Logisch-Funktionale Sprachen im Vergleich

Fakultät II für Informatik, Wirtschafts- und Rechtswissenschaften
Universität Oldenburg

von

Müllner, Nils

Betreuerin:

PD. Dr. E. Wilkeit

Tag der Anmeldung: 04. Juni 2006
Tag der Abgabe: 28. Juli 2006

Abstract

What this work is:
This work gives an insight into the small field of logical functional languages. While ALF, Curry, Erlang, Leda, Mercury and Oz/Mozart are mentioned, only Curry, Erlang and Mercury are eyed more precisely, by comparing

the effort to learn a language (Section 4.1),
the presumable future perspective (Section 4.2) and
the field of application (Section 4.3).

What this work is not:
This work is not detailed on all logical functional languages. It focuses Curry, Erlang and Mercury.
The implementation used in Section 4.4 does not work in real life. It is currently impossible to break an AES-128-CBC-Cipher by using brute-force-algorithms. The algorithms used in this work are used to measure computing-times.

This work is divided into four parts. Chapter 1 contains the motivation.
Chapter 2 gives an orientation what the paradigms logical and functional mean and why I focus on three languages.
In Chapter 3 the technologies used for this work will be in the focus. Because none of the languages is commonly known, characteristic features are discussed, which would otherwise be kept barred if one is not in touch with the applied technology.
In Chapter 4 the comparison takes place with the criteria mentioned above. Also, a benchmark has been implemented in Erlang and Java to compare the performance of the promising logical-functional sector against one standard language which is widespreaded and well known. It has not been implemented in Curry and Mercury for reasons which will be discussed in this thesis. The source code used for this work is available on the annexed CD:

- erlang/bench/bench4.erl (SingleCPU Benchmark Erlang)
- /erlang/distributed_bench/final.erl (ClusterCPU Benchmark Erlang)
- /java/bench/bench1.java (SingleCPU Benchmark Java)
- /java/db3/Client.java (ClusterCPU Client Benchmark Java)
- /java/db3/MultiServer.java (ClusterCPU Server Benchmark Java)

The sources will not be discussed for they have been documented using JavaDoc and ErlDoc.
The last chapter gives a conclusion by comparing the results from the antecedent chapter. Also my personal survey is in this chapter.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Oldenburg, den 28. Juli 2006

Inhaltsverzeichnis

1. Einleitung

Programmiersprachen lassen sich auf verschiedene Weisen klassifizieren und in Gruppen einteilen. Eine Art der Unterteilung ist, die Sprachen anhand von Paradigmen zu sortieren.
Der Begriff Paradigma bedeutet Sichtweise. Bei der Programmierung ist ein Paradigma eine Sichtweise, um ein Problem zu lösen. So gibt es verschiedene Sichtweisen, die sich für unterschiedliche Probleme unterschiedlich gut eignen. Es gibt beispielsweise Programmiersprachen, mit denen sich Benutzungsoberflächen leicht in einer ansprechenden Weise realisieren lassen, wogegen andere eher auf eine effiziente Datenhaltung oder eine sichere Kommunikation ausgelegt sind.

Bei der Entwicklung einer Programmiersprache müssen die Entwickler also vorher entscheiden, für welche Art von Problemen die zu entwickelnde Sprache gedacht ist. Der Begriff `multiparadigmatische Programmiersprache` impliziert, dass es möglich ist, mehrere Paradigmen gleichzeitig zu berücksichtigen, also dem Benutzer innerhalb einer Sprache die Möglichkeit zu geben, Problemklassen in mehreren Sichtweisen behandeln zu können, ohne die Programmiersprache zu wechseln. Im Gegensatz zu der Verwendung einer multiparadigmatischen Programmiersprache steht also die Verwendung mehrerer Sprachen, die miteinander kommunizieren können und für jeweils unterschiedliche Anwendungsfelder entwickelt wurden.

Logisch-funktionale Programmiersprachen zeichnen sich dadurch aus, dass sie zwei Paradigmen, das logische und das funktionale, gleichzeitig realisieren. Ein Vorteil beider Paradigmen ist die Parallelität. Sowohl im logischen als auch im funktionalen Paradigma wird Wert darauf gelegt, Algorithmen unterteilen zu können um deren Berechnung parallel durchführen zu können. Im Gegensatz dazu ist es in imperativen Sprachen üblich, Algorithmen sequentiell abzuarbeiten. Auf die Eigenschaften, die den einzelnen Paradigmen zu Grunde liegen oder auch beiden zu Eigen sind, wird in dieser Arbeit immer wieder an geeigneter Stelle eingegangen.

Das logische Paradigma hat als eine grundlegende Eigenschaft, dass ein Axiomensystem und einige Regeln durch den Benutzer an die Sprache gegeben werden. An dieses System aus Axiomen und Regeln lassen sich unabhängig von der Befehlsfolge Anfragen stellen, die das System mit Hilfe des vorhandenen Wissens beantworten kann. Ein einfaches Axiom könnte lauten:

`Der Himmel ist blau.`

Eine einfache Regel könnte lauten:

`Wenn der Himmel blau ist, dann ist das Wetter schön.`

Stelle ich nun an das System die Anfrage:

`Ist der Himmel blau?`

wird das System mit `Ja` antworten. Außerdem lassen sich aus vorhandenen Regeln und Axiomen weitere Regeln und Axiome folgern und ableiten. So kann ich auch fragen:

`Ist das Wetter schön?`.

In diesem Fall hat das System die Möglichkeit, die vorhandene Regel mit dem vorhandenen Axiom zu kombinieren und als Antwort `Ja` daraus abzuleiten.

Dabei ist es wichtig, zu beachten, dass das System nur die ihm gegebenen Axiome und Regeln kennt und nur aus diesen Wissen benutzen und ableiten kann. Damit gilt für logische Sprachen die `Annahme einer geschlossenen Welt`, welche in der Fachliteratur als `closed world assumption` bezeichnet wird.

Logische Programmiersprachen, wie beispielsweise Prolog, eignen sich also hervorragend zum Lösen einiger mathematischer Probleme und weniger zum Erstellen benutzerfreundlicher Oberflächen.

In dem funktionalen Paradigma sind Programmiersprachen zusammengefasst, die nicht wie logische Sprachen Wert auf Parallelisierbarkeit legen, also gewährleisten, dass Algorithmen parallel ausgeführt werden können und nicht sequentiell, wie in imperativen Programmiersprachen, ausgeführt werden müssen. Ein Augenmerk liegt darauf, die Komplexität der Verifizierbarkeit möglichst gering zu halten, also den Aufwand, der zum Beweisen, dass ein Programm den vorgegebenen Spezifikationen entspricht, notwendig ist, gering zu halten. Dieser Bestandteil des funktionalen Paradigmas erleichtert später zwar die Verifikation, jedoch schränkt dieser Vorteil den Anwender ein. Um den Vorteil der vereinfachten Verifikation zu gewährleisten, muss auf Schleifen, wie sie in imperativen Programmiersprachen üblich sind, verzichtet werden. Variablen dürfen in jeder Instanz einer Funktion nur einmal initialisiert werden. Soll eine Variable mit einem neuen Wert belegt werden, so muss die Funktion neu instanziiert werden, da in der neuen Instanz der Parameter noch nicht initialisiert ist.

In Kapitel 2 werden die Eigenschaften des logischen und des funktionalen Paradigmas weiter vertieft. Außerdem werden hier die sechs Sprachen, die im logisch-funktionalen Paradigma von Bedeutung sind (Kapitel 2.3), sowie die Vergleichsgebiete (Kapitel 2.5) vorgestellt. Der Vergleich beschränkt sich auf die Sprachen Curry, Erlang und Mercury. Die Fokussierung auf diese Sprachen wird in Kapitel 2.4 erläutert.

Kapitel 3 befasst sich mit den technischen Details. Dabei wird sowohl auf die in 4.4 benutzte Hardware wie auch auf benötigte Software und Betriebssysteme eingegangen. Das Kapitel 3.3 erklärt die notwendigen Schritte, um die für den Vergleich verwendeten Softwaresysteme zu installieren.

In Kapitel 4 werden Vergleiche aufgestellt, erklärt und die Ergebnisse erläutert. Dieses Kapitel ist in die folgenden vier Bereiche unterteilt:

- Ich habe mich mit der `Erlernbarkeit` (Kapitel 4.1) logisch-funktionaler Sprachen beschäftigt und vergleiche den zeitlichen Aufwand sowie das verfügbare Lernmaterial der drei relevanten Sprachen Curry, Erlang und Mercury.

- Die `Zukunftsperspektiven` (Kapitel 4.2) sind in dieser Arbeit ein subjektiver Vergleichspunkt. Da sich die Zukunft hier nicht beweisen lässt, werden an Hand von Fakten Schlüsse gezogen. Fakten sind dabei zum Beispiel die Frequenz der Releases und das Aufkommen in den Entwickler-Mailinglisten, die ein gewisses Maß an Arbeitsaufwand indizieren. Auch der Inhalt der Mailingliste ist Bestandteil der Auswertung. So lassen sich Schlüsse über bestehende Projekte, Probleme und geplante Erweiterungen ziehen. Jedoch sind auch die weiteren drei Vergleichspunkte dieser Arbeit, die Erlernbarkeit, die Anwendungsgebiete und die Performanz, von Bedeutung.

- Für die fokussierten Sprachen Curry, Erlang und Mercury werden die `Anwendungsgebiete` (Kapitel 4.3) verglichen. Diese sind auf Grund ihrer vielen Einflüsse aufschlussreich. Zum Einen bieten die Paradigmen an sich verschiedene Anwendungsgebiete. Bei einer multiparadigmatischen Sprache ist jedoch nicht unbedingt eine Schnittmenge oder Vereinigung der Anwendungsgebiete das Resultat der Entwicklung.
  Zum Anderen wurden die verschiedenen Sprachen unter verschiedenen Bedingungen mit unterschiedlichen Ansprüchen entwickelt und zielen dadurch zum Teil auf unterschiedliche Anwendungsfelder ab.

- Abschließend wird in dem Vergleich ein `Benchmark` (Kapitel 4.4) erläutert, durch den Erlang als logisch-funktionale Sprache mit Java als objekt-orientierter und weit verbreiteter Sprache verglichen wird. Dieser Benchmark bildet den Hauptvergleichspunkt und damit den Kern dieser Arbeit. Für den Benchmark wurde ein Verfahren entwickelt, das die Zeit misst, die ein Rechner benötigt, um einen Text n mal zu entschlüsseln, wobei n als Element der natürlichen Zahlen größer 0 vorgegeben ist. Dabei wurden die Chiffrier-Module beziehungsweise -Klassen aus den Standard-Bibliotheken verwendet. Einer der wichtigsten Gründe für dieses Vorgehen ist, dass es repräsentativ ist. Microsoft verwendet ein ähnliches Verfahren zur Klassifizierung von Rechnern, um eine ausreichende Performanz der Benutzungsoberfläche des neuen Betriebssystems Windows Vista zu gewährleisten [20] [26]. Zum Anderen ist es wichtig, soweit wie möglich vorhandene mitgelieferte Funktionen zu nutzen, um ein Verfremden der Ergebnisse durch eigene Programmierung minimal zu halten.
  Eine Vorgabe für Java war es, so nah wie möglich am logisch-funktionalen Paradigma zu bleiben, also auf Konstrukte wie Schleifen oder multiple instanzielle Initialisierung so weit wie möglich zu verzichten, um paradigmenbezogene Vorbeziehungsweise Nachteile möglichst auszuschließen.

2. Orientierung

Um das logisch-funktionale Paradigma zu verstehen, muss erst der Hintergrund logischer und funktionaler Sprachen erläutert werden. Dabei ist es von Bedeutung, die Entstehung der relevanten Sprachen zu betrachten. Während sich einige Sprachen in Syntax und Semantik an bereits existierenden Sprachen orientieren, haben andere eine eigene Syntax, bei der ein Einfluss von weiteren Sprachen kaum auszumachen ist. So orientiert sich Curry in seiner Syntax sehr an der funktionalen Sprache Haskell und kann auch Haskell-Quelltexte übersetzen. Mercury hat das logische Prolog zum Vorbild und kann unter Umständen sogar Prolog-Quelltext kompilieren. Bei Erlang erinnert die Syntax allenfalls entfernt an die Sprache C.

Bei der Entwicklung einer multiparadigmatischen Sprache muss also schon im Vorfeld festgelegt werden, welches Paradigma wie viel Einfluss auf die zu entwickelnde Sprache haben darf, um eine Sprache für ein bestimmtes Anwendungsgebiet zu konzipieren. Auch ist der Verlauf der Entwicklung bei der Entstehung von Sprachen teilweise sehr unterschiedlich. Bei Curry beispielsweise gibt es zwei Hauptentwicklungslinien. Während in der einen versucht wird, sich an dem Entwurf zu orientieren und alles direkt in Curry umzusetzen, versucht ein anderer Ansatz, möglichst viel vorhandenen Prolog-Quelltext zu übernehmen, um ein breiteres Spektrum anzubieten. Diese Vorgehensweise ist nicht immer entwurfskonform. Dieser Unterschied wird in Abschnitt 2.3.2 vertieft.

Auch die geographische Verteilung der Forschungseinrichtungen bei der Entwicklung eines Sprachfeldes ist von Bedeutung. Eine zentrale Rolle gerade für die logisch-funktionalen Sprachen Curry, Erlang und Oz/Mozart spielt das Swedish Institute of Computer Science (SICS). Zum Einen arbeitet der ehemalige Entwicklungsleiter und geistige Vater von Erlang, Joe Armstrong, an diesem Institut. Zum Anderen basiert eine Distribution von Curry auf SICStus Prolog, einer kommerziellen Prolog-Distribution des SICS. Eine weitere logisch-funktionale Sprache, die unter dem Namen Oz entstand und später in den USA als Mozart weiterentwickelt worden ist, wurde Ende der neunziger Jahre bis 2004 am SICS entwickelt.

Neben dieser zentralen Bedeutung einer Institution für das Feld der logisch-funktionalen Sprachen ist die dezentrale Entwicklung gerade für Curry von Bedeutung. Das Portland-Aachen-Kiel-Curry-System (PAKCS) wird global entwickelt. Dies wird

besonders deutlich beim Lesen der Mailingliste oder der Teilnehmerlisten entsprechender Kongresse (http://www.informatik.uni-kiel.de/~mh/wcflp2005/). Auch in der Erlang-Mailingliste ist die Teilname global. Lediglich in der Mercury-Mailingliste scheint die Teilnahme auf Australien, der Heimat von Mercury, beschränkt zu sein.

Ein Grund, warum ich mich für eine überschaubare Sprachklasse entschieden habe, ist die Vergleichbarkeit. Laut Wikipedia.com gibt es 361 verschiedene Programmiersprachen bis jetzt (http://en.wikipedia.org/wiki/Category:Programming_languages). Dabei sind verschiedene Dialekte nicht mitgezählt. Um die Menge aller Programmiersprachen übersichtlicher zu gestalten, lassen sich Sprachen nach Paradigmen unterteilen.

Die Klassifizierung einzelner Sprachen nach Paradigmen ist hilfreich, um sich bei einer unbekannten Sprache etwas an Hand ihres Paradigmas beziehungsweise ihrer Paradigmen vorstellen zu können. Jedoch gibt es teilweise große Unterschiede in der Klassifizierung. So wird Erlang in einigen Quellen nur als funktional, Mercury hingegen oft nur als logisch bezeichnet. Beide sind jedoch im logisch-funktionalen Paradigma. Dies hängt mit dem Ursprung zusammen. Erlang entstand als funktionale Sprache und erfüllt mittlerweile auch die Anforderungen logischer Programmierung. Mercury wurde als logische Sprache entwickelt und erfüllt jetzt auch funktionale Kriterien. Es ist also möglich, diese Sprachen exemplarisch für ein Paradigma anzuführen und zeitweise den multiparadigmatischen Charakter zu vernachlässigen. Eine klare Abgrenzung zwischen den Paradigmen ist nicht immer vorhanden und einige Grenzen sind noch nicht definiert.

Auch innerhalb der Paradigmen gibt es unter Entwicklern Unstimmigkeiten. Im funktionalen Paradigma beispielsweise verfasste Amr Sabry, der den Begriff der `reinen funktionalen Sprache` erst definieren musste [33], eine Arbeit, die Standards setzte und erklärte.

In der vorliegenden Arbeit werden keine Standards definiert wie in der Arbeit von Sabry. Jedoch wird auf unklare Grenzen hingewiesen, die aus ungenauen Definitionen resultieren.

Um im Folgenden die Unterschiede zwischen den Paradigmen zu verdeutlichen, gehe ich auf die Arbeitsweise einzelner Sprachen ein. Diese wird in dem folgenden Bild veranschaulicht. Dabei wird die Vorgehensweise der Sprachen behandelt, wie Quelltext in Maschinenesprache übersetzt wird. Bei allen Sprachen des logisch-funktionalen Paradigmas handelt es sich um höhere Programmiersprachen, also `high level programming languages`. Für diese Sprachen ist es typisch, nicht selbst gleich in Maschinensprache zu übersetzen, sondern andere Sprachen oder virtuelle Maschinen als Zwischenschritt zu nutzen.

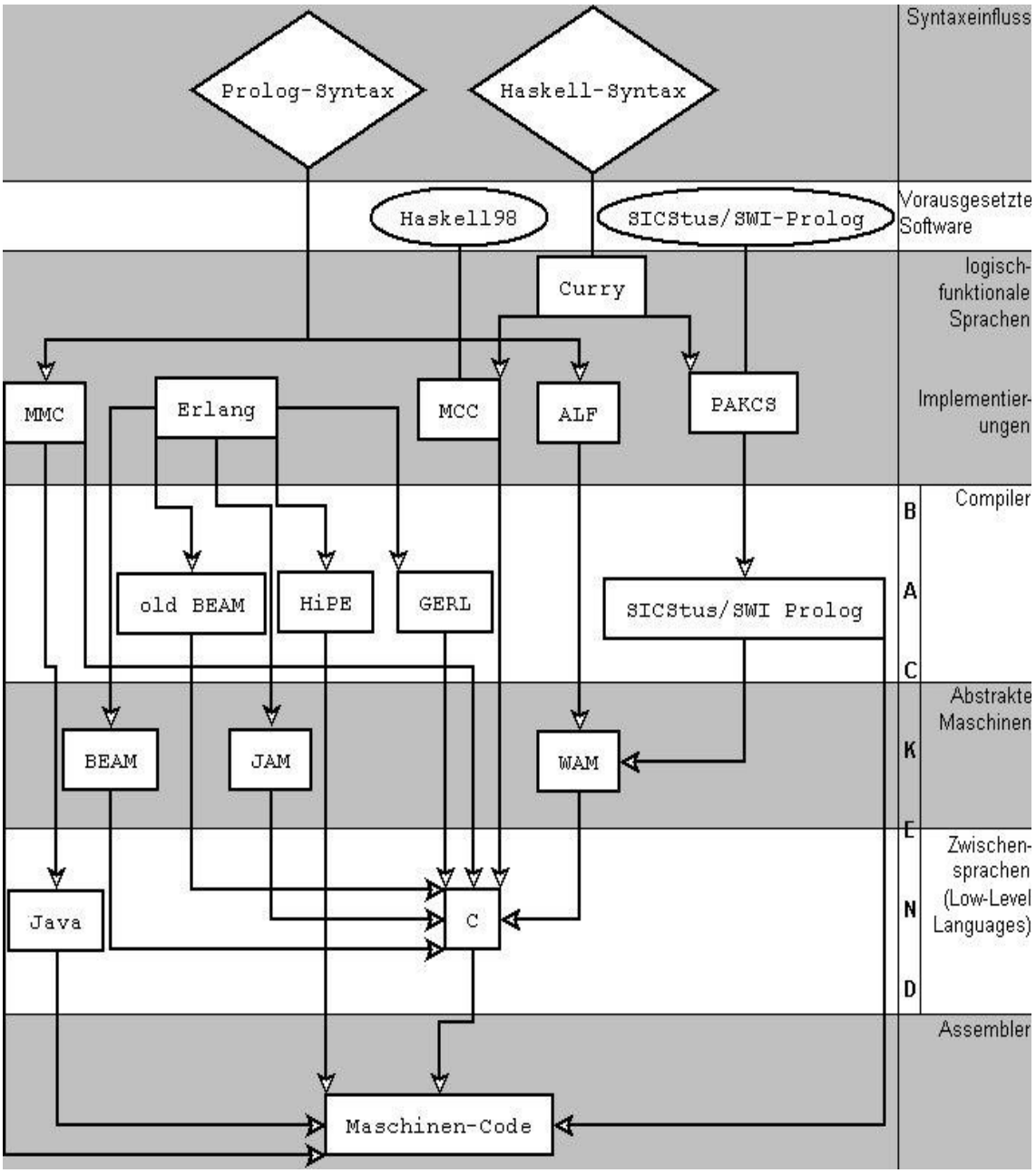

Abbildung 2.1: Die oberen zwei Schichten des Diagrams geben den Einfluss der Syntax anderer Sprachen auf die entsprechende logisch-funktionale Sprache wieder, sowie Voraussetzungen, welche Softwaresysteme für eine Sprache notwendig sind. In diesem Bild ist zu sehen, wie einzelne High-Level-Sprachen vorgehen, um Maschinen-Code zu erzeugen. Dabei lassen sich im Wesentlichen vier Arten unterscheiden:
Zum Einen ist es möglich, in eine Zwischensprache zu übersetzen, welche dann in Maschinen-Code übersetzt, wie beispielsweise MCC => C => Maschinen-Code. In diesem Fall spricht man von `kompilierten Sprachen`.
Eine weitere Möglichkeit ist, den Quelltext mit Hilfe einer abstrakten Maschine zu übersetzen, welche dann eine Zwischensprache verwendet, wie beispielsweise Erlang => BEAM => C => Maschinen-Code. Hier spricht man von `interpretierten Sprachen`.
Eine dritte Möglichkeit ist, zuerst zu kompilieren und anschließend in einer abstrakten Maschine zu interpretieren, wie zum Beispiel bei PAKCS => SICStus/SWI Prolog => WAM => C => Maschinen-Code.
Die vierte Möglichkeit sieht vor, Quelltext ohne Umwege für die Architektur zu übersetzen, wie beispielsweise Erlang => HiPE => Maschinen-Code.
Dieses Diagram erhebt keinen Anspruch auf Vollständigkeit. Es ist zur Veranschaulichung komplexer Zusammenhänge gedacht.

2.1 Das logische Paradigma

Logische Programmiersprachen zeichnen sich dadurch aus, dass sie nicht aus einer Folge von Befehlen bestehen, wie etwa imperative Programmiersprachen. Sie basieren auf einem Axiomensystem sowie einer Menge von Regeln, die sich auf die Axiome anwenden lassen. An dieses System aus Regeln und Fakten können Anfragen gestellt werden. Axiome, Regeln und Anfragen durch den Benutzer sind aus der Prädikatenlogik bekannte Hornklauseln.

Eine wichtige Eigenschaft logischer Sprachen ist die Anzahl möglicher Auswertungen eines Ausdrucks. So gibt es in der logischen Programmierung Attribute wie det, semidet und multi. Diese beziehen sich darauf, ob ein Ausdruck

- deterministisch, was bedeutet, dass der Ausdruck entscheidbar ist und es genau eine Lösung gibt,
- semideterministisch, was bedeutet, dass es genau eine oder keine Lösung gibt, oder
- multi ist, was bedeutet, dass es eine oder beliebig viele Lösungen möglich sind.

Dieser Bestandteil des logischen Paradigmas ist jedoch nicht in allen Sprachen des logisch-funktionalen Paradigmas enthalten! Mercury hat dieses Schema von Prolog übernommen. Regeln müssen mit den Attributen `det`, `semidet` oder `multi` versehen sein [52].

Curry hingegen gibt immer die erste Lösung gefolgt von einem „?“ an. So ergibt die Auswertung von

```
prelude> 4+3 == 8
Result: False ?
```

Dem Anwender obliegt nun, die Vorteile logischer Programmierung zu nutzen, um Curry nach weiteren möglichen Lösungen suchen zu lassen [44, p. 8]. Die Entscheidbarkeit einer Funktion muss also nicht im Voraus bekannt sein.

In Erlang lassen sich solche Konstrukte nachbilden und werden ähnlich wie in Mercury ausgewertet.

> Some use the first class labels in the GNU C compiler to goto from one function to another, e.g. the Erlang and Mercury compilers. [29]

Sie sind jedoch nicht wie in Mercury vorgesehen oder notwendig. Erlang wertet auch den Quelltext aus und gibt als Wert „OK“ aus, wenn dieser erfolgreich war. Eine weitere Möglichkeit besteht in der Nutzung einer Software namens Erlog, welche es Erlang ermöglicht, Prolog zu emulieren und aus dieser Emulation heraus wieder Erlang zu verwenden. Diese Entwicklung befindet sich derzeit noch in einem frühen Stadium und wird daher nicht weiter betrachtet.

Um alle möglichen Antworten zu erhalten, wird eine baumartige Struktur abgesucht. An Entscheidungspunkten wird der Zustand gespeichert und später bei einem Fehlschlag wiederhergestellt, um ihn dann in einer anderen Richtung abzusuchen. Die

Suche nach einer richtigen Antwort funktioniert also in logischen Sprachen nach dem so genannten `Backtracking`. Dieser Algorithmus ist in sämtlichen logischen Sprachen enthalten und ermöglicht es der Sprache, alle möglichen Antworten herauszufinden. Die Art des Backtracking, also das System, nach dem ein Baum abgesucht wird, kann zwischen den einzelnen Sprachen variieren.

Logische Sprachen sind daher für gewöhnlich weniger performant als funktionale Sprachen, wenn es nicht um die Auswertung von Audrücken geht. Sie sind allerdings auch für andere Anwendungsgebiete ausgelegt. Es gibt Ansätze, wie beispielsweise in Mercury, einzelne deterministische Abschnitte zu identifizieren und separat zu übersetzen, um die Effizienz zu steigern. Der Muenster Curry Compiler sieht diese Möglichkeit vor, hat sie aber noch nicht umgesetzt. Im Portland Aachen Kiel Curry System ist sie vorhanden [23].

Einige weitere charakteristische Eigenschaften sind:

- logische Variablen, also Platzhalter für logische Terme, was bedeutet, sie können mit logischen Termen unifiziert, also belegt, werden, und [42]
- partielle Datenstrukturen. Diese sind Teile einer Liste oder eines Tupels mit unbestimmten Rest.

Der unterschiedlich gewichtete Einfluss des logischen Paradigmas, insbesondere der logischen Sprache Prolog, auf die verschiedenen Sprachen des logisch-funktionalen Paradigmas verdeutlicht, dass die Grenzen unscharf sind. Es ist nicht erforderlich, diese für das logische Paradigma essentielle Eigenschaft zur Voraussetzung zu machen, wie es Mercury vorsieht. Auf der anderen Seite wird im folgenden Kapitel erläutert, dass der Einfluss der funktionalen auf die logisch-funktionalen Sprachen ebenfalls unterschiedlich gewichtet ist.

Die Seite http://vl.fmnet.info/logic-prog/ bietet eine gute Grundlage und Möglichkeiten zur Vertiefung. Hier werden nicht nur logische Sprachen erklärt, sondern auch die Eigenschaften des Paradigmas. Eine zentrale Rolle als logische Sprache hat Prolog inne. Sie wird oft exemplarisch für das logische Paradigma angeführt.

2.2 Das funktionale Paradigma

Beim funktionalen Paradigma werden Funktionen als Algorithmen aufgefasst. Diese lassen sich in Subalgorithmen zerlegen. Durch diese Modularisierung lassen sich Programme schreiben, die leicht parallelisierbar sind. Erlang beispielsweise legt großen Wert auf Parallelität, „da die Welt parallel und verteilt ist“ [36].

Ein weiterer wichtiger Aspekt des funktionalen Paradigmas ist die Verifizierbarkeit von Quelltexten. Funktionale Programme in Erlang beispielsweise enthalten keine for- oder while-Schleifen. Diese werden durch multiple Instanziierung einzelner Funktionen realisiert. Das bedeutet, dass eine Funktion sich selbst mit neuen Parametern initialisiert und dabei rekursiv arbeiten kann. Eine weitere charakteristische Eigenschaft ist, dass Variablen instanziierte Konstanten sind. Variablen können also nur einmal pro Instanz einer Funktion initialisiert sein. Ihr Wert kann sich danach nicht weiter innerhalb der Instanz verändern. Soll einer initialisierten Variablen ein neuer

Wert zugeordnet werden, muss eine neue Instanz der entsprechenden Funktion mit veränderten Werten ins Leben gerufen werden.

Dadurch werden funktionale Programme nicht nur leicht parallelisierbar, sondern sind auch mit weniger Aufwand verifizierbar als äquivalenter Quelltext imperativer Sprachen. Der funktionale Charakter basiert im Wesentlichen auf dem λ-Kalkül. Darum ist das λ in die Logos der Sprachen Haskell und Curry aufgenommen worden.

Ein weiterer bedeutender Aspekt im funktionalen Paradigma ist, die Unterscheidung zwischen reinen und unreinen Sprachen zu kennen. Sogenannte `reine Sprachen` lassen keine Seiteneffekte zu. Des Weiteren sind reine Sprachen faul, was bedeutet, dass sie eine Vorgehensweise namens `lazy evaluation` zur Auswertung von Ausdrücken benutzen. `Lazy evaluation` bedeutet, dass ein Teilausdruck erst dann ausgewertet wird, wenn sein Ergebnis benötigt wird. Im Gegensatz dazu steht die `strict evaluation`, also die sofortige Auswertung. Die Vorteile der `lazy evaluation` sind:

- es werden unendliche Datenstrukturen möglich,
- unnötige Reduktionen werden vermieden und
- die einmalige Auswertung und die damit verbundene Wiederverwendbarkeit von Ergebnissen, siehe Abschnitt 2.3.2.

Bei reinen funktionalen Sprachen ist die referentielle Transparenz also ein wichtiges Kriterium [41]. Es gibt verschiedene Auffassungen darüber, was genau die Begriffe rein und unrein bedeuten in Bezug auf funktionale Sprachen. Im Jahr 1993 veröffentlichte Amr Sabri eine Arbeit, die sich mit diesem Thema auseinandersetzt [33] mit dem Titel `What is a Purely Functional Language?`. Im Jahr 2003 griff Martin Grabmüller diese Definition auf und verwendete sie für seine Arbeit über Multiparadigmen-Programmiersprachen [41]. Im Wesentlichen definiert Sabri eine reine funktionale Sprache wie folgt:

> A language is purely functional if (i) it includes every simply typed λ-calculus term, and (ii) its call-by-name, call-by-need, and call-by-value implementations are equivalent (modulo divergence and errors). [33, Seite 2]

Zu den bekanntesten funktionalen Sprachen zählen Scheme, Lisp und Haskell. Weitere für das funktionale Paradigma charakteristische Merkmale sind:

- verschachtelte Ausdrücke
- Funktionen höherer Ordnung

Eine sehr gute Einführung in das funktionale Schema ist auf http://www.cs.nott.ac.uk/~gmh//faq.html zu finden. Hier wird nicht nur das funktionale Schema erklärt und definiert, sondern es werden auch die charakteristischen Eigenschaften erarbeitet und Grenzbereiche definiert.

2.3 Sechs logisch-funktionale Sprachen

Es gibt neun bekannte logisch-funktionale Sprachen. Davon sind drei so unbedeutend und veraltet, dass sie gesondert in Kapitel 2.3.8 behandelt werden. Im Folgenden werden die bedeutenden sechs Sprachen, welche vor unterschiedlichen Hintergründen entwickelt wurden, vorgestellt. Ein Unterschied ist die jeweilige Herkunft. Während einige Sprachen lokal gebunden sind und nur an einzelnen Hochschulen Verbreitung finden, werden andere Sprachen multinational entwickelt. Dafür ist der Gedankenaustausch bei der Entwicklung solcher Sprachen jedoch nicht so effizient wie bei lokalen Sprachen, bei denen die Entwickler direkt miteinander kommunizieren können.

Die drei Sprachen Curry, Erlang und Mercury werden in den folgenden Kapiteln fokussiert. ALF, Leda und Oz/Mozart hingegen werden nicht mehr weiterentwickelt und sind deshalb nur der Vollständigkeit halber in dieses Kapitel aufgenommen worden.

2.3.1 ALF

ALF steht für `Algebraic Logic Functional programming language`. Das letzte veröffentlichte Dokument über die Sprache ist das Benutzerhandbuch [25], welches auf den 18.08.1995 datiert ist. ALF wurde maßgeblich von Michael Hanus, der mittlerweile in Kiel doziert und an der Entwicklung einer Curry-Distribution beteiligt ist, vorangebracht und an den Universitäten Aachen und Dortmund entwickelt. Seit 1995 ist keine weitere Entwicklung mehr verzeichnet.

ALF beruht im Wesentlichen auf der Horn-Klausel und hat einen stark logischen Charakter. Das Vorgehen in ALF beruht auf einem Top-Down-Ansatz. Die Sprache bekommt ein Faktensystem und Regeln gegeben. Anschließend können Anfragen gestellt werden. ALF versucht dann vom Ziel ausgehend auf die Prämissen zurückzuschließen.

ALF basiert auf der Warren-Abstract-Machine, welche um einige Plugins erweitert (nicht in Abbildung 2.1 enthalten) in einem Emulator unter C läuft. Die Syntax erinnert sehr an Prolog und auch das verwendete Backtrackingverfahren `depth-first left-to-right` entsprechend der Horn-Klausel wird in Prolog verwendet. Die Sprache ist akademischen Ursprungs und wird soweit bekannt nicht in der Industrie eingesetzt. Der Entwicklungsstopp vor zehn Jahren kann eine Ursache dafür sein.

ALF wurde seit 1995 nicht mehr erkennbar weiterentwickelt [25]. Der Hauptentwickler arbeitet mittlerweile an der Curry-Distribution PAKCS und auch in Vorlesungen wurde ALF seit 1996 nicht weiter verwendet [48]. Deswegen wird diese Sprache nicht in den Fokus dieser Arbeit einbezogen.

2.3.2 Curry

Curry vereinigt funktionale, logische und nebenläufige Programmierparadigmen. Die Entwicklung von Curry ist eine internationale Initiative mit dem Ziel, einen Standard im Bereich der funktional-logischen Programmiersprachen zu schaffen. Durch die mögliche verzögerte Auswertung von Funktionen mit unbekannten Argumenten unterstützt Curry einen nebenläufigen Programmierstil. Darüber hinaus verfügt

Curry über ein polymorphes Typsystem, Module, Funktionen höherer Ordnung und ein deklaratives Ein- und Ausgabekonzept [43].

Curry ist eine Sprache, die sowohl das logische als auch das funktionale Paradigma berücksichtigt. Die Syntax ist mit der funktionalen Sprache Haskell vergleichbar. Curry kann als Erweiterung beziehungsweise Weiterentwicklung von Haskell verstanden werden. Es setzt auf Haskell auf und kann auch Haskell-Quelltexte kompilieren.

Sowohl Haskell als auch Curry wurden nach dem Mathematiker Haskell Curry, welcher von 1900 bis 1982 lebte, benannt und haben beide dasselbe Logo, welches auf der Titelseite dieser Arbeit unten rechts zu sehen ist.

Es gibt zwei Implementierungen von Curry, die Muenster Curry Compiler (MCC) und Portland Aachen Kiel Curry System (PAKCS) genannt werden. PAKCS ging aus dem Projekt PACS (Portland Aachen Curry System) hervor. Die Sprache Curry ist durch einen Report dokumentiert [47].

Der erste Entwurf der Sprache Curry datiert auf den 5. Dezember 1996 [46]. Im Jahr 1999 wurde der Entwurf in Report umbenannt.

Die aktuelle Version des PAKCS vom 28.03.2006 [45] liegt in Version 1.7.2 vor und enthält neben einer ausführlichen Einführung auch Beispielquelltexte. Die aktuelle Version des MCC vom 10.05.2006 liegt in Version 0.9.10 vor.

Seit dem 4. November 2002 ist der MCC öffentlich zugänglich. Die Entwicklung des MCC begann aber schon 1998. Die ersten Versionen wurden nur hochschulintern verwendet. Die erste Version von PACS ist im Juli 1999 veröffentlicht worden. Es gibt zwei Hauptentwicklungslinien, welche zwar beide akademischen Ursprungs sind, jedoch völlig unterschiedliche Herangehensweisen verwenden.

MCC (Muenster Curry Compiler)

Der MCC übersetzt im Gegensatz zum PAKCS den Quelltext via C in Maschinen-Code, verwendet also keine abstrakte oder virtuelle Maschine. Zur Zeit arbeitet nur Wolfgang Lux an der Entwicklung des MCC. Für die Installation wird ein C-Compiler, vorzugsweise der GNU C Compiler (GCC) [24], sowie ein Haskell98-Compiler vorausgesetzt. Näheres zur Installation ist in Abschnitt 3.3 beschrieben.

PAKCS (Portland Aachen Kiel Curry System)

Das PAKCS übersetzt im Gegensatz zum MCC von Curry nach Prolog und benötigt daher eine Prolog-Distribution als Host-System [23].

Bis zur Version 1.6.1-5 vom 23.10.05 war eine Voraussetzung ein kommerzielles Prolog, welches vom Swedish Institute of Computer Science (http://www.sics.se/) entwickelt und unter dem Namen SICStus Prolog (http://www.sics.se/isl/sicstuswww/site/index.html) veröffentlicht wurde. Seit Version 1.7.0 vom 06.12.05 akzeptiert das PAKCS neben der kommerziellen Variante auch ein freies Prolog von SWI (http://www.swi-prolog.org/) als Host-System. Eine weitere grundlegende Veränderung seit Version 1.7.0 ist das Einbinden des Frontends des MCC in PAKCS, da der eigene Parser vor Version 1.7.0 fehlerhaft war und nicht den vollen Sprachumfang von Curry gewährleistete [23].

Zur Installation des PAKCS wird zum einen ein Prolog als Host-System vorausgesetzt. Dabei wird für SICStus Prolog volle Funktionalität gewährleistet, während bei

der Verwendung von der kostenfreien Variante von SWI-Prolog mit Einschränkungen zu rechnen ist:

> If you do not have SICStus-Prolog, but then the execution is less efficient and some functionality (e.g., real arithmetic constraint solvers) is not available. (http://www.informatik.uni-kiel.de/~pakcs/download.html)

Zum Anderen setzt das PAKCS einen vorhandenen Glasgow Haskell Compiler voraus, der von http://haskell.org/ghc/ bezogen werden kann. Näheres zur Installation ist in Abschnitt 3.3 beschrieben.

Die Unterschiede zwischen PAKCS und MCC

Ein wesentlicher Unterschied ist wie oben beschrieben die Implementierungstechnik. Dieser Unterschied hat einige Konsequenzen.

- Im PAKCS können viele der Funktionen des verwendeten Prolog-Systems quasi frei Haus verwendet werden, die in einer nativen Implementierung wie dem MCC neu implementiert werden müssten. Dabei ist insbesondere das `constraint solving` interessant.
- Wie von Wolfgang Lux, dem Entwickler des MCC, zu erfahren war, liegt dagegen der Vorteil nativer Implementierungen in effizienterem Maschinen-Code. Dies trifft nach seinen Erfahrungen

  > interessanterweise (...) für in logischem Programmierstil geschriebene Programme zu. [23]

- Ein weiterer Vorteil des MCC ist, dass einige Spracheigenschaften, wie beispielsweise die eingekapselte Suche, nur in nativer Implementierung möglich sind.
- Ein Unterschied, auf den ich durch Wolfgang Lux aufmerksam gemacht worden bin, ist die Frage, wie das Sharing für die Implementierung der zugrunde liegenden `lazy evaluation` implementiert wird. Im Jahre 1993 veröffentlichte Feixion Liu eine Arbeit mit dem Titel `Towards lazy evaluation, sharing, and non-determinism in resolution based functional logic languages` [22], in der diese Fragestellung behandelt wird.
  - Im MCC werden unausgewertete Funktionsapplikationen so implementiert, dass grundsätzlich Sharing erfolgt. Zur Zeit ist es noch nicht möglich, ein Sharing zu verhindern, wenn der Compiler feststellt, dass eine solche Applikation nur einmal ausgewertet wird.
  - Beim PAKCS erfolgt Sharing in der Regel post-facto. Sollte ein Argument wiederverwendbar sein, wird also zur Laufzeit überprüft, ob Sharing notwendig ist.
- Ein weiterer Unterschied ergibt sich durch die unterstützten Bibliotheken.

2.3.3 Erlang

Erlang ist eine logisch-funktionale Programmiersprache, die nach Agner Krarup Erlang benannt wurde.

Agner Krarup Erlang

A.K. Erlang wurde am 1.1.1878 in Lonborg in Dänemark geboren. Bereits im Alter von 14 Jahren bestand er die Aufnahmeprüfung an der Universität zu Kopenhagen mit Auszeichnung. Da er jedoch zu jung war, unterrichtete er mit seinem Vater, einem Lehrer, bis er 16 Jahre alt war. Anschließend lernte er zwei Jahre lang, um im Alter von 18 Jahren noch einmal die Aufnahmeprüfung mit Auszeichnung zu bestehen. Er promovierte 1901 in Mathematik, studierte jedoch auch Astronomie, Physik und Chemie. Die folgenden sieben Jahre lehrte er an verschiedenen Schulen und erhielt in dieser Zeit einen Preis für eine Arbeit, die er bei der Universität Kopenhagen einreichte.

Durch Zufall begegnete er Johan Jensen, einem Chefingenieur der Telefongesellschaft, bei einem Treffen der Vereinigung der dänischen Mathematiker und erhielt eine Anstellung im Jahre 1908. Bis zu seinem Tod arbeitete er dort für die Telefongesellschaft.
Während seiner Arbeit beschäftigte sich A.K. Erlang mit dem Problem, wie viele Leitungen benötigt werden, um eine feste Anzahl Einwohner ausreichend zu versorgen. Seine Berechnungen hatten einen Fehler: Er ging davon aus, dass Benutzer jede Wartezeit akzeptieren würden. Er schuf damit jedoch die Anfänge der Warteschlangenlogik.

Seine berühmtesten Formeln sind Erlang-B und Erlang-C. Mit Erlang-B lässt sich die Anzahl benötigter Leitungen beispielsweise für ein Callcenter berechnen, während sich mit Erlang-C die Anzahl der benötigten Callcenteragenten berechnen lässt.
Die bedeutendsten Schriften Erlangs sind:

- `The Theory of Probabilities and Telephone Conversations` von 1909
- `Solution of some Problems in the Theory of Probabilities of significance in Automatic Telephone Exchanges` von 1917

Ihm zu Ehren wurde die Einheit für die Auslastung eines Telefonnetzes 1946 auf den Namen Erlang getauft. Ein Erlang entspricht einer benutzten Leitung beziehungsweise eines ausgelasteten Kanals. Besteht eine Leitung beispielsweise aus 24 Kanälen, und werden davon nur zwölf gleichzeitig genutzt, ist die Auslastung 0,5 Erlang. Die Einheit für die Auslastung heißt dementsprechend Erlangstunde. Mit dieser Maßeinheit lassen sich Berechnungen anstellen, um Aussagen über den Grad eines Dienstes (Grade of Service - GoS) und die Qualität eines Dienstes (Quality of Service - QoS) zu machen.

Die Sprache Erlang

Die Programmiersprache Erlang wurde bei Ericsson unter der Leitung von Joe Armstrong entwickelt. Gesucht wurde anfänglich eine Sprache, die sich für den Einsatz in einer eingebetteten Umgebung eignet. Nachdem einige Sprachen ausprobiert worden

waren, entschlossen sich die Ingenieure, eine neue Sprache zu entwickeln. So entstand in den neunziger Jahren eine Sprache, welche logisch und funktional ist, die sich in eingebetteten Systemen implementieren lässt und die durch ihre Prägnanz besticht. Erlang ist unter Linux, Unix und Windows lauffähig und erfüllt sanfte Realzeitanforderungen. Entwickelt wurde Erlang vor einem kommerziellen Hintergrund. Jedoch ist die Sprache frei verfügbar unter der Erlang Public License (EPL, [11]) und kann kostenlos unter http://www.erlang.org heruntergeladen werden.

Im Jahre 2002 fasste Joe Armstrong die Vorteile von Erlang in seiner Arbeit `Concurrency Oriented Programming in Erlang` [36] zusammen. Diese Arbeit beinhaltet eine kleine Einführung und bereitet die wesentlichen Merkmale der Sprache auf. Ein besonderes Augenmerk wurde dabei auf Parallelität geworfen. In diesem Bereich kann Erlang populäre Sprachen wie C++ und Java an Performanz und Quelltextlänge übertreffen beziehungsweise unterbieten [13]. Armstrong weist jedoch darauf hin, dass sich Erlang besonders in einem lokalen durch eine Firewall geschützen Netzwerk als nützlich erweist. Für die Verwendung von Erlang in einem WAN sind weitere Sicherheitsvorkehrungen notwendig wie etwa der Einsatz eines virtuellen privaten Netzwerkes (VPN).

Erlang wurde anfänglich als funktionale Sprache konzipiert und hat eine Syntax, die C ähnelt. Anders als bei Prolog wird hier mehr Wert auf Eigenschaften gelegt, die schnelles Entwickeln `rapid prototyping`, einfache Verifizierbarkeit und Parallelität ermöglichen. Zwar findet auch das logische Paradigma Berücksichtigung, jedoch steht es eindeutig im Hintergrund. Das ist der Grund, warum Erlang häufig nur als funktionale Sprache aufgeführt wird.

2.3.4 Leda

Die offizielle Homepage von Leda ist http://web.engr.oregonstate.edu/~budd/leda.html. Es gibt jedoch ein weiteres Projekt unter http://www.algorithmic-solutions.com/enleda.htm mit demselben Namen, das sich mit einem anderen Gebiet beschäftigt und nicht mit der logisch-funktionalen Sprache Leda verwechselt werden sollte.

Die experimentelle Programmiersprache Leda versuchte, das logische, das funktionale und das objekt-orientierte Paradigma mit imperativer Programmierung umzusetzen. Leider sind außer auf Wikipedia keine weiteren Informationen verfügbar. Die Projektseite verweist nur auf tote Weiterleitungen.

Der Name Leda bezieht sich auf die griechische Mythologie. Leda war die Tochter des Thestios, in die sich Zeus, der Göttervater, verliebte. Er näherte sich ihr in Gestalt eines Schwanes und schwängerte sie. Daher ist das Symbol der Programmiersprache Leda ein Schwan.

Entwickelt wurde Leda von Timothy A. Budd an der `Oregon State School of Electrical Engineering and Computer Science`. Da es sich bei Leda um einen experimentellen Ansatz handelt, war Leda in der Entwicklung starken Veränderungen ausgesetzt [40].

Zu Leda ist 1994/1995 im Addison-Wesley-Verlag das Buch `Multiparadigm Programming in Leda` erschienen. Dies ist heute jedoch nicht mehr erhältlich [7]. Außerdem leiten sämtliche Verweise auf weiterführende Quellen auf der Projektseite auf nicht mehr existente Dokumente. Da die Sprache seit 1995 nicht weiterentwickelt wurde und auch Literatur kaum vorhanden ist, wird die Sprache im Folgenden nicht weiter behandelt.

2.3.5 Mercury

> Mercury ist eine streng getypte rein deklarative logische Programmiersprache, die zum einen als Weiterentwicklung der logischen Programmiersprache Prolog angesehen werden kann, die jedoch auch neue Aspekte insbesondere aus dem Gebiet der funktionalen Programmiersprachen berücksichtigt. [43]

Mercury verwendet dieselbe Syntax und Semantik wie Prolog, beherrscht aber zusätzlich funktionale Konstrukte. Damit ist es möglich, auch Prolog-Quelltexte mit dem `Melbourne Mercury Compiler` (MMC) zu kompilieren. Der MMC übersetzt Quelltexte zunächst nach C und schliesslich in Maschinen-Code, geht also ähnlich wie der MCC vor (siehe Abbildung 2.1). Weitere Backends sind unten aufgelistet.

Mercury an der Universität Melbourne unter Leitung von Zoltan Somogyi entwickelt wird. Etwa alle zwei Tage wird eine neue `Release-of-the-Day`-Version veröffentlicht. Zur Zeit wird Version 13.0 abgeschlossen. Die erste Version wurde am 8. April 1995 veröffentlicht. Die aktuelle stabile Version 12.2 wurde am 25. Januar 2006 fertiggestellt.

Wie es für logische Sprachen üblich ist, gehört Mercury zu den kompilierenden Sprachen und nicht zu den interpretierenden. Der MMC greift auf ein striktes System zurück, verwendet also keine `lazy evaluation`. Dies soll dazu beitragen, Programme schneller entwickeln zu können.

Dem historischen Vorgänger von Mercury, Prolog, war es nicht möglich, Modularisierung zu handhaben. Mittlerweile gibt es jedoch Prolog-Implementierungen, die auch Module unterstützen.

Da Mercury im Gegensatz zu Prolog zu den `reinen` Sprachen zählt, ist Mercury üblicherweise performanter als Prolog.

Das Besondere an Mercury ist, dass es nicht erst nach Prolog kompiliert wie beispielsweise das PAKCS, sondern eine Fülle an Backends anbietet, die sich allerdings zum Großteil noch in der Entwicklung befinden. So ist Mercury darauf ausgelegt, nach C zu kompilieren. Es sind auch schon Ansätze vorhanden, Mercury nach

- IL (Microsoft's Intermediate Language, verwendet von Microsoft's .NET Common Language Runtime),
- native code (Übersetzung in Assembler mit Hilfe des GCC-Backends) und
- Aditi (die relationale Sprache des Aditi deductive database system, http://www.cs.mu.oz.au/research/aditi/, ein weiteres Projekt der Universität Melbourne)

zu übersetzen. Diese sind noch in der Entwicklung.

In Abbildung 2.1 ist ersichtlich, dass es eine Möglichkeit gibt, Mercury nach Java zu übersetzen. Diese ist jedoch noch in einer frühen Phase der Entwicklung. Genaueres steht im Quelltext von Mercury unter `compiler/mlds_to_java.m` beschrieben.

2.3.6 Oz/Mozart

Oz gehört wie das PAKCS zu den multinationalen Systemen. Seit Version 2 von Oz aus dem Jahre 1996 wurde auf automatische Nebenläufigkeit verzichtet [41]. Dieser wichtige Aspekt ist exemplarisch für die Entwicklungsweise logisch-funktionaler Sprachen.

Oz ist eine multiparadigmatische Programmiersprache, die ab 1991 an der Universität des Saarlandes unter der Leitung von Gert Smolka entwickelt worden ist. 1996 wurde die Entwicklung von einer Projektgruppe des Swedish Instutute of Computer Science (SICS) unter der Leitung von Seif Haridi übernommen. Im Jahr 1999 wurde das Mozart-Konsortium gegründet, bestehend aus der Universität des Saarlandes, dem SICS sowie der Université Catholique de Louvain. Sechs Jahre später wurde Mozart Board gegründet, eine Kerngruppe, deren Aufgabe darin bestand, Mozart schnellstmöglich einer breiten Nutzergemeinde zugänglich zu machen um die Popularität zu erhöhen.

Mozart ist die Sprache, die weltweit bis jetzt die meisten Paradigmen in sich vereinigen kann. Das logische und das funktionale Paradigma sind nur zwei davon. Insgesamt sind folgende Paradigmen enthalten:

- logisch
- funktional, sowohl `lazy` als auch `eager`
- imperativ
- objekt-orientiert
- verteilt (`distributed`)
- nebenläufig im Sinne der Definition von Joe Armstrong

Die Stärken von Mozart liegen insbesondere im `constraint-programming` sowie im `distributed programming`. Nach Oz/Mozart mit acht Paradigmen ist Ada auf Platz zwei mit fünf Paradigmen, die nebenläufig, verteilt, generisch, imperativ und objekt-orientiert sind. Der historische Vorgänger von Oz/Mozart ist die funktionale Sprache Alice, die ebenfalls an der Universität des Saarlandes entwickelt wurde.

Seit 2004 wurde Oz/Mozart nicht mehr weiterentwickelt. Die Entwickler von damals arbeiten mittlerweile an anderen Projekten.

Mozart/Oz wurde am 16.07.2004 zum letzten Mal in einer neuen Version veröffentlicht. Obwohl im Oktober 2004 noch eine Konferenz stattfand und auch die Mailingliste von einigen Entwicklern noch verwendet wird (http://www.ps.uni-sb.de/pipermail/mozart-hackers/), wurde seit 2004 keine Weiterentwicklung veröffentlicht. Außerdem lässt sich an der Frequentierung der Mailingliste absehen, dass von 2001 bis 2004 eine kontinuierliche Zunahme des Verkehrs stattfand. Seit 2005 ist dieser Trend jedoch genauso kontinuierlich rückläufig, hochgerechnet für 2006, basierend auf dem Mailaufkommen bis 09.05.2006. Daher wird Oz/Mozart im Folgenden nicht weiter betrachtet.

2.3.7 Die Unterschiede zwischen Curry und Mercury

Obwohl der Melbourne Mercury Compiler (MMC) und der Muenster Curry Compiler (MCC) eine ähnliche Vorgehensweise haben, sind Curry und Mercury doch sehr verschieden:

- Curry hat eine Haskell- und Mercury eine Prolog-Syntax.
- Curry arbeitet mit `lazy evaluation`, während Mercury `strict` arbeitet.
- Curry hat keine Einschränkung bezüglich logischer Variablen. In Mercury hingegen sind die Modi zu spezifizieren.

2.3.8 λ-Prolog, Sisal und HAL

Bei HAL, Sisal und λ-Prolog handelt es sich um drei Sprachen, die ebefalls dem logisch-funktionalen Paradigma entsprechen. Sisal ist unter http://sisal.sourceforge.net/ erreichbar, λ-Prolog ist unter http://www.lix.polytechnique.fr/Labo/Dale.Miller/lProlog/ zu finden und HAL, das mit Mercury verwandt ist, ist unter http://www.csse.monash.edu.au/~mbanda/hal/index.html zu erreichen.

Bei diesen Projekten handelt es sich um Implementierungen, die so unbedeutend für das logisch-funktionale Paradigma sind, dass sie hier nur am Rand erwähnt werden.

Sisal

Sisal gehört zu den interpretierten Sprachen. Die aktuelle Version ist 14.0.0alpha1 mit dem Bugfix2-Paket vom 27.02.2006. Die Sprache ist unter Sourceforge verfügbar und wurde bisher nur 61mal heruntergeladen.

Sisal wurde entworfen, um die Vorteile der Parallelität auszunutzen. Ein Ziel war es, die Sprache Fortran, die Anfang der Neunziger sehr populär war, zu ersetzen. Unter Sisal war es möglich, Multitasking-Compiler zu nutzen, die auf einer SMP-Architektur, einem Symmetrisches Multiprozessorsystem, arbeiteten. Heute unterstützt jedes gängige Betriebssystem Multitasking und bis auf wenige Ausnahmen auch Mehrprozessorbetrieb. Viele Compiler und Interpreter unterstützen eine Technologie namens Multithreading, also das Aufteilen eines Prozesses in Subprozesse zum parallelen Abarbeiten auf mehreren Prozessoren beziehungsweise Prozessorkernen wie beispielsweise Java 1.5 oder Erlang ab Version R10B10.

Im Jahr 1996 wurde die Entwicklung von Sisal gestoppt. Der Hauptentwickler Pat Miller ist so überzeugt von Sisal, dass er inzwischen die Entwicklung wieder aufgenommen hat und jede Woche mindestens eine Nacht an der Sprache arbeiten will „bis die Hölle zufriert“:

> I will try to work on Sisal one night a week until hell freezes over to get this done.

λ-Prolog

Bei λ-Prolog handelt es sich um eine Sprache, die ähnlich wie Curry in zwei unterschiedlichen Implementierungen entwickelt wurde. Eine Implementierung von λ-Prolog ist das Terzo-Projekt, in dem Philip Wickline einen Interpreter für die Sprache entwickelte. Die aktuelle Version vom 11.01.1999 ist als Quelltext gerade einmal 132 Kilobyte groß und basiert auf der Standard Markup Language of New Jersey (SML/NJ).

Eine weitere Implementierung ist Teyjus. Diese Implementierung geht ähnlich wie ALF vor. Sie übersetzt zuerst in eine abstrakte Maschine, die anschließend nach C kompiliert. C übersetzt anschließend als Backend in Maschinen-Code. Auch dieses System hat seit 1999 keine Aktualisierung mehr erfahren.

Mit

- LP-SML (Lambda Prolog Standard Markup Language),
- eLP (Ergo Lambda Prolog) und
- LP2.7

gibt es noch drei weitere Implementierungen von λ-Prolog, zu denen keine Informationen mehr verfügbar sind. LP2.7 wurde 1988 eingestellt.

HAL

HAL gehört zu den Constraint Logic Programming Languages (CLP). Die letzte Veröffentlichung stammt aus dem Jahr 2003 [10]. HAL ist mit Mercury verwandt und wurde an der Monash University Australia unter Mitwirkung der University of Melbourne entwickelt. HAL-Quelltext lässt sich auch unter SICStus Prolog compilieren.

Die Sprache ist unausgereift. Das verdeutlicht die ToDo-Liste (siehe http://www.csse.monash.edu.au/~mbanda/hal/index.html#status). Weder akademisch noch wirtschaftlich hat diese Sprache eine Bedeutung. Weitere Informationen sind verfügbar unter:

- http://www.cs.mu.oz.au/research/dl.html#hal
- http://www.csse.monash.edu.au/~mbanda/hal/index.html

2.3.9 Fazit

Dieser Vergleich zeigt exemplarisch, wie heterogen das Feld der logisch-funktionalen Sprachen ist. Es gibt Sprachen, die eher logisch, andere die eher funktional geprägt sind. Es gibt Sprachen mit akademischem sowie solche mit kommerziellem Charakter. Die Vorgehensweise der Sprachen ist so unterschiedlich, dass ein Diagram (Abbildung 2.1) zur Veranschaulichung sehr hilfreich ist. Jede dieser Sprachen hat Vor- und Nachteile in Bezug auf verschiedene Anwendungsgebiete, und obwohl sie alle im logisch-funktionalen Paradigma sind, bieten sie jeweils unterschiedliche Sichtweisen und Möglichkeiten zur Lösung eines Problems. Die Gründe dafür sind oben angeführt. Im Folgenden werden zunächst die technischen Details, soweit sie in diesem Kapitel noch nicht beschrieben sind, abgehandelt.

2.4 Fokussierung

Von den sechs bekannten Sprachen, die sowohl dem logischen wie auch dem funktionalen Paradigma entsprechen, habe ich mich mit den drei Sprachen beschäftigt, die heute noch weiterentwickelt werden.

Zu ALF, Leda und Oz/Mozart sind nur relativ alte Informationen verfügbar. Da die Entwicklung dieser Sprachen nicht weiter stattfindet beziehungsweise absehbar ist, dass eine Sprache in naher Zukunft nicht weiter entwickelt werden wird, werden diese drei Sprachen im weiteren Verlauf dieser Arbeit nicht weiter berücksichtigt.

Im Bereich der logisch-funktionalen Sprachen gibt es drei Sprachen, die zur Zeit entwickelt werden. Teilweise sind schon ansehnliche Bibliotheken für diese Sprachen verfügbar. Um in dieser Arbeit weiter in die Tiefe der Sprachen vordringen zu können, wird darauf verzichtet, alle Sprachen des logisch-funktionalen Spektrums in der Breite zu betrachten. Stattdessen liegt der Fokus dieser Arbeit bei den Curry-Distributionen PAKCS und MCC sowie den Sprachen Erlang und Mercury.

2.5 Vergleichsgebiete

In den Vergleichskriterien werden vier Bereiche hervorgehoben. Die wesentlichen Vergleichspunkte sind

- Erlernbarkeit (Kapitel 4.1),
- Zukunftsperspektiven (Kapitel 4.2),
- Anwendungsgebiete (Kapitel 4.3),
- Performanz (Kapitel 4.4).

Ich habe mich für einen Vergleich mit Java entschieden, da Java sehr bekannt und verbreitet und ausgereift ist und über eine ansehnliche Bibliothek verfügt. Außerdem gibt es einige Parallelen zu Erlang. Java hat wichtige benötigte Funktionen auf ähnliche Weise wie Erlang integriert und die Quelltexte werden durch eine ähnliche Syntax vergleichbar.

Dabei ist Erlang ein Vertreter der logisch-funktionalen Sprachen und Java der Bezugspunkt. Es wären viele Arten von Vergleichen möglich gewesen und es haben schon viele Vergleiche stattgefunden, wie etwa

- Haskell vs. Erlang [37]
- Java vs. Erlang [13]
- C++ vs. Erlang [35]
- Call vs. Erlang [4]
- Mercury vs. Prolog [21]

Der Vergleich Erlang vs. Java wurde schon einmal angestellt. Um Aspekte wie die Performanz von verteiltem Rechnen zu betrachten, ist es allerdings notwendig, einen komplexeren Benchmark zu verwenden. Ein weiterer Grund für das erneute Untersuchen ist, dass sich inzwischen Hardware und Programmiersprachen maßgeblich gewandelt haben und die Bedeutung alter Ergebnisse keine bedeutende Relevanz für aktuelle Systeme wie die oben genannten hat.

Die Ergebnisse der Versuche legen nahe, dass es sinnvoll ist, eine relativ unbekannte Sprache des logisch-funktionalen Schemas mit einer Sprache zu vergleichen, die weit verbeitet und bekannt ist. Damit lassen sich die Vorzüge und Nachteile neuer Technologien gegenüber etablierten Sprachen aufzeigen. Oft müssen sich unbekanntere Sprachen, wie die des logisch-funktionalen Paradigmas, daher mit elaborierten Sprachen wie C++ oder Java vergleichen lassen. Dabei ist darauf zu achten, dass die Sprachen vergleichbar und nicht zu unterschiedlich sind.

Erlang ist die einzige logisch-funktionale Sprache mit einem kommerziellen Hintergrund. Sie bietet umfangreiche Dokumentation und Bibliotheken und enthält maßgebliche Funktionen, um ein Benchmarkverfahren zu implementieren. Erlang ist auf verteiltes Rechnen ausgelegt und im Gegensatz zu Curry und Mercury von der Syntax mit Java oder C++ vergleichbar.

Java als Bezugspunkt ist sinnvoll, da Java notwendige Funktionen in mitgelieferten Bibliotheken anbietet, verteiltes Rechnen möglich ist und die Syntax vergleichbar mit der von Erlang ist. Dabei ist zu beachten, dass Java als objekt-orientierte Sprache andere Möglichkeiten hat als Erlang. Wie zu Anfang des Kapitels beschrieben, folgen beide Sprachen unterschiedlichen Paradigmen. Sie bieten unterschiedliche Perspektiven zur Lösung eines Problems. Java ist objekt-orientiert und Erlang ist funktions-orientiert.

Da sich einige Problemklassen durch beide Paradigmen bewältigen lassen, ist es sinnvoll, die Vor- und Nachteile abzuwägen, um sich für spätere Aufgaben auf Grund dieser Auswertung für eine Technologie entscheiden zu können.

Ein weiterer Grund, warum Erlang mit Java als Bezugspunkt für einen Vergleich besser geeignet ist als Mercury oder Curry mit Java ist, dass sowohl Erlang als auch Java zu den interpretierenden Sprachen gehören. Auf der anderen Seite ist Java ein besserer Bezugspunkt als C, da fast alle logisch-funktionalen Sprachen über C kompilieren (siehe Abbildung 2.1).

3. Technische Details

3.1 Verwendete Hardware und Betriebssysteme

Als Basishardware wurde ein Cluster aus 19 Rechnern benutzt. Dabei handelt es sich jeweils um ein Fujitsu-Siemens W600 System mit GeForce 4 MX Grafikkarte. Als Betriebssysteme auf diesen Rechnern dienten zum einen ein auf lokalen Festplatten installiertes Windows 2000 mit allen bis dato verfügbaren Updates ohne weitere Hintergrundprogramme. Zum Anderen wurden die Tests unter einem Debian-System durchgeführt, das zentral über NFS gebootet wurde. Das Debian-System wurde mit aktuellen Patches aber ohne überflüssige Hintergrunddienste verwendet. Diese Tests können jedoch nicht als repräsentativ gelten, da kein exklusiver Zugriff auf den Rechner gewährleistet wurde und andere Benutzer das Ergebnis verfälscht haben könnten.

3.2 Verwendete Software

Als Sprache wurde zum einen Erlang gewählt als Repräsentant des logisch-funktionalen Spektrums. Um Erlang und damit die Klasse logisch-funktionaler Sprachen einordnen zu können, ist es wichtig, mit einer Sprache zu vergleichen, die zum einen bekannt ist und zum anderen in der Entwicklung fortgeschritten. Da Java als einzige Sprache an der Universität Oldenburg als Pflichtfach gelehrt wird, über einen ausreichenden Bekanntheitsgrad verfügt und eine große Auswahl an Bibliotheken mit sich bringt, wird Erlang in Kapitel 4.4 mit der objekt-orientierten Sprache Java verglichen.

Dieser Vergleich birgt einige Nachteile in sich. Beide Paradigmen-Klassen sind für unterschiedliche Anwendungsbereiche ausgelegt. Auch sind unter Java verschiedene Konstrukte enthalten, die auf Grund der Paradigmen Erlang vorenthalten bleiben, wie beispielsweise for- und while-Schleifen. Auf der anderen Seite bietet Erlang nativ eine Netzwerkkommunikation an, die in Java erst über Erweiterungen wie beispielsweise RMI, Corba oder Sockets gegeben ist. Außerdem werden zum Ver- und Entschlüsseln integrierte Module beziehungsweise Klassen verwendet. Im Wesentlichen hängt die Performanz also von diesen verwendeten Funktionen ab. Bei der Programmierung wurde darauf geachtet, soweit wie möglich vorhandene vergleichbare

Funktionen zu benutzen, um den Einfluss durch eigene Programmierung so gering wie möglich zu halten.

Bei Java wurde die Version 1.5 verwendet, die unter http://java.sun.com/j2se/1.5.0/download.jsp verfügbar ist. Unter Linux ist sie alternativ über entsprechende Paketmanager wie APT verfügbar. Java 1.5 ist multiprozessorfähig und der Nachfolger Java 1.6 scheidet als Kandidat aus, da er sich zur Zeit noch im Beta-Stadium befindet. Zusätzlich wurde RMI für die Kommunikation verwendet, sowie die Java Cryptography Extension (JCE), die seit Java 1.4 fester Bestandteil der mitgelieferten Bibliothek ist.

3.3 Installation von Curry, Erlang und Mercury

Grundsätzlich lassen sich alle drei Sprachen unter Linux, Unix und Windows betreiben. Für Windows benötigt Curry jedoch einen Linux-Emulator wie Cygwin. Da Cygwin nicht Thema dieser Arbeit ist, wird für Curry die Installation unter Linux erklärt, die analog für Cygwin unter Windows verläuft.

Die Verwendung des APT Paket-Managers bezieht sich auf die in dieser Arbeit benutzten Debian-Systeme. Andere Linux-Distributionen wie etwa Fedora oder Gentoo verwenden andere Paket-Manager, auf die in dieser Arbeit nicht eingegangen wird.

Curry

Es gibt zwei Distributionen, PAKCS und MCC, die unterschiedliche Herangehensweisen verfolgen und unterschiedliche Anforderungen haben (siehe Kapitel 2.3.2). Zudem gibt es von dem Muenster Curry Compiler (MCC) noch eine Distribution für Macintosh namens AquaCurry.

PAKCS

Eine englische aktuelle Installationsanleitung ist unter http://www.informatik.uni-kiel.de/~pakcs/INSTALL.html verfügbar.

Für das Portland Aachen Kiel Curry System (PAKCS) gibt es eine Grundsatzentscheidung bei der Installation. PAKCS basiert auf Prolog und setzt als Umgebung entweder eine Distribution vom Swedish Institute of Technology (SICS) oder von SWI voraus. Die Vorteile von SICStus Prolog sind, dass es mehr Bibliotheken anbietet, mehr Funktionalitäten hat und für gewöhnlich effizienter ist. Ein Vorteil von SWI-Prolog ist, dass es kostenfrei ist.

Da für diese Arbeit keine Lizenz von SICStus-Prolog bereitgestellt werden konnte, wurde SWI-Prolog verwendet. Eine Preisliste für SICStus-Prolog ist unter http://www.sics.se/isl/sicstuswww/site/order.html einsehbar. Außerdem wird eine aktuelle Version von make benötigt.

SWI-Prolog kann kostenfrei unter http://www.swi-prolog.org/ bezogen oder über einen Paket-Manager wie APT installiert werden. Außerdem sollte ein aktuelles make und ein Glasgow Haskell Compiler (GHC) bereitstehen, die analog entweder über Paket-Manager oder von http://ftp.gnu.org/pub/gnu/ (Make) beziehungsweise http://www.haskell.org/ghc/ (GHC) zu beziehen sind. Um die notwendige Software über APT zu installieren, sind folgende Befehle notwendig:

```
sudo apt-get install swi-prolog
```

```
sudo apt-get install make
```

```
sudo apt-get install ghc6
```

Sobald die notwendige Software bereitsteht, sind folgende Schritte notwendig:

- Beziehen des PAKCS als Quelltext
  - Eine aktuelle Version steht unter http://www.informatik.uni-kiel.de/~pakcs/pakcs_src.tar.gz zur Verfügung
- Entpacken des Tar-Balls
  - Entpacken mit dem Befehl `tar -xzvf pakcs-src.tar.gz`
- Wechseln in das Verzeichnis `pakcs`
- Installieren des PAKCS mit den Befehlen
  - `sudo ./configure-pakcs` (Dauer unter einer Minute)
  - `sudo make` (Dauer etwa zehn bis 15 Minuten)
  - `sudo make install` (Dauer unter einer Minute)

Anschließend bietet es sich an, das Verzeichnis nach `/etc/` mit dem Befehl

```
sudo mv pakcs /etc/
```

zu verschieben und im Home-Verzeichnis in die Datei `.bashrc` folgende Zeile zu ergänzen:

```
export PATH=$PATH:/etc/pakcs/bin
```

Nun ist das PAKCS installiert und über den Befehl `pakcs` verfügbar.

MCC

Eine aktuelle Installationsbeschreibung ist unter http://danae.uni-muenster.de/~lux/curry/user.html sowie im PDF-Format unter http://danae.uni-muenster.de/~lux/curry/user.pdf verfügbar.

Eine Voraussetzung ist der Glasgow Haskell Compiler (GHC), dessen Installation im vorigen Kapitel erläutert wurde. Außerdem wird eine aktuelle Version des GNU C Compilers (GCC) vorausgesetzt. Der Quelltext kann unter http://www.gnu.org/software/gcc/releases.html heruntergeladen oder über APT mit

```
sudo apt-get install gcc
```

installiert werden.

Anschließend kann MCC von http://danae.uni-muenster.de/~lux/curry/download/ heruntergeladen und entpackt werden. Für diese Arbeit wurde Version 0.9.9 verwendet. Inzwischen ist Version 0.9.10 erschienen. Da die Installation unterschiedlich ist, werden beide Vorgehensweisen erläutert

0.9.9

- Zuerst MCC entpacken
  - tar -xzvf curry-0.9.9.tar.gz
- Danach einen notwendigen Patch herunterladen und anwenden
  - Patch herunterladen von http://danae.uni-muenster.de/~lux/curry/download/curry-0.9.9/ghc-6.4.patch
  - Patch in das Verzeichnis von MCC kopieren mit `cp ghc-6.4.patch curry-0.9.9`
  - Patch anwenden mit `patch -p0 < ghc-6.4.patch`
- Nun empfiehlt es sich, MCC nach `/etc` zu verschieben und in das entsprechende Verzeichnis zu wechseln mit
  - `sudo mv curry-0.9.9 /etc/`
  - `cd /etc/curry-0.9.9/`
- Abschließend kann Curry installiert werden mit den Befehlen
  - `sudo ./configure` (Dauer unter einer Minute)
  - `sudo make` (Dauer etwa zehn Minuten)
  - `sudo make install` (Dauer unter einer Minute)

0.9.10

- Zuerst MCC entpacken
  - tar -xzvf curry-0.9.10.tar.gz
- Nun empfiehlt es sich, MCC nach `/etc` zu verschieben und in das entsprechende Verzeichnis zu wechseln mit
  - `sudo mv curry-0.9.10 /etc/`
  - `cd /etc/curry-0.9.10/`
- Abschließend kann Curry installiert werden mit den Befehlen
  - `sudo ./configure`
  - `sudo make`
  - `sudo make install`

Nun kann der Muenster Curry Compiler mit dem Befehl `cyi` aufgerufen werden. Ein Eintrag in die Datei `.bashrc` im Home-Verzeichnis ist überflüssig, da `make` schon die Pfadvariable einträgt.

AquaCurry

Zusätzlich zu den Implementierungen für Linux wurde von Wolfgang Lux eine Entwicklungsumgebung (`integrated development environment`, IDE) speziell für Mac OS unter dem Namen AquaCurry entwickelt. Diese steht unter http://danae.uni-muenster.de/~lux/curry/download/AquaCurry/ zur Verfügung und ist mit Version 1.0 aktueller als die Linux-Version.

AquaCurry bezeichnet die Entwicklungsumgebung, die standardmäßig die aktuelle Version des MCC enthält.

AquaCurry ist leider nur unter Mac OS lauffähig. Für die Entwicklung ist es viel komfortabler als ein normaler Texteditor und eine Kommandozeile. Das zu Grunde liegende Basispaket von AquaCurry ist jedoch dasselbe wie MCC unter Linux.

Erlang

Erlang ist eine Sprache, die etwa alle zwei Monate in einer neuen Version erscheint. Die zugehörige Bibliothek trägt den Namen OTP. Für gewöhnlich sind nur beide Teile zusammen verfügbar. Eine Ausnahme bildet etwa Erlang im Embedded-Bereich. Hier wird auf nicht benötigte Funktionen verzichtet.

Die letzte grundlegende Veränderung war in Version R10B10 die Unterstützung von mehreren Prozessoren. Wie in Abbildung 2.1 ersichtlich, ist es in Erlang möglich, Quelltext nach C zu kompilieren, Erlang über eine abstrakte Maschine interpretieren zu lassen oder über `High Performance Erlang` (HiPE) direkt in Assembler zu kompilieren.

Wie die Abbildung 3.1 zeigt, lässt sich durch die Verwendung von HiPE die Performanz unter bestimmten Bedingungen um das bis zu achtfache steigern [17, 30]. HiPE ist ein Projekt der `Uppsala Universitet` in Schweden und unter http://www.it.uu.se/research/group/hipe/ erreichbar. Das Besondere an HiPE ist, dass der Interpreter BEAM nicht verwendet wird. Auch wird nicht auf sogenannte `Low-Level-Languages` zurückgegriffen. HiPE übersetzt direkt in Maschinen-Code und optimiert dadurch die Performanz. HiPE ist seit Oktober 2001 in das Paket Erlang/OTP integriert.

In derselben Forschungsabteilung entstand auch `DIALYZER` (DIscrepancy AnaLYZer for ERlang programs), ein Werkzeug, das

- Typ-Fehler,
- unerreichbaren Code,
- redundante Tests und
- unsicheren Byte-Code der virtuellen Maschine

findet. Dieses Werkzeug wurde auch zur Fehlerfindung in Version R9 eingesetzt. Dialyzer ist aktuell in der Version 1.4.0 unter http://www.it.uu.se/research/group/hipe/dialyzer/ verfügbar und setzt ein gepatchtes Erlang in Version R10B10 voraus.

HiPE wird im Folgenden nicht weiter betrachtet, da für die vorliegende Arbeit BEAM verwendet wurde. Einer der Vorteile von Erlang ist, dass es über einen Interpreter verfügt. Um die Vergleichbarkeit mit Java in Kapitel 4.4 zu gewährleisten, wurde auf die Verwendung von HiPE verzichtet.

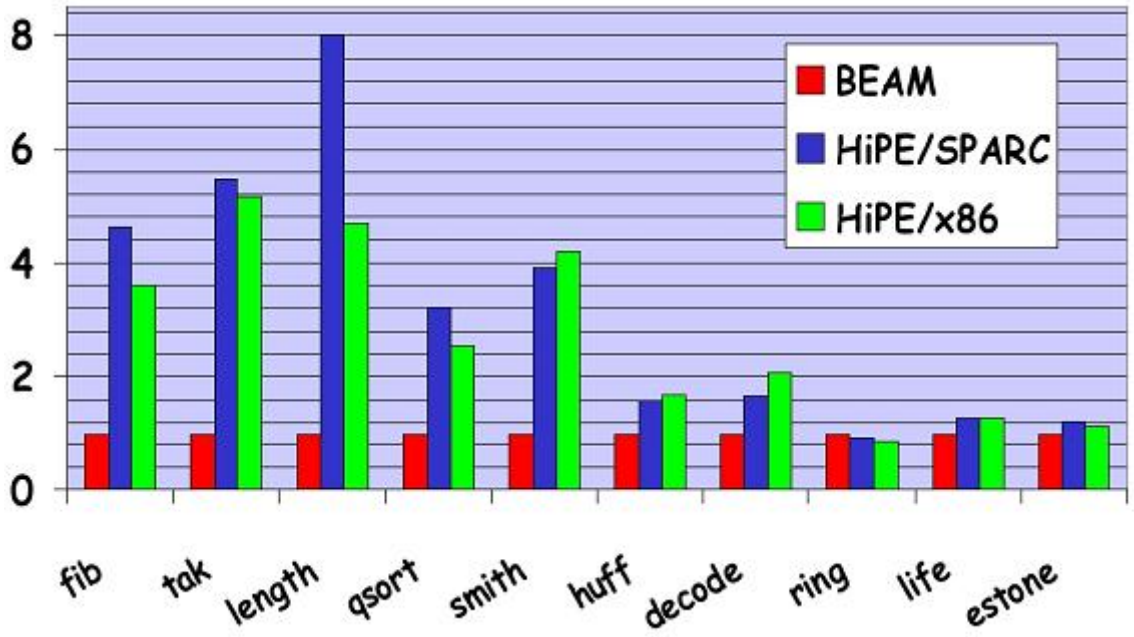

Abbildung 3.1: Performanz von BEAM und HiPE im Vergleich unter Berücksichtigung verschiedener Algorithmen und Architekturen [17, 30]

Eine englische Installationsanleitung ist unter http://www.erlang.org/doc/doc-5.5/doc/installation_guide/part_frame.html verfügbar.

Windows

Die Installation unter Windows ist sehr einfach. Die aktuelle Version ist unter http://www.erlang.org/download.html zu beziehen. Die ausführbare Datei ruft eine Setup-Routine auf, die den Benutzer durch das selbsterklärende Installationsmenü führt. Anschließend lässt sich der Erlang (BEAM) Emulator über

Start- Programme - Erlang OTP R*B - Erlang

aufrufen, wobei * für die installierte Versionsnummer steht.

Linux

Für die Installation von Erlang unter Linux werden nur ausreichend Platz und entsprechende Benutzer-Rechte vorausgesetzt. Folgende Schritte sind notwendig:

- Erstellen des Installationsverzeichnisses durch den Befehl
  - `sudo mkdir /usr/local/erlang`
- Herunterladen der Installationsdateien von http://erlang.org/download.html aus der Spalte `Source` und entpacken in das erstellte Verzeichnis durch den Befehl
  - `sudo tar -xzvf otp_src_R*.tar.gz /usr/local/erlang` (das * steht für die entsprechende Version)
- Installieren der Distribution (Eventuelle Änderungen sind unter `/usr/local/erlang/README` vermerkt)

  - Wechseln in das Verzeichnis mit `cd /usr/local/erlang/`
  - `sudo ./Install /usr/local/erlang/`
  - Während der Installation werden dem Benutzer Fragen betreffend der Umgebung gestellt. Für gewöhnlich ist es der Installationsroutine möglich, alle Parameter selbständig zu ermitteln. Dennoch wird jeder Parameter zur Sicherheit vom Benutzer abgefragt. Um die Abfrage zu beschleunigen, sind wahrscheinliche Parameter in eckigen Klammern vorgegeben, die einfach mit Enter bestätigt werden können.
  - Auf die Frage, ob eine minimale Installation oder SASL gewünscht wird, reicht es aus, wenn der Benutzer mit „minimal“ antwortet. Alle für diese Arbeit benötigten Pakete sind in der minimalen Version vorhanden. SASL ist ein Framework. Das Akronym steht für `Simple Authentication and Security Layer`. Es wird zur Authentifizierung im Internet eingesetzt.

- Abschließend ist
  - entweder der Pfad in die Pfad-Variable einzutragen, indem folgende Zeile
    `export PATH=$PATH:/usr/local/erlang/bin/`
    im Home-Verzeichnis in der Datei .bashrc ergänzt wird,
  - oder der Befehl
    `ln -s /usr/local/erlang/bin/erl /usr/local/bin/erl`

  auszuführen. Damit ist die Installation abgeschlossen.

Eine Installation über einen Paket-Manager ist nicht empfehlenswert, da oft nur alte Versionen in den Repositories vorhanden sind. APT bietet beispielsweise die Version R10B7 vom 31.08.2005 an. In dieser Arbeit wurden Tests mit aktuelleren Versionen durchgeführt.

Für die Nutzung eine Clusters muss für die Sicherheit der Kommunikation ein Cookie eingerichtet werden. Es bietet sich an, die Installation auf einem Linux-Betriebssystem, das über Network File System (NFS) gebootet wird, durchzuführen. Dadurch muss Erlang nur einmal installiert werden und auch der Cookie ist nur einmal einzurichten. Für das Erstellen eines Cookies genügt es, die Datei `.erlang.cookie` im Home-Verzeichnis zu erstellen. In diesen muss nun noch eine beliebige Phrase hineingeschrieben werden, die ausreichend lang sein sollte.

Unter Windows empfiehlt es sich, die Datei `.erlang.cookie` sowohl im Verzeichnis `C:\Windows` als auch im Benutzerverzeichnis
`C:\Dokumente und Einstellungen\Benutzername` zu erstellen. Auf die Datei in `C:\Windows` wird zurückgegriffen, wenn Erlang als System-Service gestartet wird. Andernfalls wird auf die Datei im Benutzerverzeichnis zurückgegriffen. Um sicher zu gehen, sollten beide Dateien mit dem gleichen Inhalt erstellt werden.

Die Phrase, welche die Datei `.erlang.cookie` enthält, muss auf allen Rechnern identisch sein, um die Kommunikation zu gewährleisten. Unter Umständen kann es zu Konflikten kommen, wenn unterschiedliche Versionen von Erlang auf den einzelnen Rechnern installiert sind. Daher empfiehlt es sich, dieselbe Version auf allen Rechnern zu benutzen, unabhängig davon, ob sie unter Windows oder Linux laufen.

Um Erlang kommunikationsbereit zu starten, muss der Befehl

```
erl -sname NAME
```

aufgerufen werden. Sollte entweder kein Cookie vorhanden sein oder Erlang ohne entsprechende zusätzliche Parameter gestartet werden, ist eine Kommunikation wie sie für dieses Projekt vorgesehen ist, nicht möglich.

Außerdem wird für das Ausführen des Benchmarks OpenSSL vorausgesetzt. Dies kann für Linux und Windows unter http://www.openssl.org/ bezogen oder unter Linux über einen Paket-Manager wie APT mit dem Befehl

```
sudo apt-get install openssl
```

installiert werden. Für Windows steht unter der genannten Adresse eine Installationsroutine zur Verfügung.

Mercury

Zur Zeit ist die stabile Version 12.2 vom 05.02.2006. Von Version 13.0 gibt es eine Beta-Version vom 03.04.2006 (ftp://ftp.mercury.cs.mu.oz.au/pub/mercury/beta-releases/0.13.0-beta/mercury-compiler-0.13.0-beta-2006-04-03-unstable.tar.gz) sowie eine `release-of-the-day`-Version (ROTD), die aktuell vom 22.05.2006 unter ftp://ftp.mercury.cs.mu.oz.au/pub/mercury/beta-releases/rotd/mercury-compiler-rotd-2006-05-22-unstable.tar.gz erreichbar ist.

Linux

Für die Installation wird ein GNU C Compiler (GCC) benötigt. Der Quelltext kann unter http://www.gnu.org/software/gcc/releases.html heruntergeladen oder über APT mit dem Befehl

```
sudo apt-get install gcc
```

installiert werden.

Bei der Installation des GCC ist darauf zu achten, Version 3.4.X zu installieren. Auf der Internetseite von Mercury wird davor gewarnt, andere Versionen zu verwenden. Wie aus der Mailingliste (betreffend Mails vom 10.05.2006) hervorgeht, kann es jedoch auch mit der Version 3.4.6 des GCC zu Komplikationen kommen. Hier wird als Lösung die Verwendung von Version 3.3.5 vorgeschlagen.

Bei der Installation für dieses Projekt wurde die ROTD-Version von Mercury mit GCC 3.4 verwendet.

Außerdem wird eine aktuelle Version von GNU Make in Version 3.69 oder höher vorausgesetzt. Diese lässt sich entweder über Paket-Manager installieren oder von http://ftp.gnu.org/pub/gnu/ beziehen.

Die Installation von Mercury ist einfach und ist für Version 12.2 auf Englisch unter ftp://ftp.mercury.cs.mu.oz.au/pub/mercury/mercury-INSTALL-0.12.2.txt verfügbar.

- Zuerst die gewünschte Version besorgen und entpacken.
  - Herunterladen der gewünschten Version von ftp://ftp.mercury.cs.mu.oz.au/pub/mercury/ und

  - entpacken mit dem Befehl
    `sudo tar -xzvf mercury-compiler-0.*.*.tar.gz /usr/local/mercury/`,
    wobei * für die Version steht.

- Nun kann Mercury installiert werden mit den Befehlen
  - `sudo ./configure` (Dauer unter einer Minute)
  - `sudo make` (Dauer etwa 15 Minuten)
  - `sudo make install` (Dauer etwa sieben Stunden!)
- Jetzt lassen sich optional noch die Pfade setzen. Dazu ist die Datei .bashrc um folgende Einträge zu Ergänzen, wobei * durch die jeweilige Versionsnummer zu ersetzen ist:
  - `export PATH=$PATH:/usr/local/mercury-0.*.*/bin`
  - `export MANPATH=$PATH:/usr/local/mercury-0.*.*/man`
  - `export INFOPATH=$PATH:/usr/local/mercury-0.*.*/info`
- Abschließend sollte der Befehl `make clean` ausgeführt werden, um das System von unnötigem Ballast zu befreien.

Windows

Unter Windows gibt es zwei unterschiedliche Möglichkeiten, Mercury zu installieren. Zum Einen lässt sich der Quelltext unter Cygwin kompilieren. Cygwin liefert sowohl make als auch GCC mit, so dass die Installation dieselben Schritte wie unter Linux benötigt.

Es steht jedoch auch eine Installationsroutine bereit, die den MMC in die DOS-Shell integriert. Volle Funktionalität gewährleisten die Entwickler für Windows XP. In Rahmen dieses Projekts wurde der MMC unter Windows 2000 ohne Komplikationen verwendet.

4. Vergleich

4.1 Erlernbarkeit

Bei dem Vergleich der Erlernbarkeit der drei betrachteten Sprachen Curry, Erlang und Mercury werden sowohl subjektive als auch objektive Kriterien Einfluss nehmen. Erlernbarkeit lässt sich nur schwer quantifizieren. Es lässt sich die Anzahl an aufgewendeten Stunden angeben, aber dabei würden Faktoren wie Motivation oder Vorkenntnisse unberücksichtigt bleiben.

Verglichen werden auch die offiziellen Lehrmaterialien, der Nutzen der jeweiligen Mailinglisten sowie inoffizielle Anleitungen.

Curry

Um Curry zu lernen, stehen zwei Hauptwerke kostenlos online bereit. Zum Einen gibt es von Sergio Antoy und Michael Hanus ein Lehrbuch, das unter dem Titel

`Curry - A Tutorial Introduction`

am 03.11.2004 erschienen ist [44]. Dies Buch enthält auf 75 Seiten eine übersichtliche und prägnante Einführung in die Sprache und leitet den Leser mit vielen einleuchtenden Beispielen an. Dabei ist an Vorbildung nur ein grundsätzliches Verständnis der Informatik notwendig. Vorkenntnisse in der Sprache Haskell können jedoch sehr hilfreich sein. Insgesamt neun Übungsaufgaben motivieren den Leser über das gesamte Buch und vertiefen das Verständnis. Ein weiterer positiver Aspekt ist, dass Beispiellösungen online verfügbar sind.

Ein Nachteil des Buches ist, dass es mit eineinhalb Jahren schon relativ alt und nicht aktualisiert ist. Während die Implementierung des PAKCS (siehe Kapitel 2.3.2) etwa alle zwei Monate in einer neuen Version erscheint, wurde das Handbuch schon zu lange nicht überarbeitet. Für den Einstieg in Curry ist das jedoch nicht wichtig, da sich die Syntax seit Erscheinen des Buches kaum geändert hat.

Der Muenster Curry Compiler (MCC) hingegen enthält mit dem

`Münster Curry User's Guide`

(siehe Kapitel 2.3.2) eine auf die aktuelle Version der Implementierung des MCC ausgelegte Anleitung, die zuletzt am 10.05.2006 aktualisiert wurde [51]. Dies von Wolfgang Lux geschriebene Buch umfasst auf 73 Seiten weniger eine Einführung als viel mehr technische Aspekte des Muenster Curry Compilers. Obwohl sehr viel exemplarischer Quelltext enthalten ist, lässt sich das Buch besser als Nachschlagewerk verwenden.

Eine weitere Quelle für Informationen ist die Mailingliste. Diese wird jedoch nur sehr spärlich genutzt. Nur wenige Male wurden Workshops oder neue Releases angekündigt. Fragen von Entwicklern oder ein Gedankenaustausch fanden dort nicht statt. Die Gemeinde der Benutzer von Curry ist relativ klein und so treten zur Anwendung kaum Fragen auf. Ein positiver Aspekt ist, dass die beiden deutschen Hauptentwickler Michael Hanus und Wolfgang Lux per E-Mail sehr informative Auskünfte geben und Anfragen schnell beantworten.

Curry ist auf jeden Fall eine Interesse weckende Sprache, die zur Verdeutlichung des multiparadigmatischen Ansatzes hilfreich ist. Da ausreichend Lehrmaterialien zur Verfügung stehen und die Syntax der von Haskell sehr ähnelt, ist das Erlernen nicht sehr aufwendig bei entsprechenden Vorkenntnissen. Ich habe etwa zwei bis drei Wochen benötigt, um Curry benutzen zu können.

Leider sinkt die Motivation nach dem Erlernen merklich ab, da von aktuellen Entwicklungen nichts bekannt wird und das Anwendungsgebiet sehr begrenzt ist (siehe Kapitel 4.3). Dies liegt vor allem an der kaum genutzten Mailingliste. Auf den Homepages der beiden Implementierungen sind kaum Informationen über den erfolgreichen Einsatz oder den Vergleich mit Implementierungen anderer Sprachen vorhanden.

Erlang

Zu Erlang ist bisher nur ein Lehrbuch offiziell erschienen.

`Concurrent Programming in Erlang, Second Edition`

erschien 1996 und umfasst 340 Seiten [3]. Leider ist nur der erste Teil des Buches online frei zugänglich (http://www.erlang.org/download/erlang-book-part1.pdf). Im ersten Teil sind jedoch alle notwendigen Informationen für ein grundsätzliches Verständnis der Sprache auf 205 Seiten enthalten. Die weiteren Kapitel sind für die Vertiefung in den Gebieten

- Datenbanken,
- Techniken für verteilte Systeme,
- verteilte Datenhaltung,
- Betriebssysteme,
- Realzeit-Kontrolle,
- Telephonie,
- Ein ASN.1 - Compiler,
- Graphik und

- Objekt-Orientierte Programmierung

gedacht. Das Buch eignet sich hervorragend für einen Einstieg, und so lange spezielles Wissen nicht benötigt wird, ist die freie Version völlig ausreichend. Ein Nachteil ist, dass das Buch seit zehn Jahren nicht aktualisiert wurde und daher teilweise schon veraltet ist. So konnte auf Eigenschaften und Funktionen, die später hinzukamen, wie beispielsweise Multiprozessorunterstützung und Verschlüsselung, nicht eingegangen werden. Da sich die Syntax jedoch seit Erscheinen des Buches nicht maßgeblich geändert hat, bleibt dies das Standardwerk. Vor einiger Zeit kam in der Mailingliste der Gedanke auf, ein aktuelles Buch zu verfassen. Dieser Gedanke wurde von den damaligen Autoren aufgenommen und nach einigen Tagen wieder verworfen.

Ein aktuelles Buch ist `Erlang programmation` von Mickaël Rémond [32]. Leider ist es nur in französischer Sprache erhältlich und konnte daher nicht genutzt werden.

Eine aktuelle Herangehensweise bietet Ericsson mit einem Tutorial unter http://www.erlang.org/doc/doc-5.4.13/doc/getting_started/part_frame.html [12]. Das Vorgehen ist vergleichbar dem des Buches `Concurrent Programming in Erlang` und auch die Quelltextbeispiele ähneln sich. Jedoch werden hier auch aktuelle Eigenschaften von Erlang berücksichtigt. Dies Tutorial kann sowohl als Ergänzung wie auch als Alternative zu dem Buch genutzt werden. Ein Nachteil besteht darin, dass es nicht so gut druckbar ist wie der erste Teil des Buches.

Eine weitere sehr gute Einführung, die nicht so umfangreich ist wie das Standardwerk, wurde von von Stephan Frank und Petra Hofstedt an der TU Berlin 2004 erstellt [39]. Auf den 60 Folien der Vorlesung wird eine Einführung in Erlang gegeben, die unter http://uebb.cs.tu-berlin.de/lehre/2004SPerlang/files/slides.16.4.pdf verfügbar ist. Sie lässt sich schnell durcharbeiten und gibt Aufschluss über die wichtigsten Funktionen einschließlich der Kommunikation in Erlang. Ein weiterer Vorteil ist, dass die Folien auf Deutsch sind und eine relativ aktuelle Version (R9B0) voraussetzen. In derselben Veranstaltungsreihe wurden noch weitere Vorlesungen gehalten, die aber spezieller sind und beispielsweise die Datenbank von Erlang (Mnesia) behandeln (siehe http://uebb.cs.tu-berlin.de/lehre/2004SPerlang/files/slides.23.4.pdf).

Auf der Homepage von Erlang (www.erlang.org) sind außerdem Projekte aufgelistet und als Quelltext verfügbar. Hier lassen sich Erlang-Quellen aus Projekten einsehen und Lösungen zu Problemstelllungen exemplarisch betrachten.

Die Mailingliste ist mit etwa 20 Mails pro Tag hoch frequentiert und Fragen werden oft noch am selben Tag beantwortet. Anfänger werden gut aufgenommen und einfache Fragen schnell geklärt. Hier werden neue Projekte wie die vorliegende Arbeit vorgestellt und der Verlauf bestehender Arbeiten wird kommentiert. Für Firmen gibt es ein breites kommerzielles Support-Angebot, das unter www.erlang.se offeriert wird.

Für das Erlernen der Grundlagen habe ich etwa zwei, für ein tieferes Verständnis sowie die Programmierung des Benchmarks fünf Wochen benötigt. Dabei nahm die Fehlersuche die meiste Zeit in Anspruch. An dieser Stelle ist der integrierte Debugger und eine Werkzeugleiste hervorzuheben, die sich von der Erlang-Shell aus mit den Befehlen

`debugger:start().` (Debugger)

`toolbar:start().` (Toolbar)

starten lassen. Diese werden in der Dokumentation nicht erwähnt, können aber beim Aufspüren von Fehlern sehr hilfreich sein.

Mercury

Als Lernmaterial ist von den Entwicklern von Mercury das Tutorial von Ralph Becket [5] vorgesehen. Auf 53 Seiten werden die Grundlagen anhand von Beispielen vermittelt. Für diese Einführung wird einiges Vorwissen aus dem Bereich der theoretischen Informatik wie etwa Herbrand-Universen vorausgesetzt. Das Tutorial bietet keinen tieferen Einblick in die Funktionsweise von Mercury und erklärt nur die Grundlagen der Sprache.

Für weitere Studien bieten sich die folgenden Dokumente als Nachschlagewerke an:

- Das Referenz-Handbuch [55]
- Das Benutzer-Handbuch [57]
- Die Bibliotheken-Referenz [54]
- Eine Transitionsanleitung zum Übersetzen von Prolog-Quelltexten in Mercury-Quelltext [56]
- Die FAQ [53]

Prolog-Vorkenntnisse können den Einstieg in Mercury erheblich erleichtern, da die Syntax in Mercury nahezu identisch ist. Für das Erlernen der Grundlagen habe ich ohne Vorkenntnisse von Prolog etwa eine Woche benötigt. Mit Vorkenntnissen wäre ein Verständnis weiterer Möglichkeiten, die Mercury bietet, leichter gewesen. Unter Umständen ist hier für die Vorbildung das Durcharbeiten eines Prolog-Tutorials, wie etwa das Arbeitsbuch Prolog [6], empfehlenswert.

Die Mailingliste ist mit etwa zwei Mails pro Tag nur wenig frequentiert. Das mag daran liegen, dass Mercury nicht dezentral entwickelt wird. Auf der Hompage http://www.cs.mu.oz.au/research/mercury/ sind weitere Informationen verfügbar.

Fazit

Für alle drei Sprachen sind ausreichend Lehrmaterialien vorhanden. Die Entwicklergemeinden von Erlang und Mercury sind offen und Fragen werden schnell von mehreren Mitgliedern beantwortet. Bei Curry beschränkt sich die Wahl der Ansprechpartner auf die Entwickler Wolfgang Lux und Michael Hanus, die ebenfalls schnell antworten, jedoch teilweise unterschiedliche Ansichten vertreten [24, 23, 19].

Das Maß Zeit ist hier kein Indikator für die Komplexität des Erlernens einer Sprache. Obwohl für das Erlernen von Erlang mehr Zeit investiert wurde, ist die Sprache schneller und leichter zu erlernen als Curry oder Mercury. Hier waren für die Programmieraufgabe tiefere Kenntnisse notwendig.

Vor dem Erlernen einer Sprache ist es in jedem Fall sinnvoll, sich vorher Gedanken über ihr Paradigma beziehunsgweise ihre Paradigmen und deren Ausprägung zu

machen, um sich auf die gewünschten Eigenschaften konzentrieren zu können. Wenn ich mich auf den funktionalen Charakter von Erlang einstelle und dann mit dem logischen Charakter von Mercury konfrontiert werde, müssen zwangsweise Schwierigkeiten auftreten.

> Obwohl multiparadigmatische Sprachen dieselben Perspektiven anbieten können, sind sie oft grundlegend verschieden. Das macht sich besonders beim Erlernen bemerkbar.

4.2 Zukunftsperspektive

Die Zukunftsperspektive einer Programmiersprache ist von vielen Faktoren abhängig. Faktoren, die Berücksichtigung finden sind:

- Verbreitung
- Popularität
- Anwendungsgebiete
- Erlernbarkeit
- Performanz
- Konkurrenz
- akademische Bedeutung
- wirtschaftliche Bedeutung

Es lässt sich nicht allgemein sagen, ob eine Sprache eine durchweg positive oder negative Zukunft haben wird. Ich versuche dennoch, die Zukunft der aufgeführten Programmiersprachen differenziert einzuschätzen.

Curry

Obwohl Curry global entwickelt wird, ist die Sprache noch nicht sehr verbreitet. Weitere Projekte wie beispielsweise `TEABAG: A DEBUGGER FOR CURRY` [31] von Stephen Lee Johnson, das an der Portland State University entwickelt wurde, sind zwar bezeichnend für den globalen Charakter, jedoch beschränkt sich die Enwicklung der Sprache an sich auf Deutschland. Die Entwicklung der beiden Distributionen findet in Münster und Aachen statt.

Curry wird fast nicht an Universitäten gelehrt, obwohl es sich um eine akademische Sprache handelt. Sie eignet sich sehr gut, Studenten das logisch-funktionale Paradigma näher zu bringen. Es können exemplarisch Entwurfsentscheidungen beim Entwickeln der Sprache mit Studenten diskutiert werden.

Diese Sprache wird eher selten eingesetzt. Von den Entwicklern wird sie für Web-Anwendungen und e-Learning-Systeme verwendet.

Die Dokumentation könnte reichhaltiger sein. Wie in Kapitel 4.1 beschrieben gibt es zwar ein gutes Lehrbuch und ein gutes Nachschlagewerk, aber alternative Literatur ist nicht vorhanden. Der Vorteil für die Zukunft liegt darin, dass Curry als Erweiterung von Haskell eine Nische unter den logisch-funktionalen Sprachen ausfüllt und daher keine Konkurrenz hat.

Wissenschaftlich ist die Sprache sehr interessant. Gerade die Umsetzung in zwei grundsätzlich unterschiedlichen Distributionen verdeutlicht, wie unterschiedlich Paradigmen für die Entwicklung einer Sprache sein können. In der Wirtschaft wird diese Sprache wohl in naher Zukunft keine Bedeutung erlangen, da sie

- zum Einen noch zu unpopulär ist und auch in der Vergangenheit im Vergleich zu anderen Sprachen sich nicht nennenswert weiter verbreitet hat
- und zum Anderen, da sie noch zu unausgereift ist.

Obwohl die Entwickler sehr engagiert sind, fließt im Vergleich zu anderen Sprachen zu wenig Entwicklungszeit in die Distributionen. Die Sprache entwickelt sich zu langsam, als dass sie mit anderen Sprachen, die moderne Technologien zeitnah implementieren, schritthalten könnte. Als Beispiele sind hier eingebettete Funktionen für

- Kryptographie und
- Kommunikation

zu nennen. Ein kommerzieller Einsatz wäre für die Verbreitung und Weiterentwicklung der Sprache sicher hilfreich und würde die Möglichkeiten, die Curry bereitstellt, betonen.

Erlang

Erlang ist mittlerweile weltweit verbreitet. Die Sprache kommt in einer Version für eingebettete Systeme in Mobil-Telefonen von Sony-Ericsson zum Einsatz und die Benutzer der Mailingliste sind weltweit verteilt. In der Lehre wird derzeit über die Mailingliste eine Sammlung aller Universitäten erstellt, an denen Erlang gelehrt wird (siehe http://www.erlang.org/faq/t1.html#AEN94).

Neben Mobiltelefonen sind Telefonieanwendungen und Routingsysteme Anwendungsgebiete.

Erlang bietet viele Möglichkeiten, Quelltext in Assembler zu übersetzen beziehungsweise zu interpretieren. Mit HiPE (siehe Kapitel 3.3) existiert ein Projekt, das bei der Übersetzung von Quelltext Wert auf Performanz legt, wobei allerdings die Vorteile eines Interpreters verloren gehen.

Da das Angebot an Funktionen groß ist und sich Erlang für Anwendungen, die auf eine massive Parallelisierbarkeit und Kommunikation ausgelegt sind, eignet, gelingt es der Sprache, sowohl eine Nische unter den logisch-funktionalen Sprachen zu beanspruchen durch ein Hervorheben des funktionalen Charakters, als auch gegen etablierte Sprachen als Alternative zu bestehen durch Spezialisierung in den Bereichen

des Constraint und Distributed Programming. Dies wird auch durch Untersuchungen unterstützt, wie etwa `Yet another Web-Server` (YaWS) gegen XAMP (Linux oder Windows mit Apache, MySQL und PHP) [36, p. 6] oder Erlang gegen Haskell [37]. In der Wirtschaft wird Erlang wohl gute Chancen haben, sich im eingebetteten Bereich zu etablieren, wenn Werkzeuge wie etwa Stateflow verfübar sein sollten. Im `high-level-language`-Bereich hingegen sind andere Sprachen wie etwa Java oder C zu weit verbreitet, als dass sich Erlang mit ihnen an Popularität messen könnte. Akademisch bietet sich Erlang zum einen als Primärsprache an, mit der grundlegende Konzepte verständlich vermittelt werden können. Zum Anderen lassen sich allgemeine Konzepte komplexerer Algorithmen im Hauptstudium greifbar erklären. Gerade im Bereich Rechnernetze lassen sich Kommunikation und verteiltes Rechnen anschaulich erläutern. Dabei ist es möglich, exemplarisch die Kommunikation zu einer anderen Sprache, wie etwa Python, darzulegen, um so das Prinzip offener Schnittstellen zu demonstrieren.

Mercury

Mercury ist nicht sehr weit verbreitet. Es wird an der Melbourne University entwickelt und nicht kommerziell eingesetzt. Im akademischen Bereich sind die Projekte

- `HAL` (http://www.csse.monash.edu.au/~mbanda/hal/index.html) (siehe Kapitel 2.3.8),
- `The Aditi Deductive Database` (http://www.cs.mu.oz.au/research/aditi/) sowie
- `Animating Z Using Mercury` (http://goanna.cs.rmit.edu.au/~winikoff/pipe/index.html)

zu nennen.

Die Grundlagen werden in dem Tutorial von Ralph Becket sehr gut erklärt, auch wenn mehr Literatur hilfreich wäre. Weiterführende Informationen, die über die Grundlagen hinausgehen, könnten leichter zugänglich sein und sind in der Dokumentation teilweise nur schwer zu finden. Da die Sprache nicht auf Performanz ausgerichtet ist, spielt dieser Vergleichspunkt keine Rolle für Mercury. Jedoch schafft sie es, da sie sehr durch das logische Paradigma sowie die Sprache Prolog geprägt ist, im Bereich der logisch-funktionalen Sprachen ein eigenes Gebiet zu beanspruchen.

Für die Lehre hat Mercury als Weiterentwicklung von Prolog eine akademische Bedeutung.

4.3 Anwendungsgebiete

Curry

Curry wird primär im akademischen Sektor und für Web-Anwendungen eingesetzt. Die Ähnlichkeit mit Haskell bringt einige Annehmlichkeiten wie vorhandene Bibliotheken mit sich, die auch unter Curry nutzbar sind, was das Feld potentieller Anwendungen erweitert.

Einige Vorteile von Curry sind

- kurzer, lesbarer Quelltext, was `rapid prototyping` ermöglicht, sowie
- Typsicherheit.
- Curry ist fehlerunanfällig und
- robust.

Curry eignet sich eher für Anwendungen mit hoher Anforderung an Zuverlässigkeit und weniger für Anwendungen mit hohen Anforderungen an Effizienz, wie beispielsweise Realzeitanforderungen [38]. Da Curry zu den `very high level programming languages` gehört, sind in relativ kurzer Zeit Ergebnisse dank `rapid prototyping` vorweisbar.

Außer an Hochschulen wird Curry nach aktuellem Kenntnisstand nicht eingesetzt. In der akademischen Forschung spielt Curry jedoch gerade im Bereich der logisch-funktionalen Sprachen eine bedeutende Rolle, wie aus einem Bericht von

```
The 10th ACM SIGPLAN International Conference
on Functional Programming (ICFP 2005)
```

hervorgeht, der unter http://www.informatik.uni-kiel.de/~mh/wcflp2005/ verfügbar ist.

Curry wird von den Entwicklern für Web-Anwendungen und e-Learning-Systeme eingesetzt [19].

Erlang

Erlang ist nicht nur sehr effizient sondern genügt sogar sanften Realzeitanforderungen. Auf der Homepage (http://www.erlang.org) wird das Anwendungsgebiet folgendermaßen umschrieben:

> Erlang is designed for building high-availability systems that survive software crashes.

Da Erlang für verteilte Anwendungen ausgelegt ist, sind redundante Datenhaltung sowie Bereitstellung von Ressourcen relativ leicht realisierbar. So wird an einer polnischen Hochschule gerade an der Entwicklung eines Clusters gearbeitet, um Ressourcen für ein Massively Multiplayer Online Role-Playing Game (MMPORPG) zu schaffen (siehe http://www.worldforge.org/ und http://www.erlang-fr.org/en-area/index.html). Diese Art der Anwendung ist typisch für Erlang, da viele Daten kaskadiert und gleichzeitig vielen Verbrauchern zugänglich sein müssen, was ein Rechner allein nicht bewerkstelligen könnte. Außerdem ist eine Anforderung, dass so ein System `verfügbar` sein muss. Das bedeutet, dass die Wahrscheinlichkeit der Nichtverfügbarkeit des entsprechenden Dienstes wegen eines Clusterzusammenbruchs auf Grund von Software-Abstürzen oder Ausfall einer Internetanbindung hinreichend gering ist und Ausfälle durch Redundanz aufgefangen werden können. Weitere Informationen sind unter http://www.erlang.se/euc/05/Slaski.pdf einsehbar.

Erlang ist speziell für den Einsatz in Telekommunikationssystemen wie etwa für das Steuern eines Switches entwickelt worden. Der Einsatz internetbasierter Anwendungen wie Mail Transfer Agents (MTA) oder als IMAP-4 Server oder WAP-Stacks (http://www.erlang.se/euc/99/WAP/index.htm) wurden schon mit Erlang

realisiert. Da Ericsson als entwickelnde Firma gerade im Bereich mobiler Systeme in Zusammenarbeit mit Sony auf dem Markt präsent ist, sind natürlich Anwendungen für mobile Geräte entwickelt worden, die für den Benutzer nicht sichtbar in das Betriebssystem integriert sind.

Mit Mnesia wurde ein Datenbanksystem entwickelt, das sanften Realzeitanforderungen standhält und verteilt nutzbar ist, um redundante Datenhaltung zu gewährleisten.

Darüber hinaus besticht Erlang durch Performanz, wodurch der Einsatz eines Webservers mit Yet another Webserver (YaWs) für hochfrequentierte Dienste günstiger erscheinen kann der Einsatz populärer Technologien wie XAMP (Linux oder Windows mit Apache, MySQL und PHP) wie Joe Armstrong in seiner Ausarbeitung (http://www.sics.se/~joe/apachevsyaws.html, [36]) belegt.

Erlang hat in der Wirtschaft eine Bedeutung. So setzen Firmen wie

- Bluetail,Alteon,Nortel (verteiltes, Fehler-tolerantes E-Mail System, SSL Beschleuniger),
- Cellpoint (Lokal-basierte Mobile Dienstleistungen),
- Corelatus (SS7 monitoring),
- Deutsche Flugsicherung (SNMP Agent zur Kontrolle des Air Traffic Management System),
- Finnish Meteorological Institute (Data acquisition und Realzeit monitoring),
- IDT corp. (Realzeit least-cost Routing Expertensystem),
- Mobilearts (GSM und UMTS Dienstleistungen),
- Netkit Solutions (Netzwerküberwachung und Operations Support Systems),
- Schlund + Partner (Messaging und interaktive Voice Response Services),
- T-Mobile (Erweiterte Anruf-Steuerung),
- Telia und
- Vail Systems (Interaktive Voice Response Services)

die Sprache Erlang erfolgreich in kommerziellen Systemen ein. Ericsson hat sich mittlerweile von Erlang distanziert und das damals entwickelnde Labor geschlossen. Auf der Homepage (http://www.erlang.org/faq/x1088.html#AEN1102) nehmen die aktuellen Entwickler dazu Stellung.

Mercury

Mercury hat, ähnlich wie Curry, in der freien Wirtschaft noch keine bedeutende Rolle. Als funktionale Weiterentwicklung der logischen Sprache Prolog ist die Entwicklung im logisch-funktionalen Paradigma gerade im Vergleich zu anderen Sprachen des logisch-funktionalen Paradigmas aufschlussreich. Als zu Prolog verwandte Sprache könnte Mercury gut in den Bereichen

- künstliche Intelligenz und
- Expertensysteme

eingesetzt werden. Die Entwickler zielen darauf ab, Nachteile rein logischer Sprachen durch den funktionalen Aspekt auszugleichen und somit eine Sprache zu schaffen, die

> die Reinheit und Ausdruckskraft der deklarativen Programmierung mit erweiterter statischer Analyse und Fehlererkennung kombiniert. (http://www.cs.mu.oz.au/research/mercury/)

So ist Mercury derzeit besonders unter wissenschaftlichen Gesichtspunkten bedeutend und wird wahrscheinlich erst an Popularität gewinnen müssen, um wirtschaftlich Bedeutung zu haben.

4.4 Performanz

Um zu demonstrieren, dass logisch-funktionale Sprachen an Performanz imperativen Sprachen nicht zwangsläufig unterlegen sind, wurde ein Benchmark-Algorithmus in Erlang und Java implementiert. Dabei wurden nur enthaltene Klassen zur Netzwerkkommunikation und Ver- beziehungsweise Entschlüsselung verwendet. Die Quellen sind auf der beiliegenden CD enthalten, und zwar in den Ordnern:

- `/erlang/bench/bench4.erl` (SingleCPU Benchmark Erlang)
- `/erlang/distributed_bench/final.erl` (ClusterCPU Benchmark Erlang)
- `/java/bench/bench1.java` (SingleCPU Benchmark Java)
- `/java/db3/Client.Java` (ClusterCPU Client Benchmark Java)
- `/java/db3/MultiServer.Java` (ClusterCPU Server Benchmark Java)

Während in Erlang die Verwendung der Kryptographie- und Kommunikationsfunktionen eindeutig festgelegt ist, stehen in Java mehrere Möglichkeiten zur Verschlüsselung und Kommunikation zur Verfügung. Zuerst wurde versucht, mit RMI die gewünschte Kommunikation nach dem Vorbild Erlangs zu realisieren.

`Remote Method Invocation` (RMI) sieht vor, dass `Clients` die Ressourcen eines entfernten Rechners, des `Servers`, durch das Aufrufen von Methoden nutzen. Dabei

verhält sich der `Client` als `Master` und der `Server` als `Slave`. Für die Implementierung des Benchmark-Programms ist jedoch eine andere Architektur notwendig, in welcher der `Server` in der Rolle des `Masters` viele `Clients` als dienstanbietende `Slaves` steuert.

Diese Technologie erwies sich daher als unpassend und so wurde das Benchmark-Verfahren mit Hilfe von Sockets in Java realisiert. Dazu griff ich auf die Beispielsquellen aus dem Buch `Concurrent and Distributed Computing in Java` [15, p. 97ff] zurück.

Die Benchmark-Applikationen sind in den entsprechenden Quelltexten dokumentiert. Mit ErlDoc beziehungsweise JavaDoc lässt sich eine Dokumentation in HTML generieren.

Wichtig ist an dieser Stelle erst einmal die Auswahl der Testverfahren. Folgende Vergleiche wurden durchgeführt:

1. Erlang vs. Java auf einem Single CPU Debian-System, siehe Abbildung 4.2
2. Erlang vs. Java auf einem Single CPU Windows-System, siehe Abbildung 4.3
3. Linux vs. Windows Single CPU Erlang, siehe Abbildung 4.4
4. Linux vs. Windows Single CPU Java, siehe Abbildung 4.5
5. Erlang auf einem Multi CPU Debian-Systeme (1-2-4-8 CPUs im Vergleich), siehe Abbildung 4.6
6. Java auf einem Multi CPU Debian-Systeme (1-2-4-8 CPUs im Vergleich), siehe Abbildung 4.7

Die zur Erstellung dieser Diagramme ermittelten Daten sind in Abbildung 4.1 aufgeführt. Mit den ersten beiden Tests (siehe Abbildungen 4.2 und 4.3) wird gezeigt, in welcher Zeitspanne die Sprachen mit Hilfe eines Brute-Force-Algorithmus den gesuchten Schlüssel finden können. Dabei wird noch keine Netzwerkkommunikation genutzt. Das zweite Testpaar (siehe Abbildungen 4.4 und 4.5) nutzt dieselben Ergebnisse, jedoch werden hier die verwendeten Betriebssysteme verglichen, um einen potentiellen fremden Einfluss seitens des Betriebssystems auf die Leistung der Sprachen ausschließen zu können.

Im dem dritten Testpaar (siehe Abbildungen 4.6 und 4.7) wird die Leistungssteigerung durch das Verwenden eines Clusters getestet - „wie viel Leistungssteigerung bringt der Einsatz von mehr Ressourcen?“ und „In wie weit lohnt sich der Einsatz eines Clusters für ein in Erlang entwickeltes System?“ sind hier die grundlegenden Fragestellungen.

Der gleiche Test wurde auch mit Java durchgeführt, um die Leistungssteigerung in Erlang und Java vergleichen zu können.

Um die resultierende Effizienz vergleichen zu können, musste zuerst ermittelt werden, ab wann die Messreihen in Bezug auf die Schlüssellänge Stabilität erreicht haben. Diese Berechnung wird in Abbildung 4.8 verdeutlicht.

Key	Key Java	Key Erlang
0	-128 -128 -128 -128...	...00 00 00 00
4096	-128 -112 -128 -128...	...00 00 10 00
8192	-128 -096 -128 -128...	...00 00 20 00
16384	-128 -064 -128 -128...	...00 00 40 00
32768	-128 -000 -128 -128...	...00 00 80 00
65536	-128 -128 -127 -128...	...00 01 00 00
131072	-128 -128 -126 -128...	...00 02 00 00
262144	-128 -128 -124 -128...	...00 04 00 00
524288	-128 -128 -120 -128...	...00 08 00 00
1048576	-128 -128 -112 -128...	...00 10 00 00
2097152	-128 -128 -096 -128...	...00 20 00 00
4194304	-128 -128 -064 -128...	...00 40 00 00
8388608	-128 -128 -000 -128...	...00 80 00 00
16777216	-128 -128 -128 -127...	...10 00 00 00

Keys (above) are incremented logarithmic **Time (below) is measured in seconds**

Single Erlang Linux	Single Erlang Windows	Erlang 2 PC	Erlang 4 PC	Erlang 8 PC
000,000042	000,000001	000,049976	000,001976	000,001922
000,014028	000,016000	000,018727	000,021316	000,011871
000,028382	000,031000	000,037032	000,026783	000,018940
000,062713	000,047000	000,057783	000,039947	000,024595
000,125338	000,110000	000,094407	000,054281	000,035046
000,248853	000,234000	000,163976	000,095416	000,054327
000,463744	000,453000	000,330650	000,171403	000,094505
000,936722	000,922000	000,602323	000,314799	000,173911
001,885849	001,797000	001,287821	000,640167	000,330153
003,751147	003,641000	002,428755	001,251328	000,639427
007,822381	007,594000	004,861475	002,546498	001,268243
015,614126	015,172000	009,935795	005,752700	002,506827
031,540584	031,515000	019,576745	010,660560	004,982471
063,252820	060,516000	039,165992	019,847959	010,594050

Singls Java Linux	Single Java Windows	Java 2 PC	Java 4 PC	Java 8 PC
000,540844	000,489289	000,009754	000,016206	000,016569
000,680144	000,607800	000,127358	000,081762	000,055310
000,728178	000,654043	000,168834	000,137126	000,080140
000,840250	000,756648	000,236713	000,173950	000,142585
001,024728	000,924199	000,339133	000,235050	000,164957
001,423540	001,289479	000,564639	000,344859	000,232329
002,130953	002,001201	001,039486	000,568966	000,347278
003,670110	003,425017	001,828937	000,989870	000,573592
006,656656	006,288909	003,696824	001,916876	001,049231
012,656735	011,984576	006,932223	003,564954	001,967374
024,541146	023,422153	013,404495	006,944955	003,738380
048,215258	046,181638	026,936289	014,413139	007,012879
096,788049	091,636533	053,795736	027,549481	013,620018
192,293265	182,884340	103,563880	054,973054	027,351988

Abbildung 4.1: Diese Tabelle gibt die Messergebnisse in Abhängigkeit der verwendeten Schlüssel an. In den oberen drei Spalten ist links der verwendete Schlüssel, in der Mitte der jeweils entsprechende Schlüssel in Java und rechts der entsprechende Schlüssel in Erlang abgebildet. Die Schlüssel werden zeilenweise logarithmisch inkrementiert. In Erlang und Java besteht ein Schlüssel aus 16 Blöcken, die jeweils Werte von 0 bis 255 beziehungsweise in Java von -128 bis 127 annehmen können. Aus Platzgründen sind hier nur die notwendigen Stellen aufgeführt. Ausgeschrieben sieht ein exemplarisch ausgeschriebener Schlüssel folgendermaßen aus:
`Java:` final byte[ ] keyBits = {(byte)-128, (byte)127, (byte)-128, (byte)-128,(byte)-128,(byte)-128, (byte)-128, (byte)-128,(byte)-128,(byte)-128, (byte)-128, (byte)-128,(byte)-128, (byte)-128,(byte)-128 };
`Erlang:`Key = <<16#00,16#00,16#00,16#00,16#00,16#00,16#00,16#00,16#00,16#00,16#00,16#00,16#01,16#00,16#00,16#00 >>
Darunter sind die Messergebnisse der einzelnen Tests für Erlang und Java aufgeführt. Diese Messergebnisse werden in Sekunden mit sechs Dezimalstellen angegeben.

Die Testreihen

Es wurden sechs Testreihen durchgeführt, die in drei Testpaare unterteilt sind. Anschließend wurde noch eine weitere Testreihe durchgeführt, um die Ergebnisse in Relation zu setzen.

Zuerst wurde die Performanz direkt verglichen. Dafür wurde auf verteiltes Rechnen verzichtet. In beiden Sprachen wurden nur mitgelieferte Klassen beziehunsgweise Module verwendet.

Erstes Testpaar: Java vs Erlang unter Linux und Windows

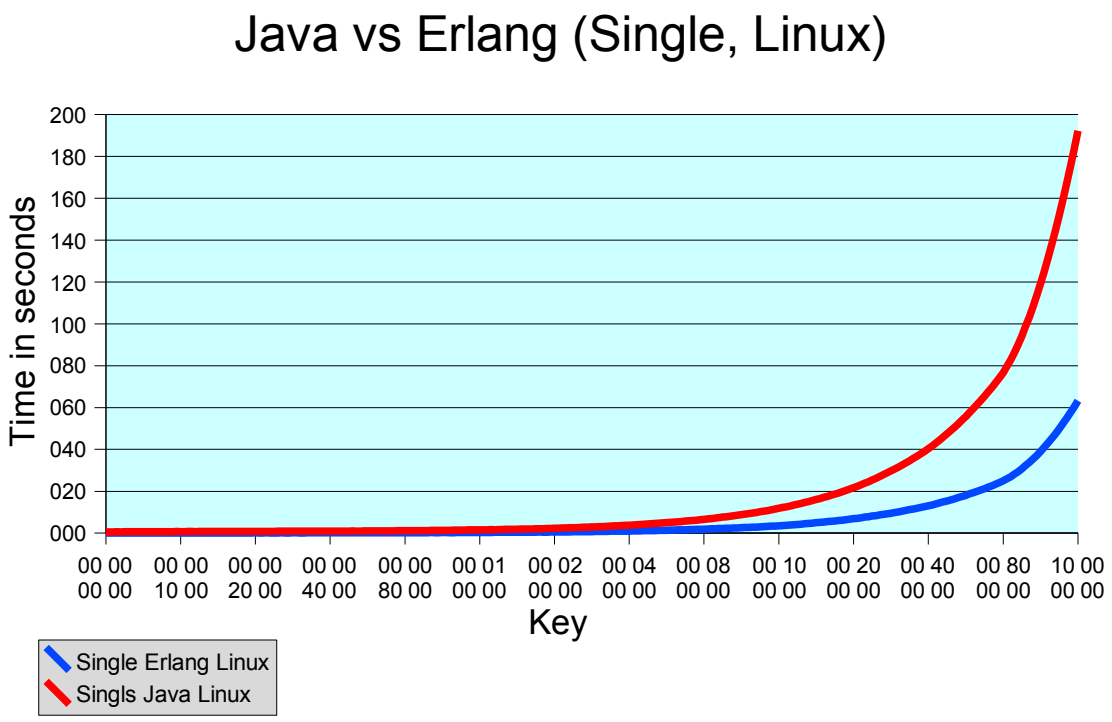

Abbildung 4.2: Diese Abbildung zeigt die gemessene Zeit in Abhängigkeit des zu findenen Schlüssels für die Sprachen Java (obere rote Kurve) und Erlang (untere blaue Kurve). Für die Ermittlung der Messwerte wurde ein Debian-System verwendet.

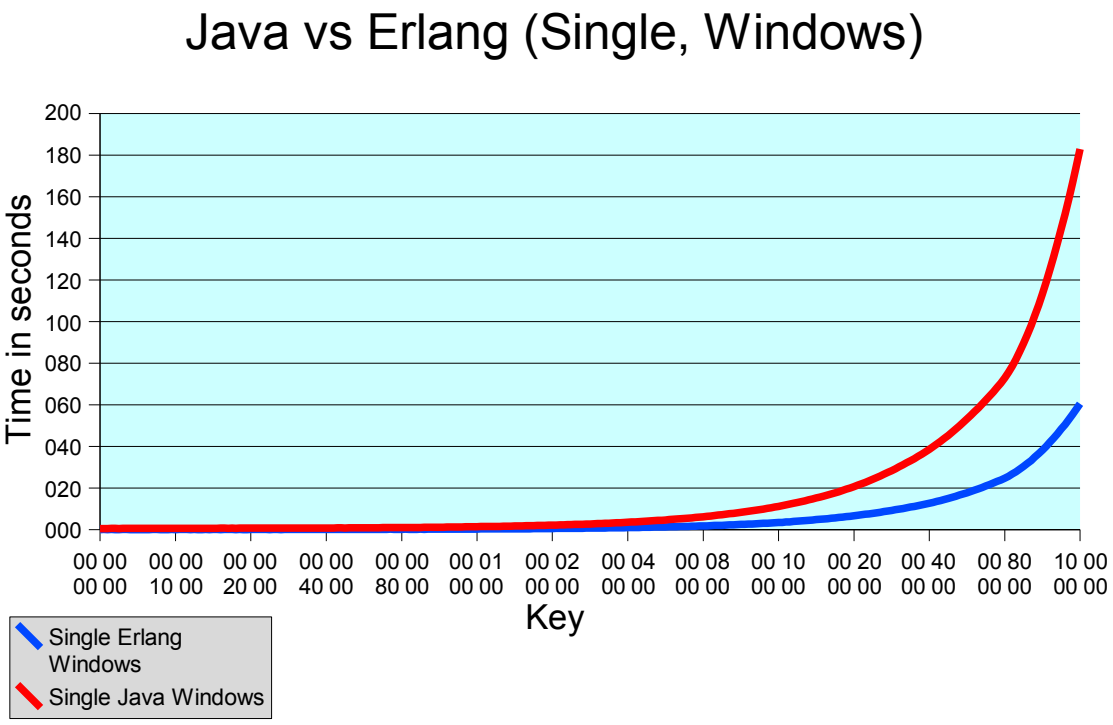

Abbildung 4.3: Diese Abbildung zeigt die gemessene Zeit in Abhängigkeit des zu findenden Schlüssels für die Sprachen Java (obere rote Kurve) und Erlang (untere blaue Kurve). Für die Ermittlung der Messwerte wurde ein Windows-System verwendet.

Wie aus diesen beiden Testreihen ersichtlich ist, benötigt Erlang nur etwa ein Drittel (genau 32,9% unter Linux, 33,1% unter Windows) der Zeit von Java. Damit ist gezeigt, dass Erlang für diese spezielle Aufgabe besser geeignet ist als Java.

Eine weitere interessante Frage ist, welche Rolle das Betriebssystem für die Performanz einer Sprache spielt.

Zweites Testpaar: Linux vs. Windows mit Erlang und Java

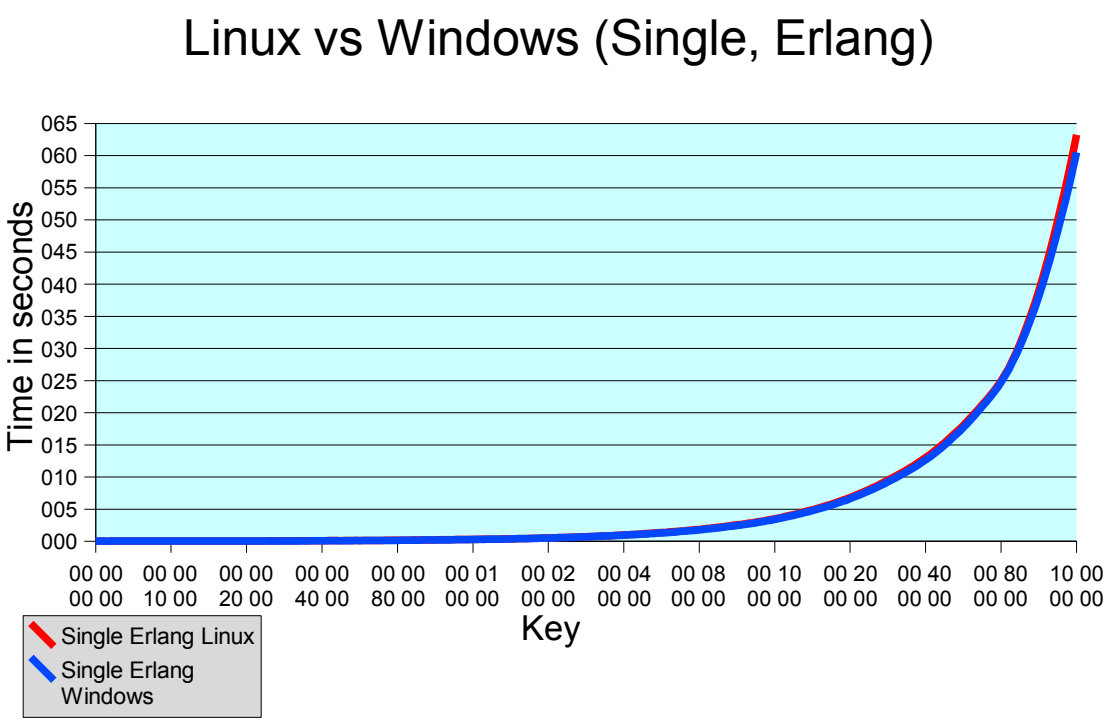

Abbildung 4.4: Diese Abbildung zeigt die gemessene Zeit für das Finden des gegebenen Schlüssels in Abhängigkeit des Betriebssystems für die Sprache Erlang. Für die Ermittlung der Messwerte wurden ein Debian-System (obere rote) Kurve sowie ein Windows-System (untere blaue) Kurve verwendet.

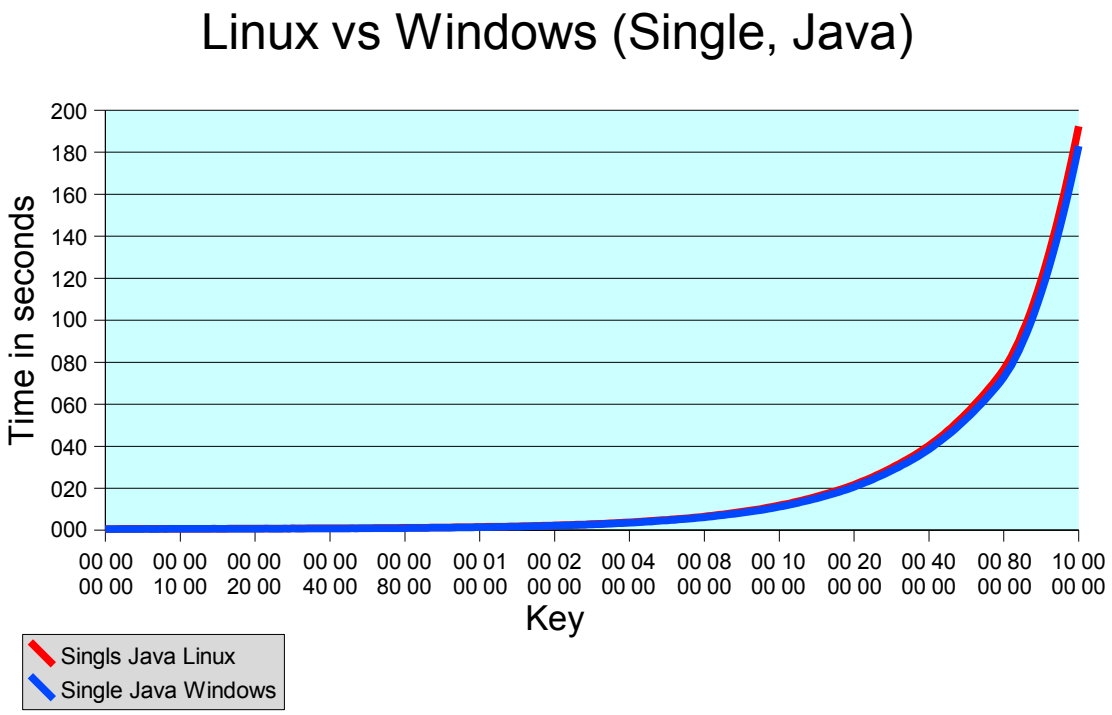

Abbildung 4.5: Diese Abbildung zeigt die gemessene Zeit für das Finden des gegebenen Schlüssels in Abhängigkeit des Betriebssystems für die Sprache Java. Für die Ermittlung der Messwerte wurden ein Debian-System (obere rote Kurve) sowie ein Windows-System (untere blaue Kurve) verwendet.

Es ist ersichtlich, dass die Graphen fast deckungsgleich sind und erst für große Schlüssel merklich unterscheidbar sind.

Wie die Abbildungen 4.4 und 4.5 zeigen, spielt die Wahl des Betriebssystems nur eine untergeordnete Rolle. Der relative Unterschied ist mit einem Faktor von etwa $6 * 10^{-7}$ unter Java beziehungsweise $2 * 10^{-7}$ unter Erlang pro berechnetem Schlüssel unbedeutend.

Drittes Testpaar: Mehr-Rechner-Betrieb Erlang vs. Java unter Linux

Die nächste bedeutende Frage ist, wie sich die Systeme verhalten, wenn mehr als ein Rechner zur Bewältigung der Aufgabe zur Verfügung steht. Dabei wurde die Performanz jeweils auf zwei, vier und acht Rechnern getestet. Zusätzlich stand jeweils ein Rechner als Server bereit.

Für die Interpretation der Ergebnisse ist es wichtig zu wissen, dass die Schlüsselbundgröße nur 256 betrug und das Netzwerk frei war für diese Versuche. Fremder Netzverkehr konnte die Versuchsergebnisse also nicht verfälschen. Mit den folgenden zwei Testreihen wird gezeigt, wie sich die Kommunikation auf die Performanz auswirkt.

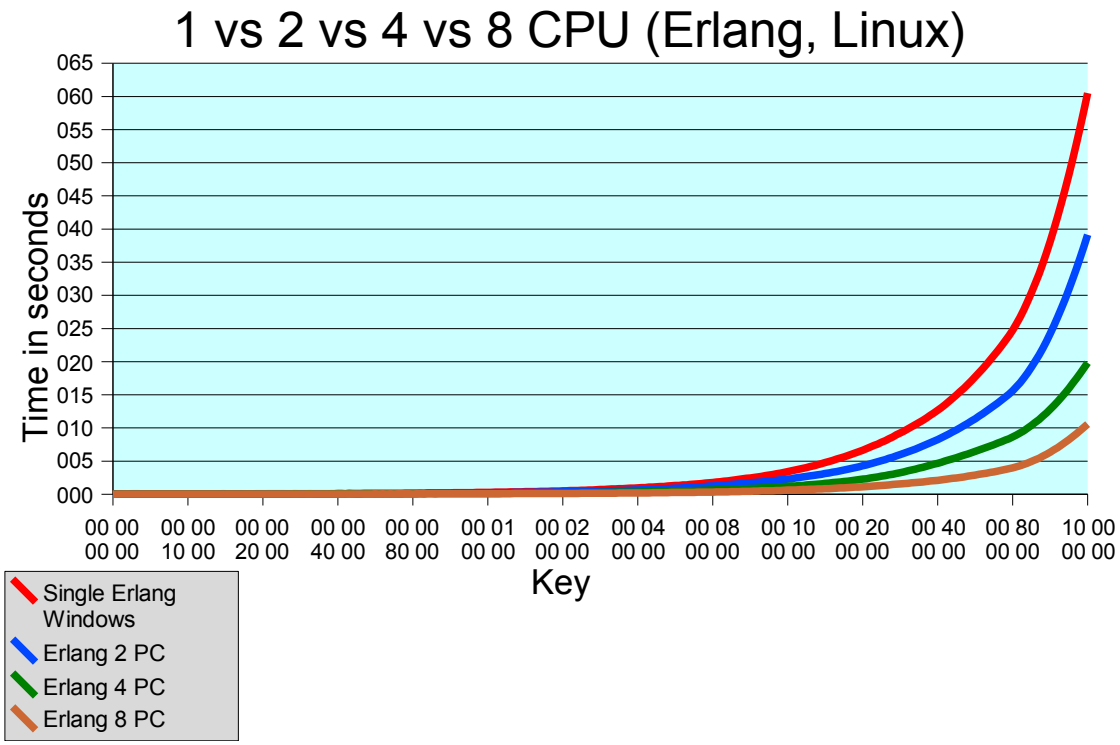

Abbildung 4.6: Diese Abbildung zeigt die benötigte Zeit in Abhängigkeit des Schlüssels. Je mehr Rechner benutzt wurden, desto weniger Zeit wurde benötigt. Dementsprechend ist die orange untere Kurve der Graph der Tests mit acht Rechnern und die obere rote Kurve der Graph der Tests mit einem Rechner.
In diesem Vergleich soll herausgefunden werden, wie groß die relative Leistungszunahme bei der Verwendung von Erlang ist.
Die relative Leistungszunahme von der Ein-Rechner-Variante auf die Zwei-Rechner-Variante ist 1,55.
Die relative Leistungszunahme von der Zwei-Rechner-Variante auf die Vier-Rechner-Variante ist 1,97.
Die relative Leistungszunahme von der Vier-Rechner-Variante auf die Acht-Rechner-Variante ist 1,87.
Im Gegensatz zu der Leistungszunahme in Java (siehe Abbildung 4.7) ist die Leistungszunahme in Erlang bei diesem Test nicht monoton.

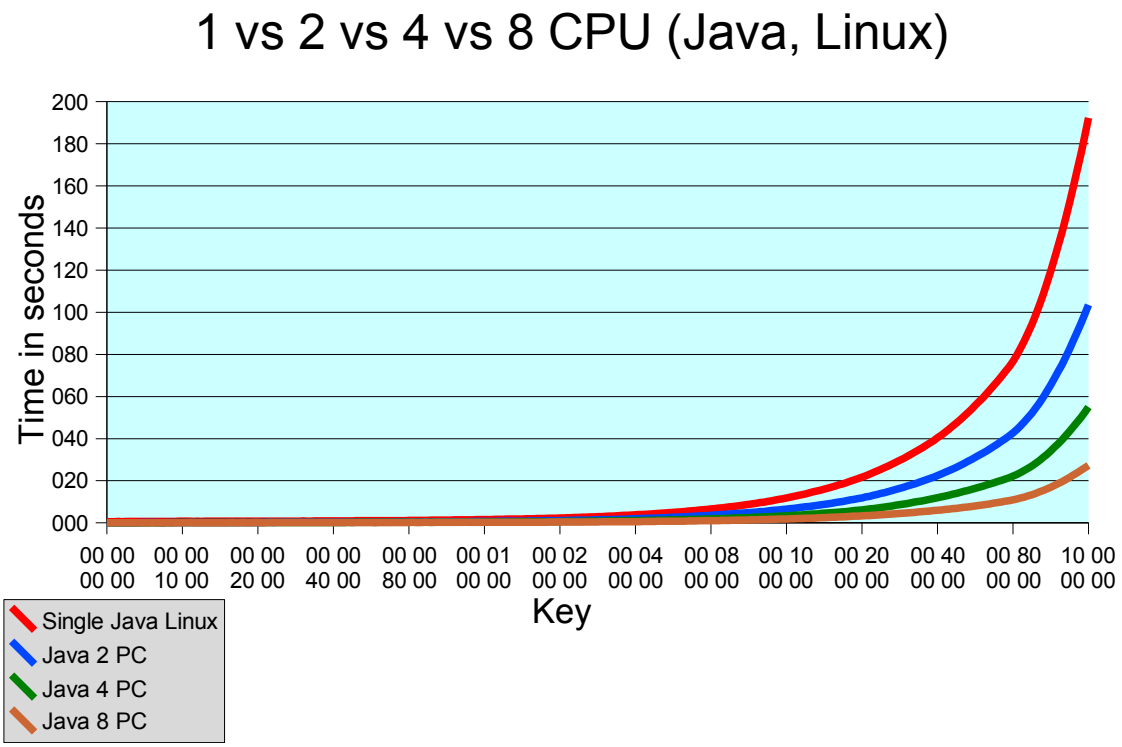

Abbildung 4.7: Diese Abbildung zeigt analog zu Abbildung 4.6 die benötigte Zeit in Abhängigkeit des Schlüssels. Wie schon in den Abbildungen 4.2 und 4.3 gezeigt, benötigt die Entschlüsselung etwa das Dreifache an Zeit wie Erlang. Daher ist die Skala der y-Achse eine Andere.
In diesem Vergleich soll herausgefunden werden, wie groß die relative Leistungszunahme bei der Verwendung von Java ist.
Die relative Leistungszunahme von der Ein-Rechner-Variante auf die Zwei-Rechner-Variante ist 1,86.
Die relative Leistungszunahme von der Zwei-Rechner-Variante auf die Vier-Rechner-Variante ist 1,88.
Die relative Leistungszunahme von der Vier-Rechner-Variante auf die Acht-Rechner-Variante ist 2,01.
Im Gegensatz zu der Leistungszunahme in Abhängigkeit von der Anzahl der Rechner in Erlang ist diese Leistungszunahme in Java bei diesem Test streng monoton wachsend. Die Kommunikation scheint hier kein limitierender Faktor zu sein.

Für Erlang bedeutet die Auswertung, dass während es sich bei den unteren zwei Testreihen offensichtlich jeweils zum nächsthöheren Graphen beinahe um eine Verdoppelung der benötigten Zeit handelt (von acht Rechnern auf vier Rechner um das 1,87fache und von vier Rechnern auf zwei Rechner um das 1,97fache), der Faktor zwischen den beiden oberen Graphen nur 1,55 beträgt.

Der Unterschied zwischen den Faktoren von acht auf vier und von vier auf zwei Rechner liegt an der Menge des Netzverkehrs. Je mehr Clients im Netz sind und Verkehr erzeugen, desto eher gehen Pakete beispielsweise durch Kollision verloren. Die Leistung nimmt also nicht im selben Maße zu wie die Anzahl der Rechner. Je mehr Rechner im Netz sind, desto weniger effizient können die einzelnen Rechner arbeiten. Abhilfe würde hier schaffen, die Größe der vom Server zum Durchsuchen verteilten Schlüsselbunde zu erhöhen. Dadurch würden die Client-Rechner mehr Zeit für das Durchsuchen eines Schlüsselbundes benötigen und daher seltener einen neuen Schlüsselbund verlangen. Das Netzwerkaufkommen würde in Folge dessen im selben Maß sinken, mit der die Größe der Schlüsselbunde zunehmen würde.

Leider konnte die Leistungszunahme für 16 Rechner nicht getestet werden, da nicht ausreichend viele Rechner zur Verfügung standen. Sonst wäre es möglich gewesen, mit Hilfe einer weiteren Messreihe die proportionale Leistungsabnahme zu bestätigen. Mit drei Messreihen lässt sich eine Tendenz ausmachen. Für eine gesicherte Aussage sind diese Messreihen jedoch nicht ausreichend.

Der Leistungszuwachs um das nur 1,55fache in Abbildung 4.6 der oberen beiden Graphen würde dem gerade Angeführten widersprechen, da die proportionale Leistungszunahme bei weniger Rechnern, also bei zwei Rechnern, größer sein müsste als bei vielen Rechnern, wie beispielsweise der Vergleich zu den Zuwachsraten bei vier (1,97) oder acht (1,87) Rechnern belegt. Dass die Applikation auf zwei Rechnern proportional langsamer ist als auf einem Rechner liegt daran, dass für die Ein-Rechner-Variante der Applikation auf jegliche Kommunikationskomponenten verzichtet wurde. Umgekehrt wurde nach der Entwicklung der Ein-Rechner-Variante der Quelltext übernommen und in ein Kommunikationsprotokol eingebettet. Da nun auch die Kommunikation berücksichtigt wurde, war die für mehrere Clients entwickelte Variante viel langsamer als die Ein-Rechner-Variante. Dadurch ist die relative Leistungszunahme von der Ein-Rechner-Variante im Vergleich zur Zwei-Rechner-Variante niedriger als die Zwei-Rechner-Variante im Vergleich zur Vier-Rechner-Variante oder die Vier-Rechner-Variante zur Acht-Rechner-Variante.

Auf Java scheint der Netzwerkverkehr keinen limitierenden Einfluss zu haben. Jedoch benötigt Java etwa dreimal so viel Zeit, um einen Schlüsselbund zu berechnen. Daher ist das Netzwerkaufkommen etwa nur 1/3 dessen Erlangs. Aufschlussreich wäre also die Testreihe mit einem noch größeren Cluster gewesen, um feststellen zu können, ab welcher Rechneranzahl der Netzwerkverkehr einen nachweisbaren Einfluss auf die Performanz von Java hat.

Zwei wichtige Ergebnisse lassen sich daraus ableiten:

- Java benötigt mit acht Rechnern nicht mehr dreimal so lange wie Erlang für das Auffinden des richtigen Schlüssels und
- die relative Leistungszunahme unter Java von der Vier-Rechner-Variante auf die Acht-Rechner-Variante mit dem Faktor 2,01 zeigt, dass es noch geringe Messungenauigkeiten gibt.

Die Messungenauigkeiten wurden eingeschränkt, da jede Versuchsreihe dreimal durchgeführt und die Ergebnisse gemittelt wurden. Eine Leistungszunahme mit einem Faktor, der größer als 2 ist bei einer Verdopplung der Anzahl der Rechner ist unwahrscheinlich und lässt sich auf Messungenauigkeiten zurückführen. Die Messungenauigkeiten in Bezug auf die Größe des verwendeten Schlüssels werden am Ende dieses Kapitels in Abbildung 4.8 in Bezug gesetzt.

Der direkte Vergleich von Java und Erlang für die relative Leistungszunahme zeigt folgende Ergebnisse:

- Für die Ein-Rechner-Variante benötigte Java 3,04-mal so viel Zeit wie Erlang.
- Für die Zwei-Rechner-Variante benötigte Java 2,64-mal so viel Zeit wie Erlang.
- Für die Vier-Rechner-Variante benötigte Java 2,77-mal so viel Zeit wie Erlang.
- Für die Acht-Rechner-Variante benötigte Java 2,58-mal so viel Zeit wie Erlang.

Fazit

Mit Hilfe der erhobenen Messdaten lassen sich Aussagen über die Effizienz der Nutzung von verteiltem Rechnen treffen. Dabei sind grundsätzlich zwei Arten von Vergleich sinnvoll.

Zum Einen kann der relative Leistungszuwachs einer für verteiltes Rechnen entwickelten Applikation gegenüber einer Applikation, die speziell für nur einen Rechner entwickelt wurde, berechnet werden.
Zum Anderen lässt sich für die Mehr-Rechner-Applikation der Leistungszuwachs in Bezug auf die Anzahl der Clients berechnen. Auf diese Weise kann man feststellen, wie viele Rechner festgelegter Kapazität notwendig sind, um einen bestimmten Workload zu bewältigen oder es lässt sich eine Kosten-Nutzen-Analyse erstellen, welche Anzahl von welchen Rechnern notwendig ist und was die einzelnen Rechnerverbände an einmaligen Kosten in der Anschaffung wie an stetigen Kosten für Wartung und Strom verursachen, um für eine Aufgabe den optimalen Cluster auswählen zu können.

Außerdem lassen sich sprachspezifische Werte ermitteln, um die Effizienz von Technologien bei verteiltem Rechnen beziffern zu können und so aus verfügbaren geeigneten Technologien, beziehungsweise Programmiersprachen, die für ein spezifisches Problem optimale wählen zu können.

In dieser Arbeit wurden bezüglich Performanz die Sprachen Erlang und Java näher betrachtet. Daraus lassen sich folgende Ergebnisse ableiten:

- Um die Zuwachsrate verteilten Rechnens gegenüber Anwendungen auf Einzelrechnern beziffern zu können, müssen die entsprechenden Messwerte in Relation gesetzt werden. Dabei sind `stabile` Ergebnisse zu verwenden, also solche, auf die äußere Bedingungen keinen signifikanten Einfluss nehmen konnten. In diesen Versuchen konnten drei bedeutende Faktoren ausgemacht werden:

1. In den aufgeführten Messreihen sind die Zeitwerte, die durch die Verwendung möglichst hoher Schlüssel zu Stande gekommen sind, wie Abbildung 4.8 zeigt, stabil. Beim Betrachten der Graphik wird deutlich, dass die Werte der x-Achse exponentiell ansteigen und somit der Raum, in dem sich die Messwerte einpendeln, in Relation zum gesamten Raum zu vernachlässigen ist.

   Wie die Grafik zeigt, pendeln sich für die distributierten Benchmarks die Werte um 0,000005 Sekunden ein, während die für den Einzelrechnerbetrieb entwickelte Variante sich bei 0,000004 einpendelt. Der Leistungszuwachs entspricht also bei der Verwendung der verteilten Applikation etwa $\frac{4}{5} * (Anzahl\_der\_Clients)$. Diese Zahl ist jedoch auf Grund der geringen Auflösung der Zeit, die in Nanosekunden gemessen wurde, zu stark gerundet. Auf andere Weise lässt sich das Ergebnis genauer bestimmen, indem folgendermaßen vorgegangen wird:
   Leistungszuwachs $= \frac{gemessene\_Zeit\_der\_Einzelrechnerapplikation}{gemessene\_Zeit * Anzahl\_der\_Clients}$
2. Ein weiterer Faktor ist die Größe des verwendeten Schlüsselbundes. Wenn zu viele Rechner zu häufig neue Schlüsselbunde anfordern, kann es zu Überlastsituationen im Netzwerk kommen. Dieses lässt sich verhindern, indem die Größe eines Schlüsselbundes ausreichend groß gewählt wird. Das bedeutet, dass für große Netzwerke mit vielen Rechnern große Schlüsselbunde gewählt werden müssen, und für kleine Netzwerke kleine Schlüsselbunde eingesetzt werden können. Der Nachteil von zu großen Schlüsselbunden ist, dass die Rechner ihren kompletten Schlüsselbund berechnen und erst anschließend beim Anfordern eines neuen Schlüsselbundes erfahren, ob der gesuchte Schlüssel schon gefunden wurde. Dabei kann es zu Situationen kommen, in denen die Client-Rechner unnötig CPU-Ressourcen verbrauchen.
3. Wie im vorangegangenen Kapitel festgestellt werden konnte, ist eine ausreichende Netzwerk-Infrastruktur notwendig. Andernfalls kann das Aufkommen an Netzwerkverkehr schnell ein limitierender Faktor werden.

- Ein weiteres Ergebnis ist, dass Erlang in allen Vergleichen schneller war als Java. Die Leistungszunahme im Mehr-Rechner-Betrieb ist jedoch unter Java größer als unter Erlang. Leider konnte nicht ermittelt werden, ob bei größeren Clustern von 16, 32 oder 64 Rechnern die Leistungszunahme unter Java streng monoton bleibt und Java im massiven Mehr-Rechner-Betrieb nicht irgendwann Erlang überholen kann.

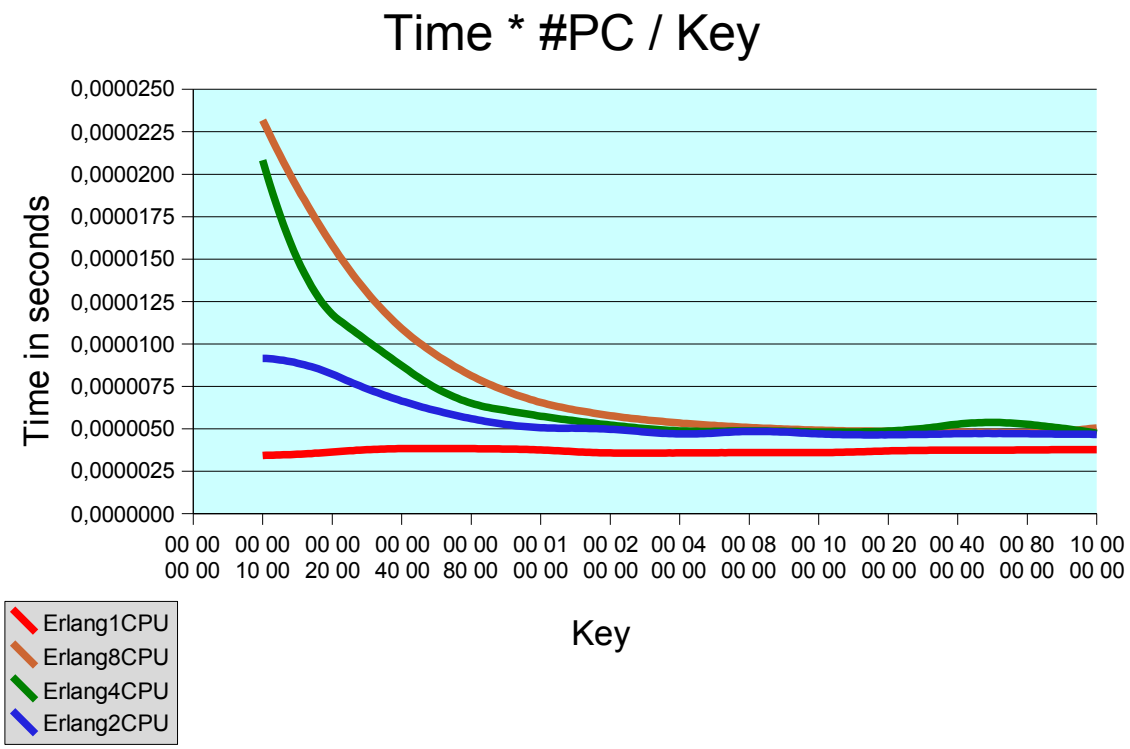

Abbildung 4.8: Diese Abbildung zeigt die zum Berechnen eines Schlüssels pro Rechner benötigte Zeit, das heißt die Zeit, die ein Rechner benötigt, um einen Schlüssel zu testen. Bei der Verwendung von mehreren Rechnern ist also die gemessene Zeit mit der Anzahl der verwendeten Rechner zu multiplizieren (Time * #PC). Bei der Verwendung von Schlüssellängen größer Null ist die gemessene Zeit durch die Anzahl der Schlüssel zu dividieren: ((Time * #PC) / Key)
Wie aus dieser Grafik ersichtlich ist, gibt es bei relativ kleinen Schlüsselmengen einen messbaren Overhead, der bei größeren Schlüsselmengen vernachlässigbar wird. Der Punkt beziehungsweise die Schlüsselgröße, ab dem dieser Fehler klein genug wird, lässt sich individuell festlegen. Da jedoch schon ab Schlüssellängen von 524.288 (also ... 00 08 00 00) ein hinreichend stabiler Wert zu messen ist, werden die an dieser Stelle auftretenden Werte verwendet.

5. Fazit

5.1 Drei logisch-funktionale Sprachen, drei Welten

Zu Beginn dieser Arbeit war mir nicht bewusst, in welche Richtung ich steuere. Das Gebiet war mir unbekannt und alle in dieser Arbeit behandelten Sprachen, außer Java, neu für mich. Es war sehr ermutigend, als ich feststellte, dass auch einige Entwickler sich nicht immer sicher im Umgang mit den entsprechenden Sprachen und Technologien waren und in den Mailinglisten entwickelten sich spannende Diskussionen.

Im Gegensatz zu Sprachen, in die sehr viele Entwickler sehr viel Zeit investiert haben, wie etwa Java, C, Haskell oder Prolog, wirken die Sprachen des logisch-funktionalen Paradigmas noch jung, stellenweise unausgereift und flexibel. Während sich etabliertere Sprachen auf ihr Paradigma berufen und dadurch versuchen, sich gegen andere Sprachen abzugrenzen, versuchen multiparadigmatische Sprachen, die Vorteile verschiedener Paradigmen nutzbar zu machen.

Dabei bieten die drei in dieser Arbeit näher betrachteten Sprachen Curry, Erlang und Mercury drei völlig unterschiedliche Ansätze. Sie alle vereinigen das logische und das funktionale Paradigma, und doch sind sie in Syntax, Anwendungsgebiet, Performanz und vielen weiteren Eigenschaften so unterschiedlich wie Sprachen unterschiedlicher Paradigmen.

In dieser Arbeit habe ich nicht nur drei für mich neue Programmiersprachen betrachten und lernen können, sondern auch zum ersten Mal in meinem Studium hinter die Funktionsweise von Programmiersprachen geblickt. Ich war überrascht von der Begeisterung und Hingabe der Entwickler und gerade beim Erlernen machte sich der Unterschied einer kleinen, hilfsbereiten Entwicklergemeinschaft gegenüber einem unüberschaubar großen unpersönlichen Support-Forum bemerkbar. Während beim Erlernen von Java Fragen häufig mit „Das wurde an anderer Stelle schon diskutiert!“ oder „So etwas Einfaches weisst du nicht?“ beantwortet wurden, gaben die Entwickler der drei näher betrachteten Sprachen hilfreiche Antworten und freuten sich sogar über das Interesse an ihren Sprachen. Besonders bei Curry waren Michael

Hanus und Wolfgang Lux sehr hilfsbereit und ihre Antworten sehr ausführlich. Bei Erlang kamen auf eine Frage bis zu zwölf Antworten, da die Mailingliste eine gewisse Verzögerung hat.

Die Zukunft von Curry, Erlang und Mercury scheint gesichert. Alle drei Sprachen verfügen über eine ausreichend große Entwicklergemeinde. Auch wenn es unter Curry und Mercury noch keine größeren öffentlichen Projekte gibt, ist es wohl nur eine Frage der Zeit, bis andere Entwickler diese Sprachen und ihre Vorteile kennen lernen.

Während Curry und Mercury wohl eher im akademischen Bereich eingesetzt werden, überzeugt Erlang im kommerziellen Bereich schon jetzt. Eine Vermutung der Entwickler von Erlang, warum Ericsson Erlang unter die EPL stellte, ist, dass Erlang wohl verbreitet werden sollte (siehe [10.4] http://www.erlang.org/faq/x1088.html#AEN1239. Mittlerweile wird Erlang nicht nur für verteiltes Rechnen und `rapid prototyping` eingesetzt. Mit dem Projekt Wings3d http://www.wings3d.com/ wurde unter Erlang ein CAD-Programm entwickelt, das sogar in der aktuellen Ausgabe der Fachzeitschrift `c't` mit einem Bericht gewürdigt wird [16].

5.2 Mein persönliches Fazit

Ich habe vor, mich in Zukunft weiter mit der Sprache Erlang zu beschäftigen. Dabei hat mich besonders die Mailingliste von den Fähigkeiten der Sprache überzeugt. Die Projekte, die mit Entwicklern auf der ganzen Welt entstehen und in der Mailingliste offen diskutiert werden, geben einen Einblick in die Möglichkeiten.

Das Erstellen dieser Arbeit hat mir eine Perspektive auf Programmiersprachen im Allgemeinen und auf Programmiersprachen des logisch-funktionalen Paradigmas im Besonderen ermöglicht, die vor dem Lernen jeder Programmiersprache hilfreich sein. In Zukunft wird es mir leichter fallen, neue Programmiersprachen zu erlernen.

Wissenschaftliches Arbeiten fiel mir zu Beginn dieser Arbeit schwer. Doch durch die konstruktive Kritik und kontinuierliche Rückkopplung mit meiner Mentorin lernte ich schnell aus meinen Fehlern.

Wenn ich die Arbeit jetzt noch einmal schreiben könnte, würde ich wohl ein anderes Konzept bevorzugen. Obwohl der praktische Anteil dieser Arbeit größer war als der wissenschaftliche, was aus dem Text nicht hervorgeht, würde ich den praktischen Anteil noch größer wählen und die Auswertung und deren Interpretation ausweiten.

Für eventuelle zukünftige Projekte, die an dieses anknüpfen, sind sicherlich die Bereiche:

- verteiltes Rechnen in Abhängigkeit vom Netzwerkaufkommen und der Clustergröße
- Technologien für verteiltes Rechnen im Vergleich (RMI, Sockets, Erlang, ...)
- verteilte Betriebssysteme basierend auf Erlang
- Entstehung und Entwicklung von Sprachen
- Warum sich ähnliche Technologien nicht immer vereinigen lassen (Curry und Mercury, MCC und PAKCS)

interessant. Auch die Gründe, warum einige Sprachen auf starke Resonanz stoßen wie beispielsweise Java, C oder Prolog, während andere Sprachen ab einem bestimmten Punkt nicht weiterentwickelt werden wie λ-Prolog und Hal, könnten in einem Projekt erforscht werden.

5.3 Nachwort

An dieser Stelle möchte ich den Leser nicht einfach aus dieser Arbeit entlassen. Ich möchte mich an erster Stelle für die intensive Betreuung durch Frau Priv.-Doz. Dr. Elke Wilkeit bedanken. Für das Korrekturlesen möchte ich mich auch bei meiner Mutter bedanken. Schließen möchte ich diese Arbeit mit einer kleinen Anekdote. Für das PAKCS ist eine Installation des SWI-Prologs notwendig, wenn man nicht die kommerzielle Variante des SICS wählt.

SWI-Prolog ist, soweit bekannt ist, die einzige Sprache, welche die korrekte Antwort auf die Frage „nach dem Leben, dem Universum und dem ganzen Rest“ aus dem Buch `Per Anhalter durch die Galaxis` von Douglas Adams [2] geben kann. Stellt man dem System eine nicht beantwortbare Frage , etwa

`?- X.` (kann auch jede andere Variable sein)

folgt die Antwort

```
% ... 1,000,000 ............ 10,000,000 years later
%
% >> 42 << (last release gives the question)
```

Literatur

[1] *Computer Language Shootout.* http://shootout.alioth.debian.org/debian/erlang.php, zuletzt besucht am 2006-07-29.

[2] Douglas Adams. *Per Anhalter durch die Galaxis.* Heyne, 1998.

[3] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition.* Prentice-Hall, 1996.

[4] Bard Technologies Inc. *Erlang Vs callLAB Simulation.* http://www.bardtech.com/Erlangs_vs_Simulation.htm, zuletzt besucht am 2006-07-29.

[5] Ralph Becket. *Mercury Tutorial.* University of Melbourne. http://www.cs.mu.oz.au/research/mercury/tutorial/book/book.pdf, zuletzt besucht am 2006-07-29.

[6] Bildungsserver Mecklenburg-Vorpommern. *Arbeitsbuch Prolog.* http://www.bildung-mv.de/download/fortbildungsmaterial/arbeitsbuch_prolog.pdf, zuletzt besucht am 2006-07-29.

[7] Timothy A. Budd. *Multiparadigm Programming in Leda.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[8] Departament de Sistemes Informàtics i Computació. *An Embedded Language Approach to Router Specification in Curry.* http://www.dsic.upv.es/users/elp/german/sofsem04/paper.pdf, zuletzt besucht am 2006-07-29.

[9] Departament de Sistemes Informàtics i Computació. *UPV-Curry.* http://www.dsic.upv.es/users/elp/upv-curry/upv-curry.html, zuletzt besucht am 2006-05-.

[10] Maria Garcia ; Christian Holzbaur Duck, Gregory; de la Banda. Extending arbitrary solvers with constraint handling rules, 2003.

[11] Ericsson. *Erlang Public License.* http://erlang.org/EPLICENSE, zuletzt besucht am 2006-07-29.

[12] Ericsson. *Getting started with Erlang.* http://www.erlang.org/doc/doc-5.4.13/doc/getting_started/part_frame.html, zuletzt besucht am 2006-07-29.

[13] Ericsson. *Performance Measurements of Threads in Java and Processes in Erlang.* http://www.sics.se/~joe/ericsson/du98024.html, Vergleich vom 02.11.1998, zuletzt besucht am 2006-07-29.

[14] Erlang IDE. *ErlIDE - The Erlang IDE. Powered by Eclipse.* http://erlide.sourceforge.net/, zuletzt besucht am 2006-07-29.

[15] Vijay K. Garg. *Concurrent and Distributed Computing in Java.* John Wiley & Sons, 2004.

[16] ghi. Gelegenheitsflieger.

[17] Google. *Graph, comparing HiPE to Beam.* http://216.239.59.104/search?q=cache:Xw0k7FwmqCUJ:www.antilost.com/community/show.php/act/ST/f/29/t/77+erlang+r11b0&hl=de&gl=de&ct=clnk&cd=1&client=firefox-a, zuletzt besucht am 2006-07-29.

[18] Michael Hanus. *PAKCS Manual.* Universität Kiel. http://www.informatik.uni-kiel.de/~pakcs/Manual.pdf, zuletzt besucht am 2006-05-23.

[19] Michael Hanus. Re: Curry; wed may 11 12:13:00 2006. E-Mail, May 2006.

[20] Heise online. *Windows Vista: Microsoft präzisiert Hardware-Voraussetzungen.* http://www.heise.de/newsticker/meldung/69336, zuletzt besucht am 2006-07-29.

[21] Lambda the Ultimate. *Mercury Vs Prolog.* http://lambda-the-ultimate.org/node/890, zuletzt besucht am 2006-07-29.

[22] Feixiong Liu. Towards lazy evaluation, sharing, and non-determinism in resolution based functional logic languages. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 201–209, New York, NY, USA, 1993. ACM Press.

[23] Wolfgang Lux. Re: Curry; thu may 06 13:27:00 2006. E-Mail, May 2006.

[24] Wolfgang Lux. Re: Curry; wed may 12 14:47:00 2006. E-Mail, May 2006.

[25] Andreas Schwab Michael Hanus. *ALF Manual.* RWTH Aachen, Universität Dortmund. http://www.informatik.uni-kiel.de/~mh/systems/ALF/manual.dvi, zuletzt besucht am 2006-07-29.

[26] Microsoft. *System Performance Assessment Tools for Windows Longhorn.* http://download.microsoft.com/download/9/8/f/98f3fe47-dfc3-4e74-92a3-088782200fe7/TWAR05002_WinHEC05.ppt, zuletzt besucht am 2006-07-29.

[27] OSGi™Alliance. *The OSGi Service Platform - Dynamic services for networked devices.* http://www.osgi.org/, zuletzt besucht am 2006-07-29.

[28] PenguinSoft. *Erlang Module Definition.* http://linux.com.hk/penguin/man/3/erlang.html, zuletzt besucht am 2006-07-29.

[29] Mikael Pettersson. A compiler for natural semantics, 1996.

[30] Mikael Pettersson, Konstantinos Sagonas, and Erik Johansson. Lecture notes in computer science - the hipe/x86 erlang compiler: System description and performance evaluation. In *Functional and Logic Programming: 6th International Symposium, FLOPS*, pages 228–244, Berlin / Heidelberg, Germany, 2002. Springer.

[31] Portland State University. *TEABAG: A DEBUGGER FOR CURRY*. web.cecs.pdx.edu/~antoy/homepage/theses/S_Johnson.pdf, zuletzt besucht am 2006-07-29.

[32] Mickaël Rémond. *Erlang programmation*. Eyrolles, 2003.

[33] Amr Sabry. What is a purely functional language? *Journal of Functional Programming*, 8(1):1–22, 1998.

[34] Scheme Cookbook. *The Erlang Cookbook*. http://schemecookbook.org/Erlang/WebHome, zuletzt besucht am 2006-07-29.

[35] School of Mathematical and Computer Sciences (MACS). *Erlang vs C++*. www.macs.hw.ac.uk/~dsg/telecoms/presentations/TFP05.ppt, zuletzt besucht am 2006-07-29.

[36] Swedish Institute of Computer Science. *Concurrency Oriented Programming in Erlang*. http://www.sics.se/~joe/talks/ll2_2002.pdf, zuletzt besucht am 2006-07-29.

[37] Tenerife Skunkworks. *Haskell vs. Erlang, Reloaded*. http://wagerlabs.com/articles/2006/01/01/haskell-vs-erlang-reloaded, zuletzt besucht am 2006-07-29.

[38] TFH Berlin. *Vorlesung 13*. http://www.wuraffel.de/tfh/ws0506/Vorlesung13.pdf, zuletzt besucht am 2006-07-29.

[39] TU Berlin. *Erlang Einführung*. http://uebb.cs.tu-berlin.de/lehre/2004SPerlang/files/slides.16.4.pdf, zuletzt besucht am 2006-07-29.

[40] TU Berlin. *Leda*. http://uebb.cs.tu-berlin.de/mps03/Multiparadigmen-Programmiersprachen_im_SoSe_2003.Ausarbeitungen/Leda.Reckmann.pdf, zuletzt besucht am 2006-07-29.

[41] TU Berlin. *Multiparadigmen-Programmiersprachen*. http://uebb.cs.tu-berlin.de/~magr/pub/Multiparadigm-TR-2003-15.pdf, zuletzt besucht am 2006-07-29.

[42] Universität Bamberg. *Programmierkurs Prolog*. http://hschaefer.fto.de/prolog95/grund_01.html, zuletzt besucht am 2006-07-29.

[43] Universität Frankfurt. *Programmiersprachen*. http://www.ki.informatik.uni-frankfurt.de/lehre/SS2005/Programmiersprachen/ThemenListe.pdf, zuletzt besucht am 2006-07-29.

[44] Universität Kiel. *Curry - A Tutorial Introduction*. http://www.informatik.uni-kiel.de/~curry/tutorial/tutorial.pdf, zuletzt besucht am 2006-07-29.

[45] Universität Kiel. *Curry, An Integrated Functional Logic Language*. http://www.informatik.uni-kiel.de/~curry/papers/report.pdf, zuletzt besucht am 2006-07-29.

[46] Universität Kiel. *Report, First draft for Curry*. http://www.informatik.uni-kiel.de/~curry/papers/report051296.dvi.Z, zuletzt besucht am 2006-07-29.

[47] Universität Kiel. *Report on Curry*. http://www.informatik.uni-kiel.de/~curry/report.html, zuletzt besucht am 2006-07-29.

[48] Universität München. *Rechnergestützte Verifikation mit ALF - Vorlesung*. http://www.tcs.informatik.uni-muenchen.de/~alti/ALF/alf.html, zuletzt besucht am 2006-05-.

[49] Universität Münster. *Aqua Curry*. http://danae.uni-muenster.de/~lux/curry/aqua.html, zuletzt besucht am 2006-07-29.

[50] Universität Münster. *Münster Curry Compiler*. http://danae.uni-muenster.de/~lux/curry, zuletzt besucht am 2006-07-29.

[51] Universität Münster. *Münster Curry User's Guide*. http://danae.uni-muenster.de/ lux/curry/user.pdf.

[52] University of Melbourne. *Determinism and Backtracking*. http://www.cs.mu.oz.au/research/mercury/tutorial/determinism-etc.html, zuletzt besucht am 2006-07-29.

[53] University of Melbourne. *Mercury FAQ*. http://www.cs.mu.oz.au/research/mercury/information/doc-latest/faq.pdf, zuletzt besucht am 2006-07-29.

[54] University of Melbourne. *Mercury Library*. http://www.cs.mu.oz.au/research/mercury/information/doc-latest/library.pdf, zuletzt besucht am 2006-07-29.

[55] University of Melbourne. *Mercury Reference Manual*. http://www.cs.mu.oz.au/research/mercury/information/doc-latest/reference_Fmanual.pdf, zuletzt besucht am 2006-07-29.

[56] University of Melbourne. *Mercury Transition Guide*. http://www.cs.mu.oz.au/research/mercury/information/doc-latest/transition_guide.pdf, zuletzt besucht am 2006-07-29.

[57] University of Melbourne. *Mercury User Guide*. http://www.cs.mu.oz.au/research/mercury/information/doc-latest/user_guide.pdf, zuletzt besucht am 2006-07-29.

[58] Wikipedia. *List of all known Logical Functional Languages*. http://en.wikipedia.org/wiki/Category:Functional_logic_programming_languages, zuletzt besucht am 2006-07-29.