1.3 Hierarchies of Abstract Machines

On the basis of what we have seen, a microprogrammed computer, on which a high-level programming language is implemented, can be represented as shown in Fig. 1.7. Each level implements an abstract machine with its own language and its own functionality.

This schema can be extended to an arbitrary number of levels and a hierarchy is thus produced, even if it is not always explicit. This hierarchy is largely used in software design. In other words, hierarchies of abstract machines are often used in which every machine exploits the functionality of the level immediately below and adds new functionality of its own for the level immediately above. There are many examples of hierarchies of this type. For example, there is the simple activity of programming. When we write a program $\mathcal{P}$ in a language, $\mathcal{L}$, in essence, we are doing no more than defining a new language, $\mathcal{L}_\mathcal{P}$ (and therefore a new abstract machine) composed of the (new) functionalities that $\mathcal{P}$ provides to the user through its interface. Such a program can therefore be used by another program, which will define new functionalities and therefore a new language and so on. It can be noted that, broadly speaking, we can also speak of abstract machines when dealing with a set of commands, which, strictly speaking, do not constitute a real programming language. This is the case with a program, with the functionality of an operating system, or with the functionality of a middleware level in a computer network.

In the general case, therefore, we can imagine a hierarchy of machines $\mathcal{M}_{\mathcal{L}_0}$, $\mathcal{M}_{\mathcal{L}_1}, \ldots, \mathcal{M}_{\mathcal{L}_n}$. The generic machine, $\mathcal{M}_{\mathcal{L}_i}$ is implemented by exploiting the functionality (that is the language) of the machine immediately below ($\mathcal{M}_{\mathcal{L}_{i-1}}$). At the same time, $\mathcal{M}_{\mathcal{L}_i}$ provides its own language $\mathcal{L}_i$ to the machine above $\mathcal{M}_{\mathcal{L}_{i+1}}$, which, by exploiting that language, uses the new functionality that $\mathcal{M}_{\mathcal{L}_i}$ provides with respect to the lower levels. Often, such a hierarchy also has the task of masking lower levels. $\mathcal{M}_{\mathcal{L}_i}$ cannot directly access the resources provided by the machines below it but can only make use of whatever language $\mathcal{L}_{i-1}$ provides.

The structuring of a software system in terms of layers of abstract machines is useful for controlling the system’s complexity and, in particular, allows for a degree of independence between the various layers, in the sense that any internal modification to the functionality of a layer does not have (or should not have) any influence on the other layers. For example, if we use a high-level language, $\mathcal{L}$, which uses an operating system’s file-handling mechanisms, any modification to these mechanisms (while the interface remains the same) does not have any impact on programs written in $\mathcal{L}$.
A canonical example of a hierarchy of this kind in a context that is seemingly distant from programming languages is the hierarchy\(^5\) of communications protocols in a network of computers, such as, for example, the ISO/OSI standard.

In a context closer to the subject of this book, we can consider the example shown in Fig. 1.8.

At the lowest level, we have a hardware computer, implemented using physical electronic devices (at least, at present; in the future, the possibility of biological devices will be something that must be actively considered). Above this level, we could have the level of an abstract, microprogrammed machine. Immediately above (or directly above the hardware if the firmware level is not present), there is the abstract machine provided by the operating system which is implemented by programs written in machine language. Such a machine can be, in turn, seen as a hierarchy of many layers (kernel, memory manager, peripheral manager, file system, command-language interpreter) which implement functionalities that are progressively more remote from the physical machine: starting with the nucleus, which interacts with the hardware and manages process state changes, to the command interpreter (or shell) which allows users to interact with the operating system. In its complexity, therefore, the operating system on one hand extends the functionality of the physical machine, providing functionalities not present on the physical machine (for example, primitives that operate on files) to higher levels. On the other hand, it masks some hardware primitives (for example, primitives for handling I/O) in which the higher levels in the hierarchy have no interest in seeing directly. The abstract machine provided by the operating system forms the host machine on which a high-level programming language is implemented using the methods that we discussed in previous sections. It normally uses an intermediate machine, which, in the diagram (Fig. 1.8), is the Java Virtual machine and its bytecode language. The level provided by the abstract machine for the high-level language that we have implemented (Java

---

\(^5\)In the literature on networks, one often speaks of a stack rather than, more correctly, of a hierarchy.
Program Transformation and Partial Evaluation

In addition to “translation” of programs from one language to another, as is done by a compiler, there are numerous transformation techniques involving only one language that operate upon programs. These techniques are principally defined with the aim of improving performance. Partial evaluation is one of these techniques and consists of evaluating a program against an input so as to produce a program that is specialised with respect to this input and which is more efficient than the original program. For example, assume we have a program \( P(X, Y) \) which, after processing the data \( X \), performs operations on the data in \( Y \) depending upon the result of working on \( X \). If the data, \( X \), input to the program are always the same, we can transform this program to \( P'(Y) \), where the computations using \( X \) have already been performed (prior to runtime) and thereby obtain a faster program.

More formally, a partial evaluator for the language \( \mathcal{L} \) is a program which implements the function:

\[
\mathcal{P}_{\text{eval}}(\mathcal{L}) : (\text{Prog} \times \mathbb{D}) \rightarrow \text{Prog} \]

which has the following characteristics. Given a generic program, \( P \), written in \( \mathcal{L} \), taking two arguments, the result of partially evaluating \( P \) with respect to one of its first input \( D_1 \) is:

\[
\mathcal{P}_{\text{eval}}(P, D_1) = P'
\]

where the program \( P' \) (the result of the partial evaluation) accepts a single argument and is such that, for any input data, \( Y \), we have:

\[
\mathcal{I}_{\mathcal{L}}(P, (D_1, Y)) = \mathcal{I}_{\mathcal{L}}(P', Y)
\]

where \( \mathcal{I}_{\mathcal{L}} \) is the language interpreter.

In this case) is not normally the last level of the hierarchy. At this point, in fact, we could have one or more applications which together provide new services. For example, we can have a “web machine” level in which the functions required to process Web communications (communications protocols, HTML code display, applet running, etc.) are implemented. Above this, we might find the “Web Service” level providing the functions required to make web services interact, both in terms of interaction protocols as well as of the behaviour of the processes involved. At this level, truly new languages can be implemented that define the behaviour of so-called “business processes” based on Web services (an example is the Business Process Execution Language). Finally, at the top level, we find a specific application, in our case electronic commerce, which, while providing highly specific and restricted functionality, can also be seen in terms of a final abstract machine.