Fortress

Eduard Frank

15. Juli 2010

Zusammenfassung Fortress ist eine neue parallele Programmiersprache, die im Rahmen des HPCS [1] Programms der DARPA [2] von Sun entwickelt wurde und insbesondere auf die Steigerung der Produktivität bei der Programmierung von Hochleistungsrechnern im Bereich des wissenschaftlichen Rechnens abzielt. Hierzu wurden beim Design von Fortress grundlegende Konzepte und Konstrukte heute gängiger Programmiersprachen im Kontext aktueller paralleler Rechnersysteme hinterfragt und dabei viele Altlasten lokalisiert und verbessert. Ergebnis ist eine neue parallele Programmiersprache, die die besten Konzepte aus mehreren Programmierparadigmen vereinigt und um zusätzliche Ideen, wie z.B. impliziter Parallelisierung und einer der Mathematik ähnlichen Programmsyntax, erweitert. Um die Komplexität heutiger Programmiersprachen zu bewältigen, wurde Fortress als sogenannte "growable" Programmiersprache konzipiert, was bedeutet, dass sie mit Leichtigkeit um weitere Sprachfeatures durch die Realisierung zusätzlicher Bibliotheken erweitert werden kann. Dieser "growable" Ansatz geht in Fortress sogar so weit, dass selbst grundlegende Basistypen und Operatoren etc. über externe Bibliotheken definiert sind, die auf einem minimalistischen Sprachkern aufsetzen.

1 Einführung

Spätestens seit die Prozessorhersteller aufgrund steigendem Energieverbrauch und Abwärme die Taktfrequenz ihrer Prozessoren nicht steigern konnten, entwickelt sich der Trend hin zu Multi-Core Systemen, also dem Verbauen mehrerer Kerne auf einem Prozessor. Schon heute verfügen handelsübliche PCs über 2 oder 4 Prozessorkerne und für die Zukunft werden Prozessoren mit 100 oder mehr Kernen erwartet. Im Bereich des Hochleistungsrechnen wurden diese Dimensionen bereits überschritten und man setzt dort mittlerweile auf sogenannte Multi-Prozessor Systeme, also Rechnersysteme, bei denen mehrere Zentralprozessoren miteinander interagieren. Zur Bearbeitung der typischen komplexen Probleme aus dem Bereich der Teilchenphysik, Metereologie, Genetik usw. wird im Bereich des Hochleistungsrechnen bereits heutzutage auf die Rechenkraft von bis zu 100 000 parallel miteinander kommunizierenden Prozessoreinheiten gesetzt.

Im Gegensatz zur Hardwareentwicklung wurde jedoch über Jahre hinweg die Entwicklung neuer Programmiersprachen zur Programmierung dieser parallelen Rechnersysteme vernachlässigt. Die meisten heute gegenwärtigen Programmiersprachen sind noch im Kontext einer strikt sequentiellen Ausführung des Programmcodes konzipiert worden, sodass es ihnen an Möglichkeiten der Parallelisierung fehlt. Zusätzliche Konstrukte und Mechanismen zur parallelen Ausführung müssen deshalb durch externe Bibliotheken, wie z.B. OpenMP oder MPI, nachgerüstet werden. Diese Bibliotheken ermöglichen zwar eine parallele Ausführung, aber notorische Probleme, die bei der parallelen Programmierung entstehen, wie z.B. Race Conditions, Deadlocks, Livelocks etc., können auch mit ihnen nicht vermieden werden. Auch was die automatische Parallelisierung von Programmen durch Compiler angeht, sind die Möglichkeiten beschränkt, da aufgrund des inhärent sequentiellen Ausführungsmodells eine automatische Parallelisierung die Semantik eines Programms verletzen könnte. Ein weiteres Problem betrifft die Entwicklung portabler paralleler Programme. Portable parallele Programme dürfen nicht im Hinblick auf die Hardwareressourcen [3] eines bestimmten Rechnersystems entwickelt werden, sondern müssen sich dynamisch bei der Ausführung an die verfügbaren Hardwareressourcen eines Rechnersystems anpassen, um

[1] High Productivity Computing Systems
[2] Defense Advanced Research Projects Agency
[3] Zum Beispiel Anzahl der Kerne bzw. Prozessoren

die meiste Performance bei der parallelen Ausführung eines Programm herauszuholen. Mit den heutigen Bibliotheken ist dies nur schwer zu realisieren.

Um die Defizite heutiger Programmiersprachen bei der Entwicklung paralleler Programme insbesondere im Fokus des Hochleistungsrechnen anzugehen, hat die DARPA 2002 das sogenannte HPCS-Programm etabliert. Ziel des HPCS-Programms ist es eine neue Programmiersprache zu realisieren, die die Programmierung von Hochleistungsrechnern erleichtert und gleichzeitig die Produktivität bei der Programmierung dieser erhöhen soll, wobei die Produktivität als Summe von Performance, Programmierbarkeit, Portabilität und Robustheit angesehen wird.

Neben IBM mit der Programmiersprache X10 und Cray mit der Programmiersprache Chapel, hat auch Sun sich am HPCS-Programm beteiligt und hierbei eine neue Programmiersprache entwickelt, die insbesondere auf das wissenschaftliche Rechnen auf Hochleistungsrechnern abzielt. Der Name dieser Programmiersprache lautet Fortress, auf die im nachfolgenden im Detail eingegangen wird.

2 Die Fortress Programmiersprache

Die Programmiersprache Fortress wird von Sun in der Öffentlichkeit als sogenanntes **"Secure Fortran"** propagiert. Intention dieser Anspielung ist es, den Softwareentwicklern mit Fortress eine vollständig neue Programmiersprache anzubieten, die sie bei der Entwicklung sicherer, effizienter und portabler Software für Hochleistungsrechner unterstützt. Erreicht werden soll dieser Leitgedanke dadurch, dass die Entwicklung von Fortress unter den Slogan "Do for Fortran what Java did for C" gestellt wird. Das Hauptziel von Fortress besteht also darin, den damaligen Erfolg der Java Programmiersprache zu wiederholen, indem beim Design von Fortress alle Altlasten und Konzepte aus heute gängigen Programmiersprachen, die damals noch im Kontext einer rein sequentiellen Ausführung des Programmcodes entwickelt wurden, nun im Hintergrund einer parallelen Ausführung des Programmcodes auf Multi-Core bzw. Multi-Prozessor Systemen hinterfragt werden. Das Ergebnis ist eine Programmiersprache, die an die Bedürfnisse des wissenschaftlichen Rechnens auf Hochleistungsrechnern angepasst wurde.

In den nachfolgenden Abschnitten werden nun die grundlegenden Eigenschaften und einige ausgewählte Aspekte der Fortress Programmiersprache im Detail beschrieben.

2.1 Grundlagen

Fortress ist eine Multiparadigmen-Programmiersprache die hauptsächlich dem objektorientierten Paradigma zugeordnet werden kann, jedoch auch viele Konzepte und Ideen aus dem funktionalen Paradigma enthält, was sich insbesondere dadurch zeigt, dass sich die Entwickler beim Design von Fortress von folgenden Programmiersprachen inspirieren ließen: Java, NextGen, Scala, Eiffel, Self, Standard ML, Objective Caml, Haskell und Scheme [4].

So baut bspw. die Objektorientierung in Fortress auf dem Traits und Objects Konzept der Self Programmiersprache auf. Traits weisen dabei eine große Ähnlichkeit zu herkömmlichen Java Interfaces auf, nur das sie zusätzliche Methodendefinitionen erlauben, während die Definition von Feldern in einem Trait nicht erlaubt ist. Objects hingegen können mit Java final Klassen verglichen werden. Sie dienen somit zur Definition von Feldern und Methoden.

Im Hinblick auf die Vererbungshierarchie ließ sich Fortress von Scala inspirieren und unterstützt die Mehrfachvererbung von Traits, während eine Vererbung von Objects nicht

möglich ist. Im DAG [4] der Vererbungshierarchie nehmen Traits somit die Rolle von Knoten ein, während Objects die Rolle von Blättern einnehmen. Das nachfolgende Listing zeigt bspw. die Definition eines List Traits und die Definition der LinkedList und ArrayList Objects, die vom List Trait erben.

```
trait List comprises { LinkedList, ArrayList }
    add(x: Object) : ()
end

object ArrayList extends List
    add(x: Object) do
        ...
    end
end

object LinkedList extends List
    add(x: Object) do
        ...
    end
end
```

Wie man an dem Listing erkennen kann, erlaubt Fortress mittels des **comprises** Schlüsselworts auch die Definition aller erbenden Objects/Traits eines Traits, wodurch einerseits die Lesbarkeit des Source Codes verbessert wird und andererseits der Compiler statische Überprüfungen und Optimierungen während der Übersetzung durchführen kann. Zusätzlich können in Fortress die Traits und Objects Definitionen mittels sogenannter Static Parameters annotiert werden, die dem Generics Konzept aus Java ähnlich sind.

Im Hinblick auf die Typisierung ließ sich Fortress ebenfalls von Scala inspirieren und ist somit eine statisch typisierte Programmiersprache, wobei jedoch genauso wie bei Scala Typinferenz unterstützt wird, sodass die Angabe eines Typen optional ist, wenn der Compiler den Typen eines Ausdrucks selbst bestimmen kann. Im Gegensatz zu Java werden in Fortress alle Typen, sogar die primitiven, über Traits bzw. Objects definiert.

Primitive Datentypen werden in Fortress genauso wie in der Mathematik über die Angabe eines Zahlenraums definiert, wobei zusätzlich die Größe eines Zahlenraums in Bits angegeben werden kann. So wäre bspw. $\mathbb{Z}32$, $\mathbb{Z}64$ der Typ eines 32 bzw. 64 Bit großen Ganzzahlenwerts, während $\mathbb{R}32$, $\mathbb{R}64$ der Typ eines 32 bzw. 64 Bit großen Fließkommawerts wäre. Zusätzlich zu den primitiven Datentypen unterstützt Fortress standardmäßig auch komplexe Datentypen wie z.B. Tuples, Arrays, Matrizen, Listen, Mengen und Maps.

Bei der Deklaration von Variablen wird in Fortress zwischen Immutable und Mutable Variablen unterschieden. So würde bspw. der folgende Ausdruck $one : \mathbb{Z}32 = 1$ eine Immutable Variable definieren, während der folgende Ausdruck $one : \mathbb{Z}32 := 1$ eine Mutable Variable definiert.

Programme in Fortress werden in sogenannte Components gegliedert, die APIs importieren bzw. exportieren können. APIs dienen dabei ausschließlich zur Beschreibung der Schnittstelle einer Component und dürfen nur Deklarationen von Traits, Objects und Funktionen enthalten. Durch das Exportieren einer API kann somit eine Schnittstelle für eine Component vereinbart werden, während durch das Importieren einer API die Funktionalitäten einer Component, die durch die Schnittstelle einer API beschrieben werden, in einer anderen Component genutzt werden können. Das nachfolgende Listing veranschaulicht das Component und API Prinzip am klassischen Hello World Programm.

[4] Direkter azyklischer Graph

```
api Executable
   run(args : String ...): ()
end
```

```
api IO
   print(str : String) : ()
end
```

```
component HelloWorld
   import print from IO
   export Executable
   run(args : String ...) = do
      print("Hello World")
   end
end
```

Damit eine Component ausführbar ist, muss sie die Executable API exportieren und hierbei die run Funktion definieren, die auch gleichzeitig als Einstiegspunkt für die Programmausführung dient. Um die print Funktion nutzen zu können, wird zusätzlich die print Funktion aus der IO API importiert.

2.2 Mathematische Notation

Die größte Auffälligkeit von Fortress Programmen zeigt sich bei der Betrachtung der Programmsyntax, die sich an der herkömmlichen mathematischen Notation orientiert.

Suns Intention bei der Einführung dieser mathematischen Notation war es, die aus dem Bereich des wissenschaftlichen Rechnens typischen komplexen Algorithmenspezifikationen mit so wenig Aufwand wie möglich auf den Computer zu übertragen. Da diese Algorithmenspezifikationen ebenfalls in einer mathematisch ähnlichen Notation vorliegen, können sie fast 1 zu 1 in Fortress Programme transformiert werden, wodurch die Anzahl der Fehler bei der Übertragung minimiert wird.

Ein einfaches Beispiel, wie diese mathematische Notation in Fortress emuliert wird, zeigt sich bei der Betrachtung der Multiplikation zweier Operanden. Im Gegensatz zu den meisten anderen Programmiersprachen wird in Fortress die Multiplikation, wie in der Mathematik üblich, durch die Aneinanderreihung zweier Operanden ausgedrückt, sodass in Fortress statt $y = 2 * x$, $y = 2x$ für die Multiplikation verwendet wird.

Fortress unterstützt standardmäßig Unicode und erlaubt somit bspw. auch griechische Buchstaben für Variablenbezeichner. Die eigentliche Eingabe von Fortress Programmen erfolgt mittels Unicode und/oder[5] einer zusätzlichen sogenannten "Twiki" Notation. Diese "Twiki" Notation ermöglicht es, alle mathematischen Ausdrücke mittels dem ASCII Zeichensatz einzugeben. In dem nachfolgenden Listing sind einige mathematische Ausdrücke sowohl in der "Twiki" Notation (links) als auch in der mathematischen Notation [6] (rechts) dargestellt.

[5] Beides kann gleichzeitig verwendet werden

[6] Einige Editoren z.B. Emacs und Vi sind bereits in der Lage, bei Eingabe des Source Codes in Unicode und/oder der "Twiki" Notation ihn in der mathematischen Notation darzustellen

`A UNION {1,2,3,4}`	$A \cup \{1,2,3,4\}$
`{k | k <- 1 : 100, prime k }`	$\{k \mid k \leftarrow 1:100, prime\, k\}$
`SUM[k <- 1:n] a[k] x^k`	$\sum_{k \leftarrow 1:n} a_k\, x^k$
`BIG MAX[(j,k)<-a.indices] |a[j,k]-b[j,k]|`	$\mathsf{MAX}_{(j,k) \leftarrow a.indices} \lvert a_{j,k} - b_{j,k} \rvert$

Die oberen Beispiele zeigen auch gleichzeitig den hohen Abstraktionsgrad von Fortress, sodass Fortress Programme nicht nur nach dem imperativen, sondern auch nach dem deklarativen Schema realisiert werden können.

2.3 Parallele Programmierung

Im Gegensatz zu den meisten Programmiersprachen, die noch im Kontext einer sequentiellen Ausführung des Programmcodes konzipiert wurden und sich somit schwer tun bei der Bestimmung, ob bestimmte Programmteile parallel ausgeführt werden können, wurde Fortress als eine vollständig parallele Programmiersprache konzipiert, die davon ausgeht, dass prinzipiell alle Programmteile, unter Beachtung von Datenabhängigkeiten, Seiteneffekten etc.[7], parallel ausgeführt werden können.

Ein explizites Erstellen von Threads ist in Fortress zwar immer noch möglich, aber viele Programmkonstrukte werden in Fortress standardmäßig implizit parallel ausgeführt. Einige der implizit parallel ausgeführten Programmkonstrukte werden in folgender Auflistung dargestellt.

- Also Blocks: **do** $f(x)$ **also do** $g(y)$ **end**
- Tuples: $(a,b,c) = (f(x), g(y), h(z))$
- Operatoren: $f(x) + g(y)$
- Funktionsargumente: $func(f(x), g(y), h(z))$

In der obigen Auflistung werden bspw. beim Funktionsaufruf der "func" Funktion implizit alle 3 Funktionsargumente parallel ausgewertet und anschließend wird die "func" Funktion aufgerufen. Sollte jedoch das Fortress Laufzeitsystem feststellen, dass die parallele Auswertung der 3 Funktionsargumente kostspieliger als die sequentielle Auswertung ist, werden die 3 Funktionsargumente implizit sequentiell ausgewertet. Ein weiteres Grundkonzept von Fortress ist, dass der Entwickler explizit mit angeben muss, falls eine sequentielle Ausführung eines Ausdrucks gewünscht ist. In der obigen Auflistung müssten dann für eine sequentielle Auswertung der Funktionsargumente die Funktionsargumente in **do**...**end** Blöcken nacheinander ausgewertet werden und anschließend die Ergebnisse dieser Auswertungen an die "func" Funktion übergeben werden.

Bei der eigentlichen Parallelisierung von Ausdrücken wird in Fortress intern das sogenannte Fork/Join Prinzip angewendet. Hierbei wird die parallele Auswertung eines Ausdrucks in Teilaufträge unterteilt, die anschließend nacheinander in die sogenannte "Work Queue" des aktuellen Threads gelegt werden. Anschließend arbeitet der aktuelle Thread alle Teilaufträge aus seiner "Work Queue" ab und fährt mit der Ausführung fort, sobald alle Teilaufträge abgearbeitet wurden. Zusätzlich wird im Fortress Laufzeitsystem ein sogenanntes "Work-Stealing" [3] Verfahren eingesetzt. Hierbei ist es möglich, dass ein freier Thread Teilaufträge aus der "Work-Queue" eines anderen Threads "stiehlt" und diesen dadurch bei der parallelen Auswertung seines Ausdrucks unterstützt, wodurch ein Load Balancing im Gesamtsystem realisiert wird. Sowohl das Fork/Join Prinzip als auch das "Work-Stealing"

[7] Wird vom Fortress Laufzeitsystem automatisch während der Laufzeit erkannt

Verfahren finden implizit statt und sind somit für den Entwickler transparent.

Ein weiteres elementares Sprachkonstrukt, das in Fortress standardmäßig implizit parallel ausgeführt wird, ist die Standard For-Schleife. Gemäß dem Grundkonzept von Fortress muss auch hier explizit mit angegeben werden, falls eine sequentielle Ausführung der For-Schleife gewünscht wird. Dies wird auch im nachfolgenden Listing veranschaulicht.

```
for i ← 1:10 do
    print i
end
```

```
for i ← seq(1:10) do
    print i
end
```

4 2 1 3 5 8 9 10 7 6

1 2 3 4 5 6 7 8 9 10

Die Parallelisierung der For-Schleife wird über sogenannte Generators gesteuert. Aufgabe der Generators ist es, für jeden Schleifendurchlauf die zugehörigen Werte für die Schleifenvariable zu generieren. Im obigen Listing werden in der Standard For-Schleife (links) die Werte $1-10$ für die Schleifenvariable parallel generiert, während in der sequentiellen For-Schleife (rechts) ein sequentieller Generator verwendet wird, der die Werte $1-10$ für die Schleifenvariable sequentiell generiert. In Fortress werden Generators mit Collections assoziiert. Die nachfolgende Auflistung zeigt bspw. einige Generators.

- $a : b$: Zahlenwerte im Intervall $[a, b]$ (Range Collection)
- $a\#b$: Zahlenwerte im Intervall $[a, a + b]$ (Range Collection)
- $\{a, b, c, d\}$: Menge von Zahlenwerten $\{a, b, c, d\}$ (Set Collection)

Zusätzlich zu den Generators existieren in Fortress die sogenannten Reductions, die eng zusammen mit den Generators interagieren und zur Verknüpfung 2er Einzelergebnisse dienen, um den Gesamtwert eines Ausdrucks zu bestimmen. Im folgenden Beispiel $\prod_{i \leftarrow 1:10} i$ generiert der Generator die Werte von $1-10$ parallel, während die Aufgabe der Reduction darin besteht, die Werte jeweils 2er generierter Werte miteinander zu multiplizieren, um das Gesamtergebnis des Ausdrucks zu bestimmen. Da der Generator die Werte parallel generiert, muss die Verknüpfungsoperation der Reduction einem Monoid entsprechen, damit der Gesamtausdruck parallel ausgewertet werden kann.

Im nachfolgenden Abschnitt wird an einem Beispiel veranschaulicht, wie mit Hilfe des Generator und Reduction Prinzips ein Fortress Ausdruck parallelisiert werden kann.

2.3.1 Beispiel: Generator, Reduction Parallelisierung

Ziel dieses Abschnitts ist es, den folgenden Fortress Ausdruck: $\sum_{i \leftarrow 1:100} i$ für eine Range Collection zu parallelisieren. Für die Parallelisierung von Ausdrücken, die nach dem Generator, Reduction Prinzip ablaufen, sind in Fortress folgende Schnittstellen definiert.

```
trait Reduction⟦L⟧
    empty(): L
    join(a: L, b: L): L
end
```

```
trait Generator⟦E⟧
    generate⟦R⟧(r: Reduction⟦R⟧, body: E → R): R
end
```

Mittels des Reduction Traits wird die Schnittstelle für eine konkrete Reduction definiert. Die Methode **empty** muss hierbei das neutrale Element bezüglich der Verknüpfungsoperation zurückgeben, während die Methode **join** das Verknüpfungsergebnis zweier Elemente zurückgeben muss.

Mittels des Generator Traits wird die Schnittstelle für einen konkreten Generator definiert. Die **generate** Methode erhält als ersten Parameter die konkrete Reduction und als zweiten Parameter einen Funktionskörper. Der grundsätzliche Ablauf der **generate** Methode besteht anschließend darin, für jedes Element einer Collection den Funktionskörper aufzurufen und die Ergebnisse jeweils zweier Funktionskörperaufrufe mittels der übergebenen konkreten Reduction zu verknüpfen, um zum Schluss das Gesamtergebnis zurückzugeben.

Zur Parallelisierung des obigen Summenausdrucks wird folgende konkrete Reduction definiert.

```
object SumZZ32Reduction extends Reduction⟦Z32⟧
    empty() : Z32 = 0
    join(a : Z32, b : Z32) : Z32 = a + b
end
```

Das neutrale Element bezüglich der Verknüpfung ist 0, während die Verknüpfungsoperation der Addition entspricht.
Für die Realisierung des Generators wird die Range Collection vom Generator Trait abgeleitet, sodass der obige Summenausdruck in folgenden **generate** Methodenaufruf transformiert werden kann.

$$(1:100).generate\big(\text{SumZZ32Reduction}, \texttt{fn}\ (i) \Rightarrow (i)\big)$$

Für die eigentliche Parallelisierung wird nach dem Recursive Subdivision Schema vorgegangen, da dieses in Kombination mit dem "Work-Stealing" Verhalten von Fortress gut skaliert [4][25]. Hierbei wird die Größe eines Ranges rekursiv und parallel in jeweils 2 gleich große Teilranges unterteilt. Sobald ein Teilrange kleiner 10 ist, werden für diesen Teilrange die Funktionskörperaufrufe sequentiell ausgeführt und mittels der Reduction verknüpft. Durch das sequentielle Ausführen kann hierbei bspw. vom lokalen Cacheverhalten eines Prozessors profitiert werden. Anschließend werden die verknüpften Teilergebnisse aller Teilranges miteinander verknüpft und das Gesamtergebnis zurückgegeben. Im nachfolgenden Listing wird die konkrete Umsetzung dieses Verfahrens veranschaulicht.

```
object Range(lo: ℤ64, hi: ℤ64) extends Generator⟦Z64⟧
    size := hi − lo + 1
    generate⟦R⟧(reduction: Reduction⟦R⟧, body: ℤ64 → R) : R =
        if size < 10 then
            r : R = reduction.empty()
            i : ℤ64 := lo
            while i ≤ hi do
                v : R = body(i)
                r := reduction.join(r, v)
                i += 1
            end
            r
        else
            mid = ⌊(lo + hi)/2⌋
            reduction.join(Range(lo, mid).generate⟦R⟧(reduction, body),
                           Range(mid + 1, hi).generate⟦R⟧(reduction, body))
        end
    end
end
```

Die Parallelisierung erfolgt hierbei im Else-Block durch die 2 **generate** Methodenaufrufe, die implizit parallel ausgeführt werden, da sie als Funktionsargumente aufgerufen werden.

Die obige **generate** Methode ist dabei universell einsetzbar und kann auch zur Parallelisierung folgender Ausdrücke verwendet werden.

`for` $i \leftarrow 1:100$ `do` $a_i := 2^i$ `end`

$A = \{\, 2^i \mid i \leftarrow 1:100 \,\}$

$z = (1:100).\mathit{generate}(\text{ForReduction}, \texttt{fn}\ (k) \Rightarrow (a_k := 2^k))$
$z = (1:100).\mathit{generate}(\text{SetReduction}, \texttt{fn}\ (k) \Rightarrow (2^k))$

Hierzu müssen nur zusätzliche Reduction Objects realisiert werden.

2.4 Growable

Um die mit den Jahren anhaltend steigende Komplexität heutiger Programmiersprache zu bewältigen und flexibel auf Wünsche und Anregungen der Entwicklergemeinde zu reagieren, verfolgen die Entwickler von Fortress folgenden Leitgedanken: "Whenever possible, implement a proposed language feature in a library rather than building it into the compiler".

Statt also alle Sprachfeatures direkt in den Compiler zu integrieren, werden die meisten auf externe Bibliotheken ausgelagert. Hierfür haben die Fortress Entwickler einen minimalistischen Sprachkern geschaffen, der als Aufsetzpunkt für diese externen Bibliotheken dient und an die Bedürfnisse der Bibliotheksentwickler ausgelegt ist. Die Zielgruppe der Fortress Entwickler sind somit nicht mehr die Anwendungsentwickler, sondern die Bibliotheksentwickler, die anschließend die notwendigen Sprachfeatures für die Anwendungsentwickler realisieren. Um möglichst natürlich aussehende Sprachfeatures realisieren zu können, die wie ein Teil der Programmiersprache aussehen, erlaubt der Sprachkern den Bibliotheksentwicklern Einfluss auf die Syntax und Semantik der Programmiersprache auszuüben und bietet hierfür auch passende Schnittstellen zu den Interna des Compilers, wie bspw. den AST [8]. Das dieser Ansatz in der Praxis funktioniert, zeigt sich insbesondere dadurch, dass fast die gesamte Sprachfunktionalität von Fortress, angefangen von primitiven Datentypen und Arrays bis hin zu Kontrollstrukturen wie der For-Schleife, über externe Bibliotheken definiert sind. Vorteil dieses Auslagerns ist, dass Fortress anschließend flexibel an die aktuellen bzw. zukünftigen Anforderungen angepasst werden kann. Dadurch ist die Sprache sozusagen in der Lage im Laufe der Zeit zu "wachsen" und sich an die aktuellen Herausforderungen dynamisch auszurichten.

Ein weiteres Ziel des obigen Leitgedanken ist es, die Open Source Community bei der Entwicklung von Fortress mit einzubeziehen, da insbesondere im Bereich des wissenschaftlichen Rechnens eine umfangreiche Auswahl an Funktionen essentiell wichtig für den Erfolg einer Programmiersprache ist, die unmöglich von Sun alleine realisiert werden können. Durch die vielseitigen Möglichkeiten bei der Entwicklung von Sprachfeatures können dieses Funktionen so realisiert werden, dass sie wie ein Teil der eigentlichen Fortress Programmiersprache aussehen.

3 Fazit

Ein klarer Vorteil von Fortress ist die mathematischen Notation, wodurch Fortress Programme verständlicher und einfacher lesbar sind als äquivalente Programme in anderen

[8] Abstract Syntax Tree

Programmiersprachen. Zudem wird dadurch auch der Lernaufwand für Wissenschaftler im Bereich des wissenschaftlichen Rechnens, die z.B. über wenig Programmierkenntnisse verfügen, reduziert, sodass sie sich schon nach kürzester Zeit an der Entwicklung von Fortress Programmen beteiligen können. In der Praxis zeigt sich der Vorteil der mathematischen Notation insbesondere darin, dass komplexe Algorithmen, die auf einem Blatt Papier oder bei Teambesprechungen auf dem Whiteboard realisiert werden, ohne große Mühe in ein lauffähiges Fortress Programm transformiert werden können. Der hohe Abstraktionsgrad der mathematischen Notation erlaubt es zudem, bei der Programmierung den Fokus auf die Realisierung des eigentlichen Algorithmus zu setzen, statt sich auf die darunterliegende Hardware zu konzentrieren.

Im Hinblick auf die Parallelisierung wird der Entwickler von der Fortress Programmiersprache in allen Belangen unterstützt. Viele Sprachkonstrukte werden hierbei bereits standardmäßig implizit parallel ausgeführt, sodass sich der Entwickler um die parallele Ausführung der Programme praktisch nicht mehr kümmern muss. In der Praxis kommt es dadurch oft vor, dass man überrascht feststellt, dass ein Codestück, von dem man vorher nicht ahnte, dass es parallel ausgeführt werden kann, plötzlich implizit von bspw. 64 Threads parallel bearbeitet wird.

Nachdem Sun 2006 aus dem HPCS-Programm ausgeschieden ist, wurde das sogenannte Project Fortress Projekt [5] etabliert und Fortress als Open Source veröffentlicht. 2008 wurde dann die 1.0 Version der Sprachspezifikation zusammen mit einem auf Java 1.5 basierten Referenzinterpreter veröffentlicht. Im Gegensatz zur Versionsnummer der Sprachspezifikation weist der Referenzinterpreter in der Praxis jedoch erhebliche Defizite im Bereich der Performance und der Typüberprüfung auf, sodass er für den produktiven Einsatz noch nicht eingesetzt werden sollte. Zusätzlich wurde auch mit der Entwicklung eines Compilers begonnen, der die JVM als Zielumgebung verwenden soll. Mittlerweile wurde auch eine erste Version des Compilers veröffentlicht, der jedoch nur einen Bruchteil der Fortress Sprachspezifikation unterstützt. Während der Entwicklung des Compilers wurden auch die Grenzen der JVM erkannt [25], sodass es fraglich ist, ob an der JVM als Zielumgebung für den Compiler weiter festgehalten wird.

Obwohl Fortress ursprünglich für den Bereich des wissenschaftlichen Rechnens auf Hochleistungsrechnern konzipiert wurde, ist es durch den "growable" Leitgedanken der Sprache nicht ausgeschlossen, dass es in Zukunft auch in anderen Bereichen eingesetzt wird.

In dieser Arbeit konnten natürlich aufgrund des Umfangs nicht alle Sprachfeatures der Fortress Programmiersprache vorgestellt werden, so enthält Fortress in der aktuellen Version bspw. folgende ebenfalls interessante Konzepte.

- Synchronisation von Threads mittels Software Transactional Memory -> Keine Deadlocks [22]
- Viele Möglichkeiten bei der Überladung von Operatoren
- Annotierung von Variablen mit physikalischen Dimensionen und Einheiten möglich, die zur Compilezeit überprüft werden. [22]
- ...

Zusammenfassend in einem Satz lässt sich sagen, dass Fortress mit seinen neuen revolutionären Ideen und Konzepten den Weg für zukünftige Programmiersprache weist, an der tatsächlichen praktischen Umsetzung dieser Visionen scheitert es jedoch **noch**.

Literatur

1. Guy Steele *"Growing a Language" keynote talk, OOPSLA 1998*
2. Maurice Herlihy, Victor Luchangco, Mark Moir *A Flexible Framework for Implementing Software Transactional Memory*
3. David Chase, Yossi Lev *Dynamic Circular Work-Stealing Deque*
4. Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele, and Sam Tobin-Hochstadt *TheFortress Language Specification*
5. Project Fortress *http://projectfortress.sun.com/*
6. Wikipedia *http://en.wikipedia.org/wiki/Fortress_%28programming_language%29*
7. Fortress Programming Language Tutorial http://research.sun.com/projects/plrg/PLDITutorialSlides9Jun2006.pdf
8. Fortress: A New Programming Language for Scientific Computing *http://labs.oracle.com/projects/plrg/Publications/1.02_steele.pdf*
9. Fortress 0.62 *http://labs.oracle.com/projects/plrg/Publications/PPoPPPanel.pdf*
10. Fortress for Productive Computing *http://labs.oracle.com/projects/plrg/Publications/Fortress-PMUA.pdf*
11. Parallelism in Fortress *http://labs.oracle.com/projects/plrg/Publications/PGAS.pdf*
12. Parallel Programming and Parallel Abstractions in Fortress *http://labs.oracle.com/projects/plrg/Publications/PACTSept2005.pdf*
13. The Fortress Programming Language *http://labs.oracle.com/projects/plrg/Publications/JapanLecture2006public.pdf*
14. Parallel Programming and Parallel Abstractions in Fortress *http://labs.oracle.com/projects/plrg/Publications/Aarhus-Fortress-Parallelism-2006public.pdf*
15. A Growable Language *http://labs.oracle.com/projects/plrg/Publications/OOPSLA-GrowableLanguage-2006public.pdf*
16. Project Fortress *http://labs.oracle.com/projects/plrg/Publications/allen-fortressintro.pdf*
17. Object-Oriented Programming in Fortress *http://labs.oracle.com/projects/plrg/Publications/allen-oo-fortress.pdf*
18. Fortress: A New Programming Language for Scientific Computing *http://labs.oracle.com/projects/plrg/Publications/SNU.pdf*
19. What's Cool about Fortress *http://labs.oracle.com/projects/plrg/Publications/2007-0410.pdf*
20. Growing the Fortress Programming Language by Example *http://labs.oracle.com/projects/plrg/Publications/2008-0157.OH08-Ryu.pdf*
21. A Growable Language *http://labs.oracle.com/projects/plrg/Publications/OOPSLA-GrowableLanguage-2006public.pdf*
22. Project Fortress: A New Programming Language from Sun Labs *http://labs.oracle.com/projects/plrg/Publications/2008-0218.JavaOne.pdf*
23. Fortress Boot Camp Material *http://labs.oracle.com/projects/plrg/Publications/BootCamp2008.html*
24. Fortress: Parallel Programming Through Extensible Bulk Operations *http://labs.oracle.com/projects/plrg/Publications/Rochester-Nov2008.pdf*
25. Run your whiteboard in parallel *http://www.infoq.com/presentations/chase-fortress*
26. The Extraordinary Algebra of List Comprehensions *http://labs.oracle.com/projects/plrg/Publications/NEPLS-Mar2009-comprehensions.pdf*
27. Growing the Fortress Programming Language by Example *http://labs.oracle.com/projects/plrg/Publications/2008-0157.OH08-Ryu.pdf*
28. The Future Is Parallel: What's a Programmer to Do? Breaking Sequential Habits of Thought *http://labs.oracle.com/projects/plrg/Publications/NEPLSMarch2009Steele.pdf*
29. A Short Hands-On Introduction to Fortress *http://labs.oracle.com/projects/plrg/Publications/MITtutorial2009.pdf*
30. Organizing Functional Code for Parallel Execution; or, foldl and foldr Considered Slightly Harmful *http://labs.oracle.com/projects/plrg/Publications/ICFPAugust2009Steele.pdf*
31. Project Fortress: A Multicore Language for Multicore Processors *http://labs.oracle.com/projects/plrg/Publications/linuxMagazine.pdf*
32. Fortress Presentation *http://www.slideshare.net/alexmiller/project-fortress*
33. The Soul of Fortress *http://labs.oracle.com/minds/2005-0302/*
34. Parallel by Default *http://blogs.sun.com/simons/entry/fortress_parallel_by_default*