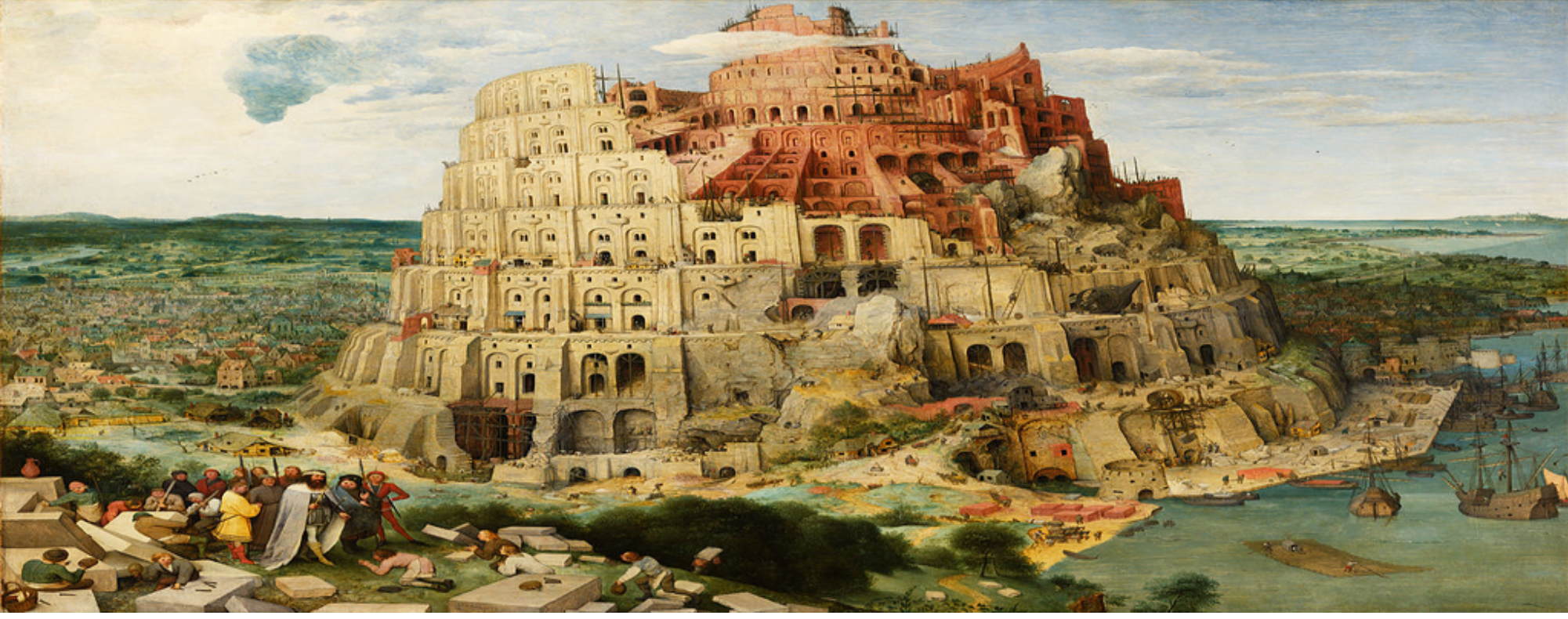

Parseo y Generación de Código

16 de noviembre de 2017

Asignación de registros
Manejo automático de memoria

Licenciatura en Informática con Orientación en Desarrollo de Software
Universidad Nacional de Quilmes

Asignación de registros

Asignación de registros

Supongamos que se tradujo un programa a código de tres direcciones, usando una cantidad ilimitada de registros. Por ejemplo:

```
for i = 1 to n
  v[i] := 2 * v[i] + v[i + 1]
end
```

```
; v = t0, n = t1, i = t2
t2 := 1
start:
jumpIf> t2 t1 end
t3 := 8 * t2
t4 := t0[t3]
t5 := 2 * t4
t6 := t2 + 1
t7 := 8 * t6
t8 := t0[t7]
t9 := t5 + t8
t10 := 8 * t2
t0[t10] := t9
t2 := t2 + 1
jump start
end:
```

Asignación de registros

- El ejemplo anterior utiliza once variables (`t0`, ..., `t10`).
- Al momento de generar código máquina, es posible que no haya tantos registros disponibles.
- Esto obliga a hacer *register spilling*, es decir, volcar el valor de algunos de dichos registros en memoria principal, haciendo operaciones de **load** y **store** cada vez que se necesita operar con ellos.
- Es deseable reducir la cantidad de registros disponibles.

Asignación de registros

Para analizar el problema se introduce la siguiente definición:

Decimos que dos variables **interfieren** si hay algún punto del programa en el que ambas están vivas.

Asignación de registros

Esto permite plantear el problema usando lenguaje de teoría de grafos:

- Considerar el **grafo de interferencia** del programa:
  - Tiene un nodo por cada variable del programa.
  - Tiene una arista (v, w) si las variables v y w interfieren.
- Las variables se pueden asignar para usar a lo sumo k registros si y sólo si los nodos del grafo de interferencia se pueden pintar utilizando a lo sumo k colores, de tal manera que nodos adyacentes tengan colores distintos.
- Este problema de grafos se conoce como k-**coloreo**.

Asignación de registros

Ejercicio. Armar el grafo de interferencia asociado al siguiente programa:

```
t1 := 9
t2 := 2
t3 := t1 * t2
t4 := t1 + t2
t4 := t4 + t2
t5 := t3 * t4
t5 := t3 * t5
```

Mostrar que el grafo es 3-coloreable.
Reescribir el programa utilizando solamente tres registros.

Asignación de registros

Nota. El problema de k-coloreo es NP-completo:
no se conoce manera de resolverlo de manera eficiente.

Sin embargo:

- Muy frecuentemente los grafos de interferencia para un procedimiento involucran un número pequeño de variables (< 30) y el problema de k-coloreo se puede resolver de manera exacta usando fuerza bruta o *backtracking*.
- Típicamente no se espera que el compilador genere código **óptimo**, sino que genere código **razonablemente bueno**. Hay heurísticas, es decir, soluciones aproximadas, para el problema de k-coloreo que son eficientes y otorgan resultados razonables en la práctica.
- Algunas de esas heurísticas están orientadas al problema de asignación de registros (ver por ejemplo la técnica de Iterated Register Coalescing).

Manejo automático de memoria

Manejo automático de memoria

Objetivo. Agregar soporte para que el entorno de ejecución se encargue de liberar automáticamente la memoria.

En un momento dado de la ejecución de un programa, decimos que una celda de memoria es **basura** si ninguna posible ejecución futura del programa la referencia.

Problema. El problema de determinar si una celda de memoria es basura es indecidible.

Manejo automático de memoria

Por ejemplo, considerar el siguiente programa:

```
x := [1, 2, 3]
while p() {
    ...          // No se usa x.
}
print(x[0])
```

- Si la ejecución del `while` nunca termina, las celdas de memoria referenciadas por `x` son basura.
- En caso contrario, dichas celdas no son basura.
- Determinar si una celda de memoria es o no es basura requeriría poder determinar si un programa termina o no termina, lo que es indecidible en general.

Manejo automático de memoria

Es imposible implementar un recolector de basura **perfecto**, en el sentido de que siempre reclame la memoria asignada a todas las celdas que son basura.

Pero hay maneras de aproximar el problema.

Raíces y alcanzabilidad

En un entorno de ejecución típico, se pueden identificar tres elementos en la memoria disponible:

- Registros del procesador.
- Pila del sistema.
- *Heap*.

La pila y el *heap* residen en la memoria principal.
La memoria principal se compone de celdas de memoria.

Cada registro y cada celda de memoria pueden contener dos tipos de valores:

- Valores inmediatos (constantes numéricas).
- Punteros a arreglos en memoria, que pueden residir ya sea en el *heap* o en la pila.

Raíces y alcanzabilidad

Los lenguajes contemporáneos utilizan este modelo de memoria prácticamente sin excepción.

Por ejemplo, en un lenguaje orientado a objetos, una representación posible para un objeto podría ser un puntero a un arreglo en memoria. El arreglo se compone de $n + 2$ elementos:

- Puntero al objeto que representa su clase.
- Constante n indicando cuántas variables de instancia tiene.
- Secuencia de n punteros a sus variables de instancia.

Raíces y alcanzabilidad

Dado un estado de un programa en ejecución, llamamos **raíces** al conjunto de celdas de memoria que se pueden referenciar de manera inmediata. Las raíces son:

- Celdas referenciadas desde los registros del procesador.
- Celdas referenciadas desde la pila del sistema.

Decimos que un objeto en memoria es **alcanzable** si es posible referenciarlo empezando desde alguna de las raíces y siguiendo punteros. Recursivamente:

- Las raíces son alcanzables.
- Si una celda A es alcanzable, y contiene un puntero a un arreglo $[B_1, \ldots, B_n]$ en memoria, entonces las celdas $B_1, \ldots, B_n$ también son alcanzables.

Raíces y alcanzabilidad

Una aproximación al problema de liberar la memoria asignada a basura es **liberar las celdas de memoria inalcanzables**.

Mencionaremos tres técnicas de **recolección de basura** (*garbage collection*):

1. Conteo de referencias.
2. *Mark and sweep*.
3. *Stop and copy*.

Conteo de referencias

Conteo de referencias

Una manera de implementar manejo de memoria automático es el **conteo de referencias**.

En tiempo de ejecución, cada objeto está acompañado de un número que indica cuántos objetos lo referencian.

Cada vez que se hace una asignación en el lenguaje fuente:

```
A[i] := B
```

el compilador emite código para:

- Incrementar en uno el conteo de referencias para el objeto `B`.
- Decrementar en uno el conteo de referencias para el objeto referenciado anteriormente por `A[i]`.
- Si el conteo de referencias para el objeto referenciado anteriormente por `A[i]` llega a 0, liberar la memoria asociada a dicho objeto.

Conteo de referencias

Cada vez que se libera la memoria de un objeto `A`, se decrementa en uno el conteo de referencias para todos los objetos referenciados inmediatamente por `A`.

Si sus conteos de referencias llegan a 0, se libera (recursivamente) la memoria asociada a dichos objetos.

Conteo de referencias

- El conteo de referencias es sencillo de implementar.
  (¡Qué bien!).
- Pero representa un costo elevado en tiempo de ejecución.
  El costo de cualquier operación de asignación se multiplica por un factor.
  (¡Qué mal!).
- Tiene como ventaja que es *real-time*:
  el proceso de recolección de basura no requiere pausar la ejecución del programa.
  (¡Qué bien!).
- Tiene inconvenientes para reclamar la memoria de estructuras de datos cíclicas.
  ... (¡Qué mal!).

Conteo de referencias

Ejemplo.
El siguiente programa construye una estructura de datos cíclica:

```
fun f() {
  x := [1, 2, 3]
  y := [4, 5, x]
  x[2] := y
}
```

- Si se invoca a `f()` y la ejecución del programa continúa, las celdas de memoria de los objetos `x` e `y` resultan inalcanzables.
- Pero el conteo de referencias de `x` es siempre ≥ 1 porque está referenciado por `y`.
- Inversamente, el conteo de referencias de `y` es siempre ≥ 1 porque está referenciado por `x`.

Conteo de referencias

Las estructuras de datos cíclicas son muy frecuentes en la realidad.

Por ejemplo, en lenguajes dinámicos como Smalltalk un objeto tiene referencias a su clase, y la clase tiene referencias a todas sus instancias.

Conteo de referencias

El conteo de referencias puede servir para lenguajes en los que hay garantía de que no se pueden construir estructuras de datos cíclicas.

Puede servir como técnica complementaria a otras técnicas de *garbage collection* (p.ej. en CPython).

Algunos entornos (p.ej. implementaciones antiguas de Smalltalk) usan únicamente conteo de referencias, y el programador debe encargarse de "romper" manualmente los ciclos para asegurar que la memoria se reclame.

Mark and sweep

Mark and sweep

La técnica de **marcado y barrido** (*mark and sweep*) se basa en un algoritmo de dos etapas:

- Marcar todas las celdas de memoria alcanzables.
- Liberar todas las celdas que no estén marcadas.

Hay muchas variantes de la técnica. Por simplicidad supondremos que el algoritmo es *stop-the-world*, es decir que se introduce una pausa en el programa para liberar la memoria.

Mark and sweep

¿Cuándo se invoca al garbage collector?

- Cada vez que se necesita reservar memoria, el manejador de memoria verifica si la memoria total reservada supera cierto umbral.
- En caso de que lo supere, se invoca al *garbage collector* que libera las celdas de memoria inalcanzables.
- El umbral inicialmente se establece en algún número arbitrario, por ejemplo 1 MB.
- Al finalizar la ronda de *garbage collection*, se debe actualizar el umbral. (¿Qué sucedería si el umbral no se actualiza?).
- El umbral debe ser proporcional a la memoria total que está utilizando el programa después de la ronda de *garbage collection*. (Por ejemplo, el umbral se puede actualizar para que sea el doble de la memoria total utilizada).

Mark and sweep

La etapa de **marcado** es esencialmente un recorrido *depth-first* por la memoria.

Se requiere una pila auxiliar.

Se requiere disponer de un bit auxiliar por cada objeto A, indicando si A es alcanzable.
El algoritmo que veremos además requiere:

- Dado el contenido de una celda de memoria, saber si se trata de una constante numérica o de un puntero a otra celda de memoria. (Se implementa con otro bit).
- Dado un objeto, saber su tamaño, es decir cuántas celdas de memoria ocupa.

Mark and sweep

Truco de implementación.
En muchas arquitecturas (p.ej. Intel x86_64) los últimos dos (o tres) bits de un puntero son siempre 0, porque las celdas de memoria están alineadas a 32 (o 64) bits. El *garbage collector* puede aprovechar estos últimos bits para guardar información, indicando si un objeto es alcanzable, para distinguirlo de una constante inmediata, etc.

Mark and sweep

El algoritmo de marcado es simplemente un DFS:

```
pila := []
foreach objeto A entre las raíces
    A.alcanzable := True
    pila.push(A)
end
while pila no es vacía
    A := pila.pop()
    for i = 1 to #celdas_que_ocupa(A)
        if es_puntero?(A[i]) and not A[i].alcanzable then
            A[i].alcanzable := True
            pila.push(A[i])
        end
    end
end
```

Mark and sweep

- El algoritmo de borrado recorre uno por uno todos los objetos, eliminando los que tienen el bit `alcanzable` en `False`.
- Una vez que finaliza el recorrido, todos los objetos supervivientes tienen el bit `alcanzable` en `True`. No es necesario volver a ponerlo en `False`: basta con invertir la manera en que se interpreta ese bit durante la siguiente ronda de *garbage collection*.

Stop and copy

Stop and copy

La técnica de *stop and copy* se basa en las siguientes ideas:

- El total de la memoria disponible se divide en dos mitades: el *from-space* y el *to-space*.
- Los objetos nuevos se ubican en el *from-space*.
- Cuando el *from-space* se llena, los objetos alcanzables se copian al *to-space*, actualizando todas las referencias para que apunten a la nueva copia.
- Por último se invierten los roles del *from-space* y el *to-space*.

Stop and copy

El algoritmo de copiado no requiere una pila auxiliar: el *to-space* se utiliza implícitamente como una cola.

Se requiere disponer de un bit auxiliar por cada objeto A, para indicar si ya fue copiado. Una vez que un objeto en el *from-space* se copia al *to-space*, la primera celda de memoria correspondiente a dicho objeto en el *from-space* se reemplaza por un puntero a su nueva ubicación en el *to-space*.

Igual que en en el caso de *mark and sweep* se requiere:

- Dado el contenido de una celda de memoria, saber si se trata de una constante numérica o de un puntero a otra celda de memoria.
- Dado un objeto, saber su tamaño.

Stop and copy

El algoritmo de copiado usa la siguiente función auxiliar:

```
function copiar(A : ptr)
-- Copia el objeto referenciado por el puntero A
-- al to_space.
    for j = 0 to #celdas_que_ocupa(A) - 1
        to_space[#to_space + j] = A[j]
    end
    A[0] := #to_space          -- forwarding pointer
    A.fue_copiado := True
    #to_space := #to_space + #celdas_que_ocupa(A)
end
```

Stop and copy

El algoritmo de copiado en sí mismo es un BFS.

```
foreach objeto A entre las raíces
    if not A.fue_copiado
        copiar(A)
    end
    reemplazar la referencia a A por A[0]
end
i := 0
while i < #to_space
    A := objeto en la posición i del to_space
    for j = 1 to #celdas_que_ocupa(A)
        if es_puntero?(A[j])
            if not A[j].fue_copiado
                copiar(A[j])
            end
            reemplazar la referencia a A[j] por A[j][0]
        end
    end
    i := i + #celdas_que_ocupa(A)
end
```