z`.

4.3.1. Criterios para verificar variables escalares

De la misma forma que antes, debe definirse un nuevo conjunto de asociaciones definición-uso en forma análoga a *dcu* y *dpu*.

Definición 2. *Sean x una variable e i un nodo tales que $x \in def(i)$. $diu(x, i)$ es el conjunto de todos los nodos j tales que $x \in$* i-uso(j) *y existe un camino libre de definición con respecto a x desde i hasta j.*

Los criterios anteriores estaban basados en los conjuntos *dcu* y *dpu*. Con las definiciones precedentes, pueden crearse criterios que tengan en cuenta, además, al conjunto *diu*. Con el uso de este último concepto, puede hacerse foco en el flujo de datos producido por los arreglos.

Podemos enunciar otros criterios análogos a los planteados en [12], contemplando la posible presencia de arreglos. Sea G un grafo de definición-uso y P un conjunto de caminos completos en G. Entonces:

- P satisface el criterio *todos-los-nodos(arr)* si todo nodo de G está incluido en algún camino de P.

- P satisface el criterio *todos-los-lados(arr)* si todo lado de G está incluido en algún camino de P.

- P satisface el criterio *todas-las-definiciones(arr)* si por cada nodo n de G y cada variable x definida en n, P contiene un camino libre de definición con respecto a x desde n hasta un nodo de $dcu(x, n)$ o hasta un nodo de $diu(x, n)$ o hasta un lado de $dpu(x, n)$

- P satisface el criterio *todos-los-p-usos(arr)* si por cada nodo n de G y cada variable x definida en n, P contiene un camino libre de definición con respecto a x desde n hasta todos los lados de $dpu(x, n)$ y hasta todos los sucesores de todos los nodos de $diu(x, n)$ donde x esté usado como subíndice de un arreglo en un predicado.

- P satisface el criterio *todos-los-c-usos/algunos-p-usos(arr)* si por cada nodo n de G y cada variable x definida en n, P contiene un camino libre de definición con respecto a x desde n hasta:

  1. todos los nodos de $dcu(x, n)$ y, si es vacío, hasta algún lado de $dpu(x, n)$, y
  2. todos los sucesores de todos los nodos de $diu(x, n)$ donde x esté usado subindizando un arreglo usado para cálculos[2] y, si no hay en $diu(x, n)$ ningún nodo que cumpla esta condición, hasta algún nodo de $diu(x, n)$ donde x esté usado como subíndice de un arreglo en un predicado.

- P satisface el criterio *todos-los-p-usos/algunos-c-usos(arr)* si por cada nodo n de G y cada variable x definida en n, P contiene un camino libre de definición con respecto a x desde n hasta:

  1. todos los lados de $dpu(x, n)$ y, si es vacío, hasta algún nodo de $dcu(x, n)$, y
  2. todos los sucesores de todos los nodos de $diu(x, n)$ donde x esté usado subindizando un arreglo usado en un predicado[3] y, si no hay en $diu(x, n)$ ningún nodo que cumpla esta condición, hasta algún nodo de $diu(x, n)$ donde x esté usado como subíndice de un arreglo usado para cómputo.

- P satisface el criterio *todos-los-usos(arr)* si por cada nodo n de G y cada variable x definida en n, P contiene un camino libre de definición

[2] es decir, el elemento referido del arreglo es usado para cálculos
[3] es decir, el elemento referido del arreglo es usado en un predicado

con respecto a x desde n hasta todos los nodos de $dcu(x, n)$, hasta todos los lados de $dpu(x, n)$ y hasta todos los nodos de $diu(x, n)$.

- P satisface el criterio *todas-las-definiciones-usos(arr)* si por cada nodo n de G y cada variable x definida en n, P contiene todos los caminos libres de bucles y libres de definición con respecto a x desde n hasta todos los nodos de $dcu(x, n)$, hasta todos los lados de $dpu(x, n)$ y hasta todos los nodos de $diu(x, n)$.

- P satisface el criterio *todos-los-caminos(arr)* si por cada nodo n de G y cada variable x definida en n, P contiene todos los caminos completos de G.

Como puede concluirse por la observación de los enunciados precedentes, al aplicar estos criterios a un programa sin arreglos, los conjntos *diu* resultarán vacíos. Así, los conjuntos de caminos asociados a cada criterio serán exactamente los mismos que los propuestos originalmente en [12].

Debe aclararse cuál es el motivo de introducir estos nuevos criterios. Es decir, qué es lo que se testea ahora que no era testeado antes. Los criterios anteriores hacían referencia a los *c-uso*s y a los *p-uso*s; esto es, se ejercitaban caminos comprendidos entre la definición de una variable y su uso en un cómputo o en un predicado. Pero, en el lenguaje presentado, puede haber otro uso de una variable: como subíndice en un arreglo. Entonces, se captó con los nuevos criterios la necesidad de ejercitar el camino que comprende desde la definición de una variable hasta su uso en subíndices; pueden detectarse ahora errores que provengan de la errónea subindización y hubieran pasado inadvertidos con los criterios anteriores.

Para terminar de enunciar los criterios, debe considerarse la simpleza que brinda la forma SSA para el testing, como se sugiere en [5]. En ese trabajo, se enuncia un conjunto de criterios análogos a los propuestos en [12] que, con mayor simpleza, logran asociarse con los mismos conjuntos de prueba. Aquí tratará de hacerse lo mismo, proponiendo criterios análogos a los recientemente mostrados.

Antes de hacerlo, debe decidirse cómo tratar los arreglos al pasar el programa a la forma SSA. La forma Array SSA propuesta en [7] aparece como una buena opción, pero fue concebida para la construcción de compiladores paralelos y no se obtendría una gran mejora al aplicarla al testing. El enfoque adoptado, como se mencionó antes, es distinguir entre variable escalares y arreglos. Se contemplará el caso en el cual los subíndices de los arreglos son variables escalares o constantes, dejando para algún trabajo futuro la subindización de arreglos con elementos de arreglos.

Análogamente a [5], se definirá un análogo al *i-uso* para la forma SSA.

Definición 3. *En la forma SSA se dirá que un nodo n tiene un* i-uso-ssa *de una variable x_i si y sólo si en n aparece la variable x_i, o alguna de las variables de ϕ-vars(x_i), como subíndice de un arreglo.*

Con esta nueva defición, pueden plantearse criterios análogos a los tres grupos anteriores, que contemplen el uso de arreglos y consideren al programa en la forma SSA.

Sea S un grafo resultante de obtener la forma SSA de un programa y P un conjunto de caminos completos en S. Entonces:

- P satisface el criterio *todos-los-nodos(arr-ssa)* si todo nodo de S está incluido en algún camino de P.
- P satisface el criterio *todos-los-lados(arr-ssa)* si todo lado de S está incluido en algún camino de P.
- P satisface el criterio *todas-las-definiciones(arr-ssa)* si por cada nodo n de S y cada variable x_i definida en n (que no corresponda a la definición de una ϕ-función) existe un camino en P desde n hasta un nodo que contenga un *c-uso-ssa*, un *p-uso-ssa* o un *i-uso-ssa* de x_i.
- P satisface el criterio *todos-los-p-usos(arr-ssa)* si por cada nodo n de S y cada variable x_i definida en n (que no corresponda a la definición de una ϕ-función) existen en P caminos desde n hasta todos los sucesores de todos los nodos que contengan un *p-uso-ssa* de x_i y hasta todos los sucesores de todos los nodos que contengan un *i-uso-ssa* de x_i donde x_i esté usado como subíndice de un arreglo en un predicado.
- P satisface el criterio *todos-los-c-usos/algunos-p-usos(arr-ssa)* si por cada nodo n de S y cada variable x_i definida en n (que no corresponda a la definición de una ϕ-función) existen en P caminos desde n hasta:
  1. todos los sucesores de todos los nodos que contengan un *c-uso-ssa* de x_i y, si ningún nodo cumple esta condición, hasta un sucesor de algún nodo que contenga un *p-uso-ssa* de x_i, y
  2. todos los nodos que contengan un *i-uso-ssa* de x_i, donde x_i esté usado subindizando un arreglo usado para cálculos y, si ningún nodo contiene un *i-uso-ssa* computacional de x_i, hasta un sucesor de algún nodo que contenga un *i-uso-ssa* de x_i donde x_i esté usado como subíndice de un arreglo en un predicado.
- P satisface el criterio *todos-los-p-usos/algunos-c-usos(arr-ssa)* si por cada nodo n de S y cada variable x_i definida en n (que no corresponda a la definición de una ϕ-función) existen en P caminos desde n hasta:

1. todos los sucesores de todos los nodos que contengan un *p-uso-ssa* de x_i y, si ninguno cumple esta condición, hasta algún nodo que contenga un *c-uso-ssa* de x_i, y
2. todos los sucesores de todos los nodos que contengan un *i-uso-ssa* de x_i donde x_i esté usado subindizando un arreglo usado en un predicado y, si ninguno cumple esta condición, hasta un sucesor de algún nodo que contenga un *i-uso-ssa* de x_i donde x_i esté usado como subíndice de un arreglo usado para cómputo.

- P satisface el criterio *todos-los-usos(arr-ssa)* si por cada nodo n de S y cada variable x_i definida en n (que no corresponda a la definición de una ϕ-función) existen en P caminos desde n hasta todos los nodos que contengan un *c-uso-ssa* de x_i, hasta todos los sucesores de todos los nodos que contengan un *p-uso-ssa* de x_i y hasta todos los nodos que contengan un *i-uso-ssa* de x_i.

- P satisface el criterio *todas-las-definiciones-usos(arr-ssa)* si por cada nodo n de S y cada variable x_i definida en n (que no corresponda a la definición de una ϕ-función) P contiene todos los caminos libres de bucles desde n hasta todos los nodos que contengan un *c-uso-ssa* de x_i, hasta todos los sucesores de todos los nodos que contengan un *p-uso-ssa* de x_i y hasta todos los nodos que contengan un *i-uso-ssa* de x_i.

- P satisface el criterio *todos-los-caminos(arr-ssa)* si P contiene todos los caminos completos de S.

Estos criterios, como puede observarse, son análogos a los tres grupos de criterios planteados anteriormente. Pueden usarse para verificar definiciones y usos de las variables escalares dentro de un programa con arreglos. El último paso para poder verificar completamente este tipo de programas es definir criterios que contemplen las cadenas DUA.

4.3.2. Cadenas definición-uso de arreglos (DUA)

Supóngase que, en un programa con arreglos, se conocen de antemano todos los valores de los distintos subíndices usados. Esto es prácticamente lo mismo que suponer que todos los subíndices son constantes. La pregunta es, ¿qué cadenas definición-uso deben tenerse en cuenta? En este caso, la respuesta es simple: basta sólo con considerar las cadenas que consisten en la definición de un elemento `A(k)` a los usos de `A(k)`, con `k` constante. Es totalmente innecesario considerar, por ejemplo, la cadena formada por la

definición de `A(4)` a un uso de `A(5)`, porque se está definiendo un elemento y usando otro.

Lamentablemente, no siempre se conocen de antemano los índices de los arreglos; más aun, esto no pasa casi nunca. Podemos diferenciar tres clases de cadenas definición-uso de arreglos (DUA):

cadena definida: cuando, como en el caso anterior, se define un elemento de un arreglo y se usa el mismo elemento. El subíndice puede ser constante o una variable escalar, mientras exista la seguridad de que tiene siempre el mismo valor.[4] La figura 4.1(a) muestra estos casos.

cadena indefinida: si no se tiene certeza sobre la igualdad o desigualdad del subíndice de la definición y del uso. Esto pasa cuando se define un elemento con subíndice constante y se usa un elemento cuyo subíndice es una variable escalar o viceversa. También ocurre cuando se define un elemento cuyo subíndice es una variable escalar y se usa otro cuyo subíndice es otra variable. En la figura 4.1(b) pueden verse ejemplos de esto.

cadena aparente: este caso se da cuando se define un elemento y se usa otro, teniendo certeza de que nunca será el mismo elemento (esto es, los subíndices de la definición y el uso son constantes distintas). Hay una ilustración de este caso en la figura 4.1(c).

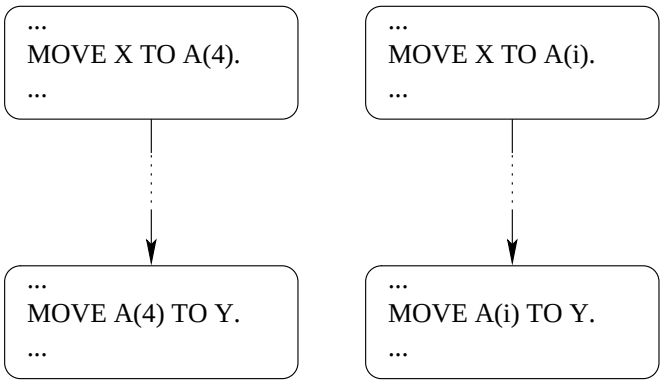
4.3.3. Criterios para verificar cadenas DUA

Como se vio en la sección 4.3.2, existen tres tipos de cadenas definición-uso de arreglos. Es trivial que no tiene ningún sentido recorrer una DUA aparente, porque es una dependencia inexistente: se están definiendo y usando datos en distinta posición de memoria. Sin embargo, es necesario verificar las DUA definidas e indefinidas; en las primeras se tiene la seguridad de que la asociación no es aparente y en las segundas se adopta una postura conservadora ante la posibilidad de que exista asociación.

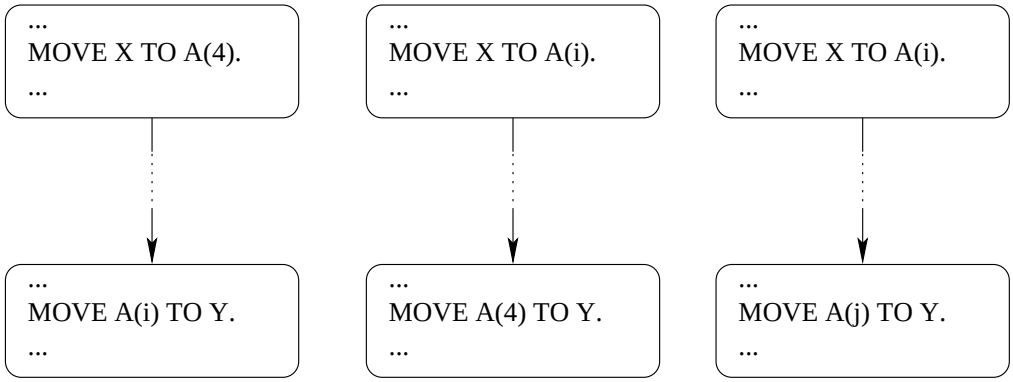
Afortunadamente, se propuso que el programa esté en la forma SSA. Esto simplifica el análisis de arreglos, porque la propagación de constantes puede transformar DUA indefinidas en definidas o aparentes (esto es, se eliminan dependencias innecesarias, una propiedad de la forma SSA).

Con estas consideraciones, podemos tratar cada DUA definida o indefinida en forma análoga a la sección 4.1 (y, como se dijo antes, pueden olvidarse

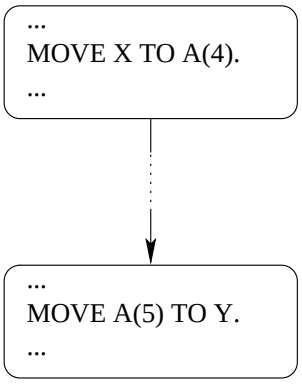
[4] Esto ocurre cuando el programa está en la forma SSA. En caso de no tener la forma SSA del programa, debe pedirse que el camino sea libre de definición con respecto a la variable escalar



(a) cadenas definidas



(b) cadenas indefinidas



(c) cadena aparente

Fig. 4.1: Ejemplos de cadenas definición-uso de arreglos (DUA)

las aparentes). Cada uso de un arreglo aparecerá en un cómputo o en un predicado (es decir, un *c-uso* o un *p-uso*) y se formarán conjuntos *dcu* y *dpu*. Los criterios para verificar cadenas DUA serán, en conclusión, exactamente los mismos que en 4.1.

4.4. Verificación de programas con arreglos

Como conclusión de la sección 4, se propone la forma de verificar programas con arreglos usando la forma SSA, el objetivo del trabajo.

Deben verificarse, naturalmente, tanto las variables escalares como los arreglos. Lo primero se llevará a cabo mediante los criterios de la sección 4.3.1 y lo último aplicando los criterios de [12] a las cadenas DUA, como se explicó en la sección 4.3.3.

Esta metodología de verificación en dos fases permite al testeador hacer foco, según su necesidad, en las variables escalares o en los arreglos. Puede también optar por elegir criterios análogos[5] para llevar a cabo la verificación de escalares y de arreglos y, así, evaluar con la misma profundidad todas las variables.

Por ejemplo, supóngase querer testear un programa que posee tanto variables escalares como arreglos. La primera fase consiste en verificar las variables escalares: se elije para esto un criterio, por ejemplo *todos-los-p-usos/algunos-c-usos(arr)* y se busca un conjunto de caminos que lo cumpla, al que llamaremos C_1. La segunda fase consiste en verificar las cadenas DUA: se elije el criterio de [12] análogo al elegido antes (en este caso, será *todos-los-p-usos/algunos-c-usos*) y se busca otra vez un conjunto de caminos, que será llamado C_2. El conjunto de caminos a ejercitar será entonces $C = C_1 \cup C_2$. En base a C, se elegirán los casos de prueba necesarios.

[5] Todos los conjuntos de criterios enunciados en el presente trabajo son análogos, cambiando la forma de llevar a cabo las pruebas o las variables que en ellas intervienen.

5. IMPLEMENTACIÓN

5.1. Implementación de la forma SSA

En esta sección se discutirá la implementación de la forma SSA. Se tratará el tema con cierta profundidad, para más detalles se recomienda consultar libros de compiladores ([2], [10] o [11]).

El algoritmo para convertir un programa a la forma SSA primero agrega ϕ-funciones para las variables y después renombra todas las definiciones y usos (con subíndices).

5.1.1. Inserción de ϕ-funciones por convergencia de caminos

Se podría agregar una ϕ-función en cada punto de unión, es decir, en cada nodo del grafo de flujo de control con más de un predecesor. Esto no es siempre necesario, debe encontrarse una forma de agregar las ϕ-funciones apropiadas, ni una más ni una menos.

Para esto, existe un criterio de convergencia de caminos que caracteriza los nodos donde se unen caminos de flujo de control de una variable. Debe haber una ϕ-función para una variable a en un nodo z en el grafo de flujo de control exactamente cuando se cumplen todos los incisos siguientes:

1. Hay un bloque x que contiene una definición de a.
2. Hay un bloque y (distinto a x) que contiene una definición de a.
3. Hay un camino no vacío P_{xz} de lados desde x hasta z.
4. Hay un camino no vacío P_{yz} de lados desde y hasta z.
5. Los caminos P_{xz} y P_{yz} tienen sólo a z como nodo común.
6. El nodo z no aparece ni en P_{xz} ni en P_{yz} antes del final del camino (aunque puede aparecer antes del final en uno sólo de ellos).

Se considera que el nodo inicial contiene una definición implícita de cada variable.

mientras existan nodos x, y, z que satisfagan las condiciones 1 a 5
y z no contenga una ϕ-función para a
hacer insertar $a \leftarrow \phi(a, a, \ldots, a)$ en el nodo z

Fig. 5.1: Criterio de convergencia de caminos

Nótese que una ϕ-función cuenta como definición de a, o sea que el criterio precedente debe ser considerado como un conjunto de ecuaciones a ser satisfechas. El sistema puede resolverse por iteración, con el criterio iterativo de convergencia de caminos de la figura 5.1. Obsérvese que la ϕ-función tiene tantos argumentos a como predecesores tiene el nodo z.

5.1.2. Inserción de ϕ-funciones por criterios de dominancia

Se dice que un nodo x *domina* a un nodo y si cada camino de lados dirigidos desde la raíz a y debe pasar por x (se considera que el grafo de flujo de control tiene una única raíz o nodo inicial). Una propiedad esencial de la forma SSA es que las definiciones dominan a los usos. Esto es,

1. Si x es el i-ésimo argumento de una ϕ-función en el bloque n, entonces la definición de x domina al i-ésimo predecesor de n.

2. Si x es usado en un bloque n que no contiene ϕ-funciones, entonces la definición de x domina a n.

Frontera de dominancia

El criterio iterativo anterior no es práctico porque es muy costoso examinar cada terna de nodos x, y, z y cada camino desde x hasta y. Un algoritmo mucho más eficiente usa el árbol de dominancia del grafo de flujo. Para ello, se pasarán a enunciar algunas definiciones necesarias:

- x domina estrictamente a y si x domina a y y $x \neq y$.

- La frontera de dominancia de un nodo x es el conjnto de todos los nodos w tales que x domina a un predecesor de w pero no domina estrictamente a w.

La frontera de dominancia de un nodo es el "límite" entre los nodos dominados y no dominados. Existen dos criterios para determinar los nodos donde deben insertarse ϕ-funciones:

Criterio de la frontera de dominancia : Cuando el nodo x contiene una definición de una variable a, entonces algún nodo z en la frontera de dominancia de x necesita una ϕ-función para a.

Frontera de dominancia iterativa : Como una ϕ-función es una definición, se debe iterar sobre el criterio de la frontera de dominancia hasta que no haya nodos que necesiten ϕ-funciones.

Estos dos criterios especifican exactamente el mismo conjunto de nodos donde poner ϕ-funciones que el criterio iterativo de convergencia de caminos. La prueba de esto escapa a los alcances del trabajo, además de no ser necesaria.

Cómputo de la frontera de dominancia

Para insertar las ϕ-funciones necesarias en cada nodo n, se necesita su frontera de dominancia, que será llamada $DF[n]$. Dado el árbol de dominancia, pueden calcularse eficientemente las fronteras de dominancia de todos los nodos del grafo en una pasada. Se definen para esto dos conjuntos auxiliares:

- $DF_{local}[n]$: los sucesores de n que no son estrictamente dominados por n.
- $DF_{up}[n]$: los nodos en la frontera de dominancia de n que no son dominados por el dominador inmediato de n.

La frontera de dominancia de n puede ser computada entonces fácilmente:

$$DF[n] = DF_{local} \cup \bigcup_{c \in hijos[n]} DF_{up}[c].$$

donde $hijos[n]$ es el conjunto de nodos cuyo dominador inmediato es n.

Para computar $DF_{local}[n]$ más fácilmente puede usarse la identidad

$$DF_{local}[n] = \{ \text{ sucesores de } n \text{ cuyo dominador inmediato no es } n\}$$

Finalmente, puede llamarse la función recursiva de la figura 5.2 en la raíz del árbol de dominadores (el nodo inicial del grafo de flujo); recorre el árbol computando $DF_{local}[n]$, $DF_{up}[c]$ para cada hijo c y $DF[n]$ para cada nodo n. Este algoritmo corre, en la mayoría de los casos, en tiempo casi lineal, proporcional a la cantidad de lados del grafo original más el tamaño de las fronteras de dominancia que computa.

```
calcularDF[n] =
    S ← {}
    para todo nodo y en sucesores[n]
        si idom[y] ≠ n
            S ← S ∪ {y}
    para todo hijo c de n en el árbol de dominadores
        calcularDF[c]
        para todo elemento w de DF[c]
            si n no domina a w
                S ← S ∪ {w}
                DF[n] ← S
```

Fig. 5.2: Cómputo de la frontera de dominancia

Inserción de ϕ-funciones y renombramiento de variables

Deben ahora insertarse las ϕ-funciones para satisfacer el criterio iterativo de la frontera de dominancia. Se usa un algoritmo simple, con lista de trabajo (worklist), para no examinar nodos donde no han sido insertadas ϕ-funciones.

Después, debe recorrerse el árbol de dominadores renombrando las distintas definiciones (incluyendo ϕ-funciones) de la variable a en a_1, a_2, a_3 y así sucesivamente. En un programa lineal, se renombrarían todas las definiciones de a y luego cada uso de a a la definición más reciente. Para un programa con ramificaciones y uniones en el flujo de control, cuyo grafo satisface el criterio de la frontera de dominancia, se renombra cada uso de a en la definición anterior más cercana a a en el árbol de dominadores. Puede lograrse para esto un algoritmo muy simple que trabaje en tiempo lineal, proporcional al tamaño del programa.

En [2] pueden encontrarse explicaciones detalladas de los algoritmos precedentes.

5.1.3. Cálculo eficiente del árbol de dominadores

Una de las razones para usar la forma SSA es la velocidad con que se pueden realizar los análisis: en vez de usar costosos algoritmos para relacionar usos con definiciones, puede buscarse la definición única o la lista de usos de cada variable. El tiempo ahorrado en los análisis no se justifica si el cómputo de la forma SSA es lento.

Los algoritmos usados para computar SSA a partir del árbol de dominadores son aceptablemente eficientes, pero el algoritmo iterativo simple de conjuntos para calcular dominadores que se muestra en [2] no lo es.

Para remediar este problema, en [8] se muestra un algoritmo eficiente. Se basa en las propiedades del árbol generador en profundidad (depth-first spanning tree). Es el árbol de recursión recorrido implícitamente por el algoritmo de búsqueda en profundidad (depth-first search, *DFS*), que numera los nodos del grafo con números de profundidad (*dfnum*, depth-first numbers) a medida que los recorre.

Para un análisis detallado del algoritmo, se refiere al lector a [2], [8], [10] y [11].

5.1.4. Algoritmos

Se comentarán aquí, en seudocódigo, algunos algoritmos presentados en la sección 3.4 que aprovechan la forma SSA. Son muy usados en compiladores optimizantes.

Eliminación de código muerto

Dado el grafo de flujo de control de un programa, sería inútil para los fines del testing verificar nodos que nunca se ejecuten. Sabiendo qué nodos están muertos (nunca se ejecutan), el testeador puede ahorrarse el trabajo de generar algunos casos de prueba. La cantidad de casos de prueba que no necesiten ser generados depende de qué criterio se elija.

Un primer algoritmo de eliminación de código muerto puede ser el siguiente:

mientras existe alguna variable v sin usos
  y la sentencia que define a v no tiene otros efectos laterales
hacer borrar la sentencia que define a v

Cuando se borra la sentencia $v \leftarrow x \oplus y$ o la sentencia $x \leftarrow \phi(x, y)$, debe tenerse cuidado de no quitar la sentencia de la lista de usos de x y de y, porque esto puede hacer que x o y sean consideradas *muertas* si fue el último uso. Para tener esto en cuenta, debe usarse una lista de trabajo de variables que deben ser reconsideradas.

$W \leftarrow$ una lista de todas las variables del programa SSA
mientras W no es vacía
  quitar alguna variable v de W
  si la lista de usos de v es vacía
    sea S la sentencia de definición
    si el único efecto lateral de S es la asignación a v
      quitar S del programa
      por cada variable x_i usada por S
        quitar S de la lista de usos de x_i
        $W \leftarrow W \cup \{x_i\}$

Con una elección apropiada de las estructuras de datos (listas doblemente enlazadas, por ejemplo), el tiempo de ejecución será lineal y proporcional al tamaño del programa más la cantidad de variables eliminadas.

Propagación simple de constantes

En ciertos casos, pueden transformarse cadenas DUA indefinidas en definidas o aparentes, al propagar el valor de una constante (es decir, reemplazar cada aparición de la variable por su valor constante) en los subíndices. Esto eliminaría asociaciones innecesarias, haciendo que el testeador evite verificar ciertas cadenas que parecían indefinidas en el código original del programa.

El siguiente es un algoritmo de lista de trabajo, como el precedente:

$W \leftarrow$ una lista de todas las sentencias del programa SSA
mientras W no es vacía
  quitar alguna sentencia S de W
  si S es $v \leftarrow \phi(c, c, \ldots, c)$ para alguna constante c
    reemplazar S por $v \leftarrow c$
  si S es $v \leftarrow c$ para alguna constante c
    quitar S del programa
    por cada sentencia T que usa v
      reemplazar c por v en T
      $W \leftarrow W \cup \{T\}$

En este algoritmo pueden incorporarse otras transformaciones muy simples para que, en tiempo lineal, se hagan todas en una sola pasada:

Propagación de copia: Una ϕ-función de argumento único $x \leftarrow \phi(y)$ o una asignación de copia $x \leftarrow y$ pueden ser borradas, e y sustituido por cada uso de x.

Plegado de constantes: Si se tiene una sentencia $x \leftarrow a \oplus b$, donde a y b son constantes, puede evaluarse $c \leftarrow a \oplus b$ en tiempo de compilación y reemplazar la sentencia por $x \leftarrow c$.

Condiciones constantes: En el bloque L, una ramificación condicional **si** $a < b$ **goto** L_1 **else** L_2, donde a y b son constantes, puede ser reemplazada por **goto** L_1 o **goto** L_2, dependiendo del valor de $a < b$ en tiempo de compilación. El lado de flujo de control desde L hasta L_2 (o hasta L_1) debe ser eliminado; esto reduce el número de predecesores de L_2 (o de L_1) y las ϕ-funciones en el bloque deben ajustarse quitando un argumento.

Código inalcanzable: Borrar un predecesor puede hacer que el bloque L_2 quede inalcanzable. En este caso, todas las sentencias de L_2 pueden ser quitadas y las listas de usos de todas las variables usadas en esta sentencia deben ser actualizadas. Luego, el bloque puede ser borrado, reduciendo el número de predecesores de sus bloques sucesores.

Propagación condicional de constantes

El algoritmo anterior de propagación de constantes tiene el problema de asumir que tal vez un bloque sea ejecutado y, entonces, no poder correctamente verificar que cierta variable puede ser una constante. Encuentra un punto fijo, pero no siempre es el mínimo punto fijo.

La presencia de sentencias nunca ejecutadas se debe a alguna estrategia de *debug* o, lisa y llanamente, a un error.

El algoritmo de propagación condicional de constantes encuentra el mínimo punto fijo: no asume que un bloque puede ser ejecutado hasta que existe evidencia de que puede serlo, y no asume que una variable no es constante hasta que existe evidencia de que lo es.

El algoritmo traza el valor de tiempo de ejecución de cada variable de la siguiente manera:

- $\nu(v) = \bot$ cuando no hay evidencia de que alguna vez se ejecuta una asignación a v.
- $\nu(v) = c$, siendo c una constante, cuando hay evidencia de que se ejecuta $v \leftarrow c$ pero no hay evidencia de que a v le es alguna vez asignado otro valor.
- $\nu(v) = \top$ cuando hay evidencia de que v tendrá al menos dos valores diferentes o algún valor impredecible en tiempo de compilación.

También se traza la ejecutabilidad de cada bloque:

- $\varepsilon(B) = falso$ cuando todavía no hay evidencia de que el bloque B es ejecutado alguna vez.
- $\varepsilon(B) = verdadero$ cuando hay evidencia de que B puede ser ejecutado.

El algoritmo comienza con $\nu(v) = \perp$ para toda variable v y $\varepsilon(B) = falso$ para todo bloque B. Luego recorre el programa, buscando evidencia de valores de variables y ejecutabilidad de bloques, siguiendo reglas que permitan cambiar valores de ν y de ε. Como resultado, se ignorarán las expresiones o sentencias de los bloques nunca ejecutados y las ϕ-funciones ignoran cualquier operando proveniente de los predecesores no ejecutables.

Puede implementarse fácilmente usando listas de trabajo (una para variables y otra para bloques). Funcionaría eligiendo una variable o on bloque de la lista, aplicando las reglas y repitiendo el proceso hasta que ambas listas estén vacías.

El correcto funcionamiento del algoritmo requiere la propiedad de *único sucesor o predecesor*. Esto es, nunca hay un lado en el grafo de flujo de control que se dirija desde un nodo con más de un sucesor hasta un nodo con más de un predecesor. El algoritmo para convertir un grafo en otro que posea esta propiedad es muy simple: para cada lado $a \to b$ tal que a tiene más de un sucesor y b tiene más de un predecesor, crear un nuevo nodo vacío z y reemplazar el lado $a \to b$ por los lados $a \to z$ y $z \to b$. Se dice que este grafo está en *forma SSA de lado dividido* (edge-split SSA form). La división de lados puede hacerse tanto antes como después de insertar las ϕ-funciones.

5.2. Implementación de una herramienta

En esta sección se describen los aspectos más importantes de la construcción de un prototipo de una herramienta destinada a asistir en el proceso de verificación de programas.

El funcionamiento del programa es simple. Dado el código del programa a verificar (en COBOL), lo pasa a la forma SSA como primer paso. Luego, analiza las cadenas definición-uso como se describió a lo largo del trabajo. Finalmente, toma un conjunto de caminos y analiza, usando la información obtenida de los análisis precedentes, si éste cumple con los criterios enunciados en el trabajo.

Cabe destacar que la presente implementación de muestra no testea, sino que sólo realiza los análisis previos. Completar el proceso de testing hubiera requerido ejecutar el programa COBOL para comparar su salida con la esperada.

```
DATA DIVISION.
   declaraciones-de-variables
PROCEDURE DIVISION.
   programa
```

Fig. 5.3: Formato del código aceptado por la herramienta

5.2.1. Lenguaje elegido

Como se mencionó durante casi todo el trabajo, los análisis que se realizan al programa a verificar son muy parecidos a los realizados en algunos compiladores. Por este motivo, debió elegirse un lenguaje propicio para el desarrollo de compiladores, es decir, que cumpla ciertos requisitos:

- facilidad de manejar estructuras de datos complejas y de gran tamaño
- buen desempeño del código compilado
- existencia de herramientas que simplifiquen ciertos pasos en el proceso de compilación como parser y lexer
- librerías que permitan trabajar con múltiples estructuras de datos (especialmente, conjuntos y mapas)
- estructura modular que simplifique el proceso constructivo de grandes piezas de software

Se decidió entonces usar OCAML. Este lenguaje posee, además de las características pedidas, innumerables ventajas sobre otros lenguajes (C, C++, Java) para escribir compiladores. Implementa un recolector de residuos de desempeño muy eficiente, recursión de cola optimizada, manejo de excepciones, inferencia de tipos (Hindley-Milner) y polimorfismo. Para una descripción más detallada del lenguaje, pueden verse [3] y [9].

5.2.2. Aspectos importantes de la implementación

La herramienta acepta como entrada un archivo de texto con la sintaxis descripta en el capítulo 2. Para que el lenguaje aceptado se asemeje un poco más a COBOL, se agregó el reconocimiento de dos *secciones*: la primera describe las variables que se usarán a lo largo del programa y la segunda describe el programa en sí.

El archivo de entrada, por lo tanto, tendrá la forma de la figura 5.3. Las variables declaradas pueden ser numéricas o alfanuméricas, escalares o arreglos. Una variable escalar se declarará colocando:

nivel nombre `PIC` *máscara* `.`

donde *nivel* es el entero definido comúnmente en COBOL (que no tiene relevancia en la presente implementación), *nombre* es el identificador único de la variable y *máscara* es una cadena formada por carácteres `X` o `9` que indican si la variable será alfanumérica o numérica, respectivamente.

Un arreglo se declarará de la misma forma que una variable escalar, pero indicando su tamaño con las palabras `OCCURS` *tamaño* `TIMES` antes del punto.

Hecha esta aclaración sobre la sintaxis aceptada, se verá ahora la implementación. El apéndice A describe el diseño de los módulos implementados. En primer lugar, el archivo de entrada es analizado por el *lexer* de OCAML; la salida es luego analizada por el *parser* de OCAML y el resultado es el *árbol de sintaxis abstracta* (AST) del programa. Cabe notar que el parser, además, asocia con cada sentencia una lista de *i-uso*s de variables.

El AST contiene las declaraciones de las variables y el programa. Las primeras son examinadas por el módulo *Vars* y guardadas en una estructura de datos que será usada por fases posteriores del análisis.

Con los datos anteriores, el módulo *Flujocontrol* construye el grafo de flujo de control y un mapa (asociación) etiquetas-nodos para identificar más fácilmente la relación entre los pasos del código original y los nodos del grafo construido. Este grafo es guardado en un archivo con formato *Graphviz*[1], al igual que todos los grafos que se guardan como resultado de los análisis.

Como paso siguiente, el grafo es analizado y se construye uno nuevo (que también se guarda en un archivo) que resulta de reemplazar las sentencias `PERFORM`[2] de COBOL por las sentencias que contiene dicha sentencia.

Esta información es suficiente para construir el árbol de dominadores. Esta ardua tarea es realizada por la función *arbdom* del módulo *Arbdom*. Primero, se calcula un árbol generador primero-profundo del grafo de flujo de control. Luego, se construye el árbol de semidominadores y, finalmente, se obtiene el árbol de dominadores inmediatos. Para lograr lo anterior, se usó un algoritmo basado en los teoremas de Lengauer y Tarjan [8], que termina

[1] *Graphviz* es un potente software libre para visualización de grafos. Más información, fuentes y binarios pueden encontrarse en *www.graphviz.org*.

[2] Esta sentencia ejecuta un conjunto de sentencias. No fue incluida en el lenguaje presentado en la sección 2 porque es redundante en la estructura de control, es decir, su uso puede reemplazarse por un grupo de sentencias equivalentes. Puede verse [14] para más información sobre esta función.

siendo, en la práctica, lineal en tiempo y espacio con respecto a la cantidad de nodos del grafo (en la teoría, su complejidad es del orden $n \log n$). El árbol de dominadores se guarda en un archivo en forma de grafo, con el formato antes decripto.

El próximo paso es construir las fronteras de dominancia; de esto se encarga el módulo *Frontdom*. La función *domfr* retorna un mapa nodos-conjunto de nodos que representa la frontera de cada nodo. Para más información sobre este paso, puede verse [4]. Las fronteras de dominancia se guardan en un archivo de texto.

Con toda la información antes obtenida, se está ahora en condiciones de calcular la forma SSA del programa. El módulo *Ssa* es el encargado de esta parte. La función *agr_phif* devuelve un nuevo grafo de flujo de control en el cual cada nodo tiene asociada información sobre las variables que necesitan ϕ-funciones. Para terminar de pasar a la forma SSA, la función *renomb_var* devuelve un árbol con las variables renombradas a partir de la información del grafo anterior. La forma de implementar estas dos funciones fue discutida en la sección 5.1.

El paso siguiente está presente en la implementación sólo para mostrar las ventajas de tener el código en la forma SSA: la función *cmuerto* del módulo *Cmuerto* devuelve una lista de nodos cuyo código no contribuye al resultado final. Con estos datos, se guarda en otro archivo un nuevo grafo con los nodos muertos resaltados. La implementación es relativamente simple y puede ser de gran utilidad para otros análisis que se quieran hacer al programa. Pueden implementarse fácilmente, en este punto, todos los algoritmos de la sección 5.1.4.

Todos los cálculos anteriores tenían solamente un propósito, que no era el análisis para la optimización sino el perseguido en el resto del trabajo: calcular las cadenas definición-uso presentes en el código de entrada. El módulo *Medidas* realiza estos cómputos. La función *calcular* devuelve una terna de valores, que representan los conjuntos *dcu*, *dpu* y *diu*. El funcionamiento puede parecer complejo a primera vista pero este módulo fue implementado en poco más de 200 líneas de código (incluyendo comentarios). La diferencia con cualquier cálculo de asociaciones definición-uso radica en que la forma SSA mantiene las propiedades de dominación (cada definición siempre domina a sus usos) y, teniendo en cuenta esto, el cálculo es casi directo. Finalmente, los tres conjuntos de asociaciones definición-uso se guardan en un archivo de texto.

6. CONCLUSIONES

El objetivo del presente trabajo fue el desarrollo de una técnica que logre capturar la esencia de los datos estructurados en el marco de la selección de casos de prueba para el testing de programas presentada en [12]. Para esto, fue necesario definir un nuevo tipo de cadenas definición-uso.

Además, se tuvieron en cuenta las modificaciones descriptas en [5] para usar la forma SSA como representación intermedia.

Se consideraron algunos algoritmos que hacen uso de la forma SSA para realizar optimizaciones en compiladores y se propuso su aplicación para detectar ciertas falencias en el código examinado.

También se presentaron medidas análogas a las presentadas en [5] y [12] que hacen uso de los conjuntos *dcu*, *dpu* y *diu*.

Finalmente, se presentó y describió una implementación que muestra la aplicación de las técnicas precedentes en una herramienta.

6.1. Trabajos futuros

Pueden agruparse los caminos a seguir para continuar la investigación aquí descripta en dos grupos: desarrollos teóricos e implementación.

Para continuar la teoría, es necesario incorporar características más generales al lenguaje de programación usado. Esto es, por ejemplo, considerar la presencia de estructuras o de punteros, así como enfocar el proceso desde el punto de vista más abstracto que puede proporcionar la programación modular. Para agregar estructuras, puede adoptarse el enfoque de este trabajo y crear nuevos conceptos (como asociaciones definición-uso). En el caso de los punteros, se necesitaría una visión más general, que contemple la organización de la memoria; sin embargo, este enfoque puede englobar arreglos, estructuras y otros tipos de estructuras de datos.

La implementación presentada es un prototipo cuya utilidad real no puede ser más que educativa. Sin embargo, el núcleo de una herramienta completa no necesita ser mucho más compleja que la actual; sólo es necesario agregar nuevas características al lenguaje de entrada (especialmente, teniendo en cuenta la expresividad de COBOL) y una interfaz gráfica que permita utilizar

con comodidad el programa. Los algoritmos usados para el cálculo de la forma SSA, junto con las estructuras de datos presentes en todo el programa tienen la robustez necesaria para hacer fácil la adaptación del mismo.

La investigación desarrollada en este trabajo es parte de un proyecto conjunto con Hernán Dacharry y Mariano Bertoni, estudiantes de Ciencias de la Computación al momento de comenzar la investigación. La tesina de Dacharry [5] contiene el comienzo y el presente trabajo es el siguiente paso del proyecto. Inicialmente, el objetivo fue exclusivamente la implementación de una herramienta completa para identificar casos de prueba; todavía no se alcanzó, pero se desarrolló una teoría robusta y se comenzó a delinear un prolijo programa que tiende a cumplir los objetivos propuestos.

Apéndice A

DISEÑO DE LOS MÓDULOS IMPLEMENTADOS

Para comprender el funcionamiento del programa referido en la sección 5.2, en este apéndice se mostrará el diseño de cada módulo. La figura A.1 muestra las dependencias entre los módulos, los lados dirigidos representan la relación depende-de.

A.1. Módulo Cctt

Este es el módulo principal.

A.2. Módulo Cmuerto

Este módulo se encarga de encontrar cuáles nodos están muertos (nunca se ejecutan).

```
val cmuerto: Ssa.SSA.grafo -> Ssa.SSA.id_nodo list
```

Devuelve una lista de nodos que nunca se ejecutan.

A.3. Módulo Criterios

Este módulo tiene funciones que permiten saber si un conjunto de caminos cumple con los criterios enunciados anteriormente.

```
type camino = Ssa.SSA.id_nodo list
```

Un camino es una lista de nodos.

```
module CNodo: Set.S with type elt = Ssa.SSA.id_nodo
```

Estructura de datos que representa un conjunto de nodos.

```
module CLado:
  Set.S with type elt = (Ssa.SSA.id_nodo * Ssa.SSA.id_nodo)
```

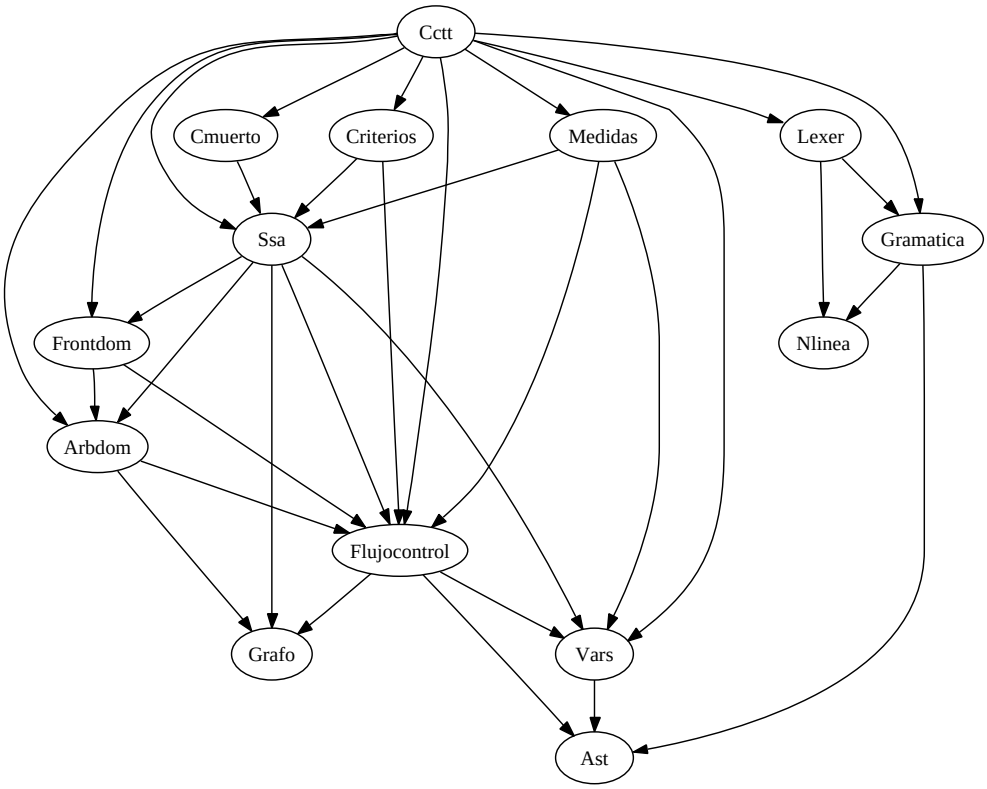


Fig. A.1: Dependencias entre los módulos

Estructura de datos que representa un conjunto de lados.

`val t_nodos: camino list -> Ssa.SSA.grafo -> CNodo.t`
Esta función implementa el criterio *todos-los-nodos*. Recibe una lista de caminos y un grafo, y devuelve un conjunto conteniendo los nodos que ningún camino recorre (si es vacío, la lista de caminos cumple este criterio).

`val t_lados: camino list -> Ssa.SSA.grafo -> CLado.t`
Esta función implementa el criterio *todos-los-lados*. Recibe una lista de caminos y un grafo, y devuelve un conjunto conteniendo los lados que ningún camino recorre (si es vacío, la lista de caminos cumple este criterio).

```
val t_defs:
  camino list -> Ssa.SSA.grafo ->
  (Ssa.ssavar * Ssa.SSA.id_nodo) list
```
Esta función implementa el criterio *todas-las-definiciones*. Recibe una lista de caminos y un grafo, y devuelve una lista con todos los pares (definición, nodo) para los cuales no existe ningún camino en la lista original que cumpla el criterio.

```
val t_pusos: camino list -> Ssa.SSA.grafo ->
  (Ssa.ssavar * Ssa.SSA.id_nodo) list
```

Esta función implementa el criterio *todos-los-p-usos*. Recibe una lista de caminos y un grafo, y devuelve una lista con todos los pares (definición, nodo) para los cuales no existe ningún camino libre de definición que cumpla los requerimientos del criterio.

```
val t_cusos_a_pusos: camino list -> Ssa.SSA.grafo ->
  (Ssa.ssavar * Ssa.SSA.id_nodo) list
```

Esta función implementa el criterio *todos-los-c-usos/algunos-p-usos*. Recibe una lista de caminos y un grafo, y devuelve una lista con todos los pares (definición, nodo) para los cuales no existe ningún camino libre de definición que cumpla los requerimientos del criterio.

```
val t_pusos_a_cusos: camino list -> Ssa.SSA.grafo ->
  (Ssa.ssavar * Ssa.SSA.id_nodo) list
```

Esta función implementa el criterio *todos-los-p-usos/algunos-c-usos*. Recibe una lista de caminos y un grafo, y devuelve una lista con todos los pares (definición, nodo) para los cuales no existe ningún camino libre de definición que cumpla los requerimientos del criterio.

```
val t_usos: camino list -> Ssa.SSA.grafo ->
  (Ssa.ssavar * Ssa.SSA.id_nodo) list
```

Esta función implementa el criterio *todos-los-usos*. Recibe una lista de caminos y un grafo, y devuelve una lista con todos los pares (definición, nodo) para los cuales no existe ningún camino libre de definición que cumpla los requerimientos del criterio.

```
val t_def_uso: camino list -> Ssa.SSA.grafo -> camino list
```

Esta función implementa el criterio *todas-las-definiciones-usos*. Recibe una lista de caminos y un grafo, y devuelve una lista con todos los caminos libres de bucles que van desde una definición hasta un uso y que no están en la lista dada.

A.4. Módulo Medidas

```
module Mdxu: Map.S with type key = (Vars.var * Ssa.SSA.id_nodo)
```

Mdxu representa las asociaciones def-uso como un mapa, es decir, como una aplicación, (nombre, id_nodo) → conjunto de id_nodos.

```
module CNodo: Set.S with type elt = Ssa.SSA.id_nodo
```

Estructura de datos que representa un conjunto de id_nodos.

```
val calcular:
  Ssa.SSA.grafo ->
  (CNodo.t Mdxu.t * CNodo.t Mdxu.t * CNodo.t Mdxu.t *
   CNodo.t Mdxu.t * CNodo.t Mdxu.t)
```

Recibe un grafo SSA y devuelve la 5-upla formada por (dcu, dpu, diu, diu_c, diu_p), donde diu_p son los diu en los cuales la variable definida está usada en un predicado y diu_c son los diu en los cuales la variable definida está usada para cómputos.

A.5. Módulo Ssa

```
type ssavar
```

Se oculta la implementación de este tipo abstracto.

```
val comp_ssavar: ssavar -> ssavar -> int
val string_of_ssavar: ssavar -> string
val gen_ssavar: Vars.var -> int -> ssavar
val camb_subind: ssavar -> int -> ssavar
val ver_subind: ssavar -> int
val var_of_ssavar: ssavar -> Vars.var
```

Operaciones varias con variables SSA.

```
module CSVar: Set.S with type elt = ssavar
```

Estructura de datos que representa un conjunto de variables SSA.

```
type vssa_info = {
  defs: CSVar.t;
  usos: CSVar.t;
  u_si: CSVar.t;
}
```

Estructura de datos que representa la información asociada a cada variable SSA.

```
type fssa_info =
  | Phif of (ssavar * ssavar list) list
  | Nada
```

Cada variable SSA puede tener asociada una ϕ-función en el grafo de flujo de control.

```
type arrssa_info = (ssavar * ssavar list) list
```
Información sobre la variable de arreglo en SSA.

```
type inodo = {
  gfc_info: Flujocontrol.GFC.info_nodo;
  vssa: vssa_info;
  fssa: fssa_info;
  assa: arrssa_info;
}
```
Cada nodo tiene asociada la información anterior.

```
type ilado = Flujocontrol.GFC.info_lado
```
Cada lado tiene asociada la misma información que en el módulo *Flujocontrol*.

```
module SSA:
  (Grafo.G with type info_nodo = inodo
   and type info_lado = ilado)
```
Estructura de datos que representa el nuevo grafo que incluye información sobre las ϕ-funciones.

```
val agr_phif: Flujocontrol.GFC.grafo -> Vars.CVar.t ->
  Frontdom.CNodo.t Frontdom.MNodo.t -> SSA.grafo
```
Toma un grafo, un conjunto de variables, un mapa nodos $\rightarrow$ conjunto de nodos (fronteras de cominancia) y devuelve un grafo en el que se le agrega a cada nodo un campo que indica si debe tener ϕ-funciones o no y, en caso que las deba tener, una lista con las variables correspondientes a cada función y la cantidad de argumentos.

```
val renomb_var: SSA.grafo -> Vars.CVar.t -> Arbdom.DOM.grafo ->
  Arbdom.DOM.id_nodo -> SSA.grafo
```
Recibe un grafo SSA, un conjunto de variables, un árbol de dominadores y un nodo del mismo (que es la raíz) y devuelve el grafo SSA con los renombres pertinentes.

A.6. Módulo Lexer

Este módulo es generado por *ocamllex*. Es el encargado de transformar el programa de entrada en una secuencia de tokens, olvidando todo lo relativo a espacios, saltos de línea y comentarios.

A.7. Módulo Gramatica

Este módulo es generado por *ocamlyacc* a partir de una descripción de la gramática.

```
type token = ...
```
Todos los tokens del lenguaje aceptado que no tienen relevancia aquí.

```
val programa :
  (Lexing.lexbuf -> token) -> Lexing.lexbuf ->
  Ast.decs * (Ast.parr list)
```
Esta función transforma el programa original en la estructura de datos que se usará para trabajar. Recibe una función que transforma la entrada en tokens (que será el lexer) y la entrada, y devuelve un par formado por las declaraciones presentes en el programa y una lista de párrafos.

A.8. Módulo Nlinea

```
val numlinea : int ref
```
Este módulo sólo guarda el número de línea del programa que está siendo parseado, es usado por el lexer y el parser.

A.9. Módulo Frontdom

```
module CNodo: Set.S with type elt = Flujocontrol.GFC.id_nodo
```
Estructura de datos que representa un conjunto de nodos.

```
module MNodo: Map.S with type key = Flujocontrol.GFC.id_nodo
```
Estructura de datos que representa un mapa nodos → ... (por lo general, será un mapa nodos → conjunto de nodos).

```
val domfr:
  Flujocontrol.GFC.grafo -> Arbdom.DOM.grafo ->
  Arbdom.DOM.id_nodo -> CNodo.t MNodo.t
```
Recibe como argumentos el grafo de control, el árbol de dominadores (representado como un grafo) y el nodo raíz. Retorna un mapa nodos → conjunto de nodos, que representa las fronteras de dominancia.

A.10. Módulo Arbdom

```
module DOM: (Grafo.G with
  type info_nodo = Flujocontrol.GFC.info_nodo
  and type info_lado = unit)
```

Estructura de datos que representa un árbol de dominadores.

```
val arbdom:
  Flujocontrol.GFC.grafo -> Flujocontrol.GFC.id_nodo ->
  DOM.grafo
```

Devuelve el árbol de dominadores inmediatos del grafo de flujo de control, partiendo del nodo raíz (segundo parámetro).

A.11. Módulo Flujocontrol

```
type instruccion = string
```

Representaremos una instrucción como una cadena alfanumérica.

```
type datos_sent = {
  instr: instruccion;
  defs: Vars.CVar.t;
  usos: Vars.CVar.t;
  u_si: Vars.CVar.t;
}
```

Datos asociados a cada sentencia.

```
type datos_salto =
  | Perform of Vars.etiqueta
  | Goto of Vars.etiqueta
```

Las sentencias de salto que se manejarán.

```
type nodo_sent =
  | Entrada of Vars.etiqueta
  | Salida of Vars.etiqueta
  | Sentencia of datos_sent
  | Salto of datos_salto
```

El tipo de los nodos que van a estar en los grafos que representen los párrafos (en Entrada y Salida va a estar el nombre del párrafo).

```
module GFC_sent: (Grafo.G with type info_nodo = nodo_sent and
type info_lado = unit)
```

Estructura de datos que representa un grafo de flujo de control. Los nodos son sentencias y tendrán asociada la información descripta más arriba.

```
type nodo_parr = {
  etiq: Vars.etiqueta;
  sentg: GFC_sent.grafo;
  entra: GFC_sent.id_nodo;
}
```

En el grafo de párrafos, cada nodo va a tener una etiqueta (el nombre del párrafo) y el grafo correspondiente a las sentencias del párrafo.

```
type lado_parr =
  | Control of datos_salto
  | Siguiente
```

En el grafo de párrafos, los lados pueden corresponder a un salto de control (Goto o Perform) o simplemente a un salto al próximo párrafo.

```
module GFC_parrf: (Grafo.G with type info_nodo = nodo_parr and
type info_lado = lado_parr list)
```

Estructura de datos que representa un grafo de párrafos.

```
type nodo_gen =
  | Comienzo of Vars.etiqueta
  | Fin of Vars.etiqueta
  | Comp of datos_sent
```

Finalmente, el grafo general sobre el que van a trabajar los algoritmos tendrá un nodo de entrada Comienzo y uno de salida Fin, y todos los otros nodos van a ser sentencias.

```
module GFC: (Grafo.G with type info_nodo = nodo_gen and
type info_lado = unit)
```

Estructura de datos que representa un grafo general.

```
module MEtiq: (Map.S with type key = Vars.etiqueta)
```

Estructura de datos que representa un mapa etiquetas $\rightarrow \ldots$.

```
val gen_gfc_parrf:
  Ast.parr list -> Vars.vardec ->
  (GFC_parrf.grafo * GFC_parrf.id_nodo MEtiq.t)
```

Esta función recibe una lista de párrafos y las declaraciones de las variables y devuelve el grafo de flujo de control, en forma de par (grafo, mapa etiquetas → nodos).

```
val sacar_perf:
  (GFC_parrf.grafo * GFC_parrf.id_nodo MEtiq.t) ->
  Vars.etiqueta -> GFC.grafo
```

Esta función recibe el par (grafo, mapa etiquetas → nodos) y la etiqueta del nodo raíz. Devuelve un grafo donde cada nodo es una sentencia con los Perform reemplazados por un subgrafo que tiene las sentencias correspondientes.

A.12. Módulo Grafo

Este módulo oculta la implementación de los grafos.

```
module type G =
  sig
    type info_nodo
    type info_lado
    type id_nodo = int
    type contexto = {n_id: id_nodo;
                     info_n: info_nodo;
                     entr: (id_nodo*info_lado) list;
                     sal: (id_nodo*info_lado) list}
    type grafo

    exception Nodo
    exception Lado

    val vacio: grafo
    val es_vacio: grafo -> bool
    val nuevo_id: grafo -> id_nodo
    val nodos: grafo -> id_nodo list
    val lados: grafo -> (id_nodo * id_nodo) list
    val info_n: grafo -> id_nodo -> info_nodo
    val camb_info_n: grafo -> id_nodo -> info_nodo -> grafo
    val info_l: grafo -> (id_nodo * id_nodo) -> info_lado
    val camb_info_l: grafo -> (id_nodo * id_nodo) ->
      info_lado -> grafo
    val cont: grafo -> id_nodo -> contexto
    val suces: grafo -> id_nodo -> id_nodo list
```

```
    val prede: grafo -> id_nodo -> id_nodo list
    val ady: grafo -> id_nodo -> id_nodo list
    val agregar_n: grafo -> info_nodo -> (grafo * id_nodo)
    val agregar_l: grafo -> info_lado -> (id_nodo * id_nodo) ->
      grafo
    val agregar_c: grafo -> contexto -> grafo
    val quitar_l: grafo -> (id_nodo * id_nodo) ->
      (grafo * info_lado)
    val quitar_n: grafo -> id_nodo -> grafo
    val quitar_nc: grafo -> id_nodo -> (grafo * contexto)
    val cant_n: grafo -> int
    val fold: (contexto * 'c -> 'c) -> 'c -> grafo -> 'c
    val gfold: (contexto -> id_nodo list) ->
      (contexto * 'c -> 'd) -> ('d * 'c -> 'c) -> 'c ->
      id_nodo list -> grafo -> 'c
    val pertenece: grafo -> id_nodo -> bool
    val union: grafo -> grafo ->
      (grafo * (id_nodo*id_nodo) list)
    val imprimir: grafo -> (info_nodo -> string) ->
      (info_lado -> string) -> string
    val impr_dot: grafo -> (id_nodo -> info_nodo -> string) ->
      (info_lado -> string) -> string
    val impr_dot2: grafo -> (id_nodo -> info_nodo -> string) ->
      (info_lado -> string) -> string
    val impr_dot_sg: grafo ->
      (id_nodo -> info_nodo -> string) ->
      (info_lado -> string) -> string
  end
```

El tipo de módulo `G` contiene la implementación de los grafos. Posee las funciones usadas para crear, modificar y recorrer los grafos.

```
  module Make (In: sig type nodo end) (Il: sig type lado end):
    G with type info_nodo = In.nodo and
    type info_lado = Il.lado
```

`Make` se usa para crear un grafo con cierto tipo de información asociada a nodos y lados.

A.13. Módulo Vars

Este módulo agrupa las estructuras de datos y las funciones necesarias para trabajar con las variables del programa.

```
type simb = Ast.symbol
```
`simb` es el mismo tipo que usa el AST para representar los nombres de variables.

```
type var =
  | Escalar of simb
  | Arreglo of simb
```
El tipo que representa las variables que pueden aparecer en el programa.

```
type vardec
```
Este es el tipo que representa la informacion de la declaracion de las variables.

```
val comp_var: var -> var -> int
```
La funcion de comparación de variables.

```
module CVar: Set.S with type elt = var
```
Estructura de datos que representa conjuntos de variables.

```
val vardecs: Ast.decs -> vardec
```
Procesa las declaraciones de variables del AST.

```
exception No_esta
```
Excepción levantada cuando falla la búsqueda de una variable en este módulo.

```
val busca_var: vardec -> simb -> CVar.t
```
Esta función toma un string (que es como aparecen las variables en el programa) y devuelve la variable a la que se refiere; para esto necesita conocer las declaraciones (primer argumento).

```
val cvar: vardec -> CVar.t
```
Esta función toma las declaraciones y devuelve un conjunto de variables declaradas.

```
val string_of_var: var -> string
```
Devuelve una representación imprimible de una variable.

```
val es_arreglo: var -> bool
val es_escalar: var -> bool
```
Funciones para saber el tipo de variable con el que se está tratando.

```
type etiqueta
val comp_etiqueta: etiqueta -> etiqueta -> int
val string_of_etiqueta: etiqueta -> string
val nueva_etiqueta: string -> etiqueta
```
Estas funciones sirven para ocultar el tipo de las etiquetas, o nombres de párrafo.

A.14. Módulo Ast

Este módulo define la estructura de datos conocida como árbol de sintaxis abstracta (AST). El AST de un programa es el resultado del lexing, que es examinado por el parser.

BIBLIOGRAFíA

[1] B. Alpern, M. Wegman, F. Zadeck, *Detecting equality of variables in programs*, Fifteenth ACM Principles on Programming Languages Symposium, January 1988.

[2] A. Appel, *Modern Compiler Implementation in ML*, Cambridge University Press, 1999.

[3] E. Chailloux, P. Manoury, B. Pagano, *Developing applications with Objective Caml*, O'Reilly & Associates, 2000.

[4] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, F. Zadeck, *An efficient method of computing Static Single Assignment form*, ACM, 1989.

[5] H. Dacharry, *La forma SSA en la selección de casos de prueba*, tesina de licenciatura, FCEIA, Septiembre 2004.

[6] C Ghezzi, M. Jazayeri, D. Mandrioli, *Fundamentals of Software Engineering*, Prentice Hall, 1991.

[7] K. Knobe, V. Sarkar, *Array SSA form and its use in Parallelization*, ACM, 1998

[8] T. Lengauer, R. E. Tarjan. *A fast algorithm for finding dominators in a flowgraph*, ACM Trans. Programming Languages Systems (TOPLAS), July 1979.

[9] X. Leroy, *The Objective Caml system release 3.08. Documentation and user's manual*, INRIA, 2004.

[10] R. Morgan, *Building an optimizing compiler*, Butterworth-Heinemann, 1998.

[11] S. Muchnick, *Advanced Compiler Design and Implementation*, Academic Press, 1997.

[12] S. Rapps, E. Weyuker, *Data Flow Analysis Techniques for Test Data Selection*, IEEE, 1982.

[13] W. Richards Adrion, M. Branstad, J. Cherniavsky, *Validation, Verification and Testing of Computer Software*, ACM Computing Surveys, June 1982.

[14] *COBOL85 V2.3A (BS2000/OSD), COBOL Compiler Reference Manual* (cb85_bs.pdf), Siemens Nixdorf Informationssysteme, 1998.