THM | CAMPUS GIESSEN | MNI
Mathematik, Naturwissenschaften und Informatik
TECHNISCHE HOCHSCHULE MITTELHESSEN

Manusskript

Rechenzeitvergleich verschiedener Ausführungsmodelle

Kurs „Wissenschaftliches Arbeiten in der Informatik“

Niklas Deworetzki

Aktuelles Datum

Zusammenfassung

Inhaltsverzeichnis

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 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 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 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.

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 verlgeichbar 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 Haskells starkes Typsystem für die Analyse verwendet.

Der ausdruckstarke Syntax in Haskell macht es einfach, mathematische Ausdrücke und Funktionen im Code darzustellen. Diese 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. Haskell verfügt zudem über eine automatische Garbage-Collection, welche die Ausführung für kurze Zeit unterbricht, wenn Speicherplatz benötigt wird. Der Garbage-Collector ist jedoch sehr simpel gehalten und kann sehr effizient arbeiten, da Haskell nur unveränderliche Datentypen unterstützt.

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 Ausdrück 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

Java als Programmiersprache ist vielseitig verwendbar, objektorientiert und mit dem Ziel entwickelt worden, einmal kompilierten Java-Code überall ausführen zu können. Der Java-Kompiler produziert dabei speziellen Bytecode, der für die Java Virtual Machine (JVM) optimiert und plattformunabhängig ist. Dieser Bytecode kann dann auf jeder beliebigen Plattform ausgeführt werden, solange es eine Implementierung der JVM für diese gibt.

Java verfügt über eine breite Spanne an primitiven Datentypen, die verwendet werden, um Zahlen darzustellen. Für Ganzzahlen gibt es Datentypen deren Größe von 8-Bit bis hin zu 64-Bit reicht. Kommazahlen werden als Gleitkommazahlen mit einer Größe von 32 oder 64-Bit dargestellt. Für die Garbage-Collection besitzt die JVM eine ganze Reihe an verschiedener Algorithmen, die stetig weiterentwickelt werden und dem Verwendungszweck angepasst werden können. Standardmäßig wird ein paralleler Garbage-Collector verwendet, welcher in einem eigenen Thread arbeitet, sodass der normale Programmablauf nicht unterbrochen wird.

Obwohl Java nicht direkt in Maschinencode übersetzt wird, ist Java ein weiterer Kandidat für sehr schnelle Programme, da der Bytecode sowie die JVM auf der dieser ausgeführt wird, stetig optimiert werden. Zudem verfügt die JVM über einen komplexen Just-in-Time Kompiliermechanismus, welcher zur Laufzeit die ausgeführten Anweisungen nicht nur in nativen Code umwandeln kann, sondern zusätzlich noch laufzeitabhängige Optimierungen durchführt.

2.3 Python

Python ist eine universelle Programmierspache, welche interpretiert ausgeführt wird und durch ein hohes Abstraktionslevel den Anspruch hat, sauberen und gut lesbaren Code

zu fördern. In Python geschriebener Quellcode wird direkt ohne weitere Analyse- oder Kompilierschritte von einem Interpreter ausgeführt.

Häufige Anwendungen für Python liegen im Bereich der Wissenschaft und Datenanalyse. Zur Darstellung von ganzen Zahlen besitzt Python einen Typen, der Ganzzahlen ohne Einschränkung der Reichweite speichern und verarbeiten kann. Da solch unbegrenzte Zahlentypen komplexe Berechnungen für Rechenoperationen benötigen ist dieser Zahlentyp für die Reichweite von 64-Bit optimiert, sodass mit Zahlen, die als eine 64-Bit Ganzzahl dargestellt werden können, auch als solche behandelt werden, was die Performance in diesen Zahlenräumen verbessert. Kommazahlen werden als normale 64-Bit Gleitkommazahlen in C dargestellt. Pythons Garbage-Collector basiert auf einem Referenzzähler, der jedoch Zyklen erkennen muss, um die Garbage-Collection korrekt durchführen zu können. Die Garbage-Collection in Python wird ausgeführt, wenn ein bestimmter Schwellenwert für den Speicherverbrauch überschritten wird.

Da Python nur interpretiert ausgeführt wird, wird die Rechenzeit für Programme in Python vermutlich am höchsten sein. Zwar gibt es teilweise Just-in-Time Kompilierung für Python Interpreter, jedoch ist diese keineswegs so ausgereift wie bei Java und auch nicht so effizient.

3 Auswahlkriterien für ein Testprogramm

Aus dem vorherigen Vergleich wird deutlich, dass alle gewählten Sprachen über einen ganzzahligen Datentypen verfügen, der eine Größe von 64-Bit besitzt oder zumindest für diese Größe optimiert ist. Zudem besitzen alle Sprachen über eine automatische Garbage-Collection, wodurch eventuelle Geschwindigkeitsvorteile oder auch Nachteile durch manuelle Speicherverwaltung wegfallen.

Diese Tatsachen führen zu der Überlegung, dass mathematische Problemstellungen als gute Grundlage für einen Vergleich der Rechenzeiten dienen, da die mathematischen Formeln und Aussagen direkt in die einzelnen Programmiersprachen übertragen werden können, ohne auf große Unterschiede in den Kapazitäten der einzelnen Sprachen zu stoßen.

Es muss lediglich darauf geachtet werden, dass bei den mathematischen Berechnungen der Zahlenraum nicht überschritten wird, der durch die 64-Bit Ganzzahlen oder Gleitkommazahlen festgelegt wird.

3.1 Fibonacci

Eine mathematische Folge, die sämtliche oben genannten Kriterien erfüllt, ist die Folge der Fibonacci-Zahlen. Eine Zahl aus dieser Folge lässt sich mit folgender Gleichung bestimmen:

$$fib_n = \begin{cases} fib_1 = 1 \\ fib_2 = 1 \\ fib_n = fib_{n-1} + fib_{n-2} \end{cases}$$

Abbildung 1: Fibonacci-Reihe rekursiv definiert in allgemeiner Form

Die Laufzeitkomplexität dieser rekursiven Funktion liegt in $O(2^n)$ und lässt sich sogar genauer als $\Theta(fib_n)$ ausdrücken.

Zur Vereinfachung einer Implementierung kann eine Funktion mit Definitionsbereich $\mathbb{N}^*$ angenommen werden. Ohne negative Parameter lässt sich die Funktion nur noch mit zwei Funktionsfällen schreiben:

$$fib(n) = \begin{cases} n \leq 2 = 1 \\ n = fib(n-1) + fib(n-2) \end{cases}$$

Abbildung 2: Fibonacci-Reihe als Funktion mit $\mathbb{N}^*$ als Definitionsbereich

Diese Umformung verändert die Laufzeitkomplexität der Funktion nicht, da sowohl der Rekursionsschritt unverändert bleibt als auch die Anzahl und Bedingung der Basisfälle.

In der Funktion treten nur Ganzzahlen und simple Addition beziehungsweise Subtraktion zur Berechnung auf, welche direkt in die einzelnen Programmiersprachen übertragen werden können.

Zu Zeigen ist nur noch, dass diese Funktion den Wertebereich einer 64-Bit Zahl nicht überschreitet, wodurch Unterschiede in den gewählten Programmiersprachen auftreten würden. Die erste Zahl, die diesen Bereich überschreitet ist $fib_{93} = fib(93) = 12.200.160.415.121.876.738$. Aufgrund der Laufzeitkomplexität der Funktion würde die Berechnung dieses Wertes einige Jahre dauern. Daher kann angenommen werden, dass in diesem Rahmen die Einschränkungen der Ganzzahlen nicht überschritten wird.

4 Implementierungen in den Programmiersprachen

Für die Implementierung der Funktion fib gilt allgemein, dass in der Programmiersprache eine neue Funktion deklariert werden muss, welche von einer Ganzzahl auf eine andere Ganzzahl abbildet.

Im nächsten Schritt muss differenziert werden, welcher Funktionsfall eintritt. Da in der reduzierten Form nur zwei Funktionsfälle definiert sind, genügt hier in einer Abfrage zu überprüfen, ob das Argument der Funktion kleiner oder gleich zwei ist. Ist dies der Fall, so ist der Rückgabewert der Funktion die Konstante 1. Ansonsten müssen zwei rekursive Aufrufe der Funktion gemacht werden, wobei einmal das Argument der Funktion um eins beziehungsweise um zwei reduziert werden muss als Parameter für die rekursiven Aufrufe. Der Einfachheit halber wird diese Abfrage als einzelner Ausdruck geschrieben.

4.1 Haskell

In Haskell kann die Definition mit Abfrage und rekursivem Aufruf direkt übernommen werden. Es muss jedoch die Signatur für die Funktion angegeben werden, um den Typen auf `Int` zu beschränken, da sonst per Typinferenz der unbegrenzte Ganzzahlentyp `Integer` inferiert wird.

Da mit `if` in Haskell auch ein Ausdruck und keine Anweisung eingeleitet wird, kann in dieser Sprache sogar der normale `if`-Syntax verwendet werden, um die Berechnungen des Funktionsfalles als Ausdruck darzustellen.

```
fib :: Int -> Int
fib n = if (n <= 2) then 1 else fib (n-1) + fib (n-2)
```

Abbildung 3: Fibonacci-Serie als Funktion in Haskell

Zwar werden die Argumente der Funktion in Haskell verzögert ausgewertet. Jedoch wird die Auswertung in der Abfrage zur Entscheidung des Funktionsfalles erzwungen, wodurch die Eigenheiten der verzögerten Auswertung hier vernachlässigt werden könne.

4.2 Java

Auch in Java kann die Definition mit Abfrage und rekursivem Aufruf einfach übernommen werden. Bei der Definition der Funktion wird noch der primitive Datentyp `int` sowohl für das Argument als auch für den Rückgabewert der Funktion festgelegt. Zudem ist es in Java sinnvoll, die Funktion als `static` zu deklarieren, da sie so unabhängig von einer Objektinstanz verwendet werden kann.

In Java muss der ternäre Operator verwendet werden, um die Funktionsfälle als einen Ausdruck darzustellen. Die Verwendung von `if` würde zwei verschiedene Anweisungen mit eigenen Ausdrücken erstellen.

```
static long fib(long n) {
  return (n <= 2) ? 1 : fib(n-1) + fib(n-2);
}
```

Abbildung 4: Fibonacci-Serie als Funktion in Java

4.3 Python

Wie zuvor auch wird die Definition mit Abfrage und rekursivem Aufruf direkt übernommen. Eine Angabe von Typen ist in Python nicht nötig, da diese zur Laufzeit bestimmt werden.

Aus selbem Grund wie in Java wird auch hier keine `if`-Anweisung verwendet zur Auswertung der Funktionsfälle.

```
def fib(n):
return 1 if n <= 2 else fib(n-1) + fib(n-2)
```

Abbildung 5: Fibonacci-Serie als Funktion in Python

5 Messergebnisse der Rechenzeiten

Aus den definierten Funktionen in den konkreten Programmiersprachen kann nun die Rechenzeit bestimmt werden. Dazu werden sogenannte Mikrobenchmarks durchgeführt, welche die Zeit messen, die für einen Funktionsaufruf benötigt wird. Die Verwendung von Mikrobenchmarks bewirkt, dass die Zeit, die für das Laden und Einrichten eines Programmes benötigt wird, vernachlässigt werden kann und nur die reine Zeit für die Berechnung als Messwert entsteht.

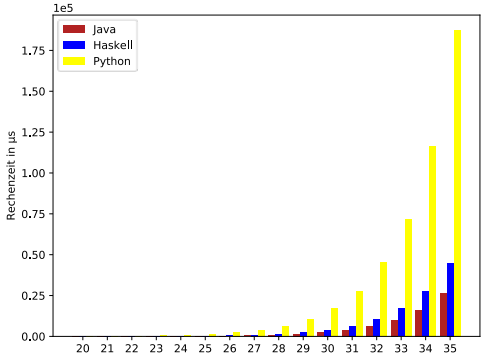

Abbildung 6: Rechenzeiten des Mikrobenchmarks (linear)

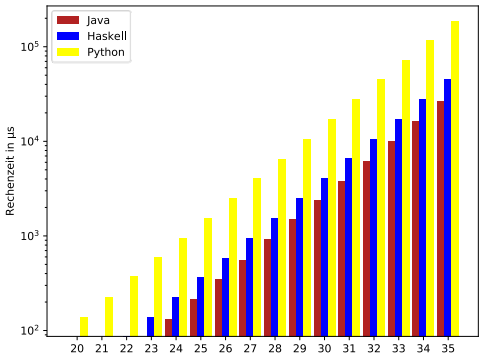

Abbildung 7: Rechenzeiten des Mikrobenchmarks (log)

Wie in Abbildung 6 zu sehen ist, benötigt der interpretierte Python-Code die meiste Rechenzeit. Schneller als der Python-Code ist der kompilierte Maschinencode von Haskell, welcher dennoch etwas langsamer ist, als der Bytecode der auf der JVM ausgeführt wird. Zudem wird deutlich, dass die Rechenzeiten als Exponentialfunktion dargestellt werden können, was der theoretischen Laufzeitkomplexität des Algorithmus entspricht.

Auf einer logarithmischen Skala wird eine Exponentialfunktion der Form $f(x) = ae^{bx}$ auf eine lineare Funktion der Form $f'(x) = log(a) + bx * log(e)$ abgebildet. Der Faktor b ist mit $b = 1$ anzunehmen, da dieser die Anzahl der rekursiven Aufrufe bestimmt, welche bei jeder Implementierung identisch ist. Die Basis e ist konstant und kann daher beim Vergleichen der Rechenzeiten vernachlässigt werden. Also bleibt die Darstellung der Rechenzeiten in der Form $f'(x) = log(a) + x$. Dadurch wird deutlich, dass die konstanten Abstände in der rechten Abbildung zwischen den Rechenzeiten durch den X-Achsenabschnitt $log(a)$ - also den Faktor a - verursacht werden.

In der Exponentialdarstellung ist der Faktor a Teil der Basis, also ein konstanter Faktor, der bei jedem Funktionsaufruf auftritt. Dies beweist also, dass konstante Geschwindigkeitsunterschiede beim ausführen der Rechenoperationen Ursache der unterschiedlichen Rechenzeiten sind.

Die tatsächliche Laufzeit eines Programms hängt jedoch nicht alleine von der Zeit ab die für die ausgeführten Funktionen benötigt wird. Das Laden und Einrichten der Laufzeitumgebung nach Programmstart, das Laden von Code und Daten sowie die Verwaltung von Speicher ist auch Teil der Laufzeit eines Programms.

Um den Einfluss der anderen Faktoren zu bestimmen, werden die Funktionen aus den vorherigen Kapiteln in ein eigenständiges Programm verpackt, welches eine Zahl als Programmparameter akzeptiert und diese als Argument für den Aufruf der Funktion fib verwendet wird. Zwar gibt es leichte Unterschiede in der Art, wie die gewählten Programmiersprachen, die Zeichenketten behandeln, über welche auf die Programmparameter zugegriffen werden kann. Jedoch kann der Einfluss einer einzigen Umwandlung

zu Programmstart angesichts der mehreren tausend Operationen während der Berechnung vernachlässigt werden.

Wird nun die Laufzeit für die Programme für die selben Eingabezahlen wie bei den Mikrobenchmarks gemessen, entstehen vergleichbare Messwerte.

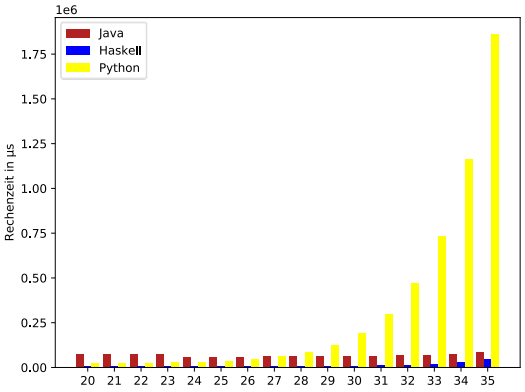

Abbildung 8: Rechenzeiten des Benchmarks (linear)

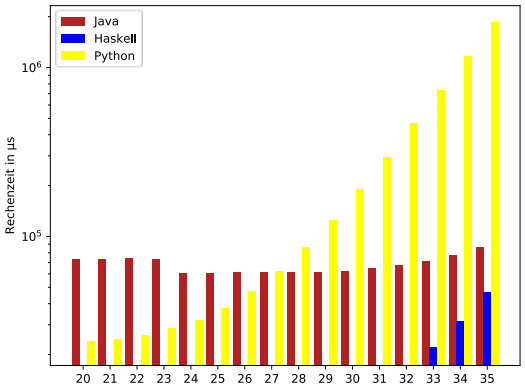

Abbildung 9: Rechenzeiten des Benchmarks (logarithmisch)

Sofort fallen Unterschiede in den Verläufen der Rechenzeiten zu den einzelnen Funktionen auf. Die Rechenzeit des interpretierten Python-Codes zeigt wieder einen exponentiellen Verlauf. Auch die Rechenzeit des kompilierten Haskell-Codes deutet exponentiellen Wachstum an, welcher jedoch kaum erkennbar ist, da der interpretierte Code deutlich langsamer ist. Die Rechenzeit des Java-Codes wirkt nahezu konstant und schwankt leicht um einen Wert. Ein Anstieg der Rechenzeit ist hier kaum zu erkennen.