Generación de Código

Albert Rubio

Procesadores de Lenguajes, FdI, UCM

Doble Grado Matématicas e Informática

Generación de código

Contenidos

Traducción de lenguajes imperativos

- Los lenguajes imperativos se basan en la existencia de un **estado**, que es transformado sucesivamente mediante la ejecución de **instrucciones**.
- El programa comienza en un estado **inicial**, y cada instrucción modifica el estado en curso y produce un nuevo estado.
- Las instrucciones se ejecutan **secuencialmente** en el orden especificado por el programador.
- Hay dos tipos de instrucciones:
  - Las que modifican propiamente el estado. La más relevante es la **instrucción de asignación**.
  - Las que evalúan el estado y en función del mismo dirigen el **flujo de control** del programa.

Estado

- El estado del programa en un momento de la ejecución está determinado por el valor de las **variables**.
  Al nivel del código máquina, esto significa los valores en **memoria** y ciertos registros como el contador del programa.
- Las variables locales y parámetros de las funciones se suelen asignar en una parte de la memoria asignada a **pila**.
- La pila contiene un **marco de activación** por cada función activa en la cadena de llamadas en curso.
  De esta forma, se pueden tener (sin conflicto) distintas **encarnaciones** de las variables de una función recursiva.
- La variables cuyo espacio se crea dinámicamente suelen estar en el **montón**, que es otra parte de la memoria.
- Podemos tener también una zona especial de la memoria reservada para variables **estáticas** que están vivas durante toda la ejecución.

Evaluación de expresiones

- Las expresiones del lenguaje se suelen evaluar mediante un recorrido en **postorden** del AST que las representa, usando un **pila**:
  - Cuando llegamos a una **hoja**, tenemos una constante o un designador (variable): apilamos su valor.
  - Cuando estamos en un **nodo interno**, con un operador $\oplus$ (supongamos binario):
    1. Desapilamos c_1 (cima) y c_2 (siguiente).
    2. Operamos $c_2 \oplus c_1$.
    3. Apilamos el resultado.
- Es fácil determinar el tamaño máximo de la pila necesario evaluar una expresión.

Instrucciones de control

La máquina a la que vamos a compilar debe proporcionar operaciones para traducir las **instrucciones de control de flujo**. Debe incluir:

- Instrucciones de **salto incondicional** (`br`).
- Instrucciones de **salto condicional** (`br_if`).
- Instrucción de **salto por llamada** a función (`call`).

Incluyen formas explícitas o implícitas de expresar etiquetas.

Pueden incluir nociones de bloque.

Máquinas reales

Las máquinas reales suelen tener unas características muy precisas que complican la traducción y que varían de unas a otras:

- Se almacena en **memoria central** o en **registros**.

  Las operaciones aritmético-lógicas suelen trabajar en registros.

  Trabajar sobre registros es mucho más rápido que sobre memoria.

- El número de registros es **finito** y depende de la máquina. No podemos tener toda la información en registros.

- El **orden** de ejecución de un **bloque básico** (un bloque de instrucciones sin saltos) es relevante para la eficiencia de ejecución (depende de la máquina).

Máquinas virtuales

La alternativa a producir código para una máquina concreta es hacerlo para una **máquina virtual**:

- Debe facilitar la traducción de lenguajes imperativos.
- Su código puede ser **interpretada** en cualquier máquina real.
- Su código puede ser **compilado** al código nativo de una máquina real.

Las máquinas virtuales facilitan la compilación y aumentan la **portabilidad** del lenguaje:

- la mayoría del trabajo del compilador es **independiente** de una máquina concreta.
- Si es interpretado se puede ejecutar en cualquier máquina que tenga un interprete de la máquina virtual (por ejemplo, la JVM).

En este curso usaremos **WebAssembly**.

Contenidos

El proyecto WebAssembly

- **WebAssembly** (wasm) es un formato binario de instrucciones para una máquina virtual de pila.
- wasm es un nuevo tipo código de bajo nivel que se puede ejecutar en los navegadores modernos con un rendimiento casi nativo (es decir, como si ejecutara directamente código ensamblador de nuestra máquina).
  - Es un proyecto en desarrollo, pero posiblemente será el estándar en el entorno web.
  - Actualmente se ejecuta a través de JavaScript
  - Aún tiene limitaciones como las llamadas a sistema, paralelización,...
- Lenguajes como C/C++ y Rust pueden generar código objeto wasm.
- En nuestro contexto:
  - nos proporciona una máquina virtual de pila completa.
  - ejecución/testeo fácil del código generado.
  - conoceremos un bytecode usado en aplicaciones reales.

Formato texto de WebAssembly

Usaremos el formato textual en lugar del binario

- Es un formato legible y comprensible
- Fácil detectar si estamos traduciendo bien.
- Se puede compilar/traducir a binario de forma muy simple.
- Utilizaremos `wat2wasm` del *WebAssembly Binary Toolkit* (wabt).

La extensión para código WebAssembly textual es `.wat`

Podéis obtener los binarios para linux (Ubuntu), MacOS o windows en:

`https://github.com/webassembly/wabt/releases`

También podéis conseguir los fuentes e instalar la última versión en:

`https://github.com/webassembly/wabt`

WebAssembly módulos

Un código wat, se encapsula con la función predefinida `module`

```
(module
...
)
```

que incluye todas las declaraciones necesaria:

1. Tipos (type): perfiles de función.
2. Importaciones (import): definiciones importadas
3. Exportaciones (export). Declaraciones exportadas.
4. Memoria. Definición de la memoria.
5. Globales (global): declaraciones globales
6. Funciones (function)
7. Inicio (start): función a ejecutar inicialmente.
8. Datos (data): definición directa de partes de la memoria.
9. Tablas (table). Tabla de funciones.
10. Elementos (elem). Elementos de una tabla.

Definición de tipos

Es opcional en el formato textual. Pero es una buena práctica.

```
(type ident (func params results))
```

Donde `params` puede ser:

- `(param i32 i64 i32)`
- `(param i32) (param i64) (param i32)`

y lo mismo para `results`

- `(result i32 i64)`
- `(result i32) (result i64)`

Las funciones pueden devolver varios resultados.

Si no tiene ni parámetros ni resultados no se pone nada.

Tipos disponibles: `i32`, `i64` (para enteros) y `f32`, `f64` (para floats).

Memoria

Podemos definir memorias lineales (array) de bytes.

Actualmente solo se puede definir una memoria. En el futuro podrán coexistir varias.

Definimos la memoria con el tamaño inicial en *número de páginas* (64KiB).

```
(memory 2000)
```

Tenemos operaciones para preguntar el tamaño de la memoria y extenderla (no las usaremos). Y otras como: `memory.fill`, `memory.copy`, ...

- Se accede con distintos load: i32.load, i64.load, f32.load, ...
  toma la posición de la pila y deja el valor en la pila
- Se modifica con distintos store: i32.store, i64.store, f32.store, ...
  toma la posición (cima-1) y el valor (cima) de la pila
- Podemos incluir un desplazamiento (offset) y/o alineamiento (align).

  i32.load offset=4 ;; obtiene el contenido de la posición en la cima + 4

Globales

Podemos definir variables globales al módulo WebAssembly.

Se indica:

- el identificador,
- el tipo y si es mutable
- la expresión de inicialización

```
(global $pos i32 (i32.const 0)) ;; constante
(global $pos (mut i32) (i32.const 0)) ;; mutable
```

Para acceder a su valor usamos `get_global` con el identificador

```
get_global $pos ;; deja el valor en la cima de la pila
```

Para modificarla (si es mutable) usamos `set_global` con el identificador

```
set_global $pos ;; le pone el valor de la cima de la pila
```

El nombre es opcional tanto en la definición como el uso.

Podemos acceder a ellas usando su número de definición (empezando en 0).

Funciones

Las funciones constan de:

- cabecera: nombre y tipo (opcional)
- nombre (opcional) y tipo de los parámetros: `(param $op1 i32)`
- tipo de los resultados: `(result i32)`
- nombre (opcional) y tipo de las variables locales: `(local $x i32)`
- cuerpo de la función

Los parámetros y locales tienen un número asignado (empezando en 0) por orden de declaración.

Es decir, el primer parámetro es 0 y cuando acaban lo parámetros seguimos numerando las locales.

Podemos usar los nombres (si los tienen) o los números indistintamente.
Se acceden y se modifican usando: `get_local` y `set_local`

Cuerpo de la función: instrucciones

Además de las que hemos visto:

1. constantes: `i32.const`, `f32.const`, ... con un valor (0,0.0,..)
2. `drop` desapila y `select` es una función if-then-else
3. operaciones de bit: `i32.clz`, `i32.shl`, `i32.and`, ...
4. operaciones aritméticas: `i32.add`, `i32.mul`, `i32.div_s`,...
5. operaciones relacionales: `i32.eqz`, `i32.eq`, `i32.le_u`,...
6. operaciones de conversión: `i64.extend_i32_u` (`i64.extend_i32_s`), `f32.convert_i64_s`, ...
7. operaciones de control: `block`, `loop`, `if`, `br`, `call`, `return`, ...

Consultad, por ejemplo:

`https://github.com/WebAssembly/design/blob/master/Semantics.md`

`https://webassembly.github.io/spec/core/text/instructions.html`

`https://pengowray.github.io/wasm-ops/`

Instrucciones de control: etiquetas y saltos

- `block` ... `end`. Bloque de instrucciones con una etiqueta al final.
- `loop` ... `end`. Bloque de instrucciones con una etiqueta al principio.

  Si están anidados las etiquetas se numeran de dentro a fuera: 0 para el bloque más cercano, 1 el siguiente, ...
- `br`. Salto incondicional: `br 0`
- `br_if`. Salto condicional: `br_if 1`
  La cima de la pila indica el valor de la condición
- `br_table` Tabla de saltos: `br_table 2 0 1`
  La cima de la pila indica el índice de la tabla
  el último es la opción por defecto si no se aplican las anteriores

Instrucciones de control: etiquetas y saltos

```
(func $add_one
  (param $pos i32)
  (param $n i32)
  (local $i i32)
  (local $temp i32)
 i32.const 0
 set_local $i
 block
  loop
   get_local $i
   get_local $n
   i32.lt_s
   i32.eqz
   br_if 1
   get_local $i
   i32.const 4
   i32.mul
   get_local $pos
   i32.add
   set_local $temp
   get_local $temp
   get_local $temp
   i32.load
   i32.const 1
   i32.add
   i32.store
   get_local $i
   i32.const 1
   i32.add
   set_local $i
   br 0
  end
 end
)
```

Instrucciones de control: sin etiquetas

- `if ... end`

```
i32.eqz
if
 get_local $i
 call $fun1
end
```

- `if ... else ... end`

```
i32.eqz
if
 get_local $res
 return
else
 get_local $i
 call $fun1
end
```

- `return`. Si queremos volver antes del final de la función. Los resultados deben estar en la cima de la pila.
- `call`. Los parámetro deben estar en la cima de la pila (en sentido inverso).

Instrucciones: S-expresiones

Se puede escribir código wasm como expresiones (funcionales) prefijas

`(op exp-arg1 ... exp-argn)`

Podemos escribir todo el código como una S-expresión o combinarlo con el formato secuencial.

```
(func $add_one
 (param $pos i32)
 (param $n i32)
 (local $i i32)
 (local $temp i32)
(set_local $i (i32.const 0))
block
 loop
 (i32.eqz (i32.lt_s (get_local $i) (get_local $n)))
 br_if 1
 (set_local $temp (i32.add (get_local $pos) (i32.mul (get_local $i) (i32.const 4))))
 (i32.store (get_local $temp) (i32.add (i32.load (get_local $temp)) (i32.const 1)))
 (set_local $i (i32.add (get_local $i) (i32.const 1)))
 br 0
 end
end )
```

Función de inicio

Webassembly nos permite indicar una función a ejecutar cuando se cargue el módulo.

Esto se hace mediante la declaración `start`:

```
(start $funname)
```

Importaciones

Podemos importar definiciones externas indicando:

- el módulo de donde se importan.
- el nombre del objeto importado.
- su definición en WebAssembly

```
(import "runtime" "print" (func $print (type $_sig_i32)))

(import "runtime" "read" (func $read (result i32)))
```

Las importaciones deben ir antes que cualquir otra definición de objetos wasm (global, memory, func o table). Pero puede tener delante definiciones de tipos (type), exports u operaciones de inicialización (data, elem o init), que no crean objetos wasm.

Se pueden importar funciones, globales, tablas o la memoria.

Solo importaremos funciones

Exportaciones

Podemos exportar definiciones indicando:

- el nombre con el que se exporta
- el objeto exportado (qué es y su nombre).

```
(export "init" (func $init))
```

Se pueden exportar funciones, globales (no mutables), tablas o la memoria.

Solo exportaremos funciones, aunque puede ser útil exportar la memoria para debug.

```
(export "memory" (memory 0))
```

Tablas y elementos

Se utilizan para realizar vínculos dinámicos.

Actualmente podemos definir, por ejemplo, un tabla de funciones:

```
(table $funcmap 2 2 funcref)
```

donde `$funcmap` es una tabla de referencias a función de tamaño inicial y máximo 2.

Se pueden definir tablas de objetos externos (no lo usaremos).

Los elementos de la tabla se definen usando `elem`:

```
(elem $funcmap (i32.const 0) $func_1 $func_2)
```

Se llama a estas funciones con

```
call_indirect $funcmap (type $_t_i32)
```

que toma el índice de la cima de la pila y el tipo debe ser el de la función indexada. En las siguientes posiciones de la pila están los parámetros.

Contenidos

Traducción de código sin funciones

Supondremos en esta primera parte que tenemos solo un programa principal (sin funciones).

Para la traducción del código del programa usaremos tres funciones recursivas:

$code_E(expresion)$	Genera código para evaluar expresiones
$code_D(designador)$	Genera código para obtener la dirección del designador
$code_I(instruccion)$	Genera código para ejecutar instrucciones

$code_E$ y $code_D$ son mutuamente recursivas ya que en un designador pueden aparecer expresiones y en una expresión pueden aparecer designadores.

Ambas son usadas por $code_I$

Para esta primera parte podemos suponer que nuestra zona de almacenamiento en la memoria wasm empieza en la dirección conocida X.

Evaluación de expresiones

La función $code_E(e)$ considera los siguientes casos sobre e:

- Estamos en un nodo de designador (a un entero) d:
  $code_D(d)$
  $i32.load$
- Estamos en un nodo de constante (entero) c:
  $i32.const\ \ c$
- Estamos en un nodo expresión E con operador OP y operandos $Args$:
  para todo a en $Args$:
  $code_E(a)$
  $wasm\_opcode(OP)$
- La operación `new`:
  Llama a una función que reserva el espacio necesario y deja en la cima la dirección de inicio de ese espacio.

Designadores

Asociamos a cada identificador (de designador) un número que indica su posición relativa al inicio de un espacio de almacenamiento de 32-bits:

- La primera variable tendrá posición 0 (podemos tomar otro inicio)
- Las siguientes tendrán la posición de la anterior más el tamaño de la anterior. Puede ser array o struct.
- Necesitamos una función $size(t)$ que nos devuelva el tamaño de t
- Para los campos de un struct, iniciamos la cuenta de 0

¿Cómo tratar los bloques?

- Cada vez que acabamos un bloque descontamos el tamaño de todas sus variables.

La función $\delta(d)$ nos devuelve este número.

Designadores

Suponed que tenemos:

```
struct p {
  int uno[2];
  int dos
};
int i;
p ap[6][2];
p aux;
```

Entonces:

$\delta(*\texttt{uno}) = 0$
$\delta(*\texttt{dos}) = 2$
$\delta(*\texttt{i}) = 0$
$\delta(*\texttt{ap}) = 1$
$\delta(*\texttt{aux}) = 37$

Designadores

Consideraremos variables de tipo entero, arrays y structs:

- Identificador:

  $code_D(id)$

  ```
  i32.const δ(*id)
  obtener inicio memoria del marco de id
  i32.add
  ```

- Acceso a apuntadores:

  $code_D(d->)$

- Acceso a struct:

  $code_D(d.id)$

Designadores

Consideraremos variables de tipo entero, arrays y structs:

- Identificador:

  $code_D(id)$

  ```
  i32.const δ(*id)
  obtener inicio memoria del marco de id
  i32.add
  ```

- Acceso a apuntadores:

  $code_D(d->)$

  $code_D(d)$

  ```
  i32.load
  ```

- Acceso a struct:

  $code_D(d.id)$

Designadores

Consideraremos variables de tipo entero, arrays y structs:

- Identificador:

  $code_D(id)$

  ```
  i32.const δ(*id)
  obtener inicio memoria del marco de id
  i32.add
  ```

- Acceso a apuntadores:

  $code_D(d->)$

  $code_D(d)$

  ```
  i32.load
  ```

- Acceso a struct:

  $code_D(d.id)$

  $code_D(d)$

  ```
  i32.const δ(*id)
  i32.add
  ```

Designadores

- Acceso a array:

  $code_D(d[e])$

Designadores

- Acceso a array:

  $code_D(d[e])$
  - $code_D(d)$
  - `i32.const` $size(tipo(d[e]))$
  - $code_E(e)$
  - `i32.mul`
  - `i32.add`

Designadores

- Acceso a array teniendo en cuenta todos los accesos:
  $code_D(d[e_1][e_2]\dots[e_n])$

Designadores

- Acceso a array:

  $code_D(d[e_1][e_2]\ldots[e_n])$
  - $code_D(d)$
  - $code_E(e_1)$
  - `i32.const` $dim(d, 1)$
  - `i32.mul`
  - $code_E(e_2)$
  - `i32.add`
  - `...`
  - `i32.const` $dim(d, n-1)$
  - `i32.mul`
  - $code_E(e_n)$
  - `i32.add`

Acceso a arrays dinámicos

- Acceso a array dinámicos:

  Lo haremos teniendo en cuenta todos los accesos.

  Los arrays dinámicos (parámetros o locales) se colocan de una forma especial en memoria.

  En el sitio que les toca, en lugar de su contenido, guardamos:

  - donde empieza su contenido al final de la memoria local
  - tamaño del array
  - longitud de la primera dimensión

    .....
  - longitud de la penúltima dimensión

  Debe rellenarse en tiempo de ejecución (cuando se saben las dimensiones): hay que generar código que lo rellene.

  Hay que mantener la primera posición libre en la pila, y se va modificando a medida que reservamos dinámicamente el espacio para

Acceso a arrays dinámicos

- $code_D(d_d[e_1][e_2]\dots[e_n])$

  $code_D(d)$

```
tee_local $darr    ;; almacena y devuelve
i32.load           ;; cargamos la posición de inicio
```

  $code_E(e_1)$

```
get_local $darr    ;;|
i32.const 2        ;;|
i32.add            ;;|
i32.load           ;;| hemos cargado dim(d, 1)
i32.mul
```

  $code_E(e_2)$

```
i32.add
...
get_local $darr    ;;|
i32.const n        ;;|
i32.add            ;;|
i32.load           ;;| hemos cargado dim(d, n − 1)
i32.mul
```

  $code_E(e_n)$

```
i32.add
```

Instrucciones

- Asignación:
  $code_I(d = e)$
  $code_D(d)$
  `i32.add`
  $code_E(e)$
  `i32.store`

- if-then-else: no es necesario crear etiquetas explícitamente.
  $code_I(ite(e, I_0, I_1))$

Instrucciones

- Asignación:
  $code_I(d = e)$
  $code_D(d)$
  `i32.add`
  $code_E(e)$
  `i32.store`

- if-then-else: no es necesario crear etiquetas explícitamente.
  $code_I(ite(e, I_0, I_1))$
  $code_E(e)$
  `if`
  $code_I(I_0)$
  `else`
  $code_I(I_1)$
  `end`

Instrucciones

- while: no es necesario crear etiquetas explícitamente.
  $code_I(while(e, I))$

Instrucciones

- while: no es necesario crear etiquetas explícitamente.
  $code_I(while(e, I))$

```
block
loop
code_E(e)
i32.eqz
br_if 1
code_I(I)
br 0
end
```

Instrucciones

- switch:

  `switch e {`
  `0 :` I_0
  ...
  $n-1$: I_{n-1}
  `default:` I_n
  `}`

Instrucciones

- $code_I(switch(e, I_0, \ldots, I_n))$

```
code_E(e)
block                          ;; block n+1
...                            ;;(n times block)   total n+2
block                          ;; block 0
br_table 0 1 ...  n
end                            ;; etiqueta 0
code_I(I_0)
br n+1
end                            ;; etiqueta 1
code_I(I_1)
br n+1
end                            ;; etiqueta 2
...
end                            ;; etiqueta n
code_I(I_n)
br n+1                         ;; no es necesaria
end                            ;; etiqueta n+1
```

Contenidos

Memoria estática y dinámica

- La memoria estática es la que podemos anticipar en tiempo de compilación.

  La guardamos en una pila de marcos de activación (ver más tarde).

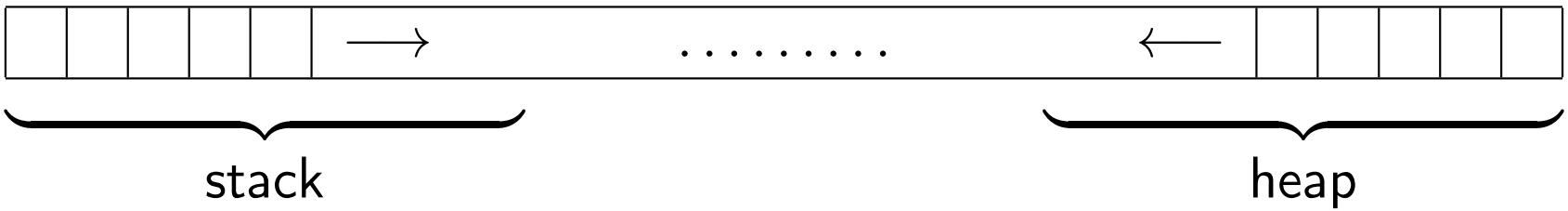
- La memoria dinámica es la que creamos y destruimos en ejecución (apuntadores u objetos en lenguajes como Java).

  La guardamos en el montón (heap) y requiere gestión de memoria (aunque no lo haremos).

- Los arrays dinámicos son un híbrido, los creamos en ejecución pero los guardamos en la pila (al final). Si tenemos arrays dinámicos en bloques internos se puede hacer una gestión más simple de la memoria recuperando memoria al cerrar cada bloque.

Memoria estática y dinámica

Memoria:



Si se cruzan se aborta la ejecución.

- En muchas máquinas virtuales de pila, la pila de ejecución (con los cálculos intermedios) forma parte de la zona de pila de la memoria.
- En WebAssembly están separadas. La pila de ejecución es independiente y la gestiona la máquina virtual.

  La memoria se usa para la información que se almacena (estática y dinámica).

  En el futuro habrá varías memorias y se podrá separar la memoria estática (que guardamos en forma de pila) de la dinámica (que guardamos como montón).

Registros

En algunas máquinas de pila existen registros predefinidos para

- el *program counter* (PC) que indica en que instrucción nos encontramos.
- el *stack pointer* (SP) que indica la última posición ocupada de la zona de pila.
- el *new pointer* (NP) indica la última posición ocupada por el montón.

En WebAssembly, no necesitamos controlar el PC (se gestiona internamente) y tenemos que crear nosotros explícitamente el SP y el NP. Se puede hacer con variables globales mutables o en las primeras posiciones de la *memory*.

Cálculo de memoria estática máxima

```
maxMemory(AST a, int & c, int & max){

}
```

Cálculo de memoria estática máxima

```
maxMemory(AST a, int & c, int & max){
  if isDef(a) {
    c += size(a);
    if c > max { max = c; }
  }
}
```

Cálculo de memoria estática máxima

```
maxMemory(AST a, int & c, int & max){
  if isDef(a) {
    c += size(a);
    if c > max { max = c; }
  } else if isBlock(a) {
    c1 = 0;
    max1 = 0;
    maxMemory(a,c1,max1);
}
```

Cálculo de memoria estática máxima

```
maxMemory(AST a, int & c, int & max){
  if isDef(a) {
    c += size(a);
    max += size(a);
  } else if isBlock(a) {
    c1 = 0;
    max1 = 0;
    maxMemory(a,c1,max1);
    if c+max1 > max {
      max = c + max1;
    }
}
```

Cálculo de memoria estática máxima

```
maxMemory(AST a, int & c, int & max){
  if isDef(a) {
    c += size(a);
    max += size(a);
  } else if isBlock(a) {
    c1 = 0;
    max1 = 0;
    maxMemory(a,c1,max1);
    if c+max1 > max {
      max = c + max1;
    }
  } else {
      for i in hijos(a) {
        maxMemory(i,c,max);
      }
  }
}
```

Contenidos

Funciones

Una **declaración** de función consta de:

- **cabecera**: con el nombre y la especificación de los parámetros con su tipo y si se pasan **por valor** o **por referencia**.
- **declaraciones locales**: tipos, variables y constantes (incluso funciones anidas).
- **bloque de instrucciones**.

La **llamada** a una función contiene el nombre de la función y los argumentos reales de la llamada:

- **Expresiones** del tipo apropiado, si el parámetro se pasa por valor.
- **Variables** del tipo apropiado, si se pasa por referencia.

Llamadas a función

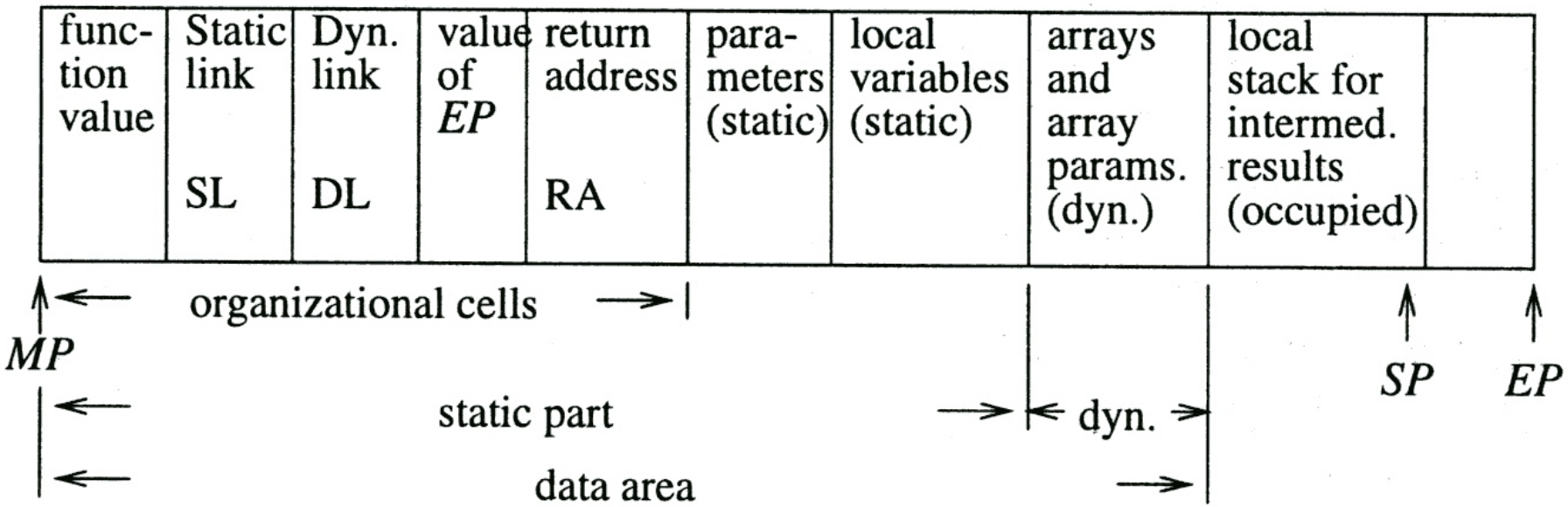
El código generado para una activación de una función, denominado **secuencia de llamada**, consiste en:

1. **Apilar** el marco de activación encima del marco en curso (el del llamante).
2. Establecer los **enlaces estáticos y dinámicos**.
3. Evaluar las expresiones de los parámetros por valor y copiar el resultado en el lugar que le corresponde en el marco de activación.
4. Copiar la dirección de los parámetros por referencia en el lugar que le corresponde en el marco de activación.
5. **Salvar** la dirección de retorno en el marco y saltar a la primera instrucción ejecutable de la función.

Algunas de estas acciones puede no ser necesario hacerlas según las instrucciones que tenga nuestra máquina virtual.
Por ejemplo, en WebAssembly tenemos una operación `call` para llamar a funciones que gestiona la dirección de retorno y el salto.

Marco de activación

Cada llamada a función tiene asociado un marco de activación que se guarda en pila. En general el marco contendrá las siguientes partes:



En WebAssembly no nos hace falta ni el *function value* (ya que lo devolverá la función) ni el *EP* asociado a la *local stack for intermediate results* (como ya hemos comentado) ni el *RA* (ya que se gestiona automáticamente), por contra deberemos guardar el valor de *SP* antes de la ejecución para poder restaurarlo.

Marco de activación

En WebAssembly nos hará falta:

1. El **enlace estático** `SL` (si tenemos funciones anidadas) y el **enlace dinámico** `DL` del marco. El primero apunta al principio del marco de su **predecesor estático**, y el segundo apunta al principio del marco de su **predecesor dinámico** (quien le ha llamado).
2. El *SP* del llamante.
3. A partir de ahí, se asignan las direcciones de los **parámetros**. Notad que al numerar los identificadores de una función, las declaraciones empiezan con los parámetros.
4. A continuación las **variables locales**.
5. Al final el espacio para el contenido de los **arrays dinámicos**.
6. Almacenaremos los valores de *SP* y *MP* en variables globales mutables.

Secuencia de entrada

Las primeras instrucciones del código WebAssembly generado para una función se denominan **secuencia de entrada** e incluyen:

1. **Creación** de los arrays dinámicos declarados como variables locales y rellenar los campos del marco de activación (no lo detallaremos).
2. **Reserva de memoria** para el nuevo marco de activación y **establecimiento** del nuevo SP, es decir el tamaño después de añadir el nuevo marco de activación. El nuevo SP se almacena en el marco de activación después del enlace dinámico.
3. **Comprobar** si el tamaño de la zona de pila (SP) colisiona con el montón (NP).

En otras máquinas virtuales puede ser necesario calcular la zona de pila necesaria para los cálculos intermedios (EP) y realizar el **salto** a la primera instrucción del bloque de la función. En WebAssembly esto lo gestiona la máquina virtual.

WebAssembly para reserva de memoria

```
(memory 2000)
(global $SP (mut i32) (i32.const 0)) ;; start of stack
(global $MP (mut i32) (i32.const 0)) ;; mark pointer
(global $NP (mut i32) (i32.const 131071996)) ;; heap 2000*64*1024-4
(func $reserveStack (param $size i32)
   (result i32)
   get_global $MP
   get_global $SP
   set_global $MP
   get_global $SP
   get_local $size
   i32.add
   set_global $SP
   get_global $SP
   get_global $NP
   i32.gt_u
   if
   i32.const 3
   call $exception
   end
)
```

Lo mismo para el montón, pero restando y sin devolver ningún valor.

Direccionamiento de variables

Para explicar el **direccionamiento de variables** locales y no locales en tiempo de ejecución, introducimos el concepto de **profundidad de anidamiento** (abreviado *PA*), que se aplica tanto a declaración como uso:

$$\begin{cases} \text{PA}(id \text{ declarado/usado en programa principal}) & = 0 \\ \text{PA}(id \text{ declarado/usado en unidad con PA}= n) & = n+1 \end{cases}$$

Si no hay funciones anidadas, solo tenemos 0 para el programa principal y 1 para el resto. En tal caso tampoco es necesario mantener el enlace estático: usamos solo el inicio del principal (0) y *MP*.

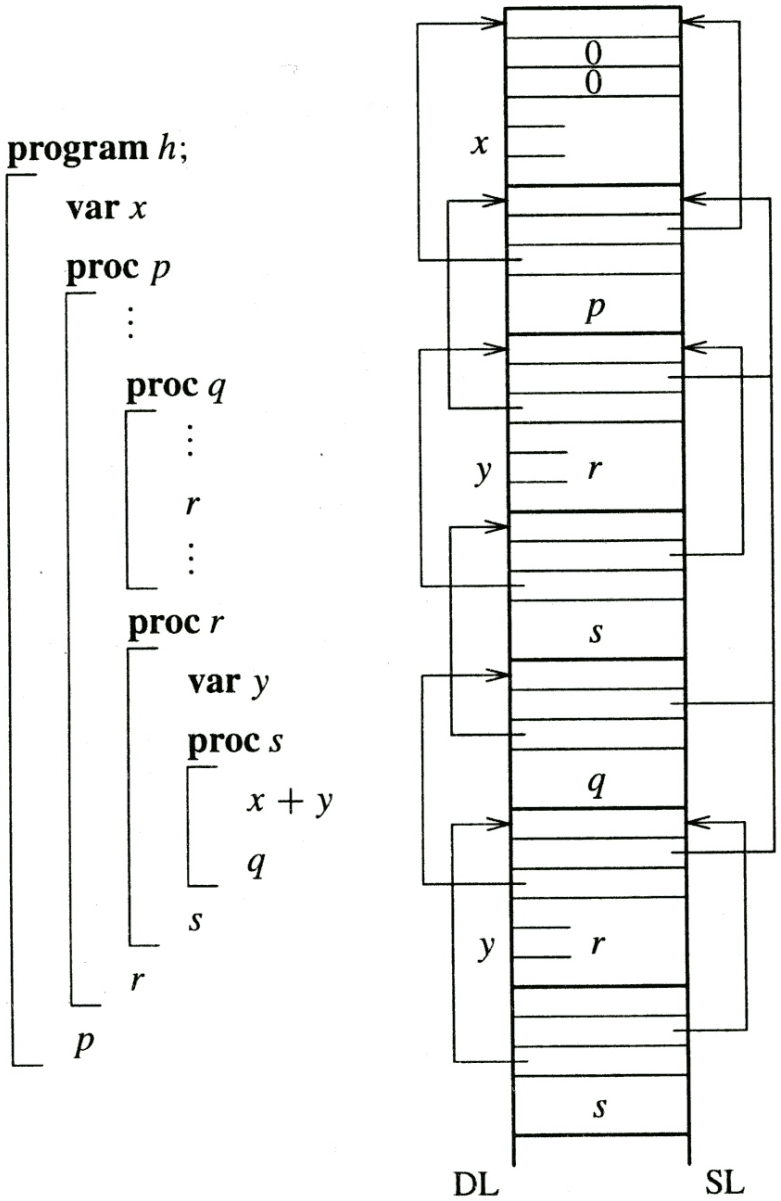
Si hay funciones anidadas, para cada uso de una variable x necesitamos:

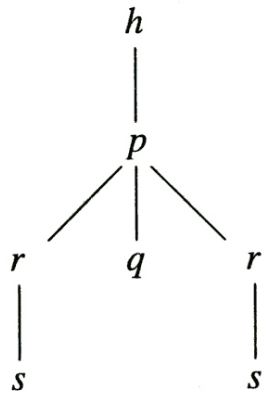
- $PA(\text{ap. de uso de } x) = m$.
- $PA(\text{ap. de definición de } x) = n$.
- $\rho(x)$ en la unidad en que está declarada x.

Hay que remontar $m - n$ enlaces estáticos en la pila de marcos e ir a $\rho(x)$.

55

Direccionamiento de variables





Definición de funciones en WebAssembly

Cabecera de la correspondiente función WebAssembly:

- Como norma general, pasaremos los parámetros por pila (ya que puede haber arrays o structs por valor).
- Si solo se permiten parámetros por referencia (se valorará menos) entonces podemos usar los propios parámetros de las funciones WebAssembly, ya que solo son direcciones y simplifica el trabajo.
- Se pueden combinar las dos cosas.
- usamos el resultado de la función WebAssembly para devolver el resultado (asumiendo que es un tipo simple o una dirección)

Una vez definida la cabecera WebAssembly de la función incluiremos:

1. las instrucciones de la **secuencia de entrada**,
2. las instrucciones del bloque de la función y
3. las instrucciones de eliminación del marco actual.

Código de llamada a función en WebAssembly

1. Evaluamos las expresiones de los parámetros pasados por valor.
2. Copiamos los parámetros. En caso de parámetros pasados por valor, hay que copiar todo su contenido a la zona que les corresponde del marco de activación (si son arrays dinámicos se rellenan los descriptores y se copia el contenido en la zona final de la pila). Notad que, aunque aún no hemos hecho la reserva de memoria, sabemos que los parámetros van a partir de $SP + 8$ (ya que SP apunta a la primera posición libre de la zona de pila y necesitamos 8 bytes para el enlace dinámico y el SP del nuevo bloque). Si guardáis alguna información más hay que añadir los bytes al 8.
3. Guardamos el valor del enlace dinámico (y el estático si es necesario).
4. Y llamamos a la función WebAssembly asociada con `call`.