Komplexe Zahlen für Java

Edwin Günthner und Michael Philippsen

Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation
Am Fasanengarten 5, 76128 Karlsruhe
http://wwwipd.ira.uka.de/JavaParty/

Zusammenfassung Eine Voraussetzung für den Einsatz von Java im wissenschaftlichen Rechnen ist die adäquate Unterstützung komplexer Zahlen. Die vorliegende Arbeit stellt einen Präprozessor und dessen Übersetzungsschema vor, der einen neuen primitiven Datentyp `complex` mit zugehörigen Operatoren auf reguläres Java abbildet. Die offensichtliche Ersetzung einer `complex`-Variablen durch zwei `double`-Variablen reicht dazu bei weitem nicht aus.
Der vom Programmierer zu schreibende Code wird durch den primitiven Typ nicht nur lesbarer als methodenbasierte Operationen auf `Complex`-Objekten, sondern er wird auch schneller ausgeführt. Gemittelt über alle untersuchten Kern- und Anwendungs-Benchmarks laufen die auf dem Basistyp `complex` basierenden Programmversionen um den Faktor 2 bis 21 (je nach JVM) schneller als die klassenbasierten Vergleichsversionen.

1 Einleitung

In Java besteht die einzig sinnvolle Möglichkeit zur Verwendung komplexer Zahlen darin, eine `Complex`-Klasse zu erstellen, deren Objekte zwei `double`-Werte enthalten. Komplexwertige Arithmetik muß dann umständlich durch Methodenaufrufe ausgedrückt werden, wie in folgendem Code-Fragment. Nicht gangbar ist die Alternative, per Hand überall dort zwei `double`-Variablen zu verwenden, wo ein komplexwertiger Wert benutzt werden soll.

```
Complex a = new Complex(5,2);
Complex b = a.plus(a);
```

Dies hat drei Nachteile: Ohne Operatorüberladung sind erstens arithmetische Ausdrücke nach ihrer Formulierung nur schwer lesbar. Zweitens ist wegen fehlender Sprach- und Übersetzerunterstützung sogenannter Wertklassen in Java das Anlegen eines Objekts viel langsamer und verbraucht mehr Speicherplatz als das Anlegen einer Variable eines primitiven Typs. Klassenbasierte komplexwertige Arithmetik ist dadurch erheblich langsamer als Javas primitivwertige Arithmetik. Dieser negative Effekt wird noch verstärkt, da durch die Formulierung mit Hilfe von Methodenaufrufen Hilfsobjekte angelegt werden, die bei Verwendung der Stapelmaschine zur Wertspeicherung gar nicht erforderlich wären. Drittens fügen sich klassenbasierte komplexe Zahlen grundsätzlich nicht voll in das übliche Erscheinungsbild der primitiven Typen ein: Sie sind nicht in die Typbeziehungen integriert, die zwischen Javas primitiven Typen bestehen, so daß

z.B. die Zuweisung eines primitiven `double`-Wertes zu einem `Complex`-Objekt keinesfalls eine automatische Typkonvertierung auslöst, was bei primitiven komplexen Zahlen möglich wäre. Darüberhinaus ist bei einer klassenbasierten Lösung stets ein Konstruktoraufruf erforderlich, wo ein Literal zur Repräsentation eines konstanten Werts ausreichen sollte.

Da wissenschaftliches Rechnen nur einen unbedeutenden Anteil an der weltweit um sich greifenden Java-Nutzung hat, ist es sehr unwahrscheinlich, daß die Java Virtual Machine (JVM) bzw. der Bytecode um einen neuen primitiven Typ `complex` erweitert wird, was sicherlich das beste Vorgehen zur Einführung komplexer Zahlen in Java ist. Da ferner nicht abzusehen ist, ob und wenn ja wann, Java um allgemeine Überladbarkeit von Operatoren und um effiziente Unterstützung von Wertklassen erweitert wird, und da eine nahtlose Einbindung einer Klasse in das Erscheinungsbild existierender primitiver Typen grundsätzlich auch nach einer solchen Erweiterung nicht gegeben ist, ist es sinnvoll, Java auf andere Weise um den primitiven Typ `complex` zu erweitern. Sollten die genannten allgemeineren Spracherweiterungen für Java dereinst kommen, dann kann die hier vorgestellte spezifisch für komplexe Zahlen entwickelte Präprozessorlösung noch immer als ein Vergleichsmaßstab für die quantitative Bewertung der Effizienz der allgemeinen Implementierung dienen.

Im folgenden Abschnitt erfolgt eine Einordnung in das Umfeld verwandter Arbeiten. Abschnitt 3 gibt einen Überblick über den gebauten Übersetzer. Die zentralen Ideen des Transformationsschemas werden in Abschnitt 4 vorgestellt, Abschnitt 5 präsentiert die quantitativen Leistungsergebnisse.

2 Verwandte Arbeiten

Das von Sun Microsystems unterstützte Java Grande Forum [6, 10] versucht, die Tauglichkeit von Java auch für wissenschaftliches Rechnen zu verbessern. Die Schwierigkeit besteht darin, die Bedürfnisse dieser zahlenmäßig kleinen Gruppe von Java-Anwendern so aufzubereiten und zu bündeln, daß sie dennoch in der Weiterentwicklung von Java berücksichtigt werden.

Das Java Grande Forum arbeitet an einer Schnittstelle und Implementierung der Klasse `Complex`, die die klassenbasierte Formulierung komplexwertiger Arithmetik in Java ermöglicht [11, 5]. Die in der Klasse angebotenen Methoden gehen über einen ad-hoc-Ansatz hinaus, da besonderer Wert auf die numerische Stabilität der verwendeten Algorithmen gelegt wird. IBM erweitert ihren nativen Java-Übersetzer so, daß die Verwendung dieser Klassenbibliothek erkannt wird [12]. Dadurch können Methodenaufrufe und die Erzeugung von Hilfsobjekten weitgehend eingespart werden, so daß akzeptable Leistungen auch bei einer klassenbasierten komplexwertigen Arithmetik erreicht werden, allerdings nur auf einigen Rechnerarchitekturen der Firma IBM. Die übrigen Nachteile (keine Operatorüberladung und keine nahtlose Integration in den Kanon der primitiven Typen) bleiben bestehen.

Es werden Überlegungen angestellt, Wertklassen zu einem offiziellen Bestandteil von Java zu machen [4, 9]. Während die praktische Umsetzung aber auf sich

warten läßt, wird im Borneo-Projekt [3] bereits versucht, Java um Wertklassen zu erweitern. Weil es bereits objektorientierte Sprachen gibt, die Wertklassen unterstützen, z.B. Sather [8], sind die grundsätzlichen Probleme einer Übersetzung in nativen Code als gelöst zu betrachten.[1] Offen ist allerdings, wie eine solche Spracherweiterung durch Rückführung auf die gegebene Ausgangssprache effizient erledigt werden kann. Auch helfen Wertklassen alleine für komplexwertige Arithmetik nur eingeschränkt, da Operatorüberladung und nahtlose Integration in den Kanon der primitiven Typen fehlen.

3 Der Übersetzer *cj* im Überblick

Nach unserer Erfahrung stößt eine Spracherweiterung nur dann auf Akzeptanz, wenn eine Transformation in reguläres Java existiert. Eine bessere Leistung wird allerdings erreicht, wenn gezielte Optimierungen bei der Bytecode-Erzeugung angewendet werden. Unser Übersetzer *cj*, der auf *gj* aufbaut [1], hat daher zwei Ausgabeformate, wobei wir in diesem Papier die Java-Lösung darstellen und die Optimierungen nur andeuten.

3.1 Eingeführte Sprachelemente

Die Menge der primitiven Typen wird von *cj* um den Typ `complex` erweitert. Ein Wert vom Typ `complex` repräsentiert, wie auch in Fortran und dem C9X-Vorschlag zur Fortentwicklung von C [2], ein Paar von Gleitkommazahlen mit den üblichen Operationen. Real- und Imaginärteil sind über die Felder `real` und `imag` ansprechbar. Man beachte, daß die Feldbezeichnungen selbst keine neuen Schlüsselworte sind. Der primitive Typ `complex` ist Obertyp von `double`, was sich durch implizite Typumwandlung bei Verwendung eines `double`-Werts in einem `complex`-Kontext bemerkbar macht. Mit einem zweiten neuen Schlüsselwort `I` (imaginäre Einheit) können konstante komplexwertige Ausdrücke und Initialisierungen wie in folgendem Beispiel-Code formuliert werden.

```
void foo(complex x, complex y) {
  complex const = 5.0 + y.real * I;
  complex sum   = const + x + y;
  ...
```

3.2 Eine dritte Transformationsphase für *gj*

Der Übersetzer *gj* enthält bereits zwei Transformationsphasen, die zusätzlich eingeführte Sprachmittel auf reguläres Java zurückführen. Innere Klassen werden wie in jedem Java-1.1-Übersetzer auf die Ausdruckselemente von Java 1.0 zurückgeführt. Vor dieser Rückführung geschieht in *gj* eine Transformation generischer Klassen auf den Java-1.1-Sprachumfang.

Cj ergänzt eine weitere Transformation vor der Auflösung generischer Klassen.[2] Die Grundlagen dieser Transformation werden in Abschnitt 4 erläutert.

[1] In C++ kann der Programmierer mit Hilfe des Wertübergabemechanismus die Wertklassensemantik nachempfinden.

[2] Zusätzlich muß die Typprüfung um den Basistyp `complex` erweitert werden.

3.3 Getrennte Übersetzung durch Namensmodifikation

Als `complex` definierte Programmelemente werden auf die in Java existierenden Möglichkeiten abgebildet. Ein Teil der Transformation wird durch Modifikation der vom Programmierer verwendeten Namen realisiert. Durch Anhängen von `$cj$real` an einen vom Programmierer vergebenen Variablennamen wird z.B. die real-Komponente nach der Transformation bezeichnet. Analog werden Methodennamen verändert, wenn ihre Signatur komplexwertige Parameter hat.

Da auch der Bytecode-Lader und die Typüberprüfung die Modifikationsregeln für Namen kennen, ist es möglich, einzelne Java-Programme auch dann getrennt zu übersetzen, wenn `complex` im vorübersetzten Bytecode bereits aufgelöst worden ist. Wenn beispielsweise der Name einer modifizierten Methode im Bytecode gefunden wird, dann wird für die semantische Prüfung ein zusätzliches Methodensymbol angelegt, das die komplexwertige Originalsignatur hat.

4 Rekursives Übersetzungsschema für `complex`

Die Transformation komplexwertiger Berechnungen in einfache `double`-Arithmetik verursacht folgende Schwierigkeiten.

• **Nicht-lokale Transformationen.** Wird ein komplexwertiger Ausdruck an einer Stelle im Programmtext verwendet, an der nur ein *Ausdruck* zulässig ist, kann dieser nicht durch eine Folge von Anweisungen aufgelöst werden.

```
while (u == v && x == (y = foo(z))) {...}
```

Wenn die komplexwertige Bedingung der `while`-Schleife in ihren real- und imaginär-Anteil aufgespalten werden soll, dann bleibt nur der Umbau der `while`-Schleife unter Verwendung von Hilfsvariablen und deren Berechnung im Inneren des Schleifenrumpfs.

Zur Auflösung komplexwertiger Ausdrücke in Anweisungsfolgen braucht man nicht-lokale Transformationsregeln, die die umgebenden Anweisungen mit umbauen. Leichter in einen Übersetzer einzubauen und auf ihre Korrektheit zu prüfen sind lokale Transformationsregeln, die komplexwertige Ausdrücke wieder durch Ausdrücke (nicht Anweisungen) ersetzen.

• **Semantik.** Um die in Java übliche Semantik zu erhalten, kann nicht zuerst der ganze real- und anschließend der imaginär-Anteil ausgewertet werden, da arithmetische Ausdrücke strikt von links nach rechts ausgewertet werden müssen. Ein Seiteneffekt während der Auswertung ist nur rechts von seinem Auftreten sichtbar. Bei einer Ausnahmebedingung dürfen nur Seiteneffekte aus dem Teil des Ausdrucks sichtbar werden, der links von der Auslösestelle liegt.

Die Aufspaltung der Berechnung in ihren real- und imaginär-Anteil ist aus oben genannten Gründen nicht zulässig, ferner bleibt bei diesem Ansatz unklar, was mit Funktionsaufrufen geschehen soll (`foo(z)` im obigen Beispiel). Wird `foo` zweimal aufgerufen? Werden zwei Versionen von `foo` erstellt?

Eine Transformation komplexwertiger Arithmetik mit Java-gerechter Semantik muß für jeden Ausdruck sein Äquivalent in `double`-Arithmetik festlegen.

4.1 Sequenzmethoden

Um beide Kategorien von Schwierigkeiten zu umgehen, führen wir als eine der zentralen Ideen von *cj* konzeptuell sog. Sequenzmethoden ein. Jeder komplexwertige Ausdruck wird in eine Folge von Zuweisungen transformiert und diese in den Argumenten von sogenannten Sequenzmethoden versteckt, deren Rückgabewert zwar nicht `void` ist, aber trotzdem ignoriert wird. Dadurch bleibt ein Ausdruck ein Ausdruck und die Transformation ist lokal. Eine Sequenzmethode hat einen leerem Rumpf, die Operation selbst findet während der Auswertung der Argumente statt und zwar wie Java vorschreibt von links nach rechts.[3] Bei geschachtelten Ausdrücken sind die Argumente des Sequenzmethodenaufrufs selbst wieder Sequenzmethodenaufrufe für die Teilausdrücke. Damit wird es möglich, jeden einzelnen Knoten eines komplexwertigen Ausdrucksbaums einzeln bzgl. real- und imaginär-Anteil auszuwerten und dadurch die Sichtbarkeit von Seiteneffekten und Ausnahmebedingungen an die vorgeschriebene Auswertungsreihenfolge zu koppeln.

Bei der Generierung von Java-Code (*cj* als Präprozessor), bleiben die Aufrufe der (in der umgebenden Klasse `final` deklarierten) Sequenzmethoden erhalten und werden vom Just-in-Time-Übersetzer entfernt. Wenn *cj* selbst den Bytecode erzeugt, werden die Aufrufe der Sequenzmethoden wegoptimiert – nur die Auswertung der Argumente bleibt.

In C/C++ könnte statt Sequenzmethoden der Komma-Operator verwendet werden; Java bietet jedoch kein derartiges Sprachkonstrukt an. Durch das Wegoptimieren der Methodenaufrufe kann *cj* aber Bytecode erzeugen, der ähnlich effizient ist, wie Bytecode, der aus einem Komma-Operator entstanden wäre.

4.2 Sequenzmethoden am Beispiel

Konzentrieren wir uns bei dem komplexwertigen Ausdruck `z = x + y` zunächst auf die rechte Seite der Zuweisung. Um sicherzustellen, daß keine Seiteneffekte auftreten, werden zunächst alle Operanden in Hilfsvariablen gesichert. Das folgende Code-Fragment ist das (vorläufige und un-optimierte) Ergebnis der Transformation der rechten Seite der Zuweisung.

```
seq(seq(tmp1_real = x_real, tmp1_imag = x_imag),
    seq(tmp2_real = y_real, tmp2_imag = y_imag),
    tmp3_real = tmp1_real+tmp2_real, tmp3_imag = tmp1_imag+tmp2_imag)
```

Im Beispiel würden im umgebenden Block 6 `double`-Variablen deklariert, was nicht gezeigt ist. Es werden zunächst die inneren `seq`-Aufrufe von links nach rechts ausgewertet: beide Anteile von `x` und `y` werden in Hilfsvariablen gesichert. Die äußere Sequenzmethode sorgt für die anschließende Addition (im dritten und vierten Argument). Eine nachfolgende Grundblockoptimierung entdeckt die Kopienfortschreibung und eliminiert passiven Code, so daß wir mit einer fast minimalen Zahl von Hilfsvariablen und Kopieroperationen auskommen. Im Beispiel bleiben zwei Hilfsvariablen und eine Sequenzmethode übrig.

[3] Ausnahmesituationen treten also nicht in der Sequenzmethode auf, sondern im Aufrufkontext; sie müssen nicht in der Signatur der Sequenzmethode deklariert werden.

```
    seq(tmp3_real = x_real+y_real, tmp3_imag = x_imag+y_imag)
```

Betrachten wir nun die Zuweisung zu `z`, die zwei Einzelzuweisungen erfordert.

```
seq(seq(tmp3_real = x_real+y_real, tmp3_imag = x_imag+y_imag),
    z_real = tmp3_real, z_imag = tmp3_imag)
```

Auch hier führt die Grundblockoptimierung zur Reduktion der Anzahl der Hilfsvariablen und verhindert die Deklaration einer Sequenzmethode. Der resultierende Java-Code erfordert also keine Hilfsvariablen mehr und kommt mit der Deklaration einer Sequenzmethode in der umgebenden Klasse aus. Im allgemeinen Fall werden je nach den verwendeten Transformationen u.U. mehrere Sequenzmethoden erzeugt; die generierten Signaturen bzw. Typen der Parameter werden jeweils so gewählt, daß sie zu den tatsächlich vorkommenden Argumenten zuweisungskompatibel sind. Weil die Sequenzmethoden also benutzerdefinierte Typen in den Argumenten verwenden können, ist eine vordefinierte Zusammenfassung in einer Hilfsklasse nicht möglich.

```
seq(z_real = x_real + y_real, z_imag = x_imag + y_imag)
```

Wenn direkt Bytecode erzeugt wird, werden nur noch die Argumente der Sequenzmethoden ausgewertet, so daß der entstehende Bytecode der manuellen Auflösung der komplexwertigen Ausdrücke entspricht.

4.3 Einfache Transformationsregeln im Detail

Im folgenden betrachten wir einen Ausdruck E, der aus Unterausdrücken e_1, bis e_n zusammengesetzt ist. Die Umsetzungsregel $eval[E]$ beschreibt (rechts vom $\mapsto$-Symbol) die rekursive Transformation in reguläres Java, die wie jeweils angegeben $eval[e_i]$ für alle Unterausdrücke anwendet. Komplexwertige Ausdrücke werden von $eval$ meist auf Sequenzmethoden abgebildet, deren Resultat ignoriert wird. Manchmal ist es aber erforderlich, auf den real- oder imaginär-Anteil zuzugreifen. Dazu dienen $evalR$ und $evalI$, die prinzipiell dasselbe leisten wie $eval$, aber auf Sequenzmethoden (`seqREAL`) abgebildet werden, die den real- oder imaginär-Anteil zurückgeben statt einem unbeachteten Resultat. Werden $evalR$ bzw. $evalI$ auf ein komplexwertiges Feld angewendet, liefert die erzeugte Sequenzmethode ein Feld von `double`-Werten zurück. Siehe die Diskussion der Konstruktor-Methoden in Abschnitt 4.5. Nicht-komplexwertige Ausdrücke werden von $eval$, $evalR$ und $evalI$ unverändert beibehalten. Während das =-Symbol im Sinne der Programmiersprache verwendet wird, definiert $\equiv$ links einen Bezeichner, der durch den rechts angegebenen programmiersprachlichen Ausdruck zu expandieren ist.

Für die linken Seiten von Zuweisungen gibt es eine andere Transformationsregel: $access[E]$ liefert nicht den Wert sondern den kürzesten Zugriffspfad (nur höchstens eine Zeigerdereferenzierung) auf einen Unterausdruck.

Wie anhand des Beispiels weiter oben deutlich wurde, werden viele Hilfsvariablen in dem Block deklariert (und anschließend in der Regel wieder wegoptimiert), der den Ausdruck im Programmtext unmittelbar umschließt.[4] Die unten beschriebenen Transformationsregeln zeigen die Deklaration der Hilfsvariablen nicht. Stattdessen sind diese an der Namenskonvention zu erkennen: Für einen komplexwertigen Ausdruck e bezeichnen e_{real} und e_{imag} die zugehörigen beiden Hilfsvariablen vom Typ `double`. Auf zusätzlich benötigte Hilfsvariablen wird im Text hingewiesen. Komplexwertige Felder werden im nächsten, Methodenaufrufe im übernächsten Abschnitt gesondert diskutiert. Nicht gezeigt sind die recht simplen Transformationsregeln für unäre Operatoren, für Konstanten, für Literale und für initialisierte Konstanten in einem Interface. Details sind in [7] beschrieben.

• **Einfacher Bezeichner:** Für $E \equiv c$ lautet die Transformationsregel:

$$eval[c] \mapsto seq(E_{real} = c_{real}, E_{imag} = c_{imag})$$

Beide Komponenten des komplexwertigen Bezeichners c werden in den Hilfsvariablen gespeichert, die das Ergebnis des Ausdrucks E sind. Ist c das Ziel einer Zuweisung, dann genügt es, die gemäß der Namensmodifikationsregel gebildeten Namen zu benutzen.

• **Selektion:** Für $E \equiv F.e$ lautet die Transformationsregel:

$$eval[F.e] = seq(tmp = eval[F], E_{real} = tmp.e_{real}, E_{imag} = tmp.e_{imag})$$

Es wird F (einmal) ausgewertet und in einer Hilfsvariablen *tmp* gesichert, ehe es für den Zugriff auf beide Komponenten verwendet wird. Ist $F.e$ das Ziel einer Zuweisung, dann wird F in eine Hilfsvariable ausgewertet, die dann für die weitere Transformation der rechten Seite verwendet wird:

$$access[F.e] \mapsto tmp = eval[F] \\ \wedge\ E^{\downarrow}_{real} \equiv tmp.e_{real}, E^{\downarrow}_{imag} \equiv tmp.e_{imag}$$

Im Rahmen der Umsetzungsregel für die Zuweisung (s.u.) wird erstens der Programm-Code rechts vom $\mapsto$-Symbol dort eingesetzt, wo $access[F.e]$ ausgewertet wird, und zweitens sind die in dieser Regel verwendeten Bezeichner $E^{\downarrow}_{real}$ und $E^{\downarrow}_{imag}$ durch den Code textuell zu ersetzen, der nach dem $\equiv$-Zeichen angegeben ist. (Diese $\downarrow$-Notation und das textuelle Einsetzen von Programm-Code sollen helfen, die Diskussion der Zugriffspfadauswertung von der Diskussion der Zuweisung gedanklich zu trennen.)

• **Zuweisung:** Für $E \equiv e_1 = e_2$ lautet die Transformationsregel:

$$eval[e_1 = e_2] \mapsto seq(access[e_1], eval[e_2], E_{real} = e^{\downarrow}_{1real} = e_{2real}, E_{imag} = e^{\downarrow}_{1imag} = e_{2imag})$$

Erst wird der Zugriff auf e_1, dann die rechte Seite der Zuweisung ausgewertet. In den letzten zwei Schritten erfolgt die Zuweisung beider Anteile des komplexwertigen Ausdrucks. Da die Zuweisung selbst wieder ein Ausdruck ist, müssen zusätzlich die Hilfsvariablen gesetzt werden, die zu E gehören. Die $e^{\downarrow}$ werden gemäß der *access*-Umsetzung textuell eingefügt.

Beispielsweise wird aus `X.Y.z = x` nachdem die durch die Auswertung von `x` eingefügten Hilfsvariablen und Sequenzmethodenaufrufe wegoptimiert sind:

[4] Bei statischem Code oder der Initialisierung von Instanzvariablen werden die Hilfsvariablen nicht zu Instanzvariablen, sondern durch die Verwendung von statischen und/oder dynamischen Blocks mit eingeschränkter Lebensdauer und Sichtbarkeit versehen.

```
seq(tmp = X.Y, tmp.z_real = x_real, tmp.z_imag = x_imag)
```

Die Speicherung in der Hilfsvariablen `tmp` ist nur dann erforderlich, wenn `X.Y` Seiteneffekte haben kann.

• **Kombination aus Zuweisung und Operation:** Für $E \equiv e_1 \diamond = e_2$, wobei $\diamond \in \{+, -, *, /\}$, lautet die Transformationsregel:

$$eval[e_1 \diamond = e_2] = seq(access[e1], e_1^{\downarrow} = eval[e_1^{\downarrow} \diamond e_2])$$

Die Adressse der linken Seite der Zuweisung wird ausgewertet; überall wo $e_1^{\downarrow}$ steht wird der resultierende Programm-Code textuell eingesetzt. Die Adresse geht als linker Operand in die Operation ein. Abschließend wird das Ergebnis der Operation an die zuvor berechnete Adresse geschrieben. Diese Reihenfolge ist nötig, um mehrfache Seiteneffekte bei der Berechnung von e_1 zu umgehen.

• **Vergleich:** Für $E \equiv e_1 == e_2$ lautet die Transformationsregel:

$$eval[e_1 == e_2] \mapsto seq_{\&\&}(eval[e_1], eval[e_2], e_{1real} == e_{2real}, e_{1imag} == e_{2imag})$$

Im Gegensatz zu den bisher verwendeten Sequenzmethoden hat diese einen nicht-leeren Rumpf. Zurückgegeben wird das Ergebnis einer logischen UND-Verknüpfung des dritten und vierten Argumentwerts. Es wird also `true` zurückgegeben, wenn die beiden komplexwertigen Ausdrücke e_1 und e_2 in ihren beiden Komponenten übereinstimmen. Analog erfordert die Ungleich-Verknüpfung eine $seq_{||}$-Sequenzmethode. Die Rümpfe beider speziellen Sequenzmethoden werden bei der direkten Erzeugung von Bytecode unmittelbar eingebaut („inlining“).[5]

• **Addition und Subtraktion:** Für $E \equiv e_1 \diamond e_2$, wobei $\diamond \in \{+, -\}$, lautet die Transformationsregel:

$$eval[e_1 \diamond e_2] \mapsto seq(eval[e_1], eval[e_2], E_{real} = e_{1real} \diamond e_{2real}, E_{imag} = e_{1imag} \diamond e_{2imag})$$

• **Multiplikation:** Für $E \equiv e_1 * e_2$ lautet die Transformationsregel:

$$eval[e_1 * e_2] \mapsto seq(eval[e_1], eval[e_2], E_{real} = e_{1real} * e_{2real} + e_{1imag} * e_{2imag}, E_{imag} = e_{1real} * e_{2imag} - e_{1imag} * e_{2real})$$

• **Division:** Die Regel für die Division ist strukturell genau wie die für die Multiplikation, nur mit komplizierteren Ausdrücken. Neben der Standard-Divisionsregel, die numerisch instabil ist, bietet *cj* auch das langsame aber genauere Verfahren aus der Referenzimplementierung von [11, 2] an. Aus Platzgründen drucken wir hier keine der beiden Varianten ab. (Die Meßergebnisse in Abschnitt 5 beziehen sich auf die schnelle Division, außer wenn explizit *slowdivision* angegeben ist.)

• **Typwandlung:** Da `complex` als Obertyp von `double` definiert wird, werden automatische Typumwandungen eingefügt wo dies erforderlich ist. Ferner werden explizite Typumwandlungen nach `complex` entfernt, wenn der betroffene Ausdruck ohnehin komplexwertig ist. Lediglich für den verbleibenden Fall ($E \equiv (complex)\ e$) muß die Transformationsregel benutzt werden:

$$eval[(complex)\ e] \mapsto seq(eval[e], E_{real} = e, E_{imag} = 0)$$

• **String-Konkatenation:** Die wenig zeitkritische String-Konkatenation erledigt *cj* durch Erzeugung eines Objekts der Klasse `Complex` und den Aufruf der

[5] Im Beispiel der komplizierten `while`-Bedingung aus Abschnitt 4 resultiert in etwa:
`while ( seq`$_{\&\&}$`(u_real==v_real,u_imag==v_imag)`
`&& seq`$_{\&\&}$`(`$eval[E \equiv$ `y=foo(z)],x_real==e_real,x_imag==e_imag)) {...}`
Aus Gründen der Lesbarkeit verzeichten wir hier darauf, auch die Zuweisung und den Funktionsaufruf zu expandieren.

`toString`-Methode. Dadurch ist man ohne Änderung des Übersetzers in der Lage, das Ausgabeformat zu verändern. Die Transformationsregel lautet (analog für vertauschte Operanden und die `+=`-Operation):

$$eval[str + e] \mapsto str + (new\ Complex(evalR[e], e_{imag}).toString())$$

Durch $evalR[e]$ wird e ausgewertet und der real-Anteil zurückgeliefert. Darüberhinaus wird von $evalR$ die Hilfsvariable e_{imag} deklariert und mit dem imaginär-Anteil von e initialisiert. Die Asymmetrie ist erforderlich, um doppelte Auswertung von e zu verhindern.

4.4 Transformationsregeln für Felder

Während eine einzelne komplexwertige Variable sinnvollerweise durch ein Paar von `double`-Werten repräsentiert wird, ist die Lösung für komplexwertige Felder nicht offensichtlich. Es gibt zwei grundsätzliche Möglichkeiten: Entweder wird ein komplexwertiges Feld durch zwei `double`-Felder ersetzt oder es wird ein einziges `double`-Feld doppelter Länge benötigt.

Die Verwendung eines Feldes erhält die Anzahl anzulegender Objekte. Der Aufwand steigt aber bei jedem Feldzugriff, weil beim Zugriff auf ein komplexwertiges Feldelement zwei Bereichstests erforderlich sind. Ferner ist festzulegen, ob zusammengehörige `double`-Werte an benachbarten Feldpositionen abgespeichert werden, was eventuell Vorteile beim Cache-Verhalten bringt, oder ob erst alle real-Anteile im Feld angeordnet werden, ehe die imaginär-Anteile folgen.

Die Verwendung von zwei Feldern erzeugt zwei Objekte, was in der Regel etwas langsamer ist. Andererseits können wegen der exakt gleichen Größe beider Felder die doppelten Bereichstests von vielen JITs wegoptimiert werden, was bei einem Feld doppelter Größe nur schwer zu schaffen ist.

Weil zukünftige JITs bei der Objekterzeugung immer besser werden und immer mehr Bereichstests für Felder einsparen können und weil dann die Transformation einfacher zu implementieren ist, verwenden wir zwei Felder.

• **Feld-Erzeugung und Initialisierung:** Java bietet verschiedene Syntax-Elemente an, um Felder zu erzeugen oder auch gleich zu initalisieren. Betrachten wir zunächst die Transformationsregel für die reine Felderzeugung:

$$\begin{aligned} &eval[new\ complex[e_1]\dots[e_n]] \mapsto \\ &\quad seq(E_{real} = new\ double[e'_1 = eval[e_1]]\dots[e'_n = eval[e_n]], \\ &\qquad E_{imag} = new\ double[e'_1]\dots[e'_n]) \end{aligned}$$

Bei der Berechnung von E_{real} werden weitere Hilfsvariablen e'_i angelegt/benutzt, um die Größenangaben im imaginär-Anteil wiederzuverwenden. Für die Feldinitialisierung wird folgende Transformationsregel verwendet:

$$\begin{aligned} &eval[new\ complex\ []\dots[]\{e_1, \dots, e_n\}] \mapsto \\ &\quad seq(E_{real} = new\ double[]\dots[]\{evalR[e_1], \dots, evalR[e_n]\}, \\ &\qquad E_{imag} = new\ double[]\dots[]\{e_{1imag}, \dots, e_{nimag}\}) \end{aligned}$$

$EvalR$ kommt mit inneren Feldinitialisierungen zurecht, da es diese wie anonyme Feldinitialisierungen kleinerer Dimensionalität transformiert.

• **Feldzugriff als Ziel einer Zuweisung:** Solche Feldzugriffe könnten unter Seiteneffekten leiden, wenn die Auswertung eines Index-Ausdrucks das Feld selbst verändert. Daher ist im allgemeinen Fall die Speicherung der Referenz auf das Feld (bis zur innersten Dimension, e_{n-1}) in einer Hilfsvaria-

blen erforderlich. Für den allgemeinen Fall lautet die Transformationsregel:

$$access[F[e_1]\ldots[e_n]] \mapsto tmp = eval[F[e_1]\ldots[e_{n-1}]] \\ \wedge\ E^{\downarrow}_{real} \equiv tmp_{real}[e'_n = eval[e_n]], E^{\downarrow}_{imag} \equiv tmp_{imag}[e'_n])$$

Wiederum werden neue Hilfsvariable e'_i benutzt, um den Index-Ausdruck nur einfach auszuwerten, und mit Hilfe der $\downarrow$-Notation wird ausgedrückt, daß auf der rechten Seite einer Zuweisung die angegebenen Ausdrücke textuell eingesetzt werden. Für den Zugriff auf ein eindimensionales Feld gilt $tmp = eval[F]$.

• **Feldzugriff:** Für $E \equiv F[e_1]\ldots[e_n]$ lautet die Transformationsregel:

$$eval[F[e_1]\ldots[e_n]] \mapsto seq(eval[F], E_{real} = F_{real}[e'_1 = eval[e_1]]\ldots[e'_n = eval[e_n]], \\ E_{imag} = F_{imag}[e'_1]\ldots[e'_n])$$

4.5 Transformationsregeln für Methodenaufrufe

Bei Methodenaufrufen muß man komplexwertige Parameter und Rückgabewerte unterscheiden. Ferner erfordern Konstruktor-Methoden eine Sonderbehandlung.

• **Komplexwertiger Rückgabewert:** In Java können keine zwei `double`-Werte auf einmal aus einer Methode zurückgegeben werden. Der offensichtliche Ansatz, im Inneren der Methode ein Objekt einer `Complex`-Klasse oder ein zweielementiges `double`-Feld für die Rückgabe zu generieren, ist i.A. ungünstig, weil dann bei jedem Methodenaufruf ein Objekt erzeugt würde, das unmittelbar nach der Rückkehr verworfen werden kann. Die in *cj* verwendete Transformation deklariert für jeden textuell vorkommenden Aufruf einer Methode mit komplexwertigem Rückgabewert ein zwei-elementiges `double`-Feld. Die Deklaration erfolgt nicht im unmittelbar den Methodenaufruf umgebenden Block sondern am Anfang derjenigen Methode, die die Aufrufstelle umschließt. Anstelle der ursprünglich aufgerufenen Methode *foo* wird eine Methode $\widehat{foo}$ mit veränderter Signatur aufgerufen, der als zusätzliches Argument eine Referenz auf das Hilfsfeld übergeben wird. Das Hilfsobjekt wird pro Aufruf der umgebenden Methode nur einmal angelegt und evtl. wiederverwendet. Die Transformationsregel für $E \equiv foo()$ lautet (die Transformation von Parametern wird unten beschrieben):

$$eval[foo()] \mapsto seq(\widehat{foo}(tmp), E_{real} = tmp[0], E_{imag} = tmp[1])$$

Weil das Hilfsfeld erstens lokal zur umgebenden Methode angelegt ist und zweitens pro textuellem Aufruf von *foo* ein eigenes Hilfsfeld angelegt wird, sind rekursive Aufrufe unproblematisch. Auch wenn innerhalb der Methode mehrere Threads angelegt werden, landet das frisch angelegte Hilfsfeld im Inneren der `run`-Methode der Threads, so daß jeder Thread ein eigenes Exemplar nutzt.

Die veränderten Methoden haben nicht den Rückgabetyp `void` sondern geben einen Hilfstyp zurück, damit sie im Inneren von Ausdrücken verwendbar sind. Die `return`-Anweisungen im Inneren der Methoden geben `null` zurück, nachdem die Elemente des Hilfsfelds gesetzt worden sind.

• **Komplexwertiger Parameter:** Es wird die auf der Hand liegende Transformation durchgeführt. Die Signatur der Methode wird so geändert, daß statt eines komplexwertigen Arguments zwei `double`-Werte übergeben werden. Entsprechend werden die Aufrufstellen der Methode modifiziert. Wichtig ist, daß nicht nur die Parameterleiste der Methode sondern auch ihr Name verändert wird, um eine Kollision mit einer gleichnamigen Methode zu vermeiden, die

zufällig die entstehende Parametertypisierung hat. Die Transformationsregel lautet (entsprechend für Methoden mit mehreren komplexwertigen Parametern):

$$eval[bar(e)] \mapsto \widehat{bar}(evalR[e], e_{imag})$$

• **Konstruktor-Methode:** Aufrufe von Konstruktor-Methoden lassen sich abgesehen von den unveränderten Methodennamen i.allg. nach dem selben Schema behandeln.[6] Lediglich der Aufruf eines anderen Konstruktors kann als erster Befehl einer Konstruktor-Methode einer Sonderbehandlung bedürfen wenn wie im folgenden Code-Fragment die Auswertung seiner Argumente selbst eine Transformation erfordert, die Hilfsvariablen einführt.

```
public Foo(complex x, complex z) {
  super((x+x)+z);
}
```

In diesem Fall würde die Transformation zur Auswertung des komplexwertigen Ausdrucks `(x+x)+z` Hilfsvariablendeklarationen *vor* dem `super`-Aufruf anlegen, die in Java dort nicht erlaubt sind. Als Ausweg wird (verkürzt dargestellt) eine zusätzliche Konstruktor-Methode erzeugt, die neben den Parametern des ursprünglichen Konstruktors (`complex` aufgelöst) noch die benötigten Hilfsvariablen enthält.

```
private Foo(double x_real, double x_imag, // Parameter des ersten
            double z_real, double z_imag, // Konstruktors
            double tmp1_real, double tmp1_imag, // Hilfsvariablen
            double tmp2_real, double tmp2_imag) {
  super(seqREAL(seq(tmp1_real = x_real + x_real,
                tmp1_imag = x_imag + x_imag),
                tmp2_real = tmp1_real + z_real,
                tmp2_imag = tmp1_imag + z_imag),
        tmp2_imag);
}
```

Der `super`-Aufruf wurde oben ebenfalls modifiziert, um statt eines komplexwertigen Arguments zwei `double`-Werte zu akzeptieren.

5 Leistungsmessungen

5.1 Meßaufbau

Für die Leistungsmessungen verwenden wir einen Pentium 100 mit 64 MB Hauptspeicher und 512 KB Cache. Auf diesem Rechner sind Linux 2.0.36 (Suse 6.0) mit einer Vorabversion des JDK 1.2 und Windows NT Version 4 (Service Pack 4) mit diversen JDKs installiert. Um eine möglichst allgemeine Aussage zum Leistungsverhalten machen zu können, untersuchen wir auf der Windows-Plattform SUN's JDK 1.2.1 (sogenanntes Java 2), das im Internet-Explorer (Version 5) eingebaute JDK, ein von der IBM herausgegebenes JDK und schließlich eine Beta-Version des HotSpot (neuer Just-in-Time-Übersetzer von Sun).

[6] Um trotz Namensgleichheit Kollisionen durch Veränderung der Parametertypisierung auszuschließen, ergänzt *cj* die Parameterleiste um einen neuen Hilfstyp. Dies fehlt im Beispiel.

Als Benchmark-Programme untersuchen wir diverse Kern-Benchmarks, welche Feldzugriffe, die komplexwertige Basisarithmetik und die Leistung von Funktionsaufrufen mit komplexwertigen Argumenten/Rückgabewerten messen. Ferner nehmen wir Zeitmessungen für einige Anwendungskerne (Microstrip-Potentialfeldberechnung, komplexwertige Matrixmultiplikation und komplexwertige FFT) vor. Für alle Programme haben wir mindestens zwei Versionen: eine Version verwendet den Basistyp `complex`, die andere ist klassenbasiert.

5.2 Ergebnis

Gemittelt über alle Benchmarkprogramme ergibt sich, daß die Versionen, die auf dem Basistyp `complex` beruhen, *im Durchschnitt 2 bis 21 mal so schnell* (je nach JVM) ausgeführt werden, wie die klassenbasierten Lösungen. Die kleinere Verbesserung wird von guten Java-Implementierungen erreicht (HotSpot und Internet-Explorer), die Optimierungen der Bereichsüberprüfung bei Feldzugriffen vornehmen, eine schnellere Objekterzeugung haben und Methodenrümpfe vermehrt einbauen („inlining“). Die größten Verbesserungen erreichen die Basistyp-Versionen in der Regel mit dem JDK 1.2 auf der Windows-Plattform.

5.3 Ergebnisse im Detail

In Abbildung 1 sind die Ergebnisse zusammengestellt – aufgeteilt nach den sechs verschiedenen Benchmarks (a) bis (f). In jeder der sechs Teilabbildungen gibt es fünf Balkengruppen, je eine für die fünf untersuchten JVMs. Eine Balkengruppe besteht aus einem schwarzen Balken, der die relative Laufzeit der Programmversion angibt, die auf `Complex`-Objekten basiert. Der Faktor, um den die von *cj* übersetzte Version mit primitivem Datentyp (benachbarter grauer Balken) schneller ist, kann über dem schwarzen Balken abgelesen werden. Bei manchen Benchmarks gibt es auch noch eine Programmversion, bei der `complex` manuell durch zwei `double` ersetzt wurde. Diese handoptimierten Programmversionen (weiße Balken) sind nur geringfügig schneller als der von *cj* generierte Code.

Es ist in Teilabbildung (a) und (c) zu beobachten, daß die Beschleunigung geringer ausfällt als in den übrigen Benchmarks. Ferner ist zu sehen, daß die guten Java-Implementierungen (Internet-Explorer und HotSpot) den Zusatzaufwand der Objekt-Erzeugung in klassenbasierten Lösungen recht gut eliminieren können. Die Leistung von *cj* ist aber noch immer um mindestens 10-40% besser.

Während in (a) und (c) der Aufwand für Zugriffe auf komplexwertige Felder und der Aufwand für den Methodenaufruf mit komplexwertigem Rückgabeparameter untersucht wurde, untersuchen (b), (d), (e) und (f) vorwiegend die Leistung der Arithmetik, wobei (d) und (e) auch ein gewisses Maß an Feldzugriffen haben. Hier fällt auf, daß durch den in *cj* realisierten Einbau aller Methodenaufrufe bei möglichst geringer Verwendung von Hilfsvariablen eine deutlich höhe-

re Geschwindigkeit erzielt werden kann, als es bei klassenbasierten Lösungen möglich ist.

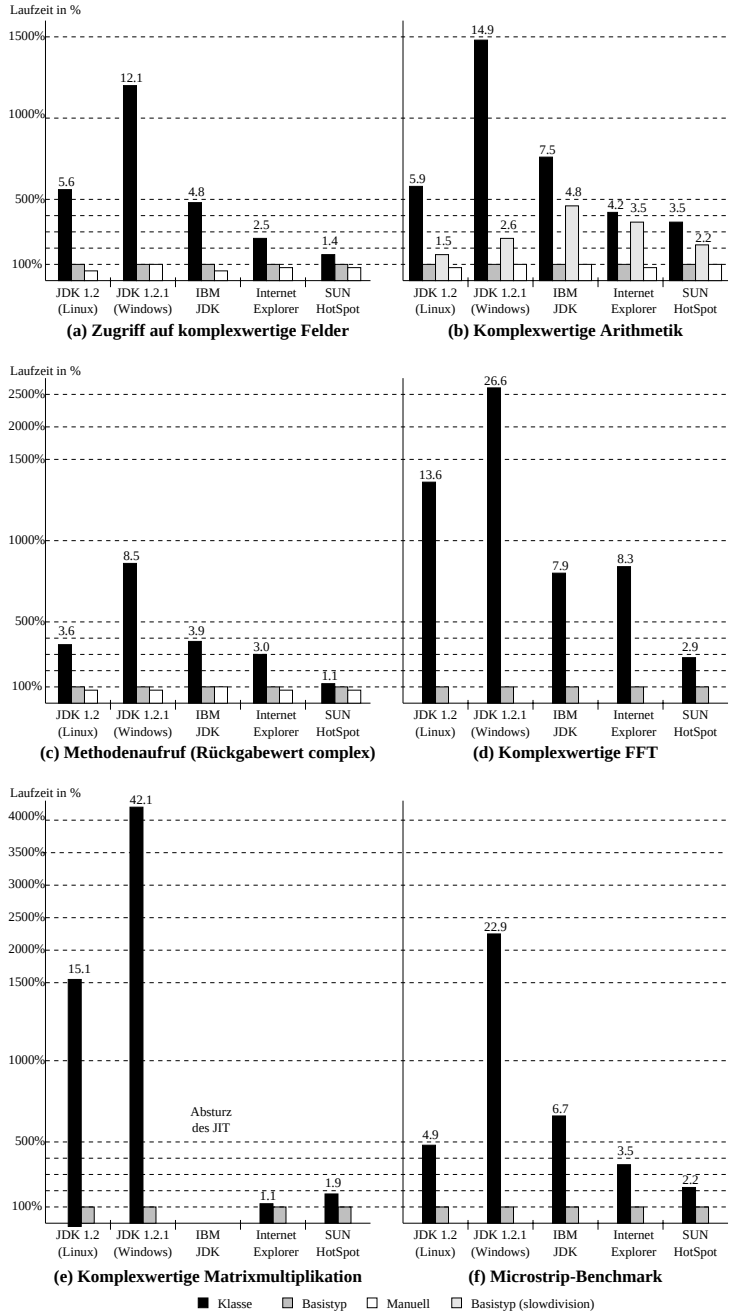

Abbildung 1: Leistung der Benchmark-Programme

Selbst die guten Java-Implementierungen werden mindestens um etwa Faktor 3 geschlagen. Mit den schlechteren Java-Implementierungen kann teilweise ein Faktor von über 26 erreicht werden. Dieses Optimierungspotential liegt wohl an den Methodenaufrufen und der zwangsläufig erforderlichen Instantiierung termporärer Objekte, die *cj* vollständig einsparen kann.

6 Zusammenfassung

Dieser Beitrag zeigt, wie komplexe Zahlen nahtlos und effizient zu Java ergänzt werden können. Wichtig dabei ist, daß die einfache Überführung in verdoppelte Operationen auf den real- und imaginär-Anteilen nicht ausreicht. Mit den Sequenzmethoden ist eine Notation für die erforderlichen lokalen Programmtransformationen vorgestellt worden. Die Technik zur Realisierung komplexwertiger Rückgabewerte aus Methoden zeigt sich als sehr effizient, weil die Erzeugung vieler temporärer Objekte eingespart werden kann. Insgesamt laufen die von *cj* übersetzten Programme 2 bis 21 mal so schnell wie klassenbasierte Vergleichsimplementierungen.

Danksagungen

Das Java Grande Forum und Siamak Hassanzadeh von Sun Microsystems unterstützten die Diskussion über mögliche komplexe Zahlen in Java finanziell. Martin Odersky gebührt unser Dank für das Bereitstellen von *gj*. Bernhard Haumacher und Lutz Prechelt gaben wertvolle Anregungen zur Verbesserung der Darstellung.

Literatur

1. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. of OOPSLA'98*, October 1998. http://www.cis.unisa.edu.au/~pizza/gj/.
2. C9x proposal. ftp://ftp.dmk.com/DMK/sc22wg14/c9x/complex/ and http://anubis.dkuug.dk/jtc1/sc22/wg14/.
3. J. D. Darcy and W. Kahan. Borneo language. http://www.cs.berkeley.edu/~darcy/Borneo.
4. J. Gosling. The evolution of numerical computing in Java. http://java.sun.com/people/jag/FP.html.
5. IBM. Numerical intensive java. http://www.alphaWorks.ibm.com/tech/ninja/.
6. Java Grande Forum. http://www.javagrande.org.
7. JavaParty. http://wwwipd.ira.uka.de/JavaParty/.
8. S. M. Omohundro and D. Stoutamire. The Sather 1.1 specification. Technical Report TR-96-012, ICSI, Berkeley, 1996.
9. G. Steele. Growing a language. In *Proc. of OOPSLA'98*, October 1998. key note.
10. G. K. Thiruvathukal, F. Breg, R. Boisvert, J. Darcy, G. C. Fox, D. Gannon, S. Hassanzadeh, J. Moreira, M. Philippsen, R. Pozo, and M. Snir (editors). Java Grande Forum Report: Making Java work for high-end computing. In *Supercomputing'98*, Orlando, Florida, November 1998. panel handout.
11. Visual Numerics. Java grande complex reference. http://www.vni.com/corner/garage/grande/index.html, 1999.
12. P. Wu, S. Midkiff, J. Moreira, and M. Gupta. Efficient support for complex numbers in Java. In *ACM 1999 Java Grande Conference*, San Francisco, 1999. pp 109–118.