1 Wahl der Sprachen

Im Folgenden wird die Wahl der Programmiersprachen diskutiert, die in diesem Dokument verglichen werden. Hierbei wird versucht, ein möglichst breites Spektrum an verschiedenen Ausführungsmodellen abzudecken.

Es gibt verschiedene Ausführungsmodelle, die es ermöglichen ein Programm auszuführen. Jedes hat dabei eigene Vor- und Nachteile.

1.1 Ausführbarer Maschinencode

Das erste Ausführungsmodell ist Maschinencode in direkt ausführbaren Dateien. Programmiersprachen wie C, C++ oder auch Haskell werden normalerweise direkt zu ausführbaren Dateien kompiliert.

Dieses Modell ermöglicht ein sehr effizientes Ausführen, da ein Programm direkt gestartet werden kann, ohne auf abhängige Prozesse oder Umgebungen zu warten. Zudem ermöglicht der architekturnahe Maschinencode die effiziente Nutzung der zu Verfügung stehenden Ressourcen, wodurch direkt 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 verbesern 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 wie die Division durch 0 oder illegale Arrayzugriffe 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 maschinenespezifischen Prozessoranweisungen auch nutzen zu können. Ein anderer Nachteil wird bei der Softwareentwicklung deutlich. Bei Änderungen muss immer zuerst neu kompiliert werden, sodass die Entwicklung verlangsamt wird. Zudem ist es schwerer, Fehler zu debuggen, da nur Maschinencode vorliegt, dessen Struktur sehr unterschiedlich zum ursprünglichen 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 Maschinen. 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.

1.4 Haskell

Haskell ist eine kompilierte Sprache, welche in mehreren Schritten zu nativem Maschinencode kompiliert wird. Da Haskell stark typisiert ist, wird in den ersten Analyseschritten versucht, möglichst viel Information zu dem unterliegenden Quellcode zu erfassen. Dies geschieht entweder durch die direkte Analyse des Quellcodes oder durch Typinferenz, bei unbekannten oder nicht definierten Typen. Nachdem die Analyse abgeschlossen ist, muss der Haskell-Quellcode transformiert werden, um anschließend Maschinencode erzeugen zu können. Da Haskell eine funktionale Sprache ist, sind diese Transformationen komplexer als bei einfachen, imperativen Sprachen.

Zunächst wird der Code vereinfacht. Dabei werden sämtliche syntaktischen Erweiterungen in eine Basisform gebracht und die Komplexität des Codes reduziert.

Compilierter Code Mathematische Sprache Funktionale Sprache

Integer (Systemstandard 64-Bit) Double (64-Bit Gleitkommazahl) Lazy Listen

1.5 Java

Basierend auf VM Allrounder/Vielseitige Sprache (Enterprise) Objektorientiert

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

1.6 Python

Interpretiert Allrounder, Schwerpunkt Mathematik/Statistik Objektorientiert / Skriptsprache

Ein int Typ. Unbounded aber optimiert innerhalb 2ĥ4 double als standard C-double (64-Gleitkommazahl) range für lazy int reihen

2 Fibonacci (rekursiv)

Pseudocode einfügen (mathematische Gleichung) Laufzeitanalyse O(2n̂)

Code Python, Java, Haskell

Bytecode / Compilierten Code => Stackaufbau, wenig Berechnung

Fibonacci anders

3 Fibonacci (iterativ)

4 Notizen