

Ej. 3.17 : Recuperación "automática" de memoria (generada de un modo estático).

El tratamiento "global" de la memoria disponible asumiendo las posiciones disponibles enlazadas en una lista "potencialmente infinita", que va recorriendo el "puntero" new vís next, sin volver jamás atrás, garantiza en particular que jamás dos variables compartirán una misma dirección, lo que necesitamos para garantizar la "corrección" de esta semántica (Ej. 3.14). El "precio" que pagamos por ello es la generación de basura, en forma de montones de direcciones inaccesibles: todas las creadas en bloques de los que ya nos hemos salido. Obsérvese que esto incluiría situaciones en las que la generación de memoria puede realizarse de un modo estático, como por ejemplo si tenemos un bloque dentro de un bucle, o dentro de procedimientos iterativos.

La recuperación de memoria en este marco resulta sorprendentemente sencilla: basta "memorizar" el valor de next en el entorno de variables, y "seguir utilizando" la regla [compns] que "recupera" el valor de dicho entorno cuando "nos salimos" de un bloque. Al recuperar el valor previo de next se estaría, en efecto, recuperando toda la memoria asignada durante la ejecución del bloque (incluyendo incluso la generada "dinámicamente" en el marco de un procedimiento recursivo).

Asumiendo que, como es el caso, nunca podremos "reanudar" una posición en uso, el único peligro sería el que pudiesemos "reutilizar" valores supuestamente "borrados", pero esto aquí tampoco pasa pues inicializamos las variables declaradas en los bloques. ¿Cuál sería entonces la mínima diferencia entre los dos modelos de memoria?

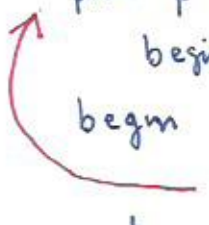
Reutilización de la memoria (Ejercicio 3.17)

Una solución precipitada de este ejercicio nos llevó a concluir que al incluir la información sobre "la primera dirección disponible" next en el entorno env_v , en lugar de en la memoria sto , lo que hacíamos era "recuperar" las direcciones ya no utilizables (al "salirnos" hacia un bloque "más externo") cuando terminas la ejecución de un bloque. Esto es en efecto lo que sucedería si manejamos bloques sólo definiendo variables "locales", pero sin procedimientos. La "corrección" de esta nueva regla para la "asignación de memoria" en los bloques (¡ojo, al ejecutarse éstos, es decir, inexorablemente de un modo dinámico!) se correspondería con el funcionamiento según lo esperado de la regla [compns], que "recupera" el entorno vigente al "empezar" cuando "termina" una ejecución (en particular, la de un bloque). La nueva regla en el enunciado de este ejercicio "mimetiza" este funcionamiento manejando los "mismos" direcciones de memoria, exactamente igual que cuando no los teníamos manejábamos los "mismos" nombres de variable.

Pero (en general) esta nueva forma pesa a ser "incorrecta" si "mezclamos" arbitrariamente variables y procedimientos en los bloques y "admitimos" llamadas a procedimientos declarados "fuera" del bloque en que nos encontramos.

Ejemplo :

```
B = begin var x := 1 ;  
      proc p is  
        begin var y := 2 ;  
        begin var z := 3 ;  
        call p  
        end  
      end
```



$env_v, env_p \vdash \langle B, sto \rangle \rightarrow sto$ "genera" env'_v con $env'_v x = env_v(2)$
(next)

y env'_p con $\text{env}'_p \vdash (\text{begin } y := 2 \text{ end}, \text{env}'_v, \text{env}_p)$. "Finalmente"
 "ejecutamos" $\text{env}'_v, \text{env}'_p \vdash (\text{begin var } z := 3 ; \text{call } p \text{ end}, \text{sto}') \xrightarrow{\text{sto}''}$
 con $\text{sto}'(\text{env}'_v x) = 1$, y "generamos" env''_v con $\text{env}''_v z = \text{env}'_v(2)$,
 procediendo con $\text{env}''_v, \text{env}'_p \vdash (\text{call } p, \text{sto}'')$, con $\text{sto}''(\text{env}''_v z) = 3$.
 Esto "genera" a su vez $\text{env}'_v, \text{env}_p \vdash (\text{begin var } y := 2 \text{ end}, \text{sto}'') \xrightarrow{\text{sto}'''} \text{sto}'''$

que produce la "reutilización incorrecta" de la posición de memoria $\text{env}'_v(2)$ que "se vuelve a asignar" a y , produciéndose sto''' con $\text{sto}'''(\text{env}'''_v y) = 2$, donde $\text{env}'''_v y = \text{env}'_v(2) = \text{env}''_v z$ por lo que (al menos en este momento) la variable z ha "perdido su valor" 3, ya que $\text{sto}'''(\text{env}'''_v z) = 2$. Ciertamente que dentro de p " z no existe", así que de momento esto no sería un problema, pero veremos que "el daño está hecho". Aún podríamos confiar en que al salir del bloque que constituye el cuerpo de p se "destruye" la variable y , al "perderse" el entorno env''_v . En efecto, al "volver" al punto de llamada "recuperamos" env_v , pero si ahora a continuación "consultáramos" el valor de z nos encontraríamos con $\text{sto}'''(\text{env}_v z) = 2 \neq 3 !!!$

El problema es que, por mucho que el tratamiento de las variables y procedimientos sea estático, la generación de memoria ha de ser dinámica, y la ejecución "global" otro tanto, con eventuales excursiones "extra-bloque", para luego "regresar" a él. ¿Tendría esto solución? Pues sí, pero sólo parcialmente. De hecho, podríamos hablar de dos soluciones con ópticas diferentes:

- Restricciones que garanticen el funcionamiento correcto de la nueva política de "asignación de memoria".

Si nos fijamos en el ejemplo que hemos construido, vemos que el problema viene de la presencia de llamadas a procedimientos "externos" a un bloque, desde otro que incluya declaraciones (previas) de variables, de manera que los procedimientos llamados también contengan "variables locales". Así que habrá que "prohibir" estas llamadas.

Observese que si tenemos un procedimiento recursivo con "variables locales", esta variante seguirá funcionando mal, pues realizaría todas las llamadas con el mismo entorno $env'_p [p \rightarrow (S, env'_v, env'_p)]$, y por tanto utilizaría (erróneamente) las mismas direcciones para las distintas "copias" de sus variables locales.

b) Hay compiladores que asignan estáticamente toda la memoria necesaria para ejecutar el programa compilado en todos los casos. Naturalmente, eso sólo es posible si no hay recursión. Se trata de que la compilación asigne memoria a cada bloque al "leerlo", lo que en nuestro marco se corresponde con una "fase preliminar" en la que, para simplificar, se renombrarían primero todas las variables de manera que pasen a tener nombres diferentes, dos a dos, y se asignaran (por inducción estructural) las posiciones de las variables de un modo inyectivo, de manera que la "reutilización" de una misma variable no suponga jamás la creación de nuevas posiciones de memoria, y tampoco se puedan producir "choques" entre variables diferentes. Y pensando un poco más vemos que a las variables de "bloques hermanos" dentro de un mismo bloque padre se les podría asignar perfectamente "zonas de memoria que solapan", pues sus variables jamás coexistirán. En tal caso, el "gato total" de memoria sería el correspondiente al "mayor" (en necesidad de memoria) de esos hermanos. Ciertamente, todo esto se puede describir del modo *¡declarativo!* en que definimos una semántica operacional, pero ha de hacerse con suma unidad y resulta muy complejo.