Tema 1.5. Máquinas virtuales

Aunque hay muchos rasgos que se podrían destacar como candidatos para formar parte de una definición del concepto de máquina virtual (a partir de ahora MV), una MV se puede definir sencillamente como una capa de abstracción que separa el funcionamiento de un ordenador de su hardware. Además, las MV se dividen en *abstractas* o teóricas, como sería la máquina de Turing (el primer ejemplo de una MV) y modelos como el calculo π o lambda, y *concretas* o prácticas (a las que se quiere normalmente se hace referencia al hablar de MV). En esta sección se tratarán las MV concretas que, como se verá, son capas de software que juegan un papel relevante tanto en el funcionamiento de los lenguajes compilados como interpretados.

1.5.1 Introducción al concepto

Las MV se construyeron para simplificar el proceso del control del hardware de un ordenador porque extienden y enmascaran la funcionalidad del hardware a través de procedimientos y datos abstractos. Se pueden identificar dos tipos de MV concretas: las que juegan un papel en la preparación de un programa para su ejecución (tiempo de compilación) y las que permiten la ejecución de dicho programa. La figura 1 muestra la diferencia entre los dos tipos (cuyos conceptos se explicarán a lo largo de esta sección):

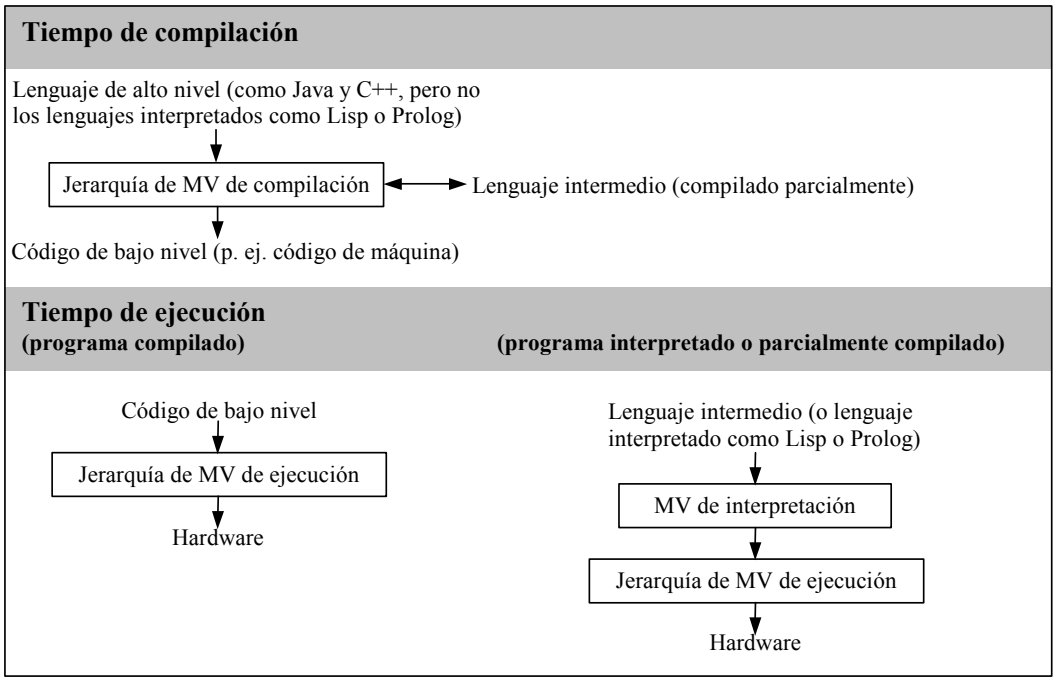

Figura 1. El papel de las MV en la compilación y ejecución de un programa

Las MV suelen aparecer en una jerarquía (como se verá en la siguiente sección). Usamos la jerarquía de MV de compilación, por ejemplo, cada vez que invocamos el compilador de C++. Y dos ejemplos muy comunes del papel de una MV en la jerarquía de MV de ejecución son PostScript (que define una MV para una impresora) y MSDOS bajo MS Windows (que define una MV que permite que antiguos programas de DOS funcionen bajo Windows). PostScript es una MV que incluye el estado gráfico, la ruta actual, el estado del diccionario y, aunque sea bastante complejo, presenta una visión de alto nivel de muchos tipos distintos de impresoras, aislando las características de las impresoras de las aplicaciones que las usan.

A la hora de implementar un lenguaje de programación, las estructuras de datos y algoritmos utilizados en la ejecución de un programa (es decir, la sintaxis y semántica de cada estructura) definen las MV (implícitamente) para este lenguaje. Y las decisiones que toma el

desarrollador afectan tanto a la traducción como a la ejecución de los programas escritos en este lenguaje. Las MV se distinguen entre sí según tres criterios:

- La concepción de las MV que tiene el desarrollador basada en la definición del lenguaje.
- Las facilidades disponibles en el hardware del ordenador.
- Las decisiones de implementación tomadas por los desarrolladores.

Aunque se puede escribir un compilador para traducir (o compilar) un programa escrito en un lenguaje de alto nivel directamente al código máquina (en el sentido de no depender de ninguna otra pieza en el sistema para producirlo), no es nada común hacerlo hoy en día por varias razones, dos de las cuales son:

- Cuando aparezca en el mercado el siguiente procesador (por ejemplo, Pentium IV después de Pentium III), habrá que modificar el compilador.
- No tiene sentido repetir el proceso de diseño y desarrollo, que es costoso, cuando ya se ha hecho lo mismo muchas veces antes para otros lenguajes para una máquina concreta.

Un caso lo constituyen los compiladores C, que ya llevan mucho tiempo en funcionamiento y son muy eficientes y estables. Por lo tanto, hay varios lenguajes como C++ (al menos en las primeras versiones) cuyo "compilador" tradujo el código fuente de C++ a C para poder luego usar el compilador C estándar. En general, sólo se suele rediseñar un compilador nuevo desde cero cuando los ingenieros del lenguaje están intentando alcanzar nuevos objetivos para ese lenguaje, en caso de que vayan más allá que el modelo funcional de los compiladores ya disponibles como, por ejemplo, es el caso de Java (que se tratará en la sección 1.5.4.).

1.5.2 Jerarquías de máquinas virtuales

La realidad es que el desarrollador de un lenguaje suele implementar las MV (de compilación y/o ejecución) de su lenguaje en términos de otras MV ya existentes. Y en el futuro, un programador de aplicaciones utilizará las MV implementadas por el desarrollador del lenguaje para producir programas que a su vez puedan servir como MV para otras aplicaciones, etc. La conclusión, como puede verse, es que una MV no suele existir aislada, sino como parte de una jerarquía de MV. A continuación se va a ver el papel de las jerarquías de MV en el tiempo de compilación y de ejecución.

En primer lugar, se va a considerar el papel de la jerarquía de MV en el tiempo de compilación. En el caso de C++, como lenguaje de alto nivel que se compila a un lenguaje tan cerca del nivel de máquina como sea posible, se puede ver que el proceso de compilación de un programa de C++ consiste en la interacción de la jerarquía de MV mostrada en la figura 2 (aunque puede haber diferencias entre compiladores de C++; por ejemplo, aquí se supone que es un proceso de compilación que usa C como código intermedio). El pre-procesador acepta el código C++ como fuente y produce otra versión del mismo código C++ con algunas extensiones e incorporaciones resueltas. El traductor convierte el código fuente C++ estándar a código fuente C. El compilador C acepta este código como fuente y lo traduce a código ensamblador (una representación simbólica del código máquina). El ensamblador traduce este código simbólico a un código máquina reubicable. Y por fin, el cargador de librerías acepta este código máquina como entrada y produce un programa simple ejecutable, compuesto por el código máquina de entrada y todos los subprogramas necesarios con direcciones de memoria contiguas. Además del proceso de compilación, la traducción completa de los programas de alto nivel (previa a su ejecución) en una forma que corre sobre la máquina, también existe otro proceso, que se llama interpretación, que tiene más que ver con el papel de las MV en el tiempo de ejecución que en el tiempo de compilación.

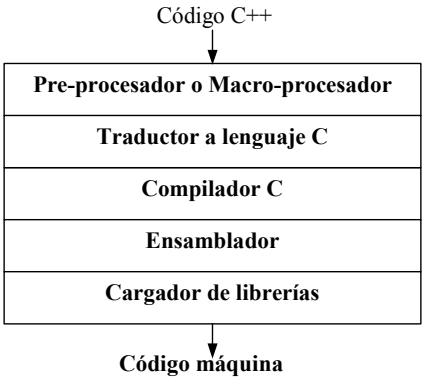

Figura 2. Ejemplo de la jerarquía de MV presente en la compilación de una aplicación

Además de los lenguajes compilados "completamente", hay otros que son compilados "parcialmente" en el sentido de que terminan el proceso de compilación en un código (o lenguaje) intermedio, en vez de en código máquina (se trata el papel de los lenguajes intermedios en la sección 1.5.3).

En segundo lugar, vamos a considerar el papel de la jerarquía de MV en el tiempo de ejecución. Se puede ver un ejemplo en la figura 3. Como se puede ver en él, lo que se podría llamar una aplicación Web (unas páginas Web con elementos interactivos incrustados) está escrita en HTML y funciona sobre una MV dada por el navegador Web, que a su vez está implementado en C o C++. Éste corre sobre una MV compuesta por las librerías de tiempo de ejecución que encapsulan el funcionamiento de los procedimientos (o métodos) y datos de navegador Web que, a su vez, utilizan las funciones de la MV del sistema operativo implementadas en código máquina. Y los programas que componen esta MV corren sobre el firmware, un conjunto de programas de micro-código que controlan directamente el hardware del ordenador.

Las jerarquías del tipo mostrado en la figura 3 tienen una estructura muy parecida a una jerarquía "usa", donde cada capa usa la capa que está directamente debajo. La diferencia es que una MV puede tener una interfaz hacia abajo, además de la interfaz hacia arriba, que especifica los servicios que requiere de la capa inferior, pero sin especificar exactamente qué implementación de los servicios es necesario. Un ejemplo sería la jerarquía de MV que compone el sistema de ventanas X, donde hay una capa que define el funcionamiento independientemente de los dispositivos y, directamente debajo, una capa que define el funcionamiento en términos de los dispositivos. Aquí X depende de la capa que depende de los dispositivos, no de una implementación en concreto de esta capa (por lo tanto, hay versiones de X para muchos tipos de hardware donde la única diferencia es la implementación de la capa relacionada con los dispositivos; las demás capas superiores son idénticas).

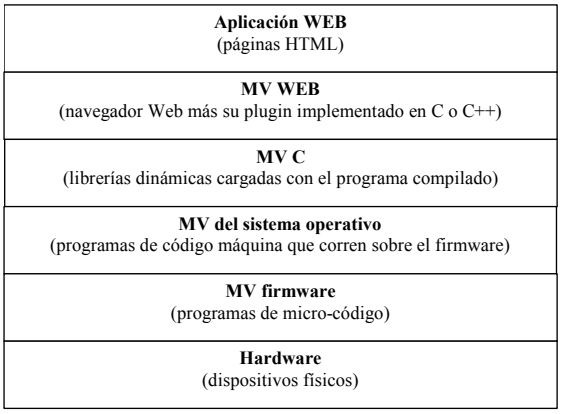

Figura 3 Ejemplo de la jerarquía de MV presente en la ejecución de una aplicación

1.5.3. Lenguajes intermedios

Un lenguaje intermedio se puede definir como una manera de representar procedimientos y estructuras de datos que sirva como entrada para una MV en alguna parte de su jerarquía, entre el lenguaje de entrada (el nivel más alto) y el código ejecutado en la máquina (el nivel más bajo) - tanto en el tiempo de compilación como en el de ejecución (lo que se puede ver en la figura 1).

Para considerar el papel de los lenguajes intermedios y sus ventajas y desventajas, conviene destacar la diferencia entre la traducción de un lenguaje de alto nivel a código máquina anteriormente a su ejecución (su compilación) y su interpretación, es decir, la conversión de cada instrucción del lenguaje a código máquina y su ejecución, una por una, al ejecutar el programa. Este proceso se realiza a través de una MV de interpretación que simula un ordenador cuyo código máquina es el lenguaje de alto nivel que está siendo interpretado. Y típicamente, esta MV se construye a través de un conjunto de programas de código máquina que representa los algoritmos y estructuras de datos necesarios para la ejecución de las instrucciones del lenguaje de alto nivel. Hay ventajas y desventajas en cada manera de convertir los lenguajes de alto nivel a código máquina, que se pueden resumir así:

Compilación		
Ventajas	1.	No hay que repetir la conversión de la misma instrucción a código máquina cada vez que aparece.
	2.	Los programas corren muy rápido.
Desventajas	1.	Pérdida de claridad e información sobre el programa.
	2.	Dificultad en localizar la fuente exacta de error.
Ejemplos → Ada, C, C++, FORTRAN, Pascal		
Interpretación		
Ventajas	1.	No hay pérdida de claridad ni de información sobre un programa ni sobre donde están los errores.
	2.	No hay que decodificar código que no se va a ejecutar.
	3.	El código es típicamente más compacto.
Desventajas	1.	Los programas corren mucho más lentamente – se paga el coste de decodificar cada instrucción.
Ejemplos → HTML, Lisp, ML, Perl, Postscript, Prolog, Smalltalk		

Estos dos casos representan los dos extremos porque, como ya se ha visto, existe también lo que se llama la compilación parcial, que es una mezcla de los dos enfoques, donde se compila el lenguaje de alto nivel a un lenguaje intermedio (más cerca de las estructuras presentes en el código máquina que las del código fuente) y luego se interpreta este lenguaje al ejecutar el programa. Como puede imaginarse, esta técnica combina las ventajas y desventajas de los dos enfoques anteriores. Un ejemplo de esta combinación existe en el lenguaje de programación Java y su entorno.

Entre otras cosas, Java empezó con la idea de liberar al programador de las dificultades de portar su aplicación a nuevas plataformas lo cual, si el programa está muy vinculado a algún aspecto del sistema operativo donde fue escrito, podría ser muy difícil. Se compilará el código fuente de Java a un código byte (*bytecode*) antes de ejecutarlo. Y a la hora de correr el programa, este código, como lenguaje intermedio, sería el lenguaje de entrada para una MV, que con un conjunto de librerías (el entorno de ejecución de Java, *Java Runtime* o JRE), la interpretaría para su ejecución. Por lo tanto, este bytecode podría correr en cualquier hardware donde haya una versión del JRE disponible. Como este bytecode está más cerca del nivel de máquina que de un lenguaje de alto nivel, los programas correrán más rápidamente que los programas completamente interpretados, aunque más despacio que los programas previamente compilados al código máquina.

Como se puede ver en la figura 1, tanto los programas compilados parcialmente a un lenguaje intermedio (como Java) como los programas escritos en lenguajes de alto nivel que se interpretan (como Lisp) requieren una MV para interpretar el programa. La principal ventaja del lenguaje intermedio en este caso es su proximidad al nivel del código máquina, en el sentido de que supone menos trabajo a la hora de ejecutarlo y, por lo tanto, los programas corren más rápidamente que los puramente interpretados.

Además del papel de los lenguajes intermedios en la compilación parcial, se puede destacar su papel en la compilación estándar. Como ejemplo se puede considerar C como lenguaje intermedio para un lenguaje compilado nuevo (como se planteó en la sección 1.5.1 para C++). Si el autor de un nuevo lenguaje decide utilizar C, por ejemplo, como su lenguaje intermedio, sólo tendrá que implementar una MV para convertir el código fuente de su lenguaje a C, ahorrando mucho trabajo. Además de las razones dadas antes, las ventajas de utilizar un lenguaje tan establecido como C como lenguaje intermedio son:

- La facilidad de portar el lenguaje a una nueva máquina (sólo hay que tener un compilador C disponible allí).
- La generación de código máquina es una tarea muy compleja que requiere un conocimiento profundo de la arquitectura de la máquina en cuestión – y de cada máquina en que se quiere una versión del lenguaje.
- La facilidad de modificar algún rasgo del comportamiento del lenguaje en alguna máquina en concreto (por ejemplo, características de memoria o rendimiento - se pueden añadir librerías C customizadas sin grandes problemas).
- Las posibilidades disponibles para mapear estructuras intermedias del nuevo lenguaje a estructuras de datos de C.

Y las desventajas son:

- La depuración es muy difícil porque, entre otras cosas, los errores que ocurren en el código C no son muy fáciles de localizar en lo que ha escrito el programador originalmente en el nuevo lenguaje.
- Las características de rendimiento y eficiencia del lenguaje están determinadas por el compilador C.
- Habrá ocasiones en las que no exista una buena traducción entre una estructura en el nuevo lenguaje y las estructuras de datos en C, por lo que habrá una pérdida de eficiencia en el programa resultante (como, por ejemplo, ocurre en la mayoría de las ocasiones en que se compilan estructuras de Prolog a C – sólo se puede expresar iteración en Prolog utilizando recursión).

1.5.4. La máquina virtual de Java como ejemplo de una MV

La MV de Java es una máquina de pila. Las instrucciones interpretadas por ella manipulan datos almacenados como elementos en una pila. El contenido ejecutable de un archivo de bytecodes contiene un vector de instrucciones bytecode para cada método. Los bytecodes son instrucciones para la MV, que tiene algunos registros de variables locales y una pila para la evaluación de expresiones. Las primeras variables locales son inicializadas con los parámetros actuales. Cada variable local o elemento de la pila es una palabra que corresponde a un entero de 32 bits, a un punto flotante o a una referencia a objeto (puntero). Para puntos flotantes dobles y enteros largos se utilizan dos huecos de la pila.

Los huecos de la pila no están relacionados con un tipo de datos, es decir, en algún punto un hueco podría contener un valor entero y en otro, el mismo hueco podría contener una referencia a un objeto. Sin embargo, no se puede almacenar un entero en un hueco y luego recuperarlo reinterpretándolo como si fuera una referencia a un objeto. Aún más, en cualquier punto del programa, el contenido de cada hueco está asociado con un único tipo de datos que puede ser determinado usando un flujo estático de datos. El tipo de datos podría ser “no

asignado", con lo cual no se permite leer el valor del hueco. Estas restricciones son parte del modelo de seguridad de Java y se ven reforzadas por el verificador de bytecodes.

El código interpretado es generalmente más lento que un programa escrito en un lenguaje compilado, y Java no es distinto en este aspecto. Se han señalado muchas posibilidades para mejorar el rendimiento de los intérpretes. Una muy común hoy en día es incluir un compilador relativamente simple en el tiempo de ejecución de la MV. Es decir, en vez de interpretar los bytecodes del programa una y otra vez, se compilan una sola vez "al instante" en el interior de la MV, y la representación compilada de los métodos que corresponden al programa es ejecutada al efectuar una llamada. Esto es conocido como un compilador al instante (o JIT, *Just In Time*).