

# 1 Moderne Ausführungsmodelle

Im Folgenden werden verschiedene Ausführungsmodelle diskutiert. Ein Ausführungsmodell beschreibt, wie ein Programm aus einer Quellsprache verarbeitet werden muss, um es ausführen zu können. Der genaue Ablauf um ein Programm aus einer Quellsprache in eine ausführbare Zielsprache zu transformieren, hängt jedoch stark von der entsprechenden Quellsprache ab.

## 1.1 Ausführbarer Maschinencode

Das erste hier diskutierte Ausführungsmodell ist Maschinencode, welcher in direkt ausführbaren Dateien gespeichert wird. Programmiersprachen wie C, C++ oder auch Haskell werden normalerweise zu einer einzigen ausführbaren Datei kompiliert, die das ursprüngliche Programm in Form von plattformabhängigen Maschinencode enthält.

Dieses Modell ermöglicht ein sehr effizientes Ausführen, da ein Programm direkt gestartet werden kann, ohne auf abhängige Prozesse oder Umgebungen warten zu müssen. Zudem ermöglicht der Maschinencode die effiziente Nutzung der zu Verfügung stehenden Ressourcen auf der jeweiligen Plattform, wodurch kompilierte Programme normalerweise sehr schnell sind und wenig Arbeitsspeicher benötigen. Durch das Kompilieren wird Code mehreren Analyseschritten unterzogen, die zum einen Optimierungen ermöglichen, was sowohl Performance als auch Ressourcenverbrauch zusätzlich verbessern kann. Andererseits ist es durch diese Analyseschritte auch möglich, Programmierfehler vorab zu erkennen. So kann überprüft werden, ob die Parameteranzahl von aufgerufenen Routinen im Code korrekt ist, es kann auf falsch geschriebene oder unbenutzte Bezeichner hingewiesen werden und durch Analyse des Programmflusses ist es möglich, Rechenfehler oder illegale Indexzugriffe vorab zu erkennen.

Als Nachteil bei diesem Modell muss jedoch der verhältnismäßig hohe Aufwand bei der Codeerzeugung beachtet werden. Das Backend eines Compilers für eine kompilierte Sprache muss für jede Maschinenarchitektur angepasst werden, um die gegebenen Ressourcen und maschinenspezifischen Prozessoranweisungen auch nutzen zu können. Ein anderer Nachteil wird bei der Softwareentwicklung deutlich. Änderungen müssen immer zuerst neu kompiliert werden, sodass die Entwicklung verlangsamt wird. Zudem ist es schwerer, Fehler zu debuggen, da nur Maschinencode vorliegt, der keine der ursprünglichen Bezeichner enthält und dessen Struktur der Anweisungen sehr unterschiedlich zum Quellcode sein kann. Zudem wird die Verfügbarkeit einiger höherer Sprachfeatures eingeschränkt. Beispielsweise wird die Introspektion ohne eine Laufzeitumgebung nur dadurch möglich, dass zu allen Instanzen bereits zur Kompilierzeit Typinformationen abgelegt werden. In C++ können für diesen Zweck Präprozessoranweisungen verwendet werden, mit deren Hilfe der Programmierer im Quellcode die benötigten Zusatzinformationen über Typen und Namen der Instanzvariablen selbst in entsprechende Strukturen eintragen kann.

## 1.2 Interpretierter Quellcode

Ein weiteres Ausführungsmodell ist das Interpretieren von Quellcode durch einen Interpreter. Beispiele für interpretierte Sprachen sind Python, JavaScript und einige Lisp Dialekte. Bei interpretierten Sprachen wird der Quellcode direkt geladen und von einem Interpreter ausgeführt, der einzelne Anweisungen und Ausdrücke erkennt und mit den dazugehörigen Aktionen verbindet.

Dies hat den Vorteil, dass interpretierte Sprachen unabhängig von der zugrundeliegenden Maschine ausgeführt werden können, solange es einen passenden Interpreter gibt. Zudem kann der Interpreter anfallende Hintergrundaufgaben wie Speicherverwaltung automatisch übernehmen, was das Programmieren angenehmer machen kann. Auch die Introspektion wird deutlich vereinfacht, da alle Informationen über Struktur des Programmes direkt zur Laufzeit vom Interpreter erfasst und verwaltet werden können. Python bietet beispielsweise eine `type` Funktion an, die zu jeder Laufzeitinstanz den zugrundeliegenden Typen anzeigen kann. Diese Zusatzinformationen zur Laufzeit sowie das Vorliegen des Quellcodes zu jeder ausgeführten Aktion erleichtert das Debuggen von interpretierten Programmen natürlich enorm. Da zudem der Schritt des Kompilierens wegfällt, wird die Softwareentwicklung deutlich beschleunigt und vereinfacht.

Die Nachteile dieses Modells gehen jedoch direkt aus den vorhin genannten Vorteilen hervor. Mit dem fehlenden Kompiliervorgang fällt die komplette Codeanalyse weg, die mit dem Kompilieren einhergeht. Dadurch fallen Programmierfehler wie fehlerhafte Aufrufe oder Bezeichner erst zur Laufzeit auf, entweder wenn das Programm geladen wird, oder auch erst zu dem Zeitpunkt wenn die fehlerhafte Anweisung ausgeführt werden soll. Dadurch werden interpretierte Programme häufig fehleranfälliger. Zudem fehlen auch die Optimierungsschritte, die beim Kompilieren angewandt werden, wodurch es zu Performanceeinbußen kommt. Diese können zwar durch Just-in-time-Kompilierung teilweise zur Laufzeit nachgeholt werden, jedoch benötigt der Kompiliervorgang zur Laufzeit sowie der Interpreter selbst einige Ressourcen, sodass Interpretierte Sprachen oft langsamer sind und mehr Arbeitsspeicher benötigen.

## 1.3 Virtuelle Maschinen und Bytecode

Das letzte Ausführungsmodell ist die Ausführung von Bytecode durch eine virtuelle Maschine. Dieses Modell wird von Programmiersprachen wie Java, C# oder Erlang verwendet.

In diesem Modell wird der Quellcode auch vorzeitig kompiliert. Jedoch nicht in Maschinencode sondern in speziellen Bytecode, der von einer virtuellen Maschine ausgeführt werden kann. Dies soll die Vorteile aus beiden vorher genannten Modellen verbinden.

Durch den vorangehenden Kompiliervorgang wird der Quellcode analysiert und Fehler können vor der Laufzeit entdeckt werden. Auch die Möglichkeit zur Codeoptimierung besteht während dieses Vorganges. Der erzeugte Bytecode ist auf die virtuelle Maschine

zugeschnitten und kann schnell in entsprechende Maschinenbefehle umgewandelt werden, ohne dass eine direkte abhängigkeit zur unterliegenden Architektur bestehen. Zusätzlich besteht die Möglichkeit zur Just-in-time-Kompilierung des Bytecodes, wodurch die Geschwindigkeit des ausgeführten Codes noch weiter verbessert werden kann. Durch die virtuelle Maschine ist weiterhin eine Laufzeitumgebung gegeben, die ähnlich wie beim interpretierten Code die Speicherverwaltung übernehmen kann und zudem Informationen zu Laufzeitkonstrukten und Instanzen verwaltet, wodurch Introspektion ermöglicht wird.

Als Nachteil ist der Overhead zu sehen, der durch die komplexe virtuelle Maschine erzeugt wird. Um die Laufzeitumgebung der virtuellen Maschine bereitzustellen, muss Arbeitsspeicher und Rechenleistung zusätzlich zum eigentlichen Programm verwendet werden. Zudem muss die komplette virtuelle Maschine zu Programmstart initialisiert werden, was besonders bei kleineren Programmen einen Großteil der benötigten Rechenzeit ausmacht.

## 2 Auswahl der Programmiersprachen

Um einen Vergleich zwischen den vorher genannten Ausführungsmodellen möglich zu machen, müssen nun Programmiersprachen zu jedem Modell gewählt werden, die mit vergleichbaren Konstrukten umgehen können und ähnliche Unterstützung für Features geben. Zusätzlich zu Ausführungsmodellen gibt es verschiedene Paradigmen, die unterschiedliche Arten der Problemlösung mit sich führen. Eine prozedurale Sprache wird Sprachkonstrukte für Schleifen bieten, mit denen über Datenstrukturen verarbeitet werden können, während eine funktionale Sprache für solche Aufgaben Pattern-Matching und Rekursion bieten wird. Auch die Auswahl an verschiedenen Datentypen muss berücksichtigt werden. In den meisten Programmiersprachen gibt es für Ganzzahlen Datentypen in verschiedenen Größen, wobei der Rechenaufwand bei Operationen auf 64-Bit Zahlen größer und daher nicht direkt vergleichbar mit dem auf 8-Bit Zahlen ist. Bei Gleitkommazahlen tritt eine ähnliche Problematik auf.

All dies muss Berücksichtigt werden bei der folgenden Wahl der Programmiersprachen zu jedem Ausführungsmodell.

### 2.1 Haskell

Haskell ist eine rein funktionale Sprache, welche in mehreren Schritten zu nativem Maschinencode kompiliert wird. Während des Kompilierens wird Haskell's starkes Typsystem für die Analyse verwendet.

Der ausdrucksstarke Syntax in Haskell macht es einfach, mathematische Ausdrücke und Funktionen im Code darzustellen. Dieser mathematische Schwerpunkt der Programmiersprache wird durch einen ganzzahligen Datentypen unterstützt, dessen Größe von der

Architektur der Maschine abhängt, auf der das Programm ausgeführt wird. In dieser Arbeit wird eine 64-Bit Architektur verwendet, wodurch die Größe des Zahlentyps auch 64-Bit beträgt. Kommazahlen werden standardmäßig auch als 64-Bit Gleitkommazahl dargestellt.

Da Haskell eine kompilierte Sprache ist, die für das Auswerten von mathematischen Operationen optimiert ist, wird der kompilierte Programmcode erwartungsmäßig schnell sein. Jedoch wertet Haskell jeden Ausdruck verzögert aus, was zwar als mächtiges Werkzeug beim Programmieren verwendet werden kann, jedoch auch Overhead bei der Auswertung und beim Ausführen eines Programmes erzeugt, wodurch die Performance beeinträchtigt wird.

## 2.2 Java

Basierend auf VM Allrounder/Vielseitige Sprache (Enterprise) Objektorientiert

long (64-Bit Integer) double (64-Bit Gleitkommazahl) Lazy Listen als Iterator/Stream (LongStream.range)

## 2.3 Python

Interpretiert Allrounder, Schwerpunkt Mathematik/Statistik Objektorientiert / Skriptsprache

Ein int Typ. Unbounded aber optimiert innerhalb  $2^{64}$  double als standard C-double (64-Gleitkommazahl) range für lazy int reihen

## 3 Fibonacci (rekursiv)

Pseudocode einfügen (mathematische Gleichung) Laufzeitanalyse  $O(2^n)$

Code Python, Java, Haskell

Bytecode / Compilierten Code => Stackaufbau, wenig Berechnung

Fibonacci anders

## 4 Fibonacci (iterativ)

## 5 Notizen