Paradigmas de la Programación

FaMAF 2015

programa: sintaxis y semántica

- un programa es la descripción de un proceso dinámico
  - **sintaxis**: texto del programa
  - **semántica**: cosas que hace

  una definición precisa del significado de un programa correcto sintácticamente y a nivel de tipos

lenguaje objeto y metalenguaje

- el **metalenguaje** es el que usamos para hablar de un **lenguaje objeto**
- necesitamos un lenguaje para hablar de la semántica de los lenguajes de programación

semántica

delimitación de la semántica de los lenguajes de programación

algunas observaciones sobre computabilidad (capítulo 2 de Mitchell, no es necesario leerlo pero puede resultarles interesante y es cortito)

- los programas pueden definir **funciones parciales**
  - algunos de sus valores pueden **indefinidos** (p.ej., si no terminan)
  - algunos de sus valores pueden ser **errores**

delimitación de la semántica de los lenguajes de programación

- intuitivamente, una función es computable si hay algún programa que la computa
  - problema: definición dependiente de la implementación de un lenguaje de programación concreto, con sus limitaciones y particularidades

queremos una definición independiente (libre!) de lenguaje

como definir la clase de funciones computables?

- una clase de funciones matemáticas: las **funciones recursivas parciales** (Church)
- las que se pueden computar con una máquina idealizada, abstracta: la **máquina de Turing**
  - cinta infinita, dividida en celdas
  - un cabezal de lectura – escritura
  - un controlador de estado finito
- si se puede expresar en **lambda cálculo**

diferentes aproximaciones a la semántica

- lambda cálculo
- semántica denotacional
- semántica operacional

lambda cálculo

- apartado 4.2. de Mitchell (no va a entrar en el examen)

lambda cálculo

sistema matemático que ilustra conceptos de lenguajes de programación

- notación para definir funciones
- sistema de prueba para probar ecuaciones entre expresiones
- reglas de cálculo (reducciones lambda)

lambda cálculo: funciones

una función es una regla para encontrar un valor a partir de un argumento

- el significado de una expresión no se define a partir de una variable
- los operadores de ligado ligan una variable en un determinado alcance

$\lambda x.x$	abstracción lambda de la función identidad
$\lambda x.(f(gx))$	otra abstracción lambda
$(\lambda x.x)5$	una aplicación

lambda cálculo: reducción

- la reducción es equivalencia ecuacional con dirección
- M → N significa que en un paso de computación, la expresión M se puede evaluar a la expresión N
- cuando no se pueden aplicar más reducciones, se llega a la **forma normal**

lambda cálculo: ejemplo de reducción

$(\lambda f.\lambda x.f(fx))(\lambda y.y+1)2$

$\rightarrow (\lambda x.(\lambda y.y+1)((\lambda y.y+1)x)2$

$\rightarrow (\lambda x.(\lambda y.y+1)(x+1))2$

$\rightarrow (\lambda x.(x+x+1))2$

$\rightarrow (2+1+1)$

$\rightarrow 3+1$

$\rightarrow 4$

lambda cálculo

- provee la base para muchos conceptos de lenguajes de programación
- especialmente adecuado para lenguajes funcionales
- captura la esencia del ligamiento de variables
- es un tipo de semántica operacional

diferentes aproximaciones a la semántica

- lambda cálculo
- semántica denotacional
- semántica operacional

semántica denotacional

- apartado 4.3 de Mitchell

semántica denotacional

- semántica matemática para programas, desarrollada a fines de los 1960 por Christopher Strachey y Dana Scott
- el significado de un programa es una **función matemática**, de estados a estados, no un algoritmo
- un **estado** es una función que representa los valores en memoria en un momento determinado durante la ejecución de un programa

motivación

- precisión: matemática en lugar de lenguaje natural
- significado puro, abstrayéndose de detalles de implementación: evitar particularidades de máquinas y lenguajes específicos
- análisis de programas
  - prueba de programas (sistema de tipos, flujo de control)
  - correctitud de compilador
  - comparación de lenguajes

- no sólo para imperativos: también se usa para funcionales
- en principio, cualquier programa imperativo se puede escribir como un programa funcional puro (en otro lenguaje)
- http://en.wikibooks.org/wiki/Haskell/Denotational_semantics

- útil para
  - determinar la correctitud de un programa
  - optimización y análisis estático
- no sirve para
  - calcular tiempo de ejecución o requerimientos de memoria, para eso, mejor usar semántica operacional, en la que se modelan los estados de la máquina y las transiciones paso a paso asociadas a un programa, (p.ej., reducciones de lambda cálculo)

composicionalidad

el significado de un programa se determina a partir de su texto composicionalmente

- muy pegado al **árbol sintáctico**
- cada expresión tiene una **denotación suficientemente detallada** para dar cuenta de su comportamiento en contextos mayores
- resulta útil para hacer **optimizaciones** sin cambios semánticos (sustituciones, etc.)

composicionalidad y árbol

por inducción sobre la estructura del árbol: la semántica de una expresión compuesta se construye composicionalmente de la de las expresiones más simples que componen su árbol (**NO** es un algoritmo)

semántica denotacional de números binarios

```
e ::= n | e+e | e-e
e ::= b | nb
b ::= 0 | 1
```

el árbol de la expresión `e` se escribe `[[e]]`

semántica denotacional de números binarios (2)

definimos el significado `E[[e]]` de una expresión `e`, basada en su árbol sintáctico `[[e]]`:

```
E[[0]]  = 0
E[[1]]  = 1
E[[nb]] = E[[n]] * 2 + E[[b]]
```

$\mathtt{E[[e_1+e_2]]} = \mathtt{E[[e_1]]} + \mathtt{E[[e_2]]}$

$\mathtt{E[[e_1-e_2]]} = \mathtt{E[[e_1]]} - \mathtt{E[[e_2]]}$

semántica denotacional de números binarios (3)

- el valor del árbol sintáctico de $[[e_1+e_2]]$ es la suma de los valores de los árboles $[[e_1]]$ y $[[e_2]]$ esto no es una definición circular porque los árboles $[[e_1]]$ y $[[e_2]]$ son más chicos que $[[e_1+e_2]]$.

expresiones con variables

```
e ::= v | n | e+e | e-e
n ::= d | nd
d ::= 0|1|2|3|4|5|6|7|8|9
v ::= x | y | z | ...
```

- el valor de una variable depende del estado de la máquina
- el estado de la máquina es una función de variables a números
- `E[[e]](s)` es el valor de `e` en el estado `s`

$$E[[x]](s) = s(x)$$

$$E[[0]](s) = 0$$

$$E[[1]](s) = 1$$

$$E[[nd]](s) = E[[n]](s)*10+E[[d]](s)$$

$$E[[e_1+e_2]](s) = E[[e_1]](s)+E[[e_2]](s)$$

semántica de estados, comandos

estado = variables $\rightarrow$ *valores*

comando = estado $\rightarrow$ *estado*

- por ejemplo, la semántica de la asignación se puede expresar con la función *modificar*:

modificar(s,x,a) = *λv si v=x entonces a si no s(v)*

Formalizing the Type System

- Approach: write a set of function specifications that define what it means to be type safe
- Basis for functions: Type Map, *tm*
  - $tm = \{ <v_1,t_1>, <v_2,t_2>, \dots <v_n,t_n> \}$
  - Each v_i represents a variable and t_i its type
  - Example:
    - int i,j; boolean p;
    - *tm* = { <i, int>, <j, int>, <p, boolean> }

Declarations

- How is the type map created?
  - When we declare variables
- typing: Declarations → Typemap
  - i.e. declarations produce a typemap
- More formally
  - typing(Declarations d) = $\bigcup_{i=1}^{n} < d_i.v, d_i.t >$
  - i.e. the union of every declaration variable name and type
  - In Java we implemented this using a HashMap

Semantic Domains and States

- Beyond types, we must determine semantically what the syntax means
- **Semantic Domains** are a formalism we will use
  - Environment, γ = set of pairs of variables and memory locations
    - γ = {<i, 100>, <j, 101>} for i at Addr 100, j at Addr 101
  - Memory, μ = set of pairs of memory locations and the value stored there
    - μ = {<100, 10> , <101, 50>} for Mem(100)=10, Mem(101)=50
  - State of the program, σ = set of pairs of active variables and their current values
    - σ = {<i,10>, <j, 50>} for i=10, j=50

semántica de estados

- **entorno**, γ = conjunto de pares de variables y posiciones en memoria
  – γ = {<i, 100>, <j, 101>} para i en Addr 100, j en Addr 101
- **memoria**, μ = conjunto de pares de posiciones en memoria y el valor que se aloja en cada una de ellas
  – μ = {<100, 10> , <101, 50>} para Mem(100)=10, Mem(101) =50
- **estado del programa**, σ = conjunto de pares de variables con variables activas y sus valore actuales
  – σ = {<i,10>, <j, 50>}   para i=10, j=50

ejemplo de estado

- x=1; y=2; z=3;
  - en este punto σ = {<x,1>,<y,2>,<z,3>}
  - notación: σ(y)=2
- y=2*z+3;
  - en este punto σ = {<x,1>,<y,9>,<z,3>}
- w=4;
  - en este punto σ = {<x,1>,<y,9>,<z,3>, <w,4>}

- también se pueden tener expresiones; por ejemplo σ(x>0) = true

unión que sobreescribe

un cierto tipo de transformación de estados

X $\overline{\cup}$ Y = reemplaza los pares <x,v> cuyo primer miembro se corresponde con un par <x,w> de Y por <x,w> y añadimos a X los otros pares de Y

ejemplo: $\sigma_1 = \{< x,1 >,< y,2 >,< z,3 >\}$

$$\sigma_2 = \{< y,9 >,< w,4 >$$

$$\sigma_1 \overline{\bigcup} \sigma_2 = \{< x,1 >,< y,9 >,< z,3 >,< w,4 >\}$$

semántica de Skip

- Skip

$$M(Skip\ s, State\ \sigma) = \sigma$$

semántica de asignación

- evaluar una expresión y asignarla a la variable

- ejemplo: x=a+b

$$\sigma = \{< a,3 >, < b,1 >, < x,88 >\}$$

$$M(x = a + b;, \sigma) = \sigma \overline{U} \{< x, M(a + b, \sigma) >\}$$

$$\sigma = \{< a,3 >, < b,1 >, < x,4 >\}$$

semántica del condicional

$$M(Conditional\ c, State\ \sigma)$$

$$= M(c.thenbranch, \sigma) \quad if\ M(c.test, \sigma)\ is\ true$$

$$= M(c.elsebranch, \sigma) \quad otherwise$$

```
If (a>b) max=a; else max=b
```

$$\sigma = \{< a,3 >< b,1 >\}$$

$$M(\text{if (a > b)max = a; else max = b;}, \sigma)$$

$$= M(\text{max} = a;, \sigma) \quad if\ M(a > b, \sigma)\ is\ true$$

$$= M(\text{max} = b;, \sigma) \quad otherwise;$$

semántica del condicional (2)

$$\sigma = \{< a,3 >< b,1 >\}$$

$$M(\text{if (a > b)max = a; else max = b;}, \sigma)$$

$$= M(\max = a;, \sigma) \quad \textit{since } M(a > b, \sigma) \textit{ is true}$$

$$= \sigma \overline{U} \{< \max,3 >\}$$

$$= \sigma\{< a,3 >, < b,1 >, < \max,3 >\}$$

semántica de bloque

una secuencia de sentencias

$$M(Block\ b, State\ \sigma)$$

$$= \sigma \qquad\qquad if\ b = \varphi$$

$$= M((Block)b_{2\ldots n}, M((Statement)b_1, \sigma)) \ \ if\ b = b_1 b_2 \ldots b_n$$

ejemplo de semántica de bloque

$$\left.\begin{array}{l} b_1 = \texttt{fact = fact * i;} \\ b_2 = \texttt{i = i - 1;} \end{array}\right\} \text{bloque}$$

- $M(b,\sigma) = M(b_2,M(b_1,\sigma))$

$= M(i=i-1,M(fact=fact*i,\sigma))$

$= M(i=i-1,M(fact=fact*i,\{<i,3>,<fact,1>\}))$

$=M(i=i-1,\{<i,3>,<fact,3>\})$

$=\{<i,2>,<fact,3>\}$

semántica de ciclo

los ciclos se componen de una expresión test y un cuerpo de sentencia

$$
\begin{aligned}
&M(Loop\ l, State\ \sigma) \\
&\quad = M(l, M(l.body, \sigma)) \qquad if\ \ M(l.test, \sigma)\ \text{is true} \\
&\quad = \sigma \qquad\qquad otherwise
\end{aligned}
$$

definición recursiva

ejemplo de ciclo

estado inicial: σ={<N,3>}

```
fact=1;
i=N;
while (i>1) {
 fact = fact * i;
 i = i -1;
}
```

después de las dos primeras sentencias,

σ = {<fact,1>,<N,3>,<i,3>}

ejemplo de ciclo

σ = {<fact,1>,<N,3>,<i,3>}

M(while(i>1) {…}, σ)

= M(while(i>1) {…}, M(fact=fact*i; i=i-1;, σ)

= M(while(i>1) {…}, {<fact,3>,<N,3>,<i,2>})

= M(while(i>1) {…}, {<fact,6>,<N,3>,<i,1>})

= M(σ)

={<fact,6>,<N,3>,<i,1>}

Defining Meaning of Arithmetic Expressions for Integers

First let's define ApplyBinary, meaning of binary operations:

$$ApplyBinary : Operator \times Value \times Value \rightarrow Value$$

$$ApplyBinary(Operator\ op, Value\ v_1, Value\ v_2)$$

$$= v_1 + v_2 \qquad if\ op = +$$

$$= v_1 - v_2 \qquad if\ op = -$$

$$= v_1 \times v_2 \qquad if\ op = *$$

$$= floor\left(\left|\frac{v_1}{v_2}\right|\right) \times sign(v_1 \times v_2) \qquad if\ op = /$$

Denotational Semantics for Arithmetic Expressions

Use our definition of ApplyBinary to expressions:

$$M : Expression \times State \rightarrow Value$$

$$\begin{aligned}
M(Expression\ e, State\ \sigma) & \\
= e & \qquad if\ e\ is\ a\ Value \\
= \sigma(e) & \qquad if\ e\ is\ a\ Variable \\
= ApplyBinary(e.op, & \\
M(e.term1, \sigma), & \\
M(e.term2, \sigma)) & \qquad if\ e\ is\ a\ Binary
\end{aligned}$$

Recall: op, term1, term2, defined by the Abstract Syntax
term1,term2 can be any expression, not just binary

Arithmetic Example

- Compute the meaning of x+2*y
- Current state σ={<x,2>,<y,-3>,<z,75>}

- Want to show: M(x+2*y,σ) = -4
  - x+2*y is Binary
  - From M(Expression e, State σ) this is

      ApplyBinary(e.op, M(e.term1, σ), M(e.term2,σ))
      = ApplyBinary(+,M(x,σ),M(2*y,σ))
      = ApplyBinary(+,2,M(2*y,σ))

    M(2*y,σ) is also Binary, which expands to:

      ApplyBinary(*,M(2,σ), M(y,σ))
      = ApplyBinary(*,2,-3) = -6

    Back up: ApplyBinary(+,2,-6) = -4

Java Implementation

```
Value M(Expression e, State state) {
    if (e instanceof Value)  return (Value)e;
    if (e instanceof Variable)  return (Value)(state.get(e));
    if (e instanceof Binary) {
        Binary b = (Binary)e;
        return applyBinary(b.op, M(b.term1, state),
            M(b.term2, state);
    }
    ...
```

Code close to the denotational semantic definition!

diferentes aproximaciones a la semántica

- lambda cálculo
- semántica denotacional
- semántica operacional

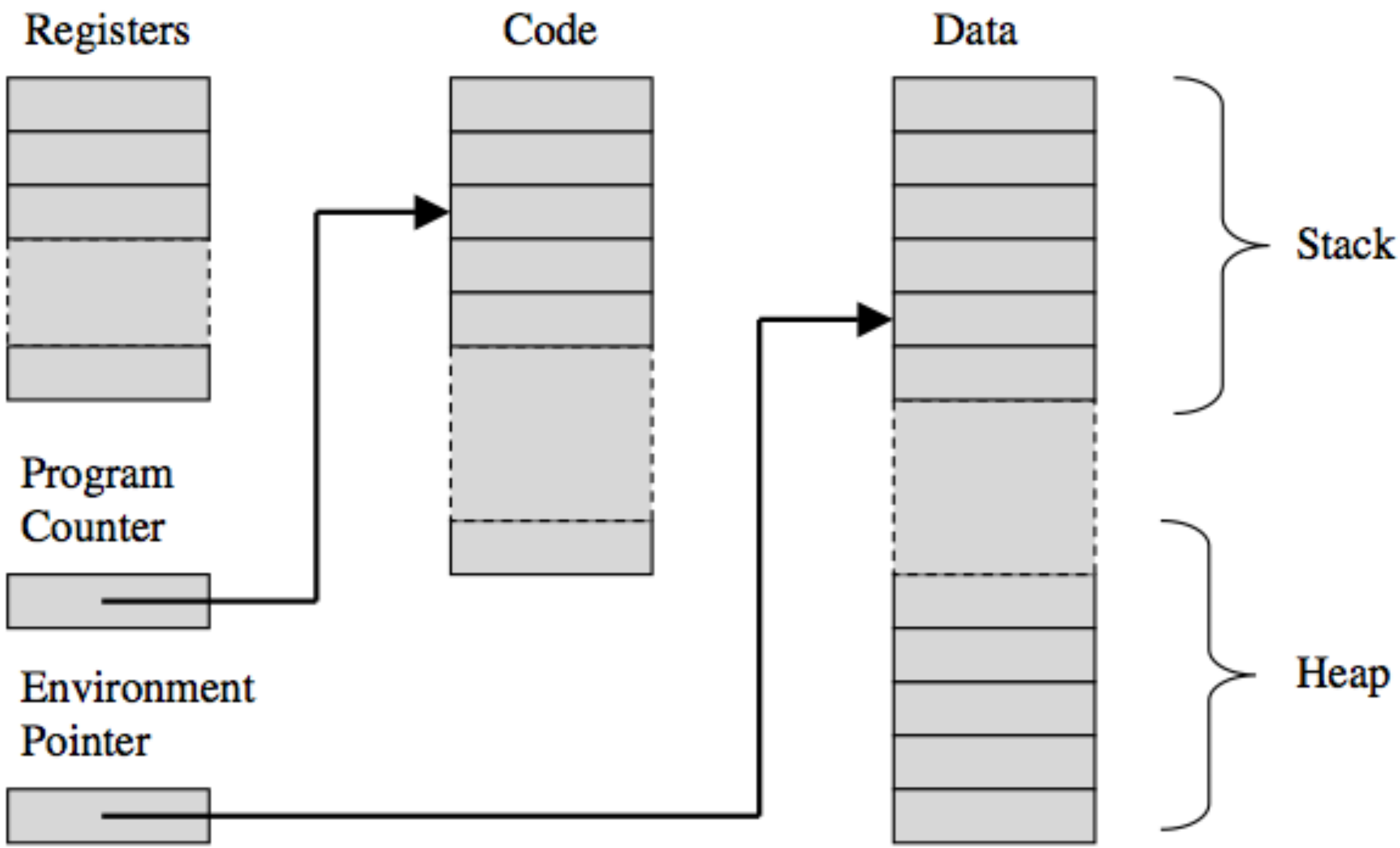
semántica operacional

- apartado 7.2 de Mitchell

semántica operacional

- una representación abstracta de la ejecución de un programa, como secuencia de transiciones entre estados (en una máquina abstracta)
- los estados son una descripción abstracta de la memoria y estructuras de datos
- las transiciones siguen la estructura de la sintaxis

una máquina abstracta



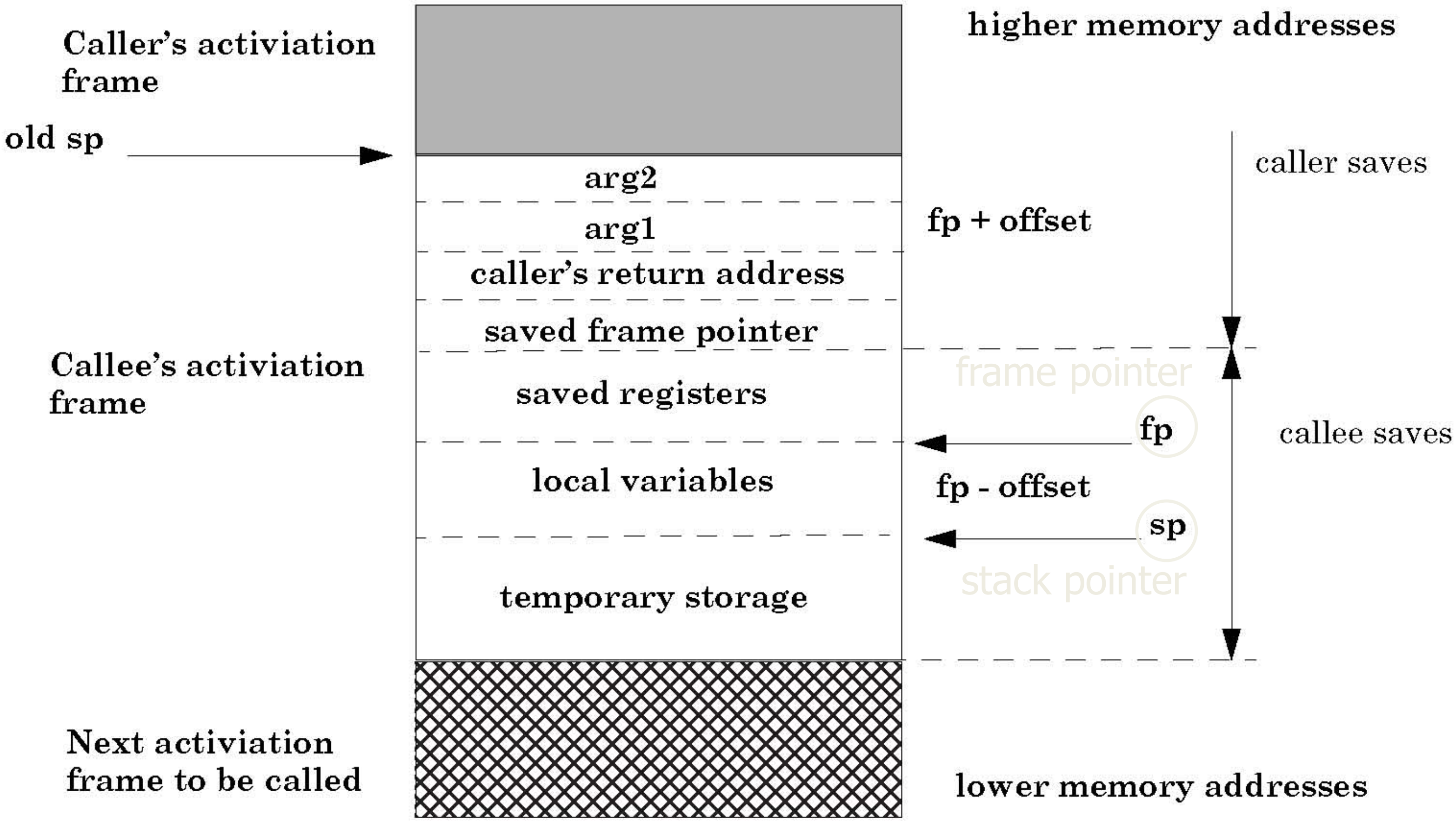
registros de activación o marcos de pila (*Activation Records* o *stack frames*)

guardan la información de un bloque:

- variables locales
- control link al que ha llamado al activation record, para volver
- variables temporales y resultados intermedios

- entran y salen de la pila (*stack*), eso hace que puedan usarse llamados anidados

Typical x86 Activation Record



pila de ejecución

- los registros de activación se guardan en la pial
  - cada nuevo bloque apila (*push*) un nuevo registro de activación en la pila
  - cada vez que se termina un bloque se saca (*pop*) el registro de arriba de la pila
  - la pila tiene todos los registros que son activos en un determinado momento de la ejecución, con el que se usó más recientemente en la punta

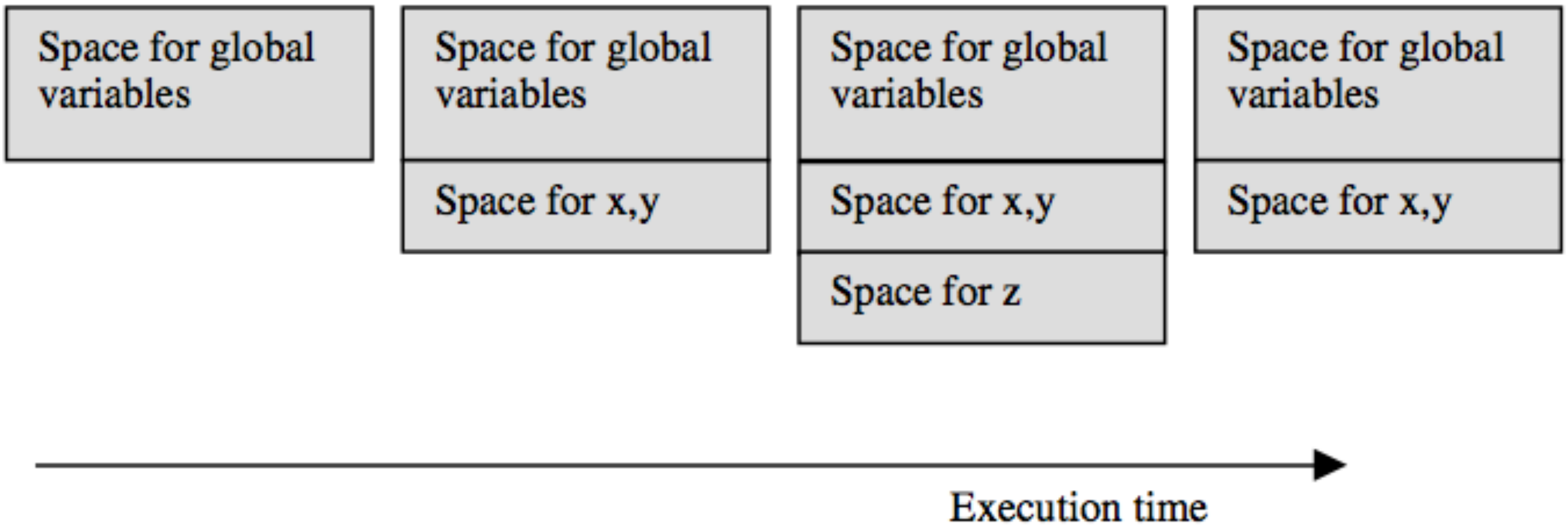
pila de ejecución: ejemplo

fact(3)

1. apila un registro, llama a fact(2)
2. esta llamada apila otro registro, llama a fact(1)
3. esta llamada apila otro registro, de forma que hay tres registros en la pila
4. cuando se termina de ejecutar el bloque del registro más reciente, se saca ese registro de la pila
5. y así sucesivamente hasta que la pila queda vacía

pila de ejecución: otro ejemplo

```
{ int x=0;
  int y=x+1;
      { int z=(x+y)*(x-y);
      };
};
```



pila de ejecución: resultados intermedios

```
{ int z = (x+y)*(x-y);
}
```

may have the form

Space for z
Space for x+y
Space for x-y

tipos

capítulo 2 de
Programming Language Design Concepts