Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 19

Reduktion des Speicherverbrauchs generierter SKilL-Zustände

Jonathan Roth

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. rer. nat. / Harvard Univ. Erhard Plödereder
Betreuer/in:	Dipl.-Inf. Timm Felden
Beginn am:	8. Dezember 2014
Beendet am:	9. Juni 2015
CR-Nummer:	C.4, D.3.3, D.3.4, E.2

Kurzfassung

Die an der Universität Stuttgart entwickelte Datenbeschreibungssprache SKilL (*S*erialization *Kil*ler *L*anguage) bietet eine Möglichkeit, große Datenmengen sprach- und plattformunabhängig zu serialisieren. Ihre aktuelle Anbindung an die Programmiersprache Scala hat aber das Problem, dass für große Datenmengen Geschwindigkeitseinbußen durch viele Garbage Collections verursacht werden und in einigen Fällen sogar Speicherüberläufe auftreten.

In dieser Arbeit werden Möglichkeiten untersucht, den Speicherverbrauch der Scala-Anbindung zu reduzieren. Der Speicherverbrauch der verschiedenen Implementierungen wird anschließend verglichen. Zu diesem Zweck wurde ein Testframework entwickelt, das die Erstellung von Tests vereinfacht und als Vorlage für entsprechende Tests in anderen Programmiersprachen verwendet werden kann.

Inhaltsverzeichnis

1. Einleitung

Die an der Universität Stuttgart entworfene Sprache SKilL (*S*erialization *Kil*ler *L*anguage) [Fel13] dient zur sprach- und plattformunabhängigen Serialisierung großer Datenmengen. Zu diesem Zweck wurden SKilL-Anbindungen für verschiedene Sprachen entwickelt, darunter Scala [SKi14a, O$^+$], Ada [Prz14], Java [Ung14] und C [Har14]. Beim Vergleich der SKilL-Anbindungen an Scala und an Ada wurde festgestellt, dass der generierte Scala-Code einen wesentlich höheren Speicherverbrauch hat als äquivalenter Ada-Code [Prz14].

Im Gegensatz zu Ada, welches eine systemnahe Programmiersprache ist, läuft Scala auf der Java Virtual Machine (JVM). Diese verwendet einen Garbage Collector und Just-In-Time-Compilation [Ora14]. Zudem erbt Scala von Java die Wurzel der Typhierarchie, die Klasse `java.lang.Object`, die bereits einen Grundspeicherverbrauch von 16 Byte pro Objekt mitbringt (auf einer 64-Bit-JVM). Dieser Speicher wird z. B. zur Synchronisation der Objektzugriffe, zur Speicherung des Alters eines Objekts und/oder des Hash-Codes verwendet [KW08].

Der von der Scala-Anbindung generierte Code benutzt viel Speicher. Das führt in Kombination mit dem Garbage Collector zu Geschwindigkeitseinbußen, da ein Großteil der Rechenleistung für teure Garbage Collections verwendet wird statt für zielführende Berechnungen. Weiterhin ist die Anzahl der gleichzeitig im Speicher gehaltenen Objekte dadurch stark begrenzt. Das kann dazu führen, dass z. B. ein Scala-Programm beim Einlesen einer von einem anderen Programm erzeugten SKilL-Datei abstürzt. Der Grund ist, dass der verfügbare Speicher vollständig aufgebraucht wird, obwohl die in der SKilL-Spezifikation aufgeführten numerischen Grenzwerte in der Datei eingehalten wurden [Fel13, Anhang D].

Nun übernimmt die JVM allerdings die gesamte Speicherverwaltung. Zudem existiert keine zuverlässige portable (d. h. für alle JVMs anwendbare) Möglichkeit, die JVM nach dem aktuellen Speicherverbrauch zu fragen. Das erschwert das Auffinden der Stellen, an denen die Scala-Anbindung Speicher verschwendet. Für die in dieser Arbeit verwendete JVM existiert aber eine nicht-portable Schnittstelle, über die der aktuelle Speicherverbrauch in regelmäßigen Zeitabständen abgefragt werden kann. Darauf basierend wurde im Rahmen dieser Arbeit ein Testframework zur Speichermessung für generierte Scala-Anbindungen entwickelt. Mit diesem wurde der Speicherverbrauch der aktuellen Implementierung evaluiert. Dazu wurden für die Speicherevaluation interessante SKilL-Spezifikationen und zugehörige Tests erstellt.

Auf den dabei gemachten Beobachtungen basierend werden in dieser Arbeit Möglichkeiten aufgezeigt, die Scala-Anbindung [SKi14a] so anzupassen, dass der generierte Code weniger Speicher verbraucht. Um die Effektivität dieser Vorschläge zu verifizieren, wurden die entwickelten Tests an die Benutzerschnittstelle der modifizierten Implementierung angepasst. Die Ergebnisse dieser Tests wurden mit den Ergebnissen für die aktuelle Implementierung verglichen.

1.1. Gliederung

In Kapitel 2 werden verwandte Arbeiten vorgestellt, sowohl zu den Tests als auch zur vorgestellten alternativen Implementierung der Scala-Anbindung.

Kapitel 3 enthält die Grundlagen der Sprachen Scala und SKilL sowie eine kurze Übersicht über den Heapaufbau der JVM.

Kapitel 4 beschreibt die alternative Implementierung. Dabei werden von der aktuellen Implementierung ausgehend die durchgeführten Veränderungen und deren Folgen beschrieben.

Kapitel 5 beschäftigt sich mit der Speichermessung auf der JVM.

Kapitel 6 beschreibt das Testframework und wie es verwendet werden kann.

In Kapitel 7 werden beide Implementierungen in Tests bezüglich ihres Speicherverbrauchs miteinander verglichen. Die dazu verwendeten Tests werden mithilfe des Testframeworks beschrieben.

Kapitel 8 schließt die Arbeit mit einer Zusammenfassung und einem Ausblick ab.

2. Verwandte Arbeiten

2.1. SKilL-Anbindung an Scala

Die Grundlage für diese Arbeit bildet die aktuelle Scala-Anbindung [SKi14a]. Die hier vorgestellte alternative Implementierung verwendet in einigen Teilen den gleichen Quellcode. Insbesondere bleibt die Benutzerschnittstelle der Basisimplementierung fast vollständig erhalten. Sie wird um ein paar neue Methoden ergänzt, die zur Erzeugung von Adaptern für Container dienen. Diese Adapter nehmen an anderen Stellen der Schnittstelle die Plätze der ursprünglichen Container ein.

Ein Vergleich der aktuellen Scala-Anbindung mit der hier vorgestellten Lösung ist im Kapitel 7 zu finden.

2.2. Benchmark-Tests

Die für den 16. Workshop Software-Reengineering und -Evolution entwickelten Benchmark-Tests [Fel14] wurden als Orientierungshilfe für die Entwicklung des Testframeworks und der verwendeten Tests benutzt. Das betrifft besonders die Implementierung der Ergebnisse (Trait `Result` und abgeleitete Klassen) aus dem Testframework. Auch die grafische Ausgabe der Ergebnisse als LaTeX-Datei baut auf der für die Benchmark-Tests implementierten Ausgabe auf.

Im Unterschied zu den Benchmark-Tests, die zur Laufzeitauswertung der generierten Scala-Anbindung gedacht sind, wird bei den hier verwendeten Tests der Speicherverbrauch gemessen. Zudem wurden die Messungen der Benchmark-Tests im selben Prozess wie der Test selbst ausgeführt. In den hier verwendeten Tests dagegen werden mehrere Prozesse verwendet.

2.3. SKilL-Anbindungen an andere Sprachen

Weitere SKilL-Anbindungen existieren zurzeit für die Programmiersprachen Ada, Java und C.

Die SKilL-Anbindung an Ada [Prz14] wurde entwickelt, um einen Vergleich verschiedener Anbindungen zu ermöglichen. Entsprechend enthält [Prz14] auch Performance-Tests zum Vergleich der Implementierungen. Im Gegensatz zu den in dieser Arbeit aufgeführten Tests wurde hier jedoch die Laufzeit, der Durchsatz und der Speicherverbrauch der erzeugten SKilL-Dateien ausgewertet.

Der Number-Test in dieser Arbeit basiert auf der selben Spezifikation wie der Number-Test in Abschnitt 6.1 von [Prz14].

Die SKilL-Anbindung an Java [Ung14] teilt einige Eigenschaften der Scala-Anbindung, da Java-Programme ebenfalls auf der JVM ausgeführt werden. Allerdings arbeitet diese mit Reflexion, um das Typsystem von SKilL zu speichern.

Die SKilL-Anbindung an C [Har14] wurde als Beweis entwickelt, dass eine Anbindung an eine nicht-objektorientierte Sprache möglich ist.

3. Grundlagen

Dieses Kapitel beschreibt in Kürze die wesentlichen Teile der Programmiersprache Scala [O$^+$], der Datenbeschreibungssprache SKilL [Fel13] und der JVM, insbesondere den Aufbau des Heaps.

3.1. Scala

Scala ist eine objektorientierte, funktionale Programmiersprache. Da sie für die JVM entwickelt wurde, teilt sie einige Eigenschaften mit der Programmiersprache Java. Insbesondere können Java-Klassen und Interfaces in Scala direkt verwendet werden. Dieser Abschnitt ist nur eine kurze Übersicht über Scala, genauere Informationen sind in der Scala-Spezifikation [O$^+$] zu finden. Es wird davon ausgegangen, dass der Leser die Programmiersprache Java kennt. Gemeinsame Elemente, wie z. B. Kommentare, werden nicht aufgeführt.

Innerhalb einer Scala-Codedatei können mehrere Klassen, Traits und Objekte definiert werden. Diese können sogar in unterschiedliche Pakete verteilt sein. [O$^+$, § 9]

3.1.1. Syntax

In Scala können Bezeichner für Klassen, Objekte und Methoden aus bestimmten Unicode-Zeichen der Basisebene bestehen, d. h. Zeichen mit den Codes U+0000 bis U+FFFF. Als Buchstaben zählen in Scala die Zeichen der Unicode-Kategorien `Ll` (Kleinbuchstaben), `Lu` (Großbuchstaben), `Lt` (Titelbuchstaben), `Lo` (Andere Buchstaben) und `Nl` (Buchstabenartige Zahlen) sowie die Zeichen _ und `$`. Die in Java üblichen Operatorzeichen sowie `#` und alle Zeichen der Unicode-Kategorien `Sm` (Mathematische Symbole) und `So` (Andere Symbole) zählen als Operatorzeichen. Ein Bezeichner kann bestehen aus:

- Buchstaben und Zahlen
- Operatorzeichen
- einer Folge von Buchstaben und Zahlen, gefolgt von _, gefolgt von Operatorzeichen
- einem beliebigen Text, eingeschlossen in ``

Schlüsselworte können nicht als Bezeichner verwendet werden, außer sie werden in `` eingeschlossen. [O$^+$, § 1.1]

Einige Beispiele für gültige Bezeichner (1. Zeile) und ungültige Bezeichner (2. Zeile, **`val`** ist ein Schlüsselwort):

Listing 3.1: Bezeichner

```
test123  X_X  _1  setter_=  ++  `val`  `das ist ein Test`
+_inv  3x  val  test++
```

Anweisungen können nicht nur mit `;` getrennt werden, sondern auch durch einen Zeilenwechsel. Zeilenwechsel als Anweisungstrenner werden aber nur innerhalb von geschweiften Klammern aktiviert und innerhalb von runden und eckigen Klammern deaktiviert. Es ist jeweils die nächste umschließende Klammerung entscheidend. Außerdem werden Zeilenwechsel in unvollständigen Ausdrücken, wie z. B. nach `.` oder einem Operator, dessen zweites Argument noch fehlt, nicht als Anweisungstrenner interpretiert. [O$^+$, § 1.2]

Anders als bei Java werden generische Typparameter nicht in spitze, sondern in eckige Klammern eingeschlossen. Ein Wildcard-Parameter wird durch `_` dargestellt. Kovarianz von Typparametern wird mit `+`, Kontravarianz mit `-` vor dem Parameter angegeben. Eine obere Grenze für einen Typparameter `T`, z. B. eine Basisklasse des Parameters, kann mit **`<:`** angegeben werden, eine untere Grenze, z. B. eine vom Parameter abgeleitete Klasse, mit **`>:`**. Arrayzugriffe werden nicht mit eckigen Klammern, sondern wie Funktionsaufrufe mit runden Klammern notiert. Typumwandlungen eines Ausdrucks `x` in einen Typ `T` werden mit `x.asInstanceOf[T]` durchgeführt. [O$^+$, § 3.2, 3.5, 4.4, 4.5, 12.1, 12.3.4]

Beispiel:

Listing 3.2: Typparameter

```
class Typen[+A,           // Kovariant
            -B,           // Kontravariant
            C >: X,       // C hat X als abgeleitete Klasse
            D <: AnyRef] { // D hat AnyRef als direkte oder indirekte Basisklasse
  /* ... */
}
```

3.1.2. Klassen und Traits

Klassen in Scala sind ähnlich den Klassen in Java: Sie können von genau einer Klasse oder einem Trait erben und beliebig viele Traits implementieren. Traits entsprechen ungefähr den Interfaces in Java. Allerdings können Traits Methoden mit einer Definition enthalten und eine Basisklasse besitzen. Implementierte Traits werden nicht wie Interfaces in Java mit `implements`, sondern mit **`with`** angegeben. [O$^+$, § 5.1, 5.4.1]

Die Definition einer Klasse ist gleichzeitig die Definition des primären Konstruktors der Klasse. Alle Inhalte der Klasse, die keine Feld- oder Methodendeklarationen sind, werden bei der Konstruktion einer Instanz der Klasse ausgeführt. Weitere Konstruktoren können innerhalb der Klasse als Methoden mit dem speziellen Namen **`this`** definiert werden, sie müssen aber als erste Anweisung einen anderen Konstruktor aufrufen, um das Objekt zu erzeugen. Die Parameter des primären Konstruktors werden hinter dem Namen der Klasse und einer eventuellen Sichtbarkeitseinschränkung für den Konstruktor angegeben. Diese Parameter können gleichzeitig Felddefinitionen sein. Hat eine Klasse (kein Trait!) eine Basisklasse, werden direkt hinter der Basisklasse die Parameter für ihren primären Konstruktor übergeben. Traits können keine Konstruktorparameter haben, da von ihnen keine Instanz erzeugt

werden kann. Ist der Inhalt einer Klassendefinition leer, können die geschweiften Klammern auch weggelassen werden. [O$^+$, § 5.1, 5.3]

Beispiel:

Listing 3.3: Klassen und Traits

```
class A(val a : Int) { /* Irgendwelche Definitionen */ }
trait T { /* Weitere Definitionen */ }
trait U extends A { /* Noch mehr Definitionen */ }
class B(b : Int) extends A(b) with T with U
```

Anders als in Java haben Klassen keine statischen Felder und Methoden. Stattdessen existieren Singleton-Objekte, die ähnlich wie Klassen ohne Typ- und Konstruktorparameter definiert werden. Dazu wird statt dem Schlüsselwort **`class`** das Schlüsselwort **`object`** verwendet. Eine Klasse oder ein Trait und ein Objekt mit dem selben Namen, die in der selben Datei definiert sind, haben Zugriff auf alle Felder und Methoden des jeweils anderen, auch private. Methoden und Felder in einem solchen Objekt können wie statische Methoden und Felder der Klasse behandelt werden. [O$^+$, § 5.5]

3.1.3. Methoden und Felder

Methoden werden mit dem Schlüsselwort **`def`** eingeführt. Überschreibt eine Methode eine Basisklassenimplementierung, muss dies explizit durch das Schlüsselwort **`override`** markiert werden. Wie im obigen Beispiel bereits zu sehen ist, werden Parameter in der Form `Name : Typ` angegeben. Steht vor dem Typ eines Parameters **`=>`**, so wird der übergebene Ausdruck erst bei der ersten Verwendung ausgewertet. Steht nach dem Typ des letzten Parameters ein `*`, so kann dieser Parameter beliebig oft wiederholt werden, analog zu `...` bei Java. Der Typ des Rückgabewerts wird mit `: Typ` nach der Parameterliste angegeben. Die Parameterliste kann vollständig weggelassen werden, falls eine Methode keine Parameter hat. Ebenso kann bei nicht-rekursiven Funktionen der Typ des Rückgabetyps vom Compiler abgeleitet und daher weggelassen werden. Die Definition von Methoden, die einen Wert zurückgeben, muss mit = beginnen; fehlt das Gleichzeichen, ist der Rückgabewert das Objekt `()` vom Typ `Unit`. [O$^+$, § 4.6, 5.2]

Für Felder gibt es zwei Schlüsselworte: **`val`** und **`var`**. **`val`** führt einen Wert ein, der nach der Initialisierung nicht mehr verändert werden kann, **`var`** dagegen einen veränderlichen Wert. Beide definieren gleichzeitig einen Scala-Getter für den Wert, **`var`** zusätzlich einen Scala-Setter. In Scala sind Getter einfache Methoden ohne Parameter und ohne Seiteneffekte, aber mit Rückgabewert, wie zum Beispiel **`def`** `x = /* ... */`. Setter sind Methoden, deren Name auf `_=` endet und die genau einen Parameter besitzen, z. B. **`def`** `x_=(X : Int) = /* ... */`. Sie werden bei Scala auf eine besondere Weise aufgerufen: Der Ausdruck `x = /* ... */` wird im gegebenen Beispiel in einen Aufruf `x_=(/* ... */)` umgewandelt. **`val`** und **`var`** werden auch zur Einführung lokaler Variablen verwendet. [O$^+$, § 4.1, 4.2, 6.15]

Einige Methodennamen haben eine spezielle Bedeutung [O$^+$, § 6.6, 6.15, 8.1.8]:

`apply` (mit beliebigen Parametern) wird aufgerufen, wenn nach einem Objekt eine passende Parameterliste steht.

update (mit beliebigen Parametern) wird aufgerufen, wenn einem Objekt mit Parameterliste etwas zugewiesen wird.

unapply (in einem Objekt, mit genau einem Parameter) wird aufgerufen, um aus dem Parameter Werte zu extrahieren, z. B. Werte von Feldern.

Beispiel:

Listing 3.4: apply und update

```
class X {
  def apply(i : Int) = /* ... */
  def update(i : Int, value : Int) = /* ... */
}

val x = new X
x(1)          // ruft x.apply(1) auf
x(1) = 2      // ruft x.update(1, 2) auf
```

Eine Klasse, ein Trait oder ein Objekt kann auch Typdefinitionen enthalten. Eine Typdefinition der Form **type** `T = Typ` definiert `T` als Alias für den Typ rechts vom Gleichzeichen. [O$^+$, § 4.3]

3.1.4. Modifizierer

Klassen-, Trait- und Objektdefinitionen können mit Modifizierern versehen sein. Diese stehen vor dem Schlüsselwort **class**, **trait** oder **object**. Nicht alle Modifizierer sind für alle Arten von Definitionen erlaubt [O$^+$, § 5.2, 5.4]:

sealed (für Klassen und Traits) gibt an, dass nur Klassen und Traits in der selben Codedatei von der Klasse bzw. dem Trait erben können.

final (für Klassen) gibt an, dass von einer Klasse nicht geerbt werden kann. Das selbe Schlüsselwort wird für Methoden verwendet, die nicht überschrieben werden können.

abstract (für Klassen) gibt an, dass von einer Klasse keine Instanz erzeugt werden kann.

case (für Klassen und Objekte) sorgt dafür, dass bestimmte Methoden vom Compiler generiert werden, unter anderem eine Überschreibung der `equals`- und `hashCode`-Methoden. Für Klassen wird im Companion-Objekt eine `apply`-Methode definiert, die ein neues Objekt der Klasse erzeugt sowie eine `unapply`-Methode, die die zur Konstruktion verwendeten Parameter aus einem Objekt extrahiert. Solche Klassen können also ohne das Schlüsselwort **new** erzeugt werden.

Standardmäßig sind alle Konstruktoren, Klassen, Traits, Objekte, Methoden und Felder öffentlich. Daher besitzt Scala kein Schlüsselwort für öffentliche Sichtbarkeit. Zum Einschränken der Sichtbarkeit können unter anderem folgende Angaben verwendet werden [O$^+$, § 5.2]:

private Nur die definierende Klasse bzw. das definierende Objekt hat Zugriff.

protected Nur die definierende Klasse und abgeleitete Klassen haben Zugriff.

private[x] Alle Klassen und Objekte, die innerhalb des umschließenden Pakets, der umschließenden Klasse bzw. des umschließenden Objekts `x` definiert sind, haben Zugriff.

`protected[x]` Alle Klassen und Objekte, die innerhalb des umschließenden Pakets, der umschließenden Klasse bzw. des umschließenden Objekts `x` definiert sind, sowie abgeleitete Klassen haben Zugriff.

3.1.5. Ausdrücke

In Scala sind alle Anweisungen Ausdrücke, einschließlich Codeblöcken und anderen üblichen Konstrukten wie **`if ... else`**, **`try ... catch`** usw. und haben einen wohldefinierten Typ. Der Typ eines Codeblocks ist der Typ der letzten Anweisung, der Typ einer **`if ... else`**-Anweisung ist der genaueste gemeinsame Typ beider Zweige. Fehlt der **`else`**-Zweig, ist dessen Typ als `Unit` festgelegt. [O$^+$, § 6.11, 6.16]

In Scala existieren zwei Formen von **`for`**-Ausdrücken:

Listing 3.5: for-Ausdrücke

```
for (value <- collection) /* Ausdruck mit value */

for (value <- collection) yield /* Ausdruck mit value */
```

Die erste Form führt für alle Werte in `collection` den Ausdruck aus, wobei `value` an den aktuellen Wert gebunden wird und hat den Typ `Unit`. Die zweite Form erzeugt eine neue Liste von Werten, meist vom selben Listentyp wie `collection`, die für jeden Wert von `collection` das Ergebnis des Ausdrucks für diesen Wert enthält. Auch hier wird `value` an den aktuellen Wert gebunden. In beiden Fällen ist `value` unveränderlich. [O$^+$, § 6.19]

`value` kann auch eine komplexere Form haben, wie zum Beispiel `X(val1, val2)`. In diesem Fall wird für das Objekt `X` die `unapply`-Methode mit dem aktuellen Wert als Argument aufgerufen und `val1` sowie `val2` an die extrahierten Werte gebunden. Hinter `collection` dürfen auch durch `;` getrennt weitere lokale Werte definiert werden, die im gegebenen Code ebenfalls unveränderlich sind. [O$^+$, § 6.19, 8.1.8]

Beispiel:

Listing 3.6: Komplexe for-Ausdrücke

```
case class X(val _1 : Int, val _2 : Int)

val collection : Array[X] = /* ... */
for (X(val1, val2) <- collection; sum = val1 + val2)
  yield sum
```

3.1.6. Vordefinierte Typen

Die Wurzel der Scala-Typhierarchie ist `scala.Any`. Diese hat genau zwei abgeleitete Klassen, `scala.AnyVal` und `scala.AnyRef`. `AnyRef` entspricht `java.lang.Object`, `Any` und `AnyVal` haben kein Java-Äquivalent. `AnyRef` ist die Basisklasse aller Referenztypen, d. h. aller Klassen, Traits und Objekte, die nicht explizit von einer anderen Klasse erben. `AnyVal` ist die Basisklasse aller Werttypen, d. h. Typen, die in Java keine Klassen sind. Da diese Typen in Scala aber gleichzeitig einen primitiven Typ

Tabelle 3.1.: Zuordnung von Scala-Werttypen zu Java-Typen

Scala-Werttyp	primitiver Java-Typ	Boxtyp
`scala.Boolean`	`boolean`	`java.lang.Boolean`
`scala.Byte`	`byte`	`java.lang.Byte`
`scala.Short`	`short`	`java.lang.Short`
`scala.Int`	`int`	`java.lang.Integer`
`scala.Long`	`long`	`java.lang.Long`
`scala.Float`	`float`	`java.lang.Float`
`scala.Double`	`double`	`java.lang.Double`
`scala.Char`	`char`	`java.lang.Character`
`scala.Unit`[a]	`void`[a]	`java.lang.Void`[a]

[a]Nur als Rückgabetyp von Funktionen äquivalent

und die zugehörige Box (d. h. ein Objekt, das einen primitiven Wert enthält, z. B. `java.lang.Integer`) bezeichnen, können sie auch als generische Parameter verwendet werden. Boxing und Unboxing geschieht bei Verwendung von Werttypen automatisch, wenn nötig. [O$^+$, § 12.1, 12.5.1]

Tabelle 3.1 listet die Werttypen von Scala mit ihren Java-Entsprechungen auf. Das einzige Objekt vom Typ `Unit` ist das leere Tupel, `()`.

Der Typ `scala.Nothing` erbt per Definition von *allen* Typen. Von diesem Typ gibt es keine Instanzen. Eine Methode, die `Nothing` zurückgibt, kann nicht normal zurückkehren, sondern muss eine Ausnahme werfen. Der Typ `scala.Null` mit der einzigen Instanz **`null`** erbt per Definition von *allen* Referenztypen.

`scala.Array[T]` in Scala entspricht dem Java-Array `T[]`. [O$^+$, § 12.3.4]

Weiterhin spielen in dieser Arbeit folgende Typen eine Rolle:

`scala.Option[T]` Repräsentiert einen optionalen Wert vom Typ `T`. Entweder eine Instanz von `scala.Some[T]`, falls ein Wert existiert oder das Objekt `scala.None`. `Some[T]` ist mit **`case`** definiert.

`scala.collection.mutable.ArrayBuffer[T]` Repräsentiert eine erweiterbare Liste von Objekten vom Typ `T` mit schnellem wahlfreien Zugriff, da intern ein Array verwendet wird.

`scala.collection.mutable.ListBuffer[T]` Repräsentiert eine erweiterbare verlinkte Liste von Objekten vom Typ `T`.

`scala.collection.mutable.HashSet[T]` Repräsentiert eine Menge von Objekten vom Typ `T`, gespeichert in einer Hash-Tabelle.

`scala.collection.mutable.HashMap[K, V]` Repräsentiert eine Map mit Schlüsseln vom Typ `K` und Werten vom Typ `V`, gespeichert in einer Hash-Tabelle.

Im Rest dieser Arbeit wird für alle in diesem Abschnitt beschriebenen Typen das enthaltende Paket weggelassen.

3.2. SKilL

SKilL [Fel13] besteht aus einer Datenbeschreibungssprache und einem Serialisierungsformat. Mit der Datenbeschreibungssprache können SKilL-Spezifikationen geschrieben werden, aus denen SKilL-Anbindungen für verschiedene Programmiersprachen generiert werden können. Sie orientiert sich an Programmiersprachen wie C++ und Java, so dass Entwickler dieser Sprachen die Spezifikationen lesen können. Um kompatibel mit möglichst vielen Programmiersprachen zu sein, ignoriert die Datenbeschreibungssprache Groß- und Kleinschreibung. Das Serialisierungsformat beschreibt den Aufbau der SKilL-Dateien, die von einer SKilL-Anbindung geschrieben und gelesen werden können.

Dieser Abschnitt beschreibt die Typen, die in SKilL-Spezifikationen verwendet werden können sowie den Grobaufbau einer generierten SKilL-Anbindung und in Kürze das Serialisierungsformat.

3.2.1. Typen

Die Typen in SKilL lassen sich in drei Bereiche aufteilen: Eingebaute Typen, zusammengesetzte Typen und Benutzertypen.

Die eingebauten Typen sind [Fel13, § 4.1]:

i8, i16, i32, i64 Ganzzahlige Werte fester Länge mit 8, 16, 32 bzw. 64 Bit.

v64 Ganzzahlige Werte mit 64 Bit Länge, die aber mit einer variablen Anzahl an Bytes serialisiert werden. Kleine Zahlen (im Intervall $[0, 128)$) werden mit einem Byte, große ($\geq 2^{55}$) und negative Zahlen dagegen mit neun Byte serialisiert.

f32, f64 Fließkommazahlen mit 32 bzw. 64 Bit, die entsprechend der Norm IEEE 754 [IEE08] kodiert sind.

bool Wahrheitswerte, die die beiden Werte wahr und falsch annehmen können.

string Zeichenketten mit variabler Länge. Der Inhalt besteht (im Serialisierungsformat) aus UTF-8-kodierten Unicode-Zeichen.

annotation Ein Zeiger auf einen beliebigen Benutzertyp. Zusätzlich zum Ziel wird der Typ des Ziels gespeichert.

Benutzertypen sind in SKilL-Spezifikationen spezifizierte oder aus SKilL-Dateien gelesene Typen. In einer SKilL-Spezifikation definierte Benutzertypen werden für die daraus generierte SKilL-Anbindung als bekannte Typen bezeichnet, alle anderen als unbekannte Typen.

Benutzertypen setzen sich aus Feldern zusammen, die einen beliebigen SKilL-Typ haben können. Felder werden wie in Java in der Form *Typ Name;* definiert. Alle Felder innerhalb einer Typdefinition müssen verschiedene Namen besitzen. Felder mit ganzzahligem Typ können auch mit ***const*** markiert werden, um Konstanten anzugeben. Der Wert von Konstanten wird in der SKilL-Spezifikation festgelegt und beim Einlesen einer SKilL-Datei überprüft. Falls der gelesene und der spezifizierte Wert verschieden sind, wird ein Fehler beim Einlesen gemeldet. Felder beliebigen Typs können mit ***auto*** markiert werden. In diesem Fall wird zwar ein entsprechendes Feld im generierten Code erzeugt, das Feld wird aber bei der Serialisierung ignoriert. [Fel13, § 3.4, 4.3]

Bekannte Benutzertypen können andere bekannte Benutzertypen erweitern, analog zum Ableiten von Klassen. Entsprechend sind zyklische Abhängigkeiten beim Erweitern nicht erlaubt. SKilL unterstützt nur einfache Vererbung, ein Benutzertypen kann also nur einen anderen Typ erweitern. Ein abgeleiteter Typ wird als Untertyp bezeichnet, der Typ, der erweitert wird, als Obertyp. Ein Typ, der keinen Obertyp besitzt, d. h. die Wurzel einer Typhierarchie ist, wird Basistyp genannt. [Fel13, § 3.3, 4.3]

Beispiel für einen Benutzertyp und Erweiterung:

Listing 3.7: Benutzertypen

```
/** Definiert einen Typ A mit einem Feld a. */
A { i32 a; }
/** Erweitert A um ein Feld b. Alternativen zu : sind extends und with. */
B : A { A b; }
```

Benutzertypen und Felder können mit vorangestellten Einschränkungen (Restrictions) und Hinweisen (Hints) versehen werden. Einschränkungen werden serialisiert, Hinweise dagegen beeinflussen lediglich die generierte SKilL-Anbindung. Einschränkungen werden mit einem *@*-Zeichen eingeleitet, Hinweise mit *!*. [Fel13, § 3.3]

Zusammengesetzte Typen bezeichnen Typen, die sich aus mehreren Einträgen des selben Typs (bzw. der selben Typen) zusammensetzen. Das sind Arrays fester Länge, Arrays variabler Länge, Listen, Mengen und Maps. Alle zusammengesetzten Typen haben Typparameter, für die eingebaute Typen oder Benutzertypen eingesetzt werden können. Das folgende Listing zeigt die Deklaration von Feldern mit zusammengesetzten Typen. [Fel13, § 3.5, 4.2]

Listing 3.8: Zusammengesetzte Typen

```
A {
  /** Ein Array fester Länge mit Elementen vom Typ string und der Länge 15. */
  string[15] a;
  /** Ein Array variabler Länge mit Elementen vom Typ i32. */
  i32[] b;
  /** Eine Liste mit Elementen vom Typ A. */
  list<A> c;
  /** Eine Menge mit Elementen vom Typ annotation. */
  set<annotation> d;
  /** Eine Map mit Schlüsseln vom Typ bool und Werten vom Typ i8. */
  map<bool, i8> e;
  /** Eine verschachtelte Map. */
  map<i16, A, f32> f;
}
```

3.2.2. Aufbau der SKilL-Anbindung an Scala

Das Diagramm 3.1 zeigt den groben Aufbau der Scala-Anbindung. Der Benutzer sieht dabei nur den Teil, der in der oberen Zeile aufgeführt ist (Frontend), der Rest wird von der Anbindung verborgen (Backend).

Benutzertypen sind hier allgemeine Klassen für unbekannte Typen sowie Klassen, die aus einer SKilL-Spezifikation generiert wurden. Für jede Typdeklaration einer SKilL-Spezifikation wird eine solche

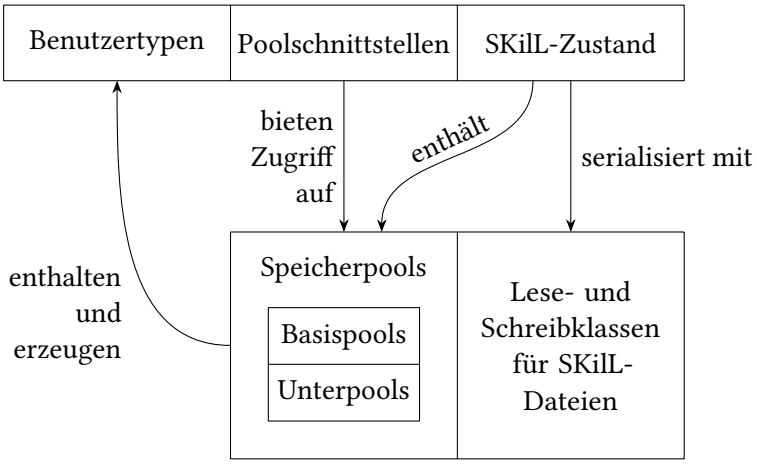

Abbildung 3.1.: Aufbau der Scala-Anbindung

Klasse generiert. Parallel dazu werden Speicherpools erzeugt. Zu jedem Basistyp wird ein Basispool generiert, für alle anderen Typen je ein Unterpool. Über die öffentliche Schnittstelle der Pools können Instanzen von bekannten Benutzertypen erzeugt werden. Außerdem kann über alle Instanzen eines Benutzertyps (einschließlich abgeleiteten Typen) iteriert werden. Der SKilL-Zustand schließlich ist die Repräsentation einer SKilL-Datei im Speicher. Er enthält für alle in einer SKilL-Datei vorkommenden und alle spezifizierten Benutzertypen je einen Speicherpool, insbesondere auch für unbekannte Typen. Ein SKilL-Zustand bietet Lese-, Schreib- und Anhängoperationen für SKilL-Dateien an und leitet die entsprechenden Aufrufe an die zuständigen internen Klassen weiter.

3.2.3. Serialisierungsformat

Eine SKilL-Datei besteht aus Blöcken. Das ermöglicht das Anhängen neuer Objekte an bestehende Daten, ohne dass an diesen etwas geändert werden muss. Außerdem ist das Format darauf optimiert, dass Teile der Datei ohne großen Aufwand übersprungen werden können. Es gibt zwei Typen von Blöcken: Zeichenkettenblöcke und Typblöcke. Diese treten immer in Paaren auf, erst ein Zeichenkettenblock, dann ein Typblock. [Fel13, § 6.2]

Ein Zeichenkettenblock beginnt mit einem ***v64*** *count*, das die Anzahl der Zeichenketten im Block angibt. Anschließend kommen *count* ***i32***-Werte, die jeweils die Endposition einer Zeichenkette relativ zum ersten Zeichen der ersten Zeichenkette angeben, gefolgt von den UTF-8-kodierten Zeichenketten. In einem Zeichenkettenblock sind sowohl Namen von Typen und Feldern der in der Datei enthaltenen Typdeklarationen (immer in Kleinbuchstabenform) als auch Benutzerzeichenfolgen enthalten, d. h. Werte von Feldern vom Typ ***string***. Zeichenketten werden in Typblöcken über ihren 1-basierten Index (gespeichert als ***v64***) in der Datei adressiert. [Fel13, § 6.2.1]

Ein Typblock beginnt ebenfalls mit einem ***v64*** *count*. Dieses gibt an, wie viele Typdeklarationen im Typblock enthalten sind. Für jeden Typ, für den in einem Typblock Instanzen enthalten sind, muss eine Typdeklaration enthalten sein. Anschließend folgen *count* Typdeklarationen und nach diesen feldweise geordnet die Daten für die einzelnen Instanzen. Die Typdeklarationen sind so geordnet, dass

ein Obertyp immer vor allen seinen Untertypen deklariert wird. Eine Typdeklaration besteht aus zwei Teilen: Einem Informationsblock über den Typ selbst, gefolgt von einer Liste von Felddeklarationen. Abbildung 3.2 veranschaulicht den Aufbau des Informationsblocks und einer Felddeklaration. [Fel13, § 6.2.2]

Informationsblock

string *name*
$?_1$ ***string*** *super*
$?_3$ ***v64*** *LBPSI*
v64 *count*
$?_1$ *R[] restrictions*
v64 *LFieldCount*

Felddeklaration

$?_2$ *R[] restrictions*
$?_2$ *TYPE type*
$?_2$ ***string*** *name*
v64 *end*

Abbildung 3.2.: Aufbau einer Typdeklaration: Informationsblock und Felddeklaration [Fel13, § 6.2]

Dabei sind mit $?_1$ gekennzeichnete Felder nur vorhanden, wenn ein Typ das erste Mal in einer SKilL-Datei vorkommt. Besitzt ein Typ keinen Obertyp, hat *super* den Wert 0. *count* im Informationsblock gibt an, wie viele neue Instanzen des Typs (einschließlich Untertypen) im Typblock dazukommen. *LFieldCount* ist die Anzahl der Felder des Typs, für die im Typblock Daten gespeichert sind. Die mit $?_2$ markierten Felder sind nur vorhanden, wenn ein Feld das erste Mal vorkommt. Felder, die neu hinzukommen, stehen hinter den bereits vorhandenen. Der Wert *end* gibt an, wo die Daten eines Feldes relativ zum Ende der letzten Typdeklaration enden. Das mit $?_3$ gekennzeichnete Feld ist nur vorhanden, wenn ein Typ einen Obertyp besitzt. In diesem Fall gibt es an, ab welchem Index (1-basiert) die Daten des Basistyps im selben Block zu diesem Typ gehören. [Fel13, § 6.2.2]

3.3. Speicherverwaltung auf der JVM

Dieser Abschnitt beschreibt in Kürze den Aufbau des Heaps und die Funktionsweise des Garbage Collectors für die verwendete JVM. Für andere JVMs wird keine Aussage gemacht; das Verhalten dürfte jedoch sehr ähnlich sein.

Der Heap ist in der JVM nicht ein großer Speicherblock, sondern in mehrere Bereiche (sogenannte Generationen) aufgeteilt. Die Generationen haben verschiedene Aufgaben. [GCT]

- Die *permanente Generation* enthält interne Daten der JVM, wie z. B. geladene Klasseninformationen. Sie hat eine feste Größe, die beim Start der JVM festgelegt wird und spielt für ein Programm auf der JVM nur eine untergeordnete Rolle.
- Die *junge Generation* enthält neue Objekte bis zu einer bestimmten Größe. Sie ist weiter unterteilt in einen Bereich für neu erzeugte Objekte („Eden“) und zwei Bereiche für Objekte, die bereits mindestens eine Garbage Collection in der jungen Generation überlebt haben („Survivor spaces“). Die Größe der jungen Generation ist verhältnismäßig klein.
- Die *alte Generation* enthält Objekte, die in der jungen Generation genug Garbage Collections überlebt haben und neue Objekte, die zu groß für die junge Generation sind.

Garbage Collections in der jungen Generation sind relativ billig, da diese relativ klein ist und zudem viele der enthaltenen Objekte nur temporäre Daten sind und nur wenige eine Garbage Collection überleben. Die alte Generation dagegen ist die größte der Generationen. In ihr befinden sich viele langlebige Objekte, wodurch sich der Aufwand für Garbage Collections erhöht. Daher werden Garbage Collections in der jungen Generation klein und Garbage Collections in der alten Generation groß genannt. [GCT]

4. Speicherreduktion von SKilL-Zuständen

Dieses Kapitel beschreibt mögliche Veränderungen in der bestehenden Scala-Anbindung, die zu einem reduzierten Speicherverbrauch führen. Dazu werden zuerst Teile der ursprünglichen Implementierung [SKi14a] betrachtet, die bezüglich ihres Speicherverbrauchs optimierbar sind. Anschließend werden darauf aufbauende Optimierungen vorgestellt. Zu den Optimierungen ist auch angegeben, inwiefern diese die Laufzeit beeinflussen können. Im Normalfall überwiegt jedoch bei großen Objektzahlen die Laufzeitverbesserung durch weniger große Garbage Collections.

4.1. Ursprüngliche Implementierung

Die ursprüngliche Implementierung ist auf GitHub (siehe [SKi14a]) zu finden. Dieser Abschnitt beleuchtet nur die für die folgenden Optimierungen wesentlichen Teile. Alle Aussagen in diesem Abschnitt beziehen sich auf die ursprüngliche Implementierung. Das vom Codegenerator erzeugte Paket wird im Folgenden mit x bezeichnet.

4.1.1. Benutzertypen

Die in der SKilL-Spezifikation definierten Benutzertypen [Fel13, § 4.3] werden auf Klassen abgebildet, die direkt oder indirekt von x`.internal.SkillType` erben. Die in einer SKilL-Spezifikation definierte Typhierarchie wird durch abgeleitete Klassen implementiert. Die Konstruktoren der Klassen sind nur innerhalb des Pakets x zugreifbar. Der vorgesehene Weg, neue Instanzen zu erzeugen, führt über die in einem SKilL-Zustand enthaltenen Objektlager, die Speicherpools. Normale Felder, im Folgenden lokale Felder genannt, sind übliche Felder innerhalb der generierten Klassen. Die Schnittstelle für den Benutzer besteht aus einer für Scala üblichen Getter/Setter-Implementierung [O^+, § 4.2]. Diese kann Fehler werfen, falls z. B. ein Feld ignoriert wird, d. h. mit einem ignore-Hint versehen ist [Fel13, § 5.2], oder eine Einschränkung verletzt wird. Verweise auf andere Objekte im selben Zustand werden durch übliche Referenzen realisiert.

Jedes Objekt enthält eine Identifikationsnummer, die SKilL-ID, vom Typ `Long` im Feld `skillID` von `SkillType`. Objekte, die aus einer SKilL-Datei gelesen oder in eine SKilL-Datei geschrieben wurden, im Folgenden alte Objekte genannt, haben eindeutige, positive SKilL-IDs. Neu erzeugte und noch nicht geschriebene Objekte, im Folgenden neue Objekte, bekommen die spezielle SKilL-ID -1. Die SKilL-ID 0 wird zur Markierung gelöschter Objekte verwendet; sie ist beim Serialisieren für Nullverweise reserviert (siehe [Fel13, § 6.3]).

Für Typen und Felder können auch Einschränkungen in der Spezifikation angegeben werden. Da diese aber in der aktuellen Form als veraltet deklariert wurden, werden sie in dieser Arbeit nicht beachtet.[1]

Laufendes Beispiel

Die folgende einfache SKilL-Spezifikation wird in diesem Kapitel immer wieder aufgegriffen. Spätere Beispiele bauen darauf auf:

Listing 4.1: Laufendes Beispiel – Hierarchie

```
A {
  i32 x;
}
B : A {
  A y;
}
```

Diese Spezifikation wird in der ursprünglichen Implementierung implementiert als

Listing 4.2: Hierarchie – Implementierung

```
sealed class A private[x](skillID: Long) extends SkillType(skillID) {
  ...
  protected var _x: Int = 0
  final def x = _x
  final def x_=(X: Int) = _x = X
  ...
}
sealed class B private[x](skillID: Long) extends A(skillID) {
  ...
  protected var _y: A = null
  final def y = _y
  final def y_=(Y: A) = { /* Prüfung: Y != null */; _y = Y }
  ...
}
```

4.1.2. Objekte in SKilL-Zuständen

Zu jedem Benutzertyp, einschließlich der nicht-spezifizierten, aus einer SKilL-Datei gelesenen Typen, existiert ein assoziierter Speicherpool in jedem SKilL-Zustand (siehe auch [Fel13, § 6.3]). Ebenso ist zu jedem Speicherpool genau ein Benutzertyp assoziiert. Speicherpools erben von der Klasse `StoragePool` über eine ihrer beiden Unterklassen `BasePool` und `SubPool`. Ein Speicherpool enthält Verweise auf alle enthaltenen Objekte, alte ebenso wie neue, und Hilfsmethoden für die Serialisierung und Deserialisierung von Speicherpools in binäre SKilL-Dateien. Er erlaubt auch den Zugriff auf unbekannte Felder per Reflexion und enthält deren Daten für jedes Objekt des assoziierten Typs,

[1] Die speicheroptimierte Implementierung enthält dennoch Code für Einschränkungen, geerbt von der ursprünglichen Implementierung, teilweise mit Anpassungen an die neue Implementierung.

d. h. unbekannte Felder sind verteilt (siehe nächsten Abschnitt). Zusätzlich enthalten Speicherpools Verweise auf alle Unterpools und ihren Oberpool, jeweils falls vorhanden.

Speicherpools gliedern sich in zwei Untertypen: Basispools (Klasse `BasePool`) gehören zu Basistypen, Unterpools (Klasse `SubPool`) zu allen anderen Typen. Unbekannte Typhierarchien verwenden direkt diese Klassen. Für bekannte Typen und deren unbekannte Untertypen werden Klassen generiert, die von einer dieser beiden Klassen erben. Ein Basispool enthält alle alten Objekte, die vom zugehörigen oder einem davon abgeleiteten Typ sind. Bei neuen Objekten ist jeder Speicherpool für alle Objekte des zugehörigen Typs zuständig.

Entsprechend enthalten Basispools ein Array, das Verweise auf alle alten Objekte des assoziierten Typs und aller abgeleiteten Typen beinhaltet. Außerdem enthalten alle Speicherpools einen `ArrayBuffer`, der Verweise auf alle neuen Objekte des assoziierten Typs enthält.

Speicherpools erlauben das Iterieren über alle enthaltenen Instanzen. Dazu gehören alle Objekte, die vom assoziierten Typ oder davon abgeleitet sind. Dabei existieren zwei Reihenfolgen:

- Dateiordnung, d. h. die Reihenfolge, in der Instanzen beim Anhängen an eine Datei geschrieben werden würden
- Typordnung, d. h. zuerst alle Objekte eines Typs, dann alle Objekte des nächsten Typs usw.

Außerdem können alte Objekte intern auch durch ihre SKilL-ID erhalten werden.

Generierte Speicherpools für bekannte Typen erlauben zusätzlich das Erzeugen neuer Objekte mittels ihrer `apply`-Methode. Davon ausgenommen sind Speicherpools für Singletons. Diese haben stattdessen eine `get`-Methode, die die einzige Instanz zurückgibt.

4.2. Verteilte vs. lokale Felder

Da in der alten Implementierung jede Instanz einer von `SkillType` abgeleiteten Klasse die gesamte Lebenszeit des enthaltenden Zustands überdauert, ist es interessant zu betrachten, wie viel Speicher die Instanzen eigentlich selbst belegen. Auf der JVM (64 Bit) benötigt jedes Objekt mindestens 16 Byte[2]. Das ist die Größe von `java.lang.Object`, der direkten oder indirekten Basisklasse jedes Objekts auf der JVM. Für SKilL-Objekte kommen nun noch die SKilL-ID und alle Felder hinzu.

Betrachtet man ein Objekt ohne Felder, so benötigt dieses bereits mindestens 24 Byte[3]. Dieser Grundspeicherbedarf kann nicht reduziert werden, solange Referenzen auf jedes dieser Objekte existieren. Daher stellt sich die Frage, ob die Objekte wirklich benötigt werden.

Die SKilL-Spezifikation [Fel13] erlaubt das Deklarieren von Feldern als „distributed“ (verteilt) mittels eines Hints wie folgt:

[2] Ermittelt mit VisualVM [Vis14] für die verwendete JVM

[3] Aufgrund der unglücklichen Benennung eines Konstruktorparameters (`skillID`) wird dieser innerhalb einer Methode verwendet statt des gleichnamigen Felds von `SkillType`. Daher kommen noch einmal 8 Byte pro Hierarchiestufe einer Klasse hinzu, d. h. erbt `B` (ohne eigene Felder) von `A`, so benötigt eine Instanz von `B` 8 Byte mehr als eine Instanz von `A`.

Listing 4.3: Distributed-Hint

```
A {
  !distributed
  i32 x;
}
```

Die ursprüngliche Implementierung unterstützt keine verteilten Felder außer den unbekannten, dort sind bekannte Felder immer lokal. Die Daten eines verteilten Felds sind im Gegensatz zu einem lokalen Feld nicht im Objekt, sondern im zugehörigen Speicherpool gespeichert. Daher müssen sie auf irgendeinem Weg dem passenden Objekt zugeordnet werden können. Eine Möglichkeit besteht darin, die Daten in einer Map zu speichern, welche mit dem Objekt indiziert wird. Diese Möglichkeit wird in der ursprünglichen Implementierung für unbekannte Felder verwendet. Vom Speicherverbrauch her besser ist aber die Verwendung eines Arrays, das die Daten des Felds enthält und mittels der SKilL-IDs der Objekte indiziert wird. Dazu muss aber jedes Objekt eines Speicherpools eine eindeutige SKilL-ID besitzen, was bisher nicht gegeben ist. Zudem erfordert die Anwesenheit eines oder mehrerer verteilter Felder, dass in jedem Objekt ein Verweis auf den zugehörigen Speicherpool enthalten ist.

Verteilte Felder sind allerdings langsamer als lokale Felder, denn statt einem Zugriff vom Objekt direkt auf die Daten muss erst auf den Speicherpool, dann auf das entsprechende Array zugegriffen werden, bevor die Daten selbst erreicht werden. Zudem sind die Daten eines Objekts nicht mehr dicht beieinander gespeichert, was die Ausnutzung von Cache-Lokalität bei der Arbeit mit einem Objekt für verteilte Felder verhindert. Werden dagegen alle Daten eines Feldes für alle Objekte durchlaufen, wie z. B. beim Serialisieren in eine SKilL-Datei, kann von der Cache-Lokalität profitiert werden, da die Daten nah beieinander liegen.

Die Ersetzung *aller* Felder durch verteilte Felder hat aber durchaus Vorteile: Da die Objekte nun keine Daten mehr enthalten, ist es unnötig, in den Speicherpools Referenzen auf sie zu speichern. Damit entfällt bei der dauerhaften Speicherung von Objektdaten der oben berechnete Grundspeicher pro Objekt. Stattdessen wird für jedes Feld ein Array benötigt, das wiederum selbst ein Objekt ist und damit 24 Byte Extraaufwand (Grundspeicher plus Größe) erzeugt. Doch innerhalb einer SKilL-Spezifikation bzw. einer SKilL-Datei ist nur eine konstante Anzahl an Feldern definiert, daher ist dies für eine konkrete Kombination von Spezifikation und Datei ein konstanter Aufwand anstatt des vorherigen linearen Aufwands.

Werden die Objekte selbst allerdings nicht mehr gespeichert, wird es nötig, für jeden Benutzerzugriff auf die Daten ein neues, relativ kleines Zugriffsobjekt zu erzeugen. Diese Objekte arbeiten als Proxy: Sie kennen die Position der Daten, kodiert in der SKilL-ID, und bieten dem Benutzer eine übliche Feldschnittstelle mit Getter und Setter an, über die er auf die Daten zugreifen kann. Sie enthalten nur noch die SKilL-ID und Verweise auf die zugehörigen Speicherpools. Dabei gehört zu einer Klasse nicht nur ihr assoziierter Speicherpool, sondern auch die zu allen direkten und indirekten Basisklassen assoziierten Speicherpools. Da der SKilL-Zustand alle zugehörigen Speicherpools kennt, wird nur ein weiterer Konstruktorparameter benötigt, nämlich das Zustandsobjekt.

Die Spezifikation in Listing 4.3 ergibt also die folgende Definition für das zugehörige Proxyobjekt:

Listing 4.4: Proxy

```
sealed class A private[x](_skillID: Long, state : SkillState) extends KnownSkillType(_skillID) {
  ...
  protected final val APool = state.A.asInstanceOf[AStoragePool]
  ...
  final def x = APool.getX(skillID)
  final def x_=(X: Int) = APool.setX(skillID, X)
  ...
}
```

`KnownSkillType` implementiert dabei die neue Infrastruktur für bekannte Typen. Alle Zugriffe auf das Feld `x` werden an den assoziierten Speicherpool weitergeleitet. Dort wird mithilfe der SKilL-ID der korrekte Datensatz adressiert.

Da die Datensätze nur noch anhand ihrer SKilL-ID und ihrem Basistyp (siehe Abschnitt 3.2.1) unterschieden werden, sind nun zwei Objekte als identisch zu betrachten, wenn sie auf die selben Daten verweisen. Entsprechend sind Änderungen an einem Objekt in jedem anderen Objekt sichtbar, das auf den selben Datensatz verweist. Die Objekte können also auch als eine Art Referenz auf den zugehörigen, unter Umständen auf mehrere Speicherpools verteilten Datensatz betrachtet werden.

4.3. SKilL-IDs und Indizierung

Um die SKilL-ID als Index für die Daten enthaltenden Arrays zu benutzen, muss jeder Datensatz (ein ursprüngliches Objekt) eine eindeutige SKilL-ID besitzen. Für alte Objekte bietet sich an, die zur Serialisierung verwendete SKilL-ID zu benutzen. Dadurch können Daten aus einer SKilL-Datei direkt so eingelesen werden, wie sie in der Datei stehen. Für neue Objekte gibt es mehrere Möglichkeiten, eindeutige SKilL-IDs festzulegen. Eine Möglichkeit ist, einfach die nächste freie positive Zahl zu verwenden, da niemals alte Objekte erzeugt werden, sondern vielmehr neue Objekte geschrieben und dadurch zu alten Objekten werden. In diesem Fall müssten die neuen Objekte sowieso umgeordnet werden, um die Bedingungen des Dateiformats zu erfüllen, dass Objekte innerhalb eines Typblocks nach Typ sortiert sein müssen (siehe [Fel13, § 6]). Umordnen bedeutet hier, dass die neuen Objekte mit neuen SKilL-IDs versehen werden.

Aufgrund dieser Umordnung ist es aber empfehlenswert, getrennte Datenarrays für alte und neue Objekte anzulegen. Daher wurde entschieden, eine andere Möglichkeit zu nutzen, nämlich neuen Objekten negative SKilL-IDs zu verteilen, die dann als negative Indizes (1-basiert) verwendet werden können. Das ermöglicht auch eine schnellere Unterscheidung, ob ein Objekt bereits geschrieben wurde oder nicht, und eine entsprechende Reaktion.

4.4. Benutzertypen und Annotationen

Da nun keine dauerhaften Objekte mehr existieren, können auch keine Referenzen mehr darauf in den Feldern gespeichert werden. Bei Referenzen ist das Ziel ein Benutzertyp, zu dem ein eindeutiger

Speicherpool gehört, der wiederum zu einem eindeutigen Basispool gehört. Innerhalb eines Basispools sind SKilL-IDs laut SKilL-Spezifikation eindeutig [Fel13, § 6.3]. Folglich reicht es aus, anstatt einer Referenz die SKilL-ID des Ziels zu speichern. Nebenbei erlaubt dies, Felder mit Referenztyp genauso einzulesen wie Felder mit primitiven Typen, nämlich durch einfaches Kopieren der Daten aus der Datei in das Datenarray des Felds.

Auch Annotationen sind Referenzen. Jedoch wird bei diesen noch zusätzlich ein Verweis auf den passenden Speicherpool benötigt, damit das Ziel eindeutig identifiziert werden kann. Hier kann ausgenutzt werden, dass jeder Speicherpool eines SKilL-Zustands einen eindeutigen Index besitzt. Für Annotationen wird folglich der Index des korrekten Speicherpools sowie die SKilL-ID des referenzierten Objekts gespeichert. Beim Lesen eines Annotationsfelds muss nun nur noch der gespeicherte Name des Speicherpools (siehe [Fel13, § 6.4]) mithilfe des SKilL-Zustands in einen Index übersetzt werden.

Werden Referenzen und Annotationen allerdings auf diese Weise gespeichert, wird ein Mechanismus benötigt, der für eine gegebene SKilL-ID ein Proxyobjekt liefert, das dem Benutzer Zugriff auf einen Datensatz gewährt. Es wurde entschieden, diesen Mechanismus in die Implementierung des SKilL-Zustands aufzunehmen, da dieser alle betroffenen Speicherpools kennt. Das ist dadurch sichergestellt, dass Querverweise zwischen verschiedenen Zuständen nicht serialisierbar und daher nicht erlaubt sind. Damit Referenzen und Annotationen in Feldern aufgelöst werden können, erhalten die Speicherpools einen Verweis auf ihren enthaltenden Zustand.

4.5. Ausnutzung von JVM-Eigenschaften

Die JVM hat einige Einschränkungen, vor allem bezüglich Objektanzahlen und Arraygrößen. So werden JVM-Arrays mit vorzeichenbehafteten 32-Bit-Ganzzahlen (`Int` in Scala) indiziert und können daher nur bis zu $2^{31} - 1$ Objekte enthalten. Folglich reicht es aus, auch für die SKilL-ID nicht die vollen 64 Bit zu benutzen, sondern lediglich 32 Bit. Entsprechend kann auch die Anzahl der Speicherpools diese Grenze nicht übersteigen. Annotationen benötigen daher nur 64 Bit bzw. zweimal 32 Bit anstatt der theoretischen 128 Bit. Nach der SKilL-Spezifikation [Fel13, § D] ist die Ausnutzung dieser Einschränkung erlaubt, ohne die Korrektheit der Implementierung zu beeinflussen.

Es wird auf der JVM außerdem deutlich zwischen primitiven Typen (`Boolean`, `Byte`, `Short`, `Int`, `Long`, `Float`, `Double`) und Objekten unterschieden. Insbesondere enthält ein Array mit primitivem Elementtyp die Einträge selbst, während ein Array mit einem Objekttyp als Elementtyp lediglich Referenzen auf die enthaltenen Objekte enthält. Daher benötigt es wesentlich weniger Speicherplatz, eine Annotation in einem einzelnen `Long`-Wert zu kodieren (8 Byte) anstatt ein Objekt mit zwei `Int`-Einträgen zu verwenden (8 Byte für eine Referenz plus 24 Byte für das Objekt).

Werden die obigen Eigenschaften der JVM ausgenutzt, können die in Tabelle 4.1 aufgeführten Scala-Typen für die entsprechenden grundlegenden SKilL-Typen verwendet werden. Dabei ist zu beachten, dass der SKilL-Typ ***string*** auch für die Speicherung im SKilL-Zustand in den Scala-Typ `String` übersetzt wird, obwohl dieser ein Objekttyp ist. Der Grund ist, dass Strings weder Benutzertypen noch zusammengesetzte Typen sind und daher als primitive Typen betrachtet werden können.

Tabelle 4.1.: Transformation von SKilL-Typen zu Scala-Typen unter Ausnutzung von JVM-Einschränkungen

SKilL-Typ	Scala-Typ (gespeichert)	Scala-Typ (Schnittstelle)
bool	`Boolean`	`Boolean`
i8	`Byte`	`Byte`
i16	`Short`	`Short`
i32	`Int`	`Int`
i64, v64	`Long`	`Long`
f32	`Float`	`Float`
f64	`Double`	`Double`
string	`String`	`String`
annotation	`Long`	`SkillType`[a]
Benutzertypen	`Int`	Benutzertyp[ab]

[a] benötigt entsprechende Auflösungsfunktionen
[b] von `SkillType` direkt oder indirekt abgeleitete Klasse

Tabelle 4.2.: Zusammengesetzte SKilL-Typen mit ihren Scala-Entsprechungen (ursprüngliche Implementierung)

SKilL-Typ	Scala-Typ
$A[n]$	`ArrayBuffer[`a`]`[a]
$A[]$	`ArrayBuffer[`a`]`
list<A>	`ListBuffer[`a`]`
set<A>	`HashSet[`a`]`
map<A, B>	`HashMap[`a`, `b`]`
map<A, B, ...>	`HashMap[`a`, HashMap[`b`, ...]]`

[a] `scala.Array` wäre eigentlich die korrekte Wahl, doch hat diese Klasse einige Besonderheiten, die die Interaktion mit generischem Code erschweren, unter anderem, dass Arrays echt spezialisiert werden für verschiedene primitive Typen. Das kollidiert mit der Typlöschung.

4.6. Zusammengesetzte Typen

SKilL besitzt noch weitere Typen, die sich aus mehreren Elementen zusammensetzen. Das sind Arrays konstanter Länge, Arrays variabler Länge, Listen, Mengen und Maps. In Tabelle 4.2 sind diese mit ihrer Scala-Entsprechung (in der ursprünglichen Implementierung) aufgelistet. A, B und C sind dabei beliebige SKilL-Typen aus Tabelle 4.1 und a, b und c ihre Scala-Entsprechungen in der Schnittstelle, n ist eine natürliche Zahl. Nach der Definition von SKilL können zusammengesetzte Typen nicht geschachtelt werden (siehe auch die Grammatik einer SKilL-Spezifikation in [Fel13, § 2]).

Werden Referenzen auf Benutzertypen und Annotationen nun als `Int` bzw. `Long` gespeichert, aber als Objekte an den Benutzer übergeben, können diese Container nicht mehr sowohl im Speicher als auch in der Schnittstelle verwendet werden. Es werden also Adapter benötigt, die intern auf die

in einem Container gespeicherten kodierten Referenzen zugreifen. Dem Benutzer stellen sie neu erzeugte Objekte zur Verfügung, über die er wiederum auf die Daten des zugehörigen Datensatzes zugreifen kann. Die Adapter müssen folglich fähig sein, sowohl Benutzertypen in kodierte Referenzen bzw. Annotationen zu konvertieren als auch kodierte Referenzen zurück in Benutzertypen.

Der zweite Teil ist ohne weitere Informationen unmöglich wegen der auf der JVM stattfindenden Typlöschung (type erasure), d. h. zur Laufzeit stehen keine Informationen mehr über die konkret eingesetzten Typparameter zur Verfügung [Ull11]. Daher benötigen die Adapter einen Zugriff auf den SKilL-Zustand, besser gesagt auf die korrekte dort definierte Auflösungsfunktion. Entsprechend wurde die Schnittstelle des SKilL-Zustands um Fabrikmethoden erweitert. Diese erzeugen korrekt typisierte und mit den korrekten Auflösungsfunktionen versehene Containeradapter mit einem neuen,

Tabelle 4.3.: Zusammengesetzte SKilL-Typen mit ihren Scala-Entsprechungen (optimierte Implementierung)

SKilL-Typ	Elementtyp(en)	Scala-Typ (Speicher)	Scala-Typ (Schnittstelle)
A[n]	primitiv[a]	`Array[`a`]`	`Array[`a`]`
	Benutzertyp	`Array[Int]`	`RefArray[`a`]`[b]
	annotation	`Array[Long]`	`AnnotationArray`[b]
A[]	primitiv[a]	`ArrayBuffer[`a`]`	`ArrayBuffer[`a`]`
	Benutzertyp	`ArrayBuffer[Int]`	`RefArrayBuffer[`a`]`[b]
	annotation	`ArrayBuffer[Long]`	`AnnotationArrayBuffer`[b]
list<*A*>	primitiv[a]	`ListBuffer[`a`]`	`ListBuffer[`a`]`
	Benutzertyp	`ListBuffer[Int]`	`RefListBuffer[`a`]`[b]
	annotation	`ListBuffer[Long]`	`AnnotationListBuffer`[b]
set<*A*>	primitiv[a]	`HashSet[`a`]`	`HashSet[`a`]`
	Benutzertyp	`HashSet[Int]`	`RefHashSet[`a`]`[b]
	annotation	`HashSet[Long]`	`AnnotationHashSet`[b]
map<*A*, *B*>	primitiv[a]	`HashMap[`a`,`b`]`	`BasicMapView[`a`,`b`]`[b]
	primitiv[a]/Benutzertyp	`HashMap[`a`,Int]`	`BasicRefMapView[`a`,`b`]`[b]
	primitiv[a]/***annotation***	`HashMap[`a`,Long]`	`BasicAnnotationMapView[`a`]`[b]
	Benutzertyp/primitiv[a]	`HashMap[Int,`b`]`	`RefBasicMapView[`a`,`b`]`[b]
	Benutzertypen	`HashMap[Int,Int]`	`RefMapView[`a`,`b`]`[b]
	Benutzertyp/***annotation***	`HashMap[Int,Long]`	`RefAnnotationMapView[`a`]`[b]
	annotation/primitiv[a]	`HashMap[Long,`b`]`	`AnnotationBasicMapView[`b`]`[b]
	annotation/Benutzertyp	`HashMap[Long,Int]`	`AnnotationRefMapView[`b`]`[b]
	annotation	`HashMap[Long,Long]`	`AnnotationMapView`[b]
map<*A*, ...>	*A* primitiv[a]	`HashMap[`a`,_]`[c]	`BasicMapMapView[`a`,_]`[bd]
	A Benutzertyp	`HashMap[Int,_]`[c]	`AnnotationMapMapView[`a`,_]`[bd]
	A ***annotation***	`HashMap[Long,_]`[c]	`AnnotationMapMapView[_]`[bd]

[a] Einer der Typen ***bool***, ***i8***, ***i16***, ***i32***, ***i64***, ***v64***, ***f32***, ***f64***, ***string***
[b] Adaptertyp
[c] Je nach weiteren Typen ist _ eine entsprechende `HashMap`
[d] Je nach weiteren Typen ist _ der diesen Typen entsprechende `...MapView`-Typ aus dieser Tabelle

leeren Hintergrundcontainer. Tabelle 4.3 listet die Repräsentation der zusammengesetzte Typen in der optimierten Implementierung auf.

4.7. Speicherpools

Um die bisher genannten Optimierungen umsetzen zu können, mussten die Speicherpools und die Serialisierung angepasst werden. Im Gegensatz zur Serialisierung, wo nur relativ kleine Änderungen vorgenommen werden mussten, meist aufgrund von Änderungen in den Speicherpools, wurden die Speicherpools komplett überarbeitet und in weiten Teilen umgestaltet. Unter anderem wurden neue Aufgaben hinzugefügt, die nötig werden, wenn die Objekte nur noch als Datensätze in den Speicherpools existieren. Tabelle 4.4 fasst die Verwaltungsbereiche, Tabelle 4.5 die Aufgaben der Speicherpools jeweils in beiden Implementierungen zusammen. Die Klasse `StoragePool` enthält dabei die Gemeinsamkeiten der Klassen `BasePool` und `SubPool`, generierte Pools sind die generierten Speicherpools für bekannte Typen (siehe auch Abschnitt 4.1.2).

Tabelle 4.4.: Verwaltungsbereiche der Speicherpools in den verschiedenen Implementierungen

	Ursprüngliche Implementierung	**Optimierte Implementierung**
`StoragePool`	• Typhierarchie (Unterpools, Oberpool) • Felddeklarationen, unbekannte Felder • Statische Instanzen des Pools • Dynamische Blockinformationen	• Typhierarchie (Unterpools, Oberpool) • Felddeklarationen, unbekannte Felder • Anzahl statischer neuer Instanzen • Statische Blockinformationen
`BasePool`	• Dynamische Instanzen des Pools	• Enthaltender SKilL-Zustand • Gültigkeit von Instanzen bzw. SKilL-IDs • Dynamische neue Instanzen
`SubPool`		• Dynamische Blockinformationen • Lokaler Poolindex (innerhalb eines Basispools)
Generierte Pools	• Singleton-Instanz (falls anwendbar)	• Singleton-Instanz (falls anwendbar) • Daten für alte dynamische Instanzen[a] • Daten für neue statische Instanzen

[a] Nur Felder, die im assoziierten Typ des Pools deklariert sind, d. h. weder Felder der Basisklassen noch Felder von abgeleiteten Klassen

Tabelle 4.5.: Aufgaben der Speicherpools in den verschiedenen Implementierungen

	Ursprüngliche Implementierung	**Optimierte Implementierung**
`StoragePool`	• Erzeugen neuer Unterpools • Hinzufügen neuer Felder • Auflösen von SKilL-IDs zu Instanzen[a] • Iteration über Instanzen	• Erzeugen neuer Unterpools, Iterieren über Unterpools • Hinzufügen neuer Felder • Auflösen von SKilL-IDs zu Instanzen • Iteration über Instanzen • Erzeugen von Proxyobjekten[a] • Infrastruktur zur Verwaltung von Felddaten (einschließlich Aktualisierung von Referenzen)
`BasePool`	• Aktualisierung von SKilL-IDs vor Serialisierung	• Aktualisierung von SKilL-IDs vor Serialisierung • Löschen und Hinzufügen von Instanzen • Aufbau der Umordnungstabellen zur Aktualisierung von Referenzen und Umordnung von Instanzen
`SubPool`		• Zuordnung von SKilL-IDs zu lokalen Datenindizes
Generierte Pools	• Erzeugen neuer Instanzen (falls anwendbar)	• Erzeugen neuer Instanzen (falls anwendbar) • Erzeugen von korrekt typisierten Proxyobjekten • Zugriff auf Felddaten für dynamische Instanzen[b]

[a] Implementiert in den abgeleiteten Klassen

[b] Nur Felder, die im assoziierten Typ des Pools deklariert sind, d. h. weder Felder der Basisklassen noch Felder von abgeleiteten Klassen

In diesem Abschnitt bezeichnen statische Instanzen eines Speicherpools alle Objekte bzw. Datensätze, die vom dazu assoziierten Benutzertyp sind. Dynamische Instanzen eines Speicherpools enthalten zusätzlich alle Objekte bzw. Datensätze der vom assoziierten Benutzertyp abgeleiteten Typen. Alte Instanzen sind gelesene oder bereits geschriebene Objekte bzw. Datensätze, alle anderen Instanzen sind neue Instanzen.

Die Speicherpools haben in der optimierten Implementierung deutlich mehr Aufgaben zu erfüllen, da die Daten aller Felder nun dort gespeichert sind. Aus diesem Grund werden in den folgenden Unterabschnitten einige Bereiche der Speicherpools detaillierter beschrieben. Das beinhaltet die Speicherung und Verwaltung der Datensätze sowie die Aufgaben der Basispools.

4.7.1. Alte Instanzen

Alte Instanzen wurden aus SKilL-Dateien gelesen bzw. in SKilL-Dateien geschrieben. Daher orientiert sich ihre Speicherung an der Spezifikation des binären SKilL-Formats [Fel13, § 6.2]. Das bedeutet, dass alle Daten eines Feldes f in Blöcken gespeichert werden, wobei jeder Block bezüglich der Typhierarchie sortiert ist, d. h. alle Daten eines Typs kommen an einem Stück direkt vor den Daten seiner abgeleiteten Typen. Da sich an diesen Blöcken nachträglich nichts mehr ändert, können alle Blöcke hintereinander in ein einziges Array geschrieben werden. Da gelöschte Instanzen nur als gelöscht markiert werden, müssen nur beim Schreiben in eine Datei die Blöcke aktualisiert werden. Der Elementtyp des Arrays ist der in den Tabellen 4.1 und 4.2 aufgeführte zum SKilL-Typ von f gehörige Scala-Typ für die Darstellung im Speicher.

Für das laufende Beispiel (Listing 4.1) ergibt sich für die angegebenen Felder der folgende Aufbau (3 Blöcke mit den Inhalten 3 `A`, 2 `B`; 0 `A`, 2 `B`; 2 `A`, 3 `B`), beschriftet mit dem jeweiligen Typ und der SKilL-ID des Objekts:

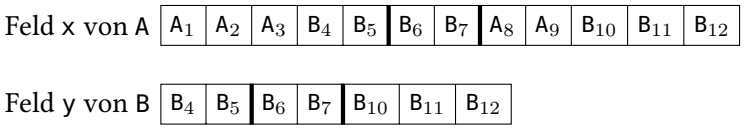

Abbildung 4.1.: Speicherung von alten Daten

Wie am Beispiel gut zu sehen ist, eignet sich die SKilL-ID als Index für Felder eines Basistyps, da alle SKilL-IDs vom Basispool vergeben werden. Für Unterpools dagegen ist die SKilL-ID ungeeignet. Die SKilL-IDs der Instanzen in einem Unterpool sind aber offensichtlich sortiert und innerhalb eines Blocks lückenlos (nach Definition des SKilL-Dateiformats, siehe auch [Fel13, § 6]). Eine Funktion zur Berechnung eines Index in ein Feld eines Unterpools aus einer SKilL-ID benötigt also zusätzlich nur Informationen über die dynamischen Blöcke des Unterpools und kann in Scala folgendermaßen implementiert werden[4]:

Listing 4.5: Index für Felder in Unterpools

```
case class BlockInfo(val bpsi : Int, val count : Int)

def indexOfOldID(skillID : Int) : Int = {
  var start = 0
  for (BlockInfo(pos, count) <- dynamicBlockInfos;
       end = pos + count)
    if (pos < skillID && skillID <= end)
      return skillID - start - 1
    else
      start += count
  -1 // wird nie erreicht (falls Eingabe gültig)
}
```

[4]In der tatsächlichen Implementierung ist eine äquivalente Variante enthalten, die statt der Methode `dynamicBlockInfos` die zugrundeliegenden Daten nutzt.

Dabei enthält `dynamicBlockInfos` vom Typ `Iterator[BlockInfo]` Informationen über alle Blöcke dynamischer Instanzen des Unterpools. `pos` bzw. `bpsi` ist dabei der Anfang des Blocks und `count` dessen Länge. Da Blockinformationen 0-basiert gespeichert werden, da in Scala Arrays 0-basiert sind, SKilL-IDs dagegen 1-basiert, wird statt dem (eigentlich eingeschlossenen) Anfang des Blocks das (eigentlich ausgeschlossene) Ende des Blocks beim Vergleich mit einbezogen. Da diese Funktion nur intern aufgerufen wird, ist die übergebene `skillID` immer in einem Block enthalten, daher wird grundsätzlich ein gültiger Index ausgegeben. Die Komplexität dieser Funktion ist im schlimmsten Fall $O(\#Blöcke)$ (Anzahl Schleifendurchläufe). Da im Normalfall nur wenige Blöcke im Verhältnis zur Anzahl der Objekte vorhanden sind – jeder Block zusätzlich zum ersten entspricht einem Anhängen an eine existierende Datei –, ist dies ein relativ kleiner Aufwand.

4.7.2. Neue Instanzen

Neue Instanzen werden während der Laufzeit neu erzeugt. Da sie in beliebiger Reihenfolge erzeugt werden können, können anders als bei alten Instanzen keine Annahmen bezüglich ihrer Ordnung getroffen werden. Würden die SKilL-IDs wie für alte Daten in Typreihenfolge vergeben werden, wären unzählige Umordnungen nötig, z. B. falls von zwei Typen, die einen gemeinsamen Basistyp haben, abwechselnd Instanzen erzeugt werden. Daher werden die SKilL-IDs in Erzeugungsreihenfolge vergeben.

Da SKilL-IDs innerhalb von Basispools eindeutig sind, sind dort zwei parallele Arrays `newPoolInfo` und `newIndexInfo` definiert, die die Zuordnung einer SKilL-ID für eine neue Instanz zu einem Paar definieren, welches aus einem Speicherpool und einem lokalen Index besteht. Dieses Paar ist die genaue Adresse eines Datensatzes. Damit das möglich ist, können die Daten einzelner Instanzen nicht wie bei alten Daten auf mehrere Speicherpools aufgeteilt sein, sondern jeder Speicherpool muss alle Daten seiner statischen Instanzen enthalten. Die Getter/Setter für Felder und die Aktualisierungsmethoden für Verweise brauchen allerdings Zugriff auf die Daten ihrer dynamischen Instanzen. Um dieses Problem zu lösen, wurden Traits eingeführt, die alle Felder eines Typs beschreiben. Diese werden von allen Pools implementiert, die zu diesem Typ oder einem davon abgeleiteten Typ assoziiert sind. Für das laufende Beispiel mit einigen neu erzeugten Objekten in der Reihenfolge `B`, `A`, `B`, `B`, `A`, `A` ergibt sich der folgende Zustand (Felder beschriftet wie im vorigen Abschnitt, Pfeile ordnen den „Adressen“ links den entsprechenden Datensatz rechts zu):

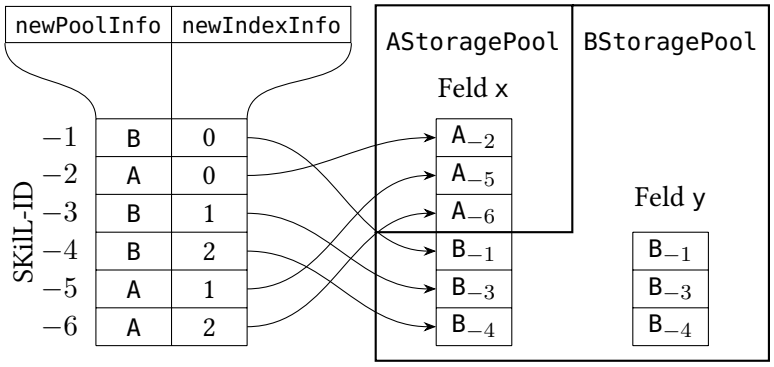

Abbildung 4.2.: Neue Daten in Speicherpools: Speicherung und Zugriff

Dabei ist zu bemerken, dass beide Speicherpools (`AStoragePool` und `BStoragePool`) ein Array für das Feld `x` besitzen, welches im `BStoragePool` parallel zum Array für das Feld `y` ist.

Die Abbildung zeigt auch, wie die Auflösung einer SKilL-ID zu Feldindizes für neue Instanzen funktioniert. Der zu erkennende indirekte Feldzugriff bedeutet allerdings, dass für neue Instanzen die Zugriffszeit auf die Daten höher ist als für alte Instanzen, da zwei Arrayzugriffe sowie zwei Feldzugriffe mehr benötigt werden (jeweils auf `newPoolInfo` und `newIndexInfo`). Bei vielen Feldzugriffen auf neue Instanzen kann daher im Vergleich zur ursprünglichen Implementierung eine längere Laufzeit erwartet werden. Außerdem brauchen neue Instanzen durch die zusätzlichen Verwaltungsstrukturen mehr Speicherplatz als alte Instanzen. Es kann daher in der optimierten Implementierung ratsam sein, in regelmäßigen Abständen den SKilL-Zustand zu schreiben und dadurch neue Instanzen zu alten Instanzen zu machen.[5]

Neue Instanzen können nur für spezifizierte, d. h. bekannte Typen erzeugt werden. Die entsprechenden Speicherpools besitzen eine Fabrikfunktion für den assoziierten Typ, falls dieser kein Singleton-Typ ist. Andernfalls wird automatisch ein neuer Datensatz sowie ein passendes Proxyobjekt erzeugt, falls noch keine Instanz des Singleton-Typs existiert. Da die `skillID` für den neuen Datensatz vom zugehörigen Basispool vergeben wird, müssen neue Instanzen dort registriert werden. Entsprechend existiert im Basispool eine Registrierungsmethode `addPoolInstance`, die von allen Fabrikfunktionen mit dem aktuellen Pool und dem nächsten lokalen Index aufgerufen wird. Sie erzeugt daraus neue Einträge in `newPoolInfo` und `newIndexInfo` und gibt die nächste freie SKilL-ID zurück.

4.7.3. Basispools

Basispools sind für alle Instanzen zuständig, die vom assoziierten Basistyp oder einem davon abgeleiteten Typ sind. Insbesondere enthalten sie die Verwaltungsstrukturen und -methoden für das Löschen und Hinzufügen von Instanzen sowie für die Zuordnung negativer SKilL-IDs zu neuen Instanzen und Hilfsmethoden für die Serialisierung von Speicherpools. Dieser Abschnitt beschäftigt sich mit den Verwaltungsaufgaben eines Basispools.

Instanzen werden innerhalb von Speicherpools nicht gelöscht, sondern lediglich als gelöscht markiert. Erst beim Serialisieren werden alle als gelöscht markierten Datensätze entfernt. Wurden außerdem alte Instanzen bearbeitet bzw. gelöscht, darf beim Serialisieren nicht an eine Datei angehängt werden. Es müssen stattdessen alle Daten neu geschrieben werden. Wie auch in der ursprünglichen Implementierung ist dieses Verhalten nicht vollständig implementiert. Lediglich gelöschte Instanzen werden erkannt. Dazu existieren in einem Basispool zwei Arrays, eines für alte (`deleted`) und eines für neue Instanzen (`newDeleted`), die für jede Instanz einen booleschen Wert enthalten, ob diese gelöscht wurde. Zur schnelleren Überprüfung, ob eine alte Instanz gelöscht wurde, hat jeder Basispool eine boolesche Variable `dirty`. Proxyobjekte (und damit der Benutzer) haben durch die Methoden `removeByID` (Löschen von Instanzen) und `isIDRemoved` (Prüfen, ob eine Instanz gelöscht ist) Zugriff auf diese Markierungen.

[5] In der ursprünglichen Implementierung ist das Speicherverhalten umgekehrt, da auf alte Instanzen zwei Referenzen, auf neue jedoch nur eine Referenz im Speicherpool gespeichert wird.

Die Verwaltungsstrukturen `newPoolInfo` und `newIndexInfo` zur Zuordnung von SKilL-IDs zu Datensätzen wurden bereits im Abschnitt 4.7.2 beschrieben und werden daher hier nicht weiter ausgeführt.

4.7.4. Serialisierung

Im Bereich der Serialisierung wurde im Vergleich zur ursprünglichen Implementierung einiges geändert, vor allem dadurch, dass Referenzen und Annotationen nun vor der Serialisierung auch in Feldern korrigiert werden müssen. Dazu wurde eine vollständig neue Infrastruktur eingeführt, so dass während der Serialisierung eines Speicherpools nun folgende Schritte durchgeführt werden:

- Schreiben einer Datei:
    1. Basispools bauen vollständige Umordnungstabellen für ihre dynamischen Instanzen auf; Instanzen werden in Typordnung sortiert.
    2. Alle Instanzen werden in einen Block komprimiert, dabei gelöschte Instanzen entfernt.
    3. Referenzen und Annotationen in bekannten und unbekannten Feldern werden mithilfe der Umordnungstabellen korrigiert, Verweise auf gelöschte Instanzen werden zu Nullverweisen korrigiert.
    4. Die Datei wird mithilfe der Klasse `StateWriter`[6] geschrieben.
- Anhängen an eine Datei:
    1. Basispools bauen Umordnungstabellen für neue dynamische Instanzen auf, in Typordnung sortiert.
    2. Die neuen Instanzen werden in einen Block komprimiert, der anschließend an die bereits vorhandenen Blöcke angehängt wird. Dabei werden gelöschte Instanzen entfernt.
    3. Referenzen und Annotationen auf neue Objekte in bekannten und unbekannten Feldern werden mithilfe der Umordnungstabelle korrigiert, Verweise auf gelöschte Instanzen werden zu Nullverweisen korrigiert.
    4. Die Datei wird mithilfe der Klasse `StateAppender`[6] geschrieben.

Es existieren zwei Formate für Umordnungstabellen, eines für alte Instanzen und eines für neue Instanzen. Die Schreiboperation verwendet beide Formate und ein Array von Paaren von Umordnungstabellen, je eine für alte und eine für neue Instanzen. Die Anhängoperation benutzt ein Array von Tabellen, aber nur das Format für neue Instanzen. Es existiert höchstens eine Umordnungstabelle pro Format und Basispool in jedem Schreib- bzw. Anhängvorgang. Für Unterpools werden die entsprechenden Tabellen ihres jeweiligen Basispools verwendet.

Eine Umordnungstabelle für alte Objekte besteht aus einer sortierten Liste von Intervallen mit zugehöriger Verschiebung bzw. einer Markierung, ob das Intervall aus gelöschten Instanzen besteht. Jedes Intervall endet mit dem Beginn des nächsten Intervalls, daher muss nur der Beginn jedes

[6] Die Klassen `StateWriter` und `StateAppender` wurden weitgehend übernommen aus der ursprünglichen Implementierung, aber angepasst an den neuen Ablauf und die neue Struktur der Speicherpools.

Intervalls gespeichert werden. Das letzte Intervall endet am Ende der Daten. Innerhalb eines Intervalls sind alle Instanzen vom selben statischen Typ. Um diese Tabellen zu verwenden, muss also für eine gegebene (positive) SKilL-ID lediglich das passende Intervall gefunden werden, das aufgrund der Konstruktion auf jeden Fall existiert. Anschließend muss, falls es sich um ein gelöschtes Intervall handelt, die Instanz gelöscht bzw. die Referenz auf Null gesetzt werden, andernfalls die Instanz bzw. die Referenz um den zum Intervall gehörenden Verschiebungswert verschoben werden. Instanzen werden gelöscht, indem ihre Daten nicht in die neu erzeugten Datenarrays kopiert werden. Da jedes Intervall 8 Byte belegt, ist die Umordnungstabelle im schlimmsten Fall, d. h. jede SKilL-ID ist ein eigenes Intervall, $8n$ Byte groß, wobei n die Anzahl der alten Instanzen ist. Dieser Fall tritt nur ein, wenn vor der Serialisierung in allen Blöcken von jedem Typ entweder nur eine Instanz existiert oder jede zweite Instanz als gelöscht markiert wurde. In üblichen Anwendungsfällen wird daher wesentlich weniger Speicher benötigt.

Da neue Instanzen in den meisten Fällen keine wohldefinierte Ordnung haben, besteht eine Umordnungstabelle für neue Instanzen lediglich aus einem Array. In diesem ist zu jeder (negativen) SKilL-ID, mittels $-\text{SKilL-ID} - 1$ in einen Index transformiert, die neue (positive) SKilL-ID oder 0 gespeichert. Offensichtlich braucht diese Umordnungstabelle immer $4n$ Byte (Referenzen werden als `Int` gespeichert), wobei n die Anzahl der neuen Instanzen ist.

Insgesamt ist die Summe des Speicherverbrauchs für die Umordnungstabellen und des Speicherverbrauchs für die Verwaltungsstrukturen pro Objekt für ausreichend viele Objekte immer noch geringer als die ursprünglichen 24 Byte Grundspeicherverbrauch für die ursprünglichen Objekte. Das gilt nicht für zusammengesetzte Typen mit Referenzen oder Annotationen als Inhalt. Aufgrund des Boxings von primitiven Scala-Typen wird hier in der optimierten Implementierung während des Schreibvorgangs wegen der Umordnungstabellen sogar mehr Speicherplatz verbraucht (siehe auch Abschnitt 7.6). Eine mögliche Lösung des Problems ist im Abschnitt 4.8.1 beschrieben.

4.8. Weitere Optimierungen

Die ursprüngliche Implementierung benutzt `ArrayBuffer`, um statische Instanzen, sowohl alte als auch neue, zu speichern. Es wurde festgestellt, dass die Klasse `ArrayBuffer` beim Vergrößern zwar automatisch das enthaltene Array vergrößert, beim Verkleinern aber keinen Speicherplatz mehr freigibt. Das kann dazu führen, dass Speicherplatz verschwendet wird. Daher verwendet die optimierte Implementierung Arrays für die Speicherung von Felddaten, die von den Speicherpools selbst verwaltet werden. Diese vergrößern die Arrays, falls nötig, aber verkleinern sie auch wieder, wenn sie nicht mehr benötigt werden. Das betrifft vor allem neue Instanzen. Bei einer Schreib- oder Anhängoperation werden alle Arrays für alte Instanzen auf die exakt benötigte Größe vergrößert, alle Arrays für neue Instanzen durch kleine, leere Arrays ersetzt.

Eine weitere Optimierung des Speicherverbrauchs ist die Verwendung spezieller Arrays für boolesche Werte, so dass die Elemente nur ein Bit benötigen. Der Zugriff auf diese Werte wird dadurch allerdings langsamer. Im Fall der internen Markierungsarrays für gelöschte Datensätze ist das üblicherweise kein großes Problem, da auf diese nur zum Löschen von Instanzen bzw. zum Prüfen, ob eine Instanz gelöscht ist, sowie beim Iterieren zugegriffen wird. Beim Iterieren wird auf das Markierungsarray für alte Daten außerdem nur zugegriffen, wenn `dirty` im entsprechenden Basispool gesetzt ist, d. h. nur

falls mindestens eine alte Instanz gelöscht wurde. Der mögliche Verwendungsbereich erstreckt sich zusätzlich über alle Felder des SKilL-Typs ***bool***. Hier wird diese Optimierung allerdings noch nicht eingesetzt.

Die Implementierung dieser spezialisierten Arrays für boolesche Werte (Klasse `BooleanArray`) verwendet intern ein `Array[Int]` zum Speichern der Einträge und belegt leer 56 Byte; ein `Array[Boolean]` benötigt leer nur 24 Byte. Der höhere Grundbedarf zahlt sich jedoch aus: Bei 40 Einträgen sind beide gleich groß (64 Byte), bei mehr Einträgen ist `BooleanArray` bereits kleiner als ein `Array[Boolean]` mit gleich vielen Einträgen.

Bei der Ausgabe in eine SKilL-Datei wird in der ursprünglichen Implementierung ein Puffer verwendet, um die Positionen zu finden, an denen die Daten eines Feldes enden. Diesen Puffer einzusparen und die Größe vorzuberechnen, wie es in einer bisher unveröffentlichten gemeinsamen Implementierung für alle JVM-Sprachen (javaCommon) gemacht wird, wäre auch vom Speicher her die beste Lösung [Fel15]. Da die optimierte Implementierung jedoch auf der ursprünglichen Implementierung basiert, wurde lediglich die Pufferklasse `OutBuffer` verbessert.

Diese in Java geschriebene Klasse speichert die geschriebenen Daten intern in Byte-Arrays. Daten können mithilfe zweier `put`-Funktionen zum Puffer hinzugefügt werden. Eine akzeptiert ein `byte` als Argument und schreibt dieses in das aktuelle Byte-Array, falls noch Platz ist, und erzeugt andernfalls ein neues Byte-Array der Größe 8 kB. Die andere `put`-Funktion erwartet ein `byte[]` als Argument und fügt dieses am Ende der Array-Liste des Puffers hinzu. Da die übergebenen Arrays aber serialisierte Formen von primitiven Datentypen sind, d. h. nur 2 bis 8 Einträge enthalten, überwiegen bei diesen Arrays die 24 Byte von `java.lang.Object` und der Arraygröße. Es ist also effizienter, diese Arrays zu verwerfen und ihren Inhalt in die größeren, bereits vorhandenen Arrays zu kopieren.

4.8.1. Nicht durchgeführte Optimierungen

Außer der Klasse `Array` existieren für keine der Klassen, die zur Repräsentation zusammengesetzter Typen verwendet werden, spezialisierte Versionen für primitive Typen. Daher ergibt sich hier ein höherer Speicherverbrauch, als nötig ist. Das liegt daran, dass alle Einträge in `ArrayBuffer`, `ListBuffer`, `HashSet` und `HashMap` wegen der Typlöschung als Referenzen auf Objekte gespeichert werden. Daher werden primitive Typen, da sie auf der JVM keine Objekte sind, in Objekte eingebettet (Boxing). Statt z. B. 8 Byte für ein `Long` werden also 8 Byte für eine Referenz auf ein `java.lang.Long`-Objekt benötigt und zusätzlich 24 Byte für dieses Objekt. Da durch die in den vorigen Abschnitten durchgeführten Optimierungen dazu führen, dass *nur* primitive Elementtypen verwendet werden (mit Ausnahme der `String`-Klasse), betrifft das fast alle Felder mit zusammengesetztem Typ. Die Lösung dieses Problems ist die Implementierung spezialisierter Varianten von allen diesen Klassen für jeden primitiven Typ und entsprechende Anpassung der Speicherpools und Containeradapter.

5. Untersuchung des Speicherverbrauchs

Zur Untersuchung des benötigten Speichers einer generierten Scala-Anbindung wurden im Rahmen dieser Arbeit Tests entwickelt. In diesen Tests werden alle für den Speicherverbrauch relevanten Teile des generierten Codes getestet. Dabei findet der Zugriff grundsätzlich nur über die generierte öffentliche Schnittstelle statt.

5.1. Speichermessung auf der JVM

Da die JVM die vollständige Verwaltung des Speichers übernimmt, ist es nicht möglich, durch modifizierte Allokations- und Deallokationsfunktionen den aktuellen Speicherverbrauch zu messen. Auch die durch Java angebotenen Funktionen `java.lang.Runtime.freeMemory` (aktuell freier Heapspeicher) und `java.lang.Runtime.totalMemory` (aktuelle Heapgröße) haben sich als unzuverlässig erwiesen, da sie ungenaue Ergebnisse liefern. Daher wurde entschieden, eine nicht-portable Schnittstelle der HotSpot-JVM [Ora14], Jvmstat [Jvm], zur Speichermessung zu benutzen. Diese ist in der Bibliothek `tools.jar` als Bestandteil des Java Delevopment Kit (JDK) 7 von Oracle enthalten. Die zur Verwendung der Schnittstelle benötigte Klasse wurde nur für die HotSpot-JVM aus der Testumgebung implementiert; eine vollständigere Behandlung der verschiedenen JVM-Versionen ist im Quellcode des Programms VisualVM [Vis14] zu finden. Der Hauptunterschied besteht in der Benennung der einzelnen „Counter“. Das sind von der JVM verwendete Instrumentationsobjekte, die verschiedene interne Daten der JVM enthalten, wie z. B. den verwendeten Speicher.

Damit der Speicherverbrauch gemessen werden kann, musste eine Klasse geschrieben werden, die die `sun.jvmstat.monitor.event.VmListener`-Schnittstelle implementiert und sich bei der Ziel-JVM, auf der der Speicherverbrauch gemessen werden soll, anmeldet. In regelmäßigen Abständen wird dann die Klasse informiert, dass sich im überwachten Prozess der Speicherverbrauch verändert hat. Diese Werte werden dann vom im nächsten Kapitel beschriebenen Testframework weiterverarbeitet.

Um den gemessenen Speicherverbrauch nicht zusätzlich durch Messdaten zu erhöhen, verwenden die in dieser Arbeit benutzten Tests mehrere Prozesse mit verschiedenen Aufgaben. Ein Prozess führt Verwaltungs- und Messaufgaben durch, d. h. er startet andere Prozesse, die den eigentlichen Test durchführen und misst deren Speicherverbrauch über die Jvmstat-Schnittstelle.

5.2. Übersicht über Jvmstat

Dieser Abschnitt gibt eine kurze, oberflächliche Übersicht über den verwendeten Teil der Jvmstat-Schnittstelle [Jvm]. Alle in diesem Abschnitt erwähnten Klassen sind im Paket `sun.jvmstat.monitor`

enthalten. Die öffentliche Schnittstelle besteht aus abstrakten Klassen, die statische Fabrikmethoden besitzen und einigen konkreten Klassen, meistens ohne zugreifbarem Konstruktor.

Die Klasse `MonitoredHost` stellt eine abstrakte Maschine dar, wie zum Beispiel den lokalen Computer, auf der virtuelle Maschinen (JVMs) ausgeführt werden können. Die statische `getMonitoredHost`-Methode liefert eine Maschine, für die man eine Adresse hat oder den eindeutigen Bezeichner einer darauf laufenden virtuellen Maschine, dargestellt durch die Klasse `VmIdentifier`. Das Testframework startet nur Prozesse auf dem lokalen Computer. Es wird daher der Aufruf `MonitoredHost.getMonitoredHost("localhost")` verwendet, um eine entsprechende Instanz der `MonitoredHost`-Klasse zu erhalten.

Eine überwachte JVM wird durch die Klasse `MonitoredVm` repräsentiert. Erzeugt werden kann eine Instanz dieser Klasse, falls man die Prozess-ID der gewünschten JVM auf einer bekannten Maschine kennt und eine `MonitoredHost`-Instanz für diese Maschine besitzt. Eine Klasse, die die `event.VmListener`-Schnittstelle implementiert, kann mittels der `MonitoredVm.addVmListener`-Methode als Beobachter einer `MonitoredVm` registriert werden.

Schließlich existieren noch Monitore (Schnittstelle `Monitor`), die einen einzelnen Wert der JVM überwachen. Sie werden in einem regelmäßigen Zeitabstand aktualisiert, welcher durch die `MonitoredVm`-Klasse festgelegt wird. Monitore können über ihren Namen von einer `MonitoredVm`-Instanz erhalten werden. Die Namen der Monitore variieren von JVM zu JVM je nach Version und Hersteller. In der JVM aus der Testumgebung entsprechen die Namen der interessanten Monitore dem regulären Ausdruck „`sun.gc.generation.[0-9]+.space.[0-9]+.used`“. Diese Monitore geben die Menge des aktuell verwendeten Speichers in jeder JVM-Generation [GCT] an. Die Summe der Werte von allen diesen Monitoren ist die gesamte Speichermenge, die aktuell von der überwachten JVM benutzt wird.

6. Testframework

In diesem Kapitel wird das Framework beschrieben, das erlaubt, mit wenig Aufwand für neue Spezifikationen angepasste Tests zu erzeugen. Zur Zeit verwendet dieses Framework nur einen Parameter, der entweder aus einer Liste stammt oder mit einer gegebenen Verteilung zufällig generiert wird. Eine Erweiterung auf mehrere Parameter ist denkbar, führt aber zu schwer auswertbaren, mehrdimensionalen Ergebnissen.

Das Testframework besteht aus mehreren Bestandteilen. Aus den folgenden Grundbausteinen kann ein Test erzeugt werden:

- Ergebnisse (Trait `Result`) speichern Messdaten in verschiedenen Formen.
- Drucker (Trait `Printer`) steuern die Ausgabe eines Prozesses.
- Aktionen (Trait `Action`) legen den Testablauf fest.
- Ein Testablauf (Klasse `Task`) kapselt die Aufgaben und Daten eines einzelnen Prozesses oder Threads.
- Die Klasse `StorageTestBase` definiert die Funktionalität, um die anderen Grundbausteine zu benutzen.

Zusätzlich existieren noch Helferklassen: Die in Kapitel 5 erwähnte Klasse zur Speichermessung mit dem Namen `ValueReporter`, sowie Verteilungen (Klasse `Distribution`) für die zufälligen Tests. Das Framework ist auf Erweiterbarkeit ausgelegt. Es können zusätzlich zu den vordefinierten Klassen eigene Ergebnisse, Aktionen und Verteilungen definiert und verwendet werden.

Dieses Kapitel beschreibt lediglich die Teile des Testframeworks, die für die Verwendung benötigt werden. Weitergehende Informationen zur Implementierung eigener Ergebnisse und Aktionen sowie genauere Informationen über Drucker und die interne Speichermessung befinden sich im Anhang.

6.1. Ergebnisse

Ergebnisse erben von einem der Traits `SingleValueResult` (für einen Messwert pro Parameter) oder `MultiValueResult` (für mehrere Messwerte pro Parameter). Die Companion-Objekte beider Traits besitzen jeweils eine Methode `saveGraph`, die eine Liste der jeweiligen Ergebnisse als Datenreihen in ein LaTeX-Diagramm speichert (benötigt das LaTeX-Paket pgfplots[1] [Feu14]).

[1] Die verwendete Version ist 1.10. Für die Verwendung der ausgegebenen Diagramme \pgfplotsset{compat=1.10} in der Präambel des LaTeX-Dokuments angeben.

Beide `saveGraph`-Methoden haben folgende Parameter (in der angegebenen Reihenfolge):

`file` (`String`) Pfad zur Ausgabedatei für das Diagramm

`caption` (`String`) Überschrift des Diagramms

`axisStyle` (`String`) Art der Achsen (normal oder logarithmisch). Akzeptiert die Werte „`axis`“, „`loglogaxis`“, „`semilogxaxis`“ oder „`semilogyaxis`“

`coordinateStyle` (`String`) Art der Datenlinien (Linien und/oder Punkte, …). Akzeptiert die Werte „`smooth`“, „`only marks`“, „`sharp plot`“, „“, … Für Beschreibungen und weitere Werte siehe Dokumentation von pgfplots [Feu14].

`data` (`Seq[...Result]`) Liste der Ergebnisse für die Datenlinien.

Jedes Ergebnis besitzt einen Namen, der im Diagramm als Beschriftung der entsprechenden Datenlinie verwendet wird.

6.1.1. Vordefinierte Ergebnisse

Die folgenden beiden kanonischen Implementierungen der beiden Traits für Ergebnisse sind bereits im Testframework vordefiniert:

`CollapsedResult` Speichert einen Messwert pro Parameter. Mehrere Messwerte für einen Parameter werden mit einer Funktion zu einem Wert verarbeitet. Der Konstruktor erwartet einen Namen für das Ergebnis und die zu verwendende Funktion als Parameter.

`MultiResult` Speichert mehrere Messwerte pro Parameter. Der Konstruktor erwartet lediglich den Namen für das Ergebnis als Parameter.

6.2. Drucker

Drucker sind nur relevant, falls nur ein Prozess verwendet werden soll, in dem sowohl die Tests ausgeführt werden als auch gemessen wird. In diesem Fall können die folgenden zwei Drucker als Parameter für die `createTask`-Methode der `StorageTestBase`-Klasse angegeben werden:

`ConsolePrint` gibt die maximale Heapkapazität und den maximalen Speicherverbrauch innerhalb des Messzeitraums auf der Konsole aus. Die Ausgabe erfolgt in tabellarischer Form, einzelne Spalten sind durch Tab-Zeichen getrennt. In der ersten Spalte steht der Parameter der aktuellen Ausführung, in der zweiten Spalte die maximale Heapkapazität in Byte, in der dritten der maximale Speicherverbrauch, ebenfalls in Byte.

`ResultPrint` gibt die maximale Heapkapazität und den maximalen Speicherverbrauch innerhalb des Messzeitraums in Ergebnisse aus, die zu Beginn der Messung gesetzt werden. Die Ergebnisse sind optional. Wird eines der Ergebnisse oder sogar beide nicht gesetzt, werden die entsprechenden Messwerte ignoriert.

6.3. Aktionen und Testabläufe

Aktionen sind das Kernstück des Testframeworks. Eine Aktion beschreibt einen Schritt bzw. eine Schrittfolge eines Testablaufs. Eine `Task`-Instanz verwaltet dabei die Daten, die von mehreren Aktionen übergreifend verwendet werden sollen.

Sowohl Aktionen als auch Testabläufe hängen vom Typ des verwendeten SKilL-Zustands ab und haben daher einen Typparameter. Da manche Aktionen keinen Zugriff auf den Zustand brauchen, existiert auch eine typlose Variante der Aktion. Alle Typparameter, die in der Definition des Traits `Action`, den abgeleiteten Traits und der Klasse `Task` vorkommen, müssen folglich den Typ eines generierten SKilL-Zustands (Trait `SkillState` im zu testenden generierten Code) oder `Nothing` annehmen. Dabei wird `Nothing` nur für typlose Aktionen verwendet.

Diese Typparameter und alle Typparameter in den folgenden Abschnitten könnten eingespart oder wenigstens mit einer oberen Schranke versehen werden, wenn alle generierten `SkillState`-Traits von einem gemeinsamen Basis-Trait (z. B. `common.SkillStateBase`) erben würden, das den gemeinsamen Teil dieser Traits enthält. Dieser gemeinsame Teil ist groß, denn die einzigen Unterschiede zwischen allen diesen Traits bestehen in den direkten Zugriffen auf die Speicherpools, die zu bekannten Typen assoziiert sind. Dadurch könnte auch der interne Teil des Testframeworks deutlich vereinfacht werden.

Aktionen können mit dem Operator `+>` verkettet werden. Das Ergebnis führt zuerst die Aktion aus, die als linker Operand gegeben ist, anschließend den rechten Operanden. Beide Aktionen müssen den selben Zustandstyp verwenden, es sei denn, mindestens eine davon ist typlos. Um eine Folge von Aktionen in eine einzige Aktion umzuwandeln, definiert das Companion-Objekt des Traits `Action` eine `fold`-Methode, die eine Liste von Aktionen erwartet. Das Ergebnis ist eine Aktion, die alle Aktionen aus der Liste in der gegebenen Reihenfolge ausführt.

Testabläufe werden durch die `Task`-Klasse dargestellt. Ein Testablauf enthält aktionsübergreifende Daten wie den verwendeten SKilL-Zustand. Objekte der `Task`-Klasse werden nur dann direkt verwendet, wenn nur ein Prozess verwendet wird. In diesem Fall werden sie mithilfe der `createTask`-Methode der `StorageTestBase`-Klasse aus einem für die Ausgabe verwendeten Drucker und einer durchzuführenden Aktion erzeugt.

6.3.1. Vordefinierte Aktionen

Im Testframework sind bereits einige typlose Aktionen vordefiniert. Außerdem enthält die Klasse `StorageTestBase` Implementierungen für bestimmte typisierte Aktionen.

Im Folgenden wird die Funktionsweise der vordefinierten Aktionen beschrieben. Die Angaben zur serialisierten Form sind nur für externe Tests relevant, d. h. Tests die mehrere Prozesse verwenden:

`DummyAction` (typlos, serialisiert als leere Zeichenfolge) führt keine Aktion aus und verschwindet beim Verketten/Serialisieren. Diese Aktion dient als neutrales Element von `+>` und wird erzeugt, wenn bei der Deserialisierung einer Aktion ein Fehler auftritt.

Pause (typlos) wartet auf eine beliebige Benutzereingabe per Konsole (Standardeingabe). Funktioniert nicht in externen Tests, da Benutzereingaben nicht an den externen Prozess weitergegeben werden können.

GC (typlos, serialisiert als „gc“) führt eine explizite Garbage Collection durch. Da die JVM frei ist, solche Anfragen zu ignorieren, geschieht unter Umständen nichts.

Delete (typlos, serialisiert als „delete“) setzt den verwendeten SKilL-Zustand auf None, d. h. kein aktiver Zustand, zurück und versucht, eine Garbage Collection durchzuführen.

Create (typisiert, serialisiert als „create“) ruft auf der umgebenden StorageTestBase-Instanz die (abstrakten) Methoden create und createElements auf.

Write (typisiert, serialisiert als „write“) ruft auf der umgebenden StorageTestBase-Instanz die (abstrakte) Methode write auf.

Read (typisiert, serialisiert als „read“) ruft auf der umgebenden StorageTestBase-Instanz die (abstrakte) Methode read auf.

CreateMore (typisiert, serialisiert als „createmore“) ruft auf der umgebenden StorageTestBase-Instanz die (abstrakte) Methode createMoreElements auf.

Append (typisiert, serialisiert als „append“) ruft auf der umgebenden StorageTestBase-Instanz die (abstrakte) Methode append auf.

Die von den typisierten Aktionen aufgerufenen Methoden werden im Abschnitt 6.5 beschrieben. Jede typisierte Aktion misst den Speicherverbrauch während ihrer Ausführung und hat entsprechend ein Ergebnis als Konstruktorparameter.

6.4. Verteilungen

Für Tests, die zufällige Werte nutzen, existieren im Testframework Verteilungen. Verteilungen erben von der abstrakten Klasse Distribution. Im folgenden Listing ist diese Klasse zusammen mit allen bereits vordefinierten Verteilungen aufgeführt. Weitere Verteilungen können erzeugt werden. Verteilungen werden vor allem als Parameter für die randomizedTest-Methoden der StorageTestBase-Klasse verwendet.

Listing 6.1: Verteilungen

```
package common.randomHelpers
import scala.util.Random

abstract class Distribution(protected val random: Random) {
  def next: Int
}

class UniformDistribution(_random: Random, val lowerBound: Int, val upperBound: Int)
extends Distribution(_random)

class LogarithmicDistribution(_random: Random, lowerBound: Int, upperBound: Int)
extends Distribution(_random)
```

Eine Verteilung benutzt den übergebenen Zufallsgenerator, um gleichverteilte (Pseudo-)Zufallszahlen als Ausgangswerte zu erhalten. Durch die `next`-Methode gibt die Verteilung dann einen Wert zurück, der entsprechend der gewünschten Verteilung angepasst wurde.

`UniformDistribution` repräsentiert eine Gleichverteilung auf dem Intervall zwischen `lowerBound` (einschließlich) und `upperBound` (ausschließlich). `lowerBound` sollte kleiner als `upperBound` sein.

`LogarithmicDistribution` repräsentiert eine logarithmische Verteilung auf dem Intervall zwischen `lowerBound` (einschließlich) und `upperBound` (ausschließlich). `lowerBound` sollte hier ebenfalls kleiner als `upperBound` und beide größer als 0 sein. Logarithmische Verteilung bedeutet hier, dass die Logarithmen der erzeugten Zahlen gleichverteilt sind.

6.5. Speichertests

Die vorgesehene Weise das Testframework zu verwenden, ist das Schreiben einer Klasse, die von `StorageTestBase` erbt und deren abstrakte Methoden implementiert. Sie enthält Methoden zum Durchführen von Tests, sowohl im selben Prozess als auch in externen Prozessen und die oben beschriebenen vordefinierten typisierten Aktionen. Außerdem sind bereits zwei vordefinierte Tests definiert, die verwendet werden können. Auch ein Speichertest hängt vom Typ des SKilL-Zustands ab, daher hat diese Klasse einen entsprechenden Typparameter.

Dieser Abschnitt erklärt die zur Verwendung des Testframeworks nötigen Schritte und die Funktionsweise der wichtigsten Teile der `StorageTestBase`-Klasse. Im Folgenden wird zuerst der grobe Aufbau der Klasse `StorageTestBase` aufgeführt, in den Unterabschnitten dann die Details zu den einzelnen Aufgabenbereichen.

Listing 6.2: Speichertest

```
package common

import common.storage._
import common.randomHelpers.Distribution

abstract class StorageTestBase[StateType](val name: String) extends CommonTest {
  type Action = storage.Action[StateType]
  type TypedAction = storage.TypedAction[StateType]
  type Task = storage.Task[StateType]

  // Tests
  ...
  // Aktionen
  ...
  // Infrastruktur
  ...
}
```

`CommonTest` (definiert in der SKilL-Scala-Testsuite [SKi14b]) erbt von der Klasse `FunSuite` aus der ScalaTest-Bibliothek [Sca] und unterstützt daher das Schreiben von Tests direkt in der Klasse mittels `test`(*Name*) { *Testinhalt* } und Ausführung der Tests als Scala-JUnit-Tests. Der Konstruktorparameter `name` wird verwendet, um Namen für die temporären Dateien zu generieren, die von den vordefinierten

Tests als Ein- und Ausgabedateien benutzt werden. Die Typdefinitionen `Action`, `TypedAction` und `Task` sollen dem Programmierer eines Speichertests das Mitschleifen des Zustandstyps ersparen und dadurch der Übersichtlichkeit dienen.

6.5.1. Testdurchführung und vordefinierte Tests

Listing 6.3: Speichertest – Tests

```
...
// Tests
def runThread(test: => Unit): Unit
def runProcess(param: String, file: Path, count: Int, action: Action): Unit

def repeatedTest(counts: Array[Int], repetitions: Int,
  task: Task): Unit
def repeatedTest(counts: Array[Int], repetitions: Int,
  param: String, tasks: Action*): Unit
def randomizedTest(samples: Int, distribution: Distribution,
  task: Task): Unit
def randomizedTest(samples: Int, distribution: Distribution,
  param: String, tasks: Action*): Unit
...
```

Die Methoden `runThread` und `runProcess` helfen bei der Ausführung eines Tests. `runThread` führt den übergebenen Ausdruck in einem neuen Thread innerhalb des selben Prozesses aus. Bei den vordefinierten Tests ist dieser Ausdruck die Ausführung der im verwendeten Testablauf festgelegten Aktion mit den entsprechenden Parametern für den aktuellen Durchlauf. `runProcess` dagegen startet einen neuen Prozess für genau einen Durchlauf der übergebenen Aktion. `param` sind Parameter, die an die neu gestartete JVM übergeben werden, `file` ist die für alle Dateioperationen zu verwendende Datei und `count` ist der Parameter des Durchlaufs. Der Name `count` wurde gewählt, da bei den in dieser Arbeit verwendeten Tests dieser Parameter immer die Anzahl der erzeugten Objekte oder Array-, Listen- oder Mengenelemente angibt.

Die beiden vordefinierten Tests `repeatedTest` und `randomizedTest` existieren in zwei Varianten: Die erste Variante nimmt einen Testablauf als Parameter und führt den Test im selben Prozess mittels `runThread` aus. Die zweite Variante führt den Test in externen Prozessen mittels `runProcess` aus. `param` wird dabei einfach weitergereicht. Für jede Aktion in `tasks` wird für jeden Durchlauf nacheinander ein eigener Prozess gestartet. Diese Prozesse verwenden alle den selben Parameter und die selbe Datei. `repeatedTest` entnimmt die Durchlaufparameter dem übergebenen Array `counts`. Für jeden Parameter wird der Test `repetitions` mal durchgeführt. `randomizedTest` generiert die Durchlaufparameter mithilfe der übergebenen Verteilung. Es werden insgesamt `samples` Testdurchläufe ausgeführt.

6.5.2. Aktionen

Die Methoden `write` und `append` könnten bereits in dieser Klasse implementiert werden, wenn alle `SkillState`-Traits von einem gemeinsamen Basis-Trait erben würden, da diese lediglich Methoden des Zustands aufrufen, die in allen `SkillState`-Traits die selbe Deklaration besitzen.

Listing 6.4: Speichertest – Aktionen

```
...
// Aktionen
def create: StateType
def createElements(state: StateType, n: Int): Unit
def write(state: StateType, f: Path): Unit
def read(f: Path): StateType
def createMoreElements(state: StateType, n: Int): Unit
def append(state: StateType): Unit

// Fabriken für typisierte Aktionen und Tasks
object Create     { def apply(res: Option[Result]) = new Create(res)     }
object Write      { def apply(res: Option[Result]) = new Write(res)      }
object Read       { def apply(res: Option[Result]) = new Read(res)       }
object CreateMore { def apply(res: Option[Result]) = new CreateMore(res) }
object Append     { def apply(res: Option[Result]) = new Append(res)     }

def createAndWrite(cr: Option[Result], wr: Option[Result])
  = Create(cr) +> Write(wr)
def readAndAppend(re: Option[Result], cr: Option[Result], ap: Option[Result])
  = Read(re) +> CreateMore(cr) +> Append(ap)

def createTask(printer: TaskBase.Printer, action: Action)
  = new Task(printer, action)
...
```

Die abstrakten Methoden `create`, `createElements`, `write`, `read`, `createMoreElements` und `append` implementieren die vordefinierten typisierten Aktionen. Die Bedeutung dieser Methoden ist wie folgt:

- `create` erzeugt einen neuen, leeren SKilL-Zustand vom entsprechenden Typ.
- `createElements` erzeugt im übergebenen (leeren) Zustand Objekte, Arrayelemente oder Ähnliches und nutzt dabei den Parameter `n`. Ob diese Objekte vom selben Typ sind, insgesamt `n` Objekte oder `n` Objekte von jedem Typ erzeugt werden, ist hier nicht festgelegt.
- `write` schreibt den übergebenen Zustand in die übergebene Datei `f`.
- `read` erzeugt einen neuen SKilL-Zustand, der aus der übergebenen Datei `f` gelesen wird.
- `createMoreElements` erzeugt im übergebenen (nicht-leeren) Zustand weitere Objekte, Arrayelemente oder Ähnliches und nutzt dabei den Parameter `n`. Ebenso wie bei `createElements` ist nichts weiteres festgelegt.
- `append` hängt die neu erzeugten Objekte an die zum gelesenen SKilL-Zustand `state` zugehörige Datei an. Diese Operation ist nur möglich, falls keine vorhandenen Objekte gelöscht oder verändert wurden.

Alle weiteren Objekte und Methoden sind Fabriken für neue Aktionen und Testabläufe. `createAndWrite` sowie `readAndAppend` bieten Abkürzungen für häufig verwendete Aktionsfolgen.

6.5.3. Infrastruktur für externe Tests

Listing 6.5: Speichertest – Infrastruktur

```
abstract class StorageTestBase ... {
  ...
  // Infrastruktur
  def getMainObject: StorageTestBase.ExternalTest[StateType]
  def stringToAction(str: String): Action
  final def externalTestMain(args: Array[String]): Unit
}
object StorageTestBase {
  trait ExternalTest[StateType] {
    def createTest: StorageTestBase[StateType]

    final def main(args: Array[String]) = createTest.externalTestMain(args)
  }
}
```

Um externe Tests korrekt ausführen zu können, benötigt die `StorageTestBase`-Klasse Informationen darüber, welches Objekt die Hauptfunktion eines externen Tests enthält. Dieses Objekt muss in der Lage sein, einen Speichertest vom richtigen Typ zu erzeugen und die benötigten Aktionen auszuführen. Dazu muss der Schreiber eines Speichertests die abstrakte Methode `getMainObject` überschreiben. Der zurückgegebene Wert muss ein statisches Objekt sein, das als Hauptobjekt verwendet werden kann, z. B. das Companion-Objekt des Speichertests. Insbesondere darf dieses Objekt kein anonymes Objekt oder ein Feld einer Klasse sein.

Das von diesem Objekt zu implementierende Trait `StorageTestBase.ExternalTest` definiert bereits eine `main`-Methode mit der benötigten Implementierung. Im Objekt muss nur noch die abstrakte Methode `createTest` definiert werden, die einen Speichertest vom richtigen Typ erzeugt. Die Methode `externalTestMain` interpretiert die durch `runProcess` generierten Parameter und erzeugt daraus einen Testablauf, der die gewünschte Aktion durchführt. Dazu wird die Methode `stringToAction` verwendet, um die übergebene Liste von serialisierten Aktionen zu deserialisieren. Werden benutzerdefinierte Aktionen verwendet, muss diese Methode überschrieben werden, um die Namen der benutzerdefinierten Aktionen zurück zu Aktionsobjekten übersetzen. Die überschreibende Methode sollte als letztes die Basisklassenimplementierung aufrufen, um die Benutzung vordefinierter Aktionen zu ermöglichen.

6.6. Beispiel

In diesem Abschnitt wird am Beispiel der folgenden SKilL-Spezifikation eine mögliche Testimplementierung vorgestellt (siehe auch den ersten Test im nächsten Kapitel):

Listing 6.6: Number-Beispiel

```
Number {
  i64 number;
}
```

Der hier angegebene Speichertest verwendet randomisierte gleichverteilte Objektanzahlen in einem externen Test. Alle in dieser Arbeit verwendeten Tests sind ähnlich aufgebaut. Es wird angenommen, dass aus der obigen Spezifikation ein Paket `number` generiert wurde, das die Implementierung der SKilL-Anbindung enthält.

Listing 6.7: Speichertest-Implementierung

```
import org.junit.runner.RunWith
import org.scalatest.junit.JUnitRunner
import scala.util.Random
import java.nio.file.Path

import number.api.SkillState
import common.storage.{SingleValueResult, CollapsedResult}
import common.randomHelpers.UniformDistribution

@RunWith(classOf[JUnitRunner]) // Ausführung als Scala-JUnit-Test
class StorageTest extends common.StorageTestBase[SkillState]("number") {
  override def create = SkillState.create
  override def createElements(state: SkillState, n: Int) =
    for (i <- 0 until n) state.Number(i)
  override def write(state: SkillState, f: Path) = state.write(f)
  override def read(f: Path) = SkillState.read(f)
  override def createMoreElements(state: SkillState, n: Int) = createElements(state, n)
  override def append(state: SkillState) = state.append

  test("Randomisierter Speichertest") {
    // Ergebnisse
    val createRes     = CollapsedResult("create",      Math.max)
    val writeRes      = CollapsedResult("write",       Math.max)
    val readRes       = CollapsedResult("read",        Math.max)
    val createMoreRes = CollapsedResult("create more", Math.max)
    val appendRes     = CollapsedResult("append",      Math.max)

    // Test
    val random = new Random
    random.setSeed(31948) // für Reproduzierbarkeit
    randomizedTest(
      100,                                            // Anzahl Durchläufe
      new UniformDistribution(random, 1, 30000000),   // Gleichverteilung in [1, 30000000)
      "-Xmx8G",                                       // 8 GB Arbeitsspeicher erlauben
      createAndWrite(Some(createRes), Some(writeRes)), // Aktionen
      readAndAppend(Some(readRes), Some(createMoreRes), Some(appendRes))
    )
    // Ausgabe
    val results = Seq(createRes, writeRes, readRes, createMoreRes, appendRes)
    SingleValueResult.saveGraph(
      "results/number.tex", // Ausgabedatei
      "Numbertest",         // Diagrammtitel
      "axis",               // Achsentyp (normale Achsen)
      "only marks",         // Diagrammtyp (nur Punkte)
      results               // Datenlinien
    )
  }
```

```
  override def getMainObject = StorageTest
}
object StorageTest extends StorageTestBase.ExternalTest[SkillState] {
  def createTest = new StorageTest
}
```

Die Implementierungen der fünf Methoden `create`, `write`, `read`, `append` und `getMainObject` sind im Normalfall identisch zu den in diesem Beispieltest gegebenen. `createElements` und `createMoreElements` dagegen enthalten sehr spezifischen Code für den zu testenden SKilL-Zustand und unterscheiden sich daher von Test zu Test deutlich.

7. Vergleich

In diesem Kapitel werden die ursprüngliche Implementierung [SKi14a] und die optimierte Implementierung aus Kapitel 4 miteinander durch Tests verglichen. Dazu werden Tests verwendet, die mithilfe des Testframeworks aus Kapitel 6 erzeugt wurden. Für jeden Test wird die verwendete SKilL-Spezifikation sowie eine kurze Beschreibung des Tests angegeben.

Der grobe Ablauf aller Tests ist gleich (siehe auch Listing 6.7):

- Es wird ein externer randomisierter Test mit 100 gleichverteilten Parametern n aus dem Intervall $[1,\ 30\ 000\ 000)$ für einfache bzw. $[1,\ 3\ 000\ 000)$ für zusammengesetzte Typen durchgeführt.
- Der erste Arbeitsprozess erzeugt einen neuen SKilL-Zustand und füllt ihn mit Objekten. Anschließend wird der Zustand geschrieben.
- Der zweite Arbeitsprozess liest den geschriebenen Zustand ein, fügt weitere Objekte hinzu und schreibt das Ergebnis mithilfe der Anhängoperation.

Die wesentlichen Unterschiede liegen in den Methoden `createElements` und `createMoreElements`. Daher sind diese oder eine Beschreibung der Elementfabriken für jeden Test angegeben.

In den Diagrammen bezieht sich die Bezeichnung Alt auf die ursprüngliche Implementierung und Neu auf die optimierte. Die Messwerte wurden mit dem Testframework erzeugt, wobei alle 10 ms ein Wert gemessen und von allen Messwerten innerhalb einer Aktion das Maximum behalten wurde. Wegen Garbage Collections auf der JVM, die automatisch ausgelöst werden, bilden die Werte nicht immer eine stetige Kurve, sondern können zwischen verschiedenen benachbarten Parametern deutlich springen.

7.1. Testumgebung

Alle Tests dieser Arbeit wurden mit der in diesem Kapitel angegebenen Testumgebung durchgeführt. Verwendet wurde ein Lenovo T510 Laptop (von 2010) mit der folgenden Ausstattung:

CPU: 2,67 GHz Intel Core i5 560M

RAM: 8 GB 667 MHz DDR3

Betriebssystem: Windows 7 Service Pack 1, 64 Bit

JVM: Java HotSpot(TM) 64-Bit Server VM (build 23.7-b01, mixed mode)

Für die Tests wurde der JVM-Parameter `-Xmx8G` verwendet, der der JVM erlaubt, bis zu 8 GB Speicher zu belegen [Jav].

7.2. Primitive Typen

Diesem Test liegt folgende Spezifikation zugrunde (siehe auch das Beispiel in Abschnitt 6.6):

Listing 7.1: Number.skill

```
Number {
  i64 number;
}
```

Dieser Test zeigt den Speicherverbrauch von einfachen Typen mit primitiven Feldern. Für andere primitive Feldtypen wird entsprechend weniger bzw. gleich viel Speicher benötigt, das Ergebnis ist jedoch analog.

Beide Elementfabriken sind identisch und erzeugen n Objekte, wobei das Feld `number` mit aufsteigenden Zahlen von 0 bis $n - 1$ gefüllt wird. Es werden also für einen Parameter n in beiden Prozessen jeweils n Elemente erzeugt.

Die Diagramme 7.1 und 7.2 zeigen den Speicherverbrauch für diesen Test für die unterschiedlichen Operationen. Zu sehen ist deutlich, dass die optimierte Implementierung außer beim Schreiben deutlich weniger Speicherplatz pro Objekt benötigt.

7.3. Mehrere Felder

Zum Vergleich mit dem vorigen Test wurde die folgende Spezifikation verwendet:

Listing 7.2: Fields.skill

```
Fields {
  i32 a;
  i32 b;
}
```

Der Typ `Fields` hat die selbe Größe wie der Typ `Number` aus dem vorigen Abschnitt. Die Elementfabriken sind wie oben definiert, mit dem einzigen Unterschied, dass beide Felder mit den selben Zahlen gefüllt werden.

In den Diagrammen 7.3 und 7.4 ist deutlich zu sehen, dass die ursprüngliche Implementierung bei mehreren Feldern übermäßig viel Speicher benötigt. Durch den effizienteren Ausgabepuffer in der optimierten Implementierung reduziert sich der Speicherverbrauch auf ein vernünftiges Maß.

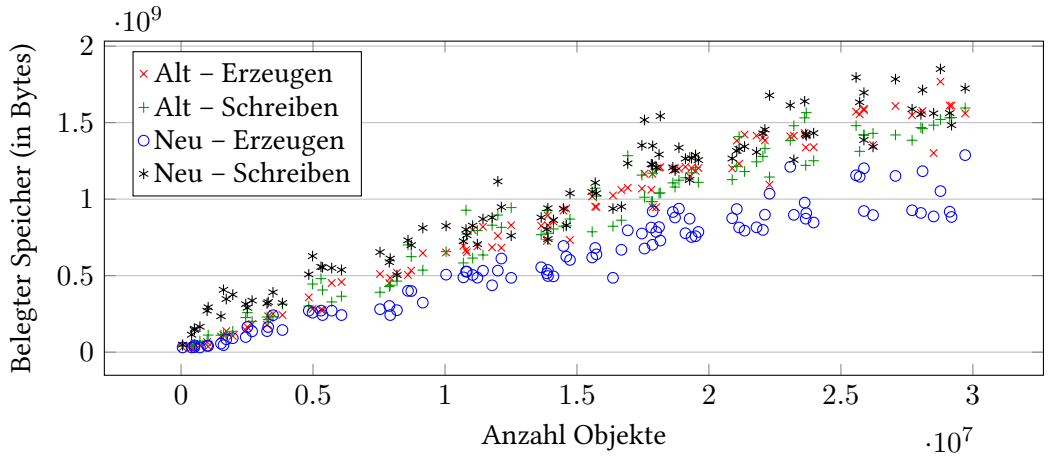

Abbildung 7.1.: Number – Erzeugen und Schreiben von Objekten

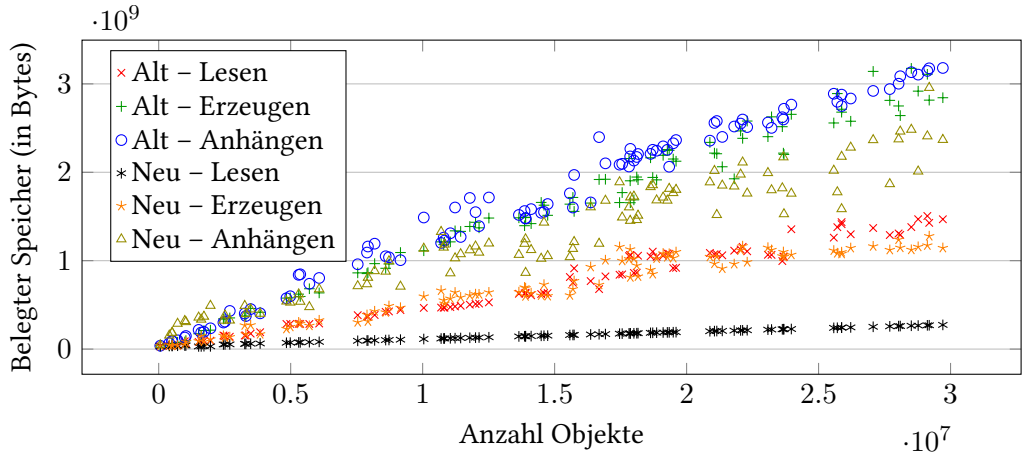

Abbildung 7.2.: Number – Lesen, Erzeugen von weiteren Objekten und Anhängen

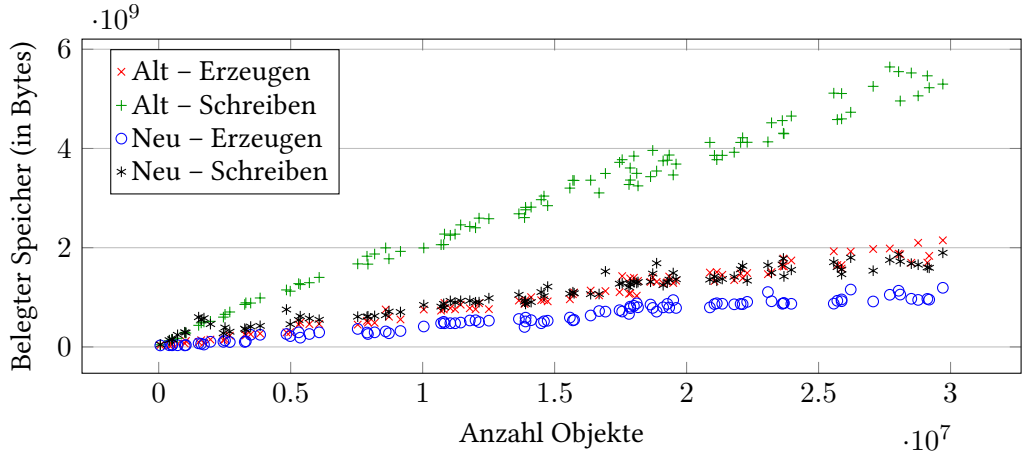

Abbildung 7.3.: Fields – Erzeugen und Schreiben von Objekten

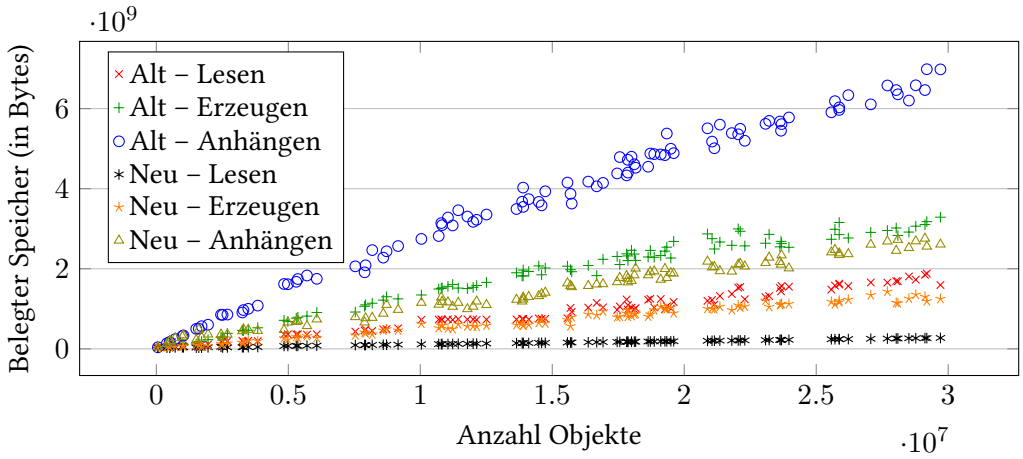

Abbildung 7.4.: Fields – Lesen, Erzeugen von weiteren Objekten und Anhängen

7.4. Referenzen

Referenzen brauchen auf einem 64-Bit-System eigentlich genauso viel Speicher wie der SKilL-Typ i64. Da aber bei der verwendeten JVM standardmäßig der Parameter `-XX:+UseCompressedOops` [Jav] gesetzt ist, mit dem die JVM statt 64-Bit-Zeigern 32-Bit-Offsets für Referenzen verwendet[1], belegen Referenzen lediglich 32 Bit. Der Typ aus der folgenden Spezifikation braucht daher theoretisch weniger Speicher als die obigen Typen.

Listing 7.3: Cycle.skill

```
Cycle {
  Cycle prev;
}
```

Die Elementfabriken sind hier definiert als

Listing 7.4: Referenzen – Elemente

```
def createElements(state: SkillState, n: Int) {
  var c = state.Cycle(null)
  for (i <- 1 until n) c = state.Cycle(c)
  state.Cycle.head.prev = c
}
def createMoreElements(state: SkillState, n: Int) {
  var c = state.Cycle.last
  for (i <- 0 until n) c = state.Cycle(c)
  state.Cycle.head.prev = c
}
```

`head` gibt dabei die erste Instanz eines Speicherpools zurück. Es wird also eine einfach verlinkte zyklische Liste ohne Inhalte erzeugt.

[1] Diese Kompression kann durch den Parameter `-XX:-UseCompressedOops` deaktiviert werden. Es wurde aber entschieden, bis auf den Parameter `-Xmx8G` die Standardwerte der JVM zu verwenden.

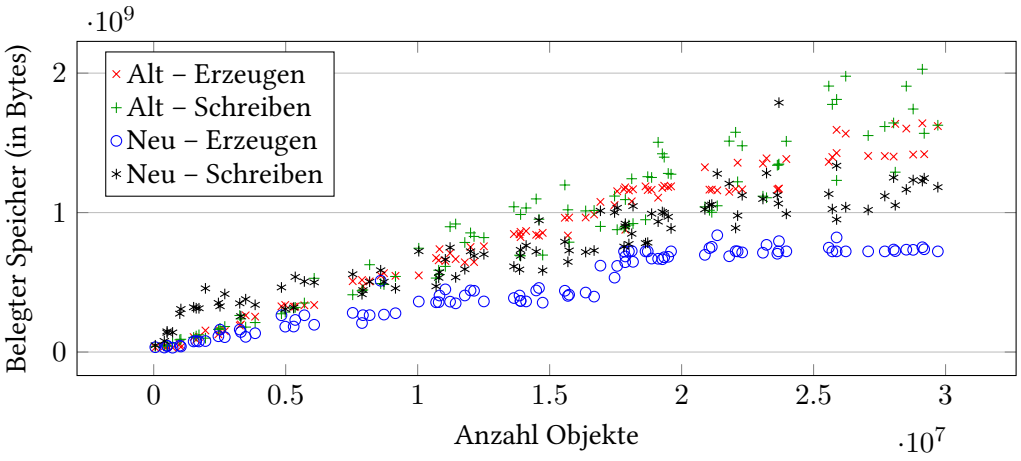

Abbildung 7.5.: Cycle – Erzeugen und Schreiben von Objekten

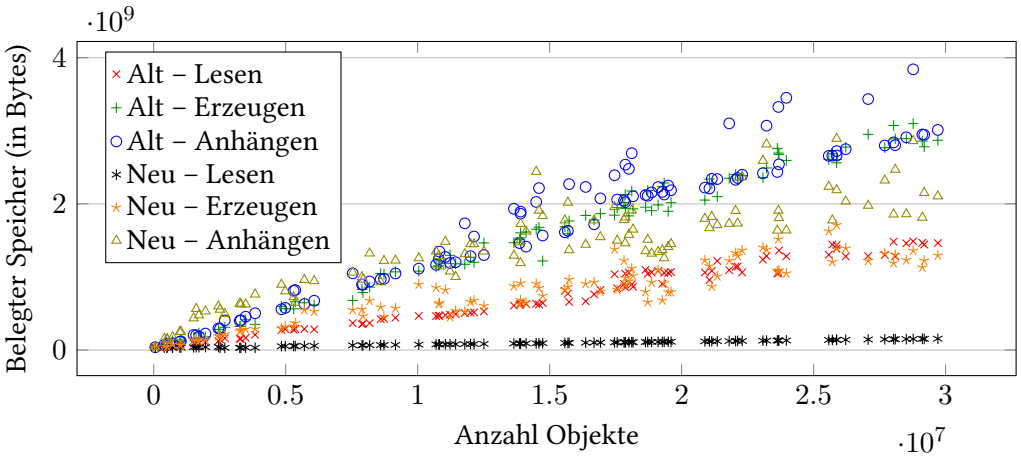

Abbildung 7.6.: Cycle – Lesen, Erzeugen von weiteren Objekten und Anhängen

7.5. Abgeleitete Typen

Dieser Abschnitt enthält mehrere Tests, die alle auf der folgenden Spezifikation aufbauen:

Listing 7.5: Derivation.skill

```
Base {
  i32 base;
}

Derived : Base {
  i32 derived;
}

Empty : Derived {
}
```

Der erste Test erzeugt nur Objekte vom Typ `Derived` und dient zum Vergleich des Verhaltens der verschiedenen Implementierungen bei Typhierarchien.

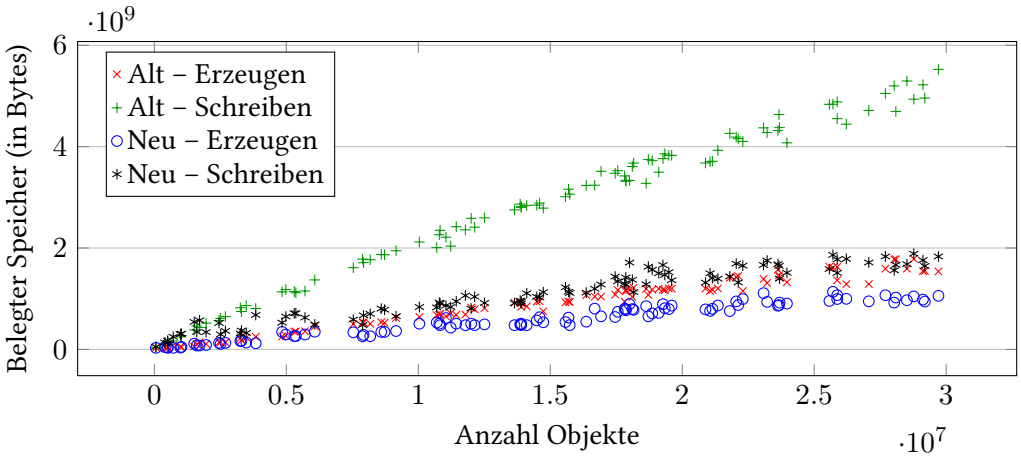

Abbildung 7.7.: Derivation Test 1 – Erzeugen und Schreiben von Objekten

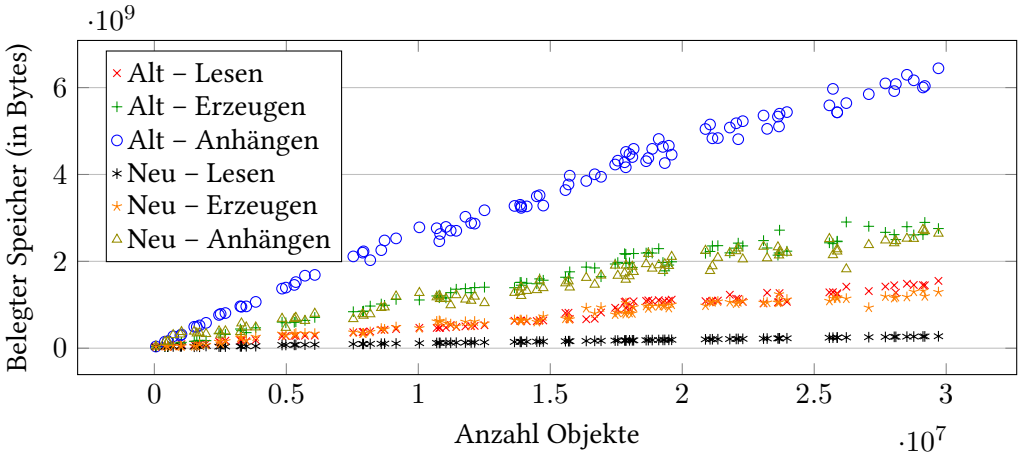

Abbildung 7.8.: Derivation Test 1 – Lesen, Erzeugen von weiteren Objekten und Anhängen

Wie in den Diagrammen 7.7 und 7.8 zu sehen ist, braucht in der ursprünglichen Implementierung der Schreib- bzw. Anhängvorgang übermäßig viel Speicher. Das ist wie beim Fields-Test auf die vielen kleinen Arrays im Puffer zur Größenberechnung (siehe Abschnitt 4.8) zurückzuführen.

Im zweiten Test wird der Speicherverbrauch unter Verwendung von Objekten des Typs `Empty` mit den Ergebnissen vom ersten Test verglichen. Da die ursprüngliche Implementierung bei diesem Test beim Schreiben aufgrund einer `java.lang.NullPointerException`[2] abgestürzt ist, sind für diese keine Ergebnisse vorhanden.

[2]Die Ausnahme ist beim Schreiben des *base*-Felds von *Base* in der generierten Datei `StateWriter.scala` aufgetreten.

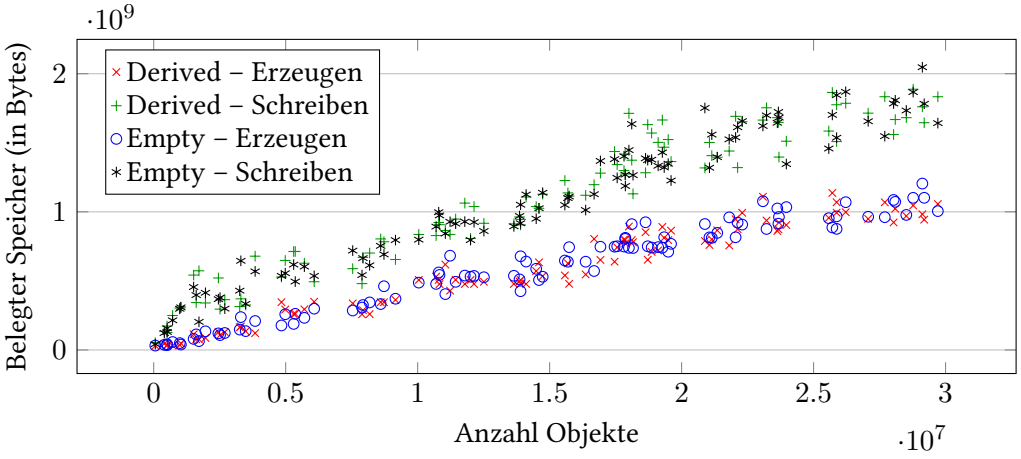

Abbildung 7.9.: Derivation Test 2 – Erzeugen und Schreiben von Objekten

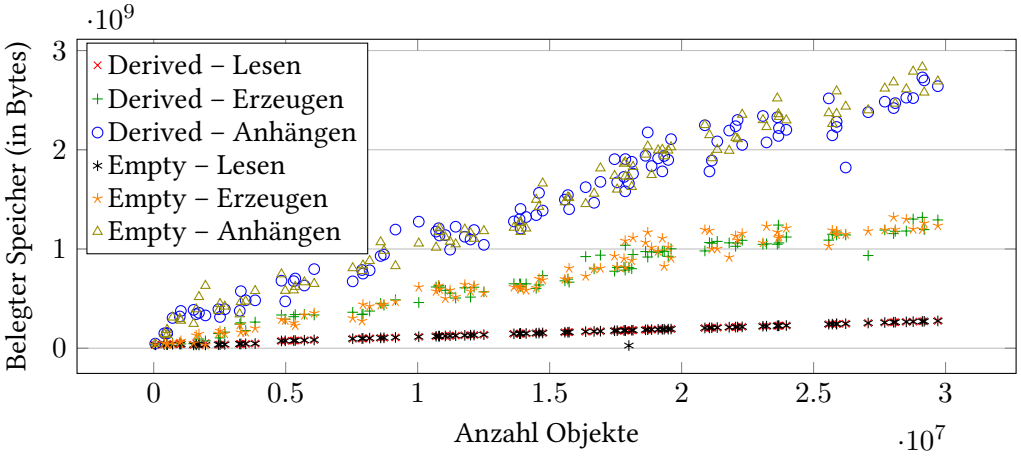

Abbildung 7.10.: Derivation Test 2 – Lesen, Erzeugen von weiteren Objekten und Anhängen

Die Diagramme 7.9 und 7.10 zeigen, dass in der optimierten Implementierung leere (Unter-)Typen bis auf den assoziierten Speicherpool keinen zusätzlichen Speicher brauchen. Der zusätzliche Speicher für den Speicherpool ist aber vernachlässigbar, da er konstant und klein ist.

7.6. Zusammengesetzte Typen

Für zusammengesetzte Typen werden außer bei Arrays fester Größe immer zwei Tests durchgeführt. Im ersten Test wird die Anzahl der Elemente pro Objekt auf 10 festgesetzt und die Anzahl der Objekte variiert, im zweiten wird die Anzahl der Objekte auf 10 festgesetzt und die Anzahl der Elemente pro Objekt variiert. Da im zweiten Test die Objekte bearbeitet werden, ist hier Anhängen nicht möglich, daher entfällt dieser Schritt. Der Einfachheit halber sind für Arrays und Listen alle Einträge Annotationen, die auf nichts verweisen. Es könnten auch Einträge mit gültigen Zielen verwendet werden, z. B. Selbstverweise, aber das erhöht nur die Komplexität des Tests und ändert am

Speicherverbrauch nichts. Bei Mengen und Maps ist es dagegen nicht möglich, solche Einträge zu verwenden, da diese jedes Element bzw. jeden Schlüssel nur einmal erlauben. Mengen verwenden daher i64 als Einträge, welche in der optimierten Implementierung genau gleich viel Speicherplatz belegen wie Annotationen. Da sich Mengen und Maps prinzipiell sehr ähnlich verhalten und der Test für Mengen mit der ursprünglichen Implementierung bereits 39 Stunden benötigte, wurde kein getrennter Test für Maps durchgeführt.

7.6.1. Arrays fester Größe

Dieser Test verwendet die folgende Spezifikation:

Listing 7.6: FixedArray.skill

```
Fixed {
  annotation[10] fixed;
}
```

Die Elementfabriken produzieren auch hier n Objekte und sind identisch für beide Erzeugungsphasen. Da in diesem Fall aber die Benutzerschnittstelle der beiden Implementierungen unterschiedlich sind, mussten die Elementfabriken an die jeweilige Implementierung angepasst werden. Die folgenden beiden Listings zeigen die verschiedenen Varianten für `createElements`:

Listing 7.7: Arrays fester Größe – Elemente, ursprüngliche Implementierung

```
def createElements(state: SkillState, n: Int) =
  for (i <- 0 until n) {
    val data = ArrayBuffer[SkillType](null, null, null, null, null, null, null, null, null, null)
    state.Fixed(data)
  }
```

Listing 7.8: Arrays fester Größe – Elemente, optimierte Implementierung

```
def createElements(state: SkillState, n: Int) =
  for (i <- 0 until n) {
    val data = state.makeAnnotationArray(10)
    state.Fixed(data)
  }
```

Durch die interne Verwendung der Klasse `Array` statt `ArrayBuffer` verringert sich der Speicherverbrauch in der optimierten Implementierung beim Erzeugen neuer Objekte, ansonsten verändert sich der Speicherverbrauch hier kaum (siehe Diagramme 7.11 und 7.12).

7.6.2. Arrays variabler Größe und Listen

Für Arrays variabler Größe und Listen werden fast identische Tests verwendet. Die zugrundeliegenden Spezifikationen sind:

Listing 7.9: VariableArray.skill

```
Variable {
  annotation[] variable;
}
```

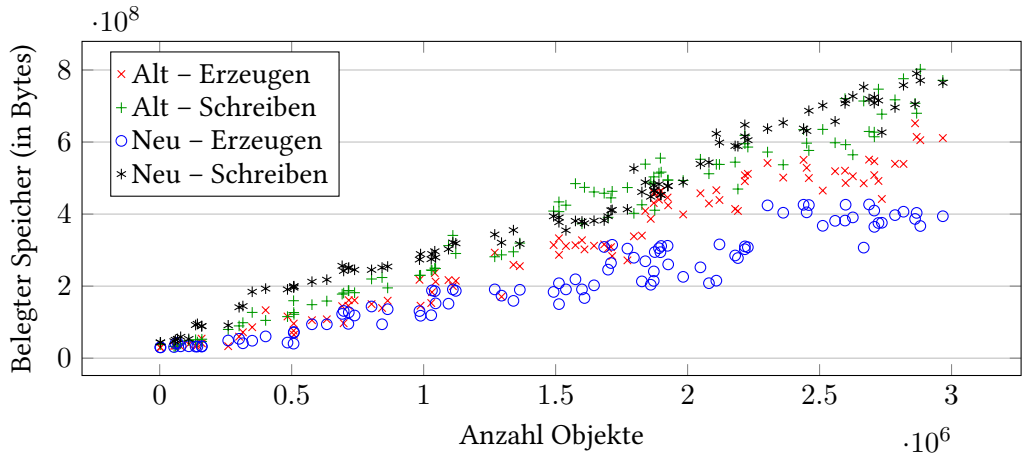

Abbildung 7.11.: FixedArray – Erzeugen und Schreiben von Objekten

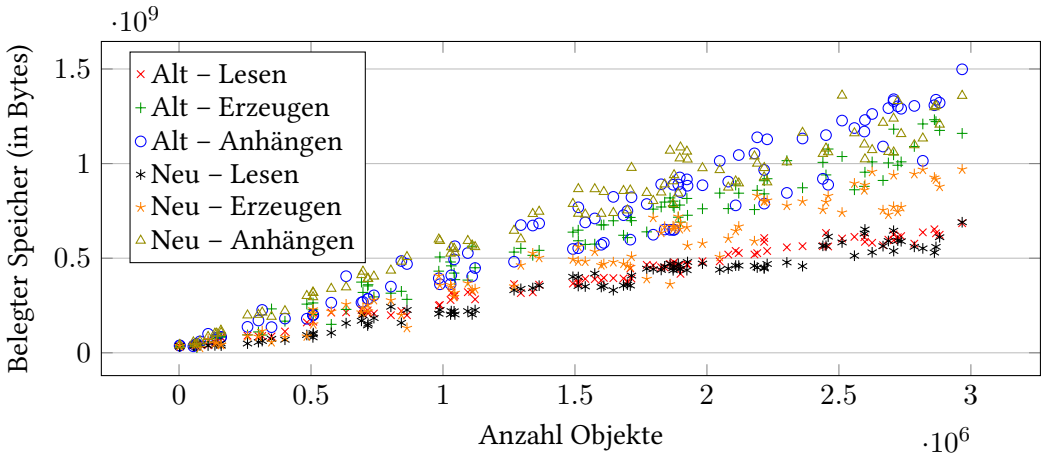

Abbildung 7.12.: FixedArray – Lesen, Erzeugen von weiteren Objekten und Anhängen

Listing 7.10: List.skill

```
AList {
  list<annotation> aList;
}
```

Im ersten Test werden in beiden Elementfabriken n Objekte mit jeweils 10 Elementen im enthaltenen Array bzw. in der enthaltenen Liste erzeugt. Dazu werden folgende Definitionen von `createElements` verwendet:

Listing 7.11: Arrays variabler Größe und Listen – Elemente, ursprüngliche Implementierung (Test 1)

```
def createElements(state: SkillState, n: Int) =
  for (i <- 0 until n) {
    val data = ArrayBuffer[SkillType]() // bzw. ListBuffer[SkillType]()
    for (j <- 0 until 10) data += null
    state.Variable(data)
  }
```

Listing 7.12: Arrays variabler Größe und Listen – Elemente, optimierte Implementierung (Test 1)

```
def createElements(state: SkillState, n: Int) =
  for (i <- 0 until n) {
    val data = state.makeAnnotationVarArray() // bzw. state.makeAnnotationList()
    for (j <- 0 until 10) data += null
    state.Variable(data)
  }
```

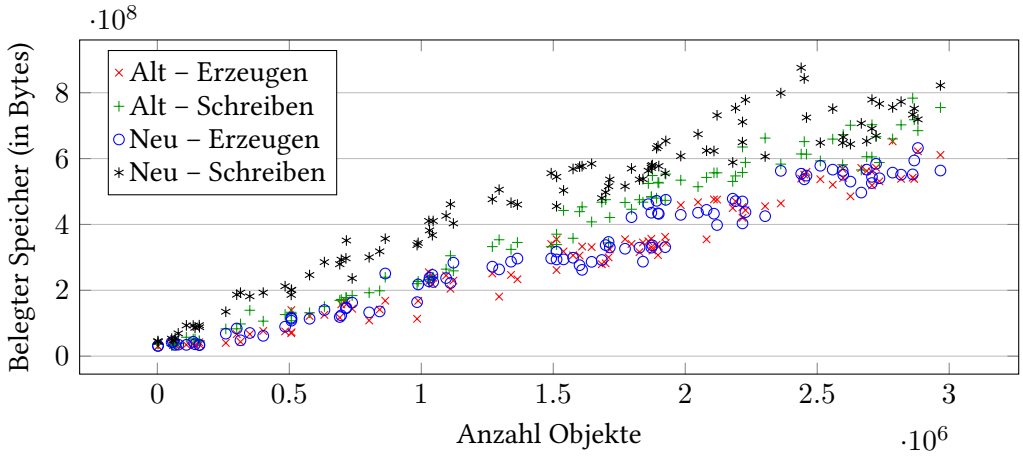

Abbildung 7.13.: VariableArray Test 1 – Erzeugen und Schreiben von Objekten

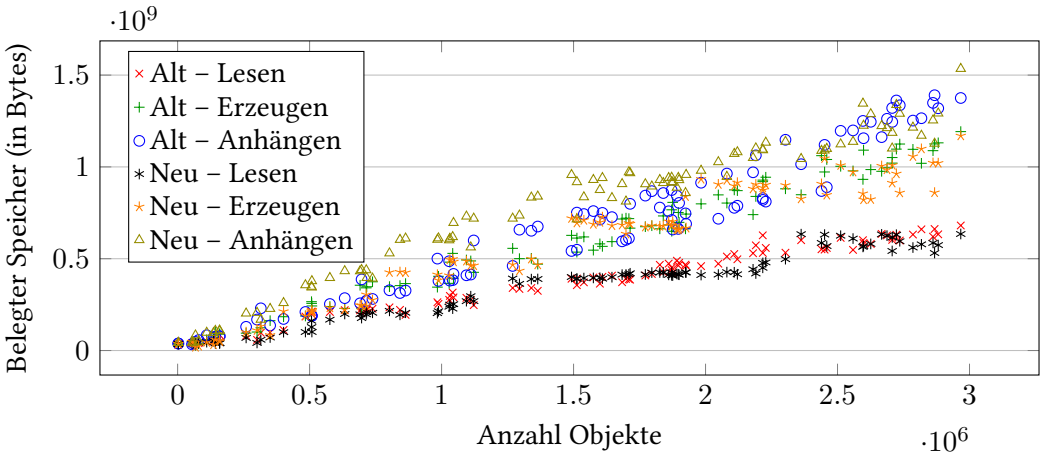

Abbildung 7.14.: VariableArray Test 1 – Lesen, Erzeugen von weiteren Objekten und Anhängen

Die Diagramme 7.13 bis 7.16 zeigen deutlich den in Abschnitt 4.7.4 beschriebenen Effekt, dass durch Boxing der primitiven Typen in der optimierten Implementierung kein Speicher eingespart werden kann. Im Gegenteil wird der Speicherverbrauch beim Schreiben sogar noch durch die Umordnungstabellen erhöht.

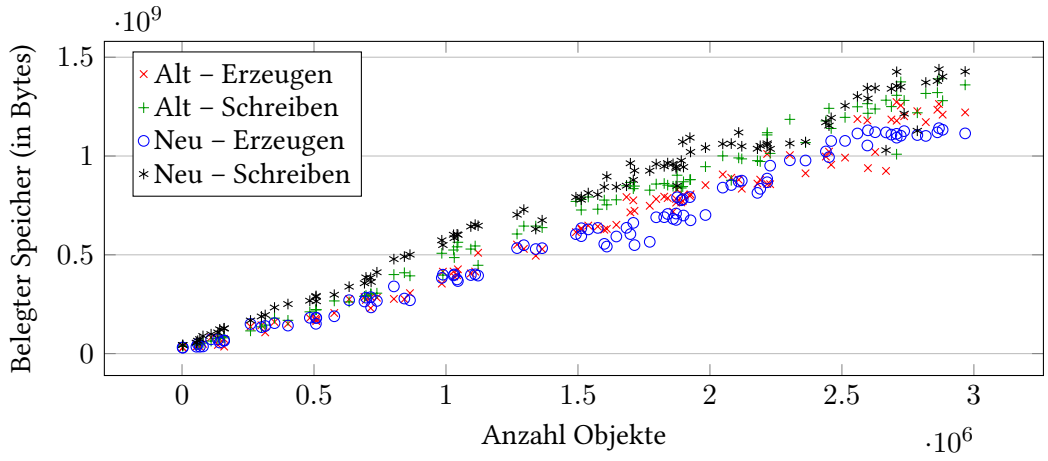

Abbildung 7.15.: List Test 1 – Erzeugen und Schreiben von Objekten

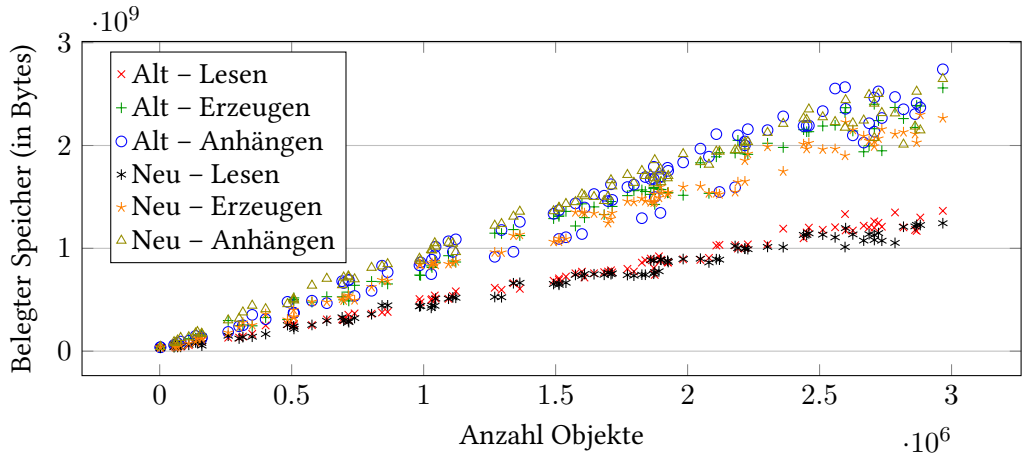

Abbildung 7.16.: List Test 1 – Lesen, Erzeugen von weiteren Objekten und Anhängen

Im zweiten Test werden in der ersten Elementfabrik 10 Objekte mit jeweils n Elementen im enthaltenen Array bzw. in der enthaltenen Liste erzeugt. Die zweite Elementfabrik erzeugt in den bereits vorhandenen 10 Objekten n weitere Array- bzw. Listenelemente.

Listing 7.13: Arrays variabler Größe und Listen – Elemente, ursprüngliche Implementierung (Test 2)

```
def createElements(state: SkillState, n: Int) =
  for (i <- 0 until 10) {
    val data = ArrayBuffer[SkillType]() // bzw. ListBuffer[SkillType]()
    for (j <- 0 until n) data += null
    state.Variable(data)
  }
def createMoreElements(state: SkillState, n: Int) =
  for (obj <- state.Variable.all) { // bzw. obj <- state.AList.all
    val data = obj.variable // bzw. obj.aList
    for (j <- 0 until n) data += null
  }
```

Listing 7.14: Arrays variabler Größe und Listen – Elemente, optimierte Implementierung (Test 2)

```
def createElements(state: SkillState, n: Int) =
  for (i <- 0 until 10) {
    val data = state.makeAnnotationVarArray() // bzw. state.makeAnnotationList()
    for (j <- 0 until n) data += null
    state.Variable(data)
  }
def createMoreElements(state: SkillState, n: Int) =
  for (obj <- state.Variable.all) { // bzw. obj <- state.AList.all
    val data = obj.variable // bzw. obj.aList
    for (j <- 0 until n) data += null
  }
```

Wie die Diagramme 7.17 bis 7.20 zeigen, braucht die optimierte Implementierung in den meisten Fällen in etwa gleich viel Speicher wie die ursprüngliche Implementierung. Hier ist der zusätzliche Speicherverbrauch durch die Umordnungstabellen kaum zu bemerken, da diese nur 10 Einträge enthalten, also sehr klein sind.

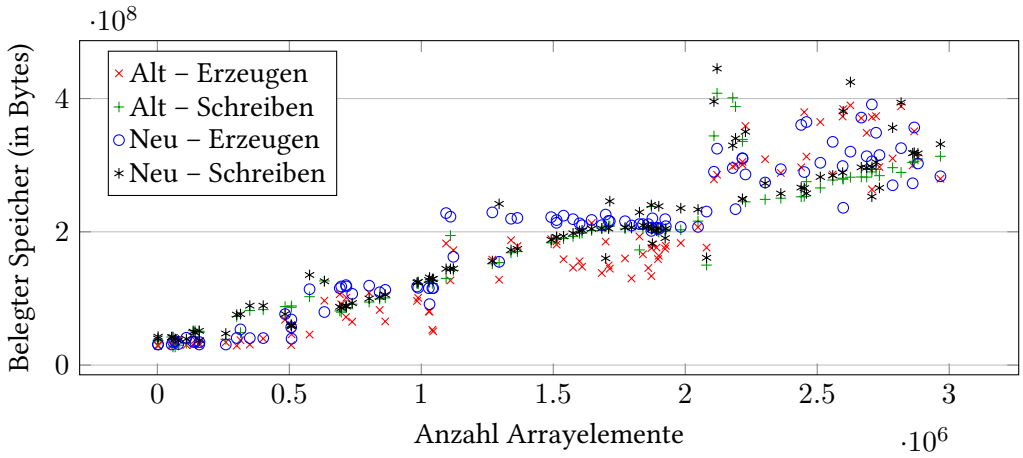

Abbildung 7.17.: VariableArray Test 2 – Erzeugen und Schreiben von Objekten

Diese Tests zeigen deutlich, dass durch spezialisierte Datenstrukturen für primitive Typen noch deutlich Speicherplatz eingespart werden kann. Für Basistypen, die keine bekannten Untertypen besitzen, könnten zusätzlich noch die Umordnungstabellen für neue Objekte eingespart werden, da nur von bekannten Typen Objekte erzeugt werden können.

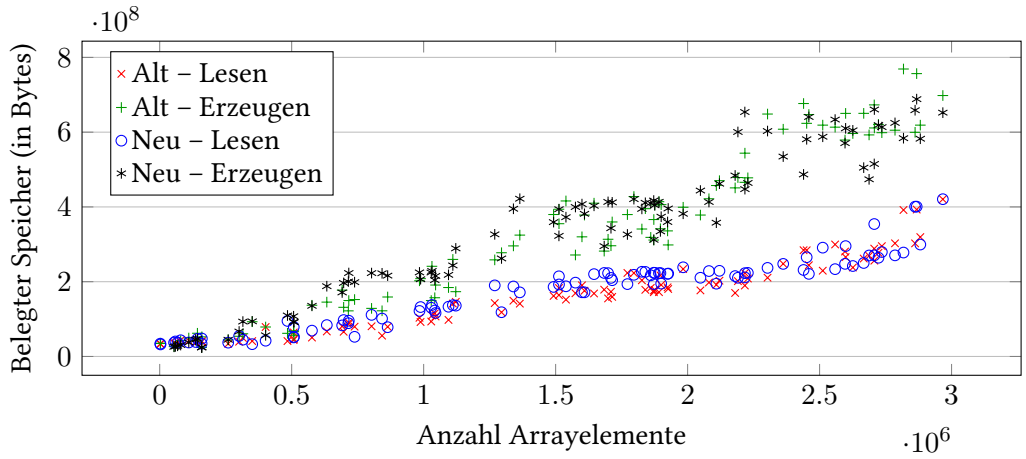

Abbildung 7.18.: VariableArray Test 2 – Lesen und Erzeugen von weiteren Objekten

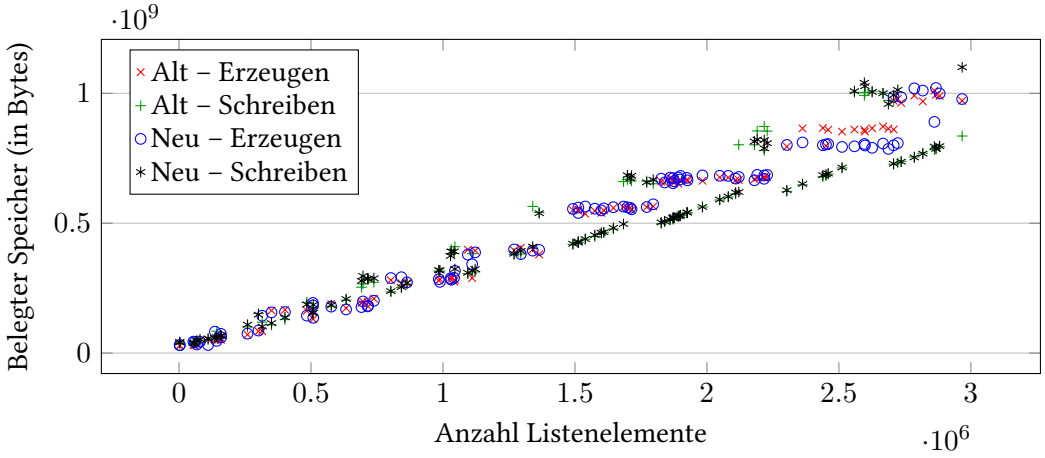

Abbildung 7.19.: List Test 2 – Erzeugen und Schreiben von Objekten

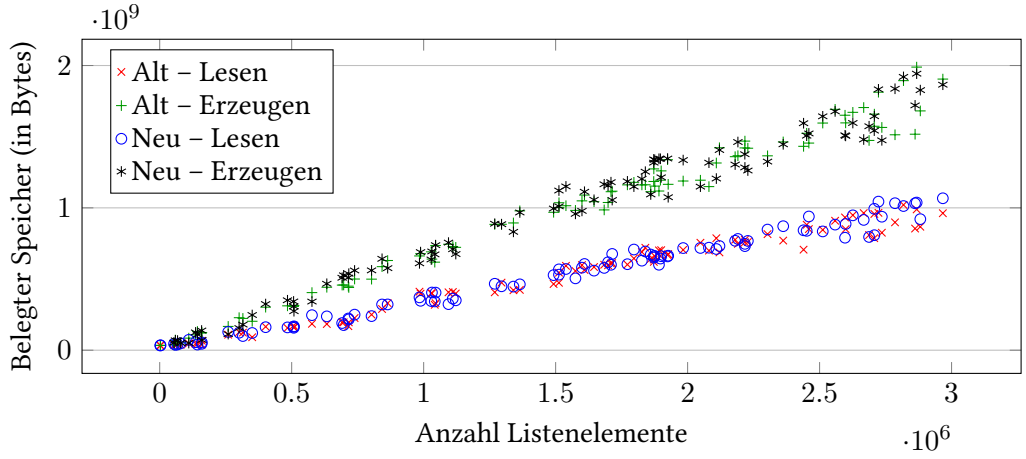

Abbildung 7.20.: List Test 2 – Lesen und Erzeugen von weiteren Objekten

7.6.3. Mengen

Für Mengen wird die folgende Spezifikation verwendet:

Listing 7.15: Set.skill

```
ASet {
  set<i64> aSet;
}
```

Wie im vorigen Abschnitt erzeugen auch hier im ersten Test die Elementfabriken n Objekte mit jeweils 10 Einträgen im enthaltenen `HashSet`. Allerdings sind hier für beide Implementierungen die Elementfabriken gleich:

Listing 7.16: Mengen – Elemente (Test 1)

```
def createElements(state: SkillState, n: Int) =
  for (i <- 0 until n) state.ASet(HashSet(0, 1, 2, 3, 4, 5, 6, 7, 8, 9))
```

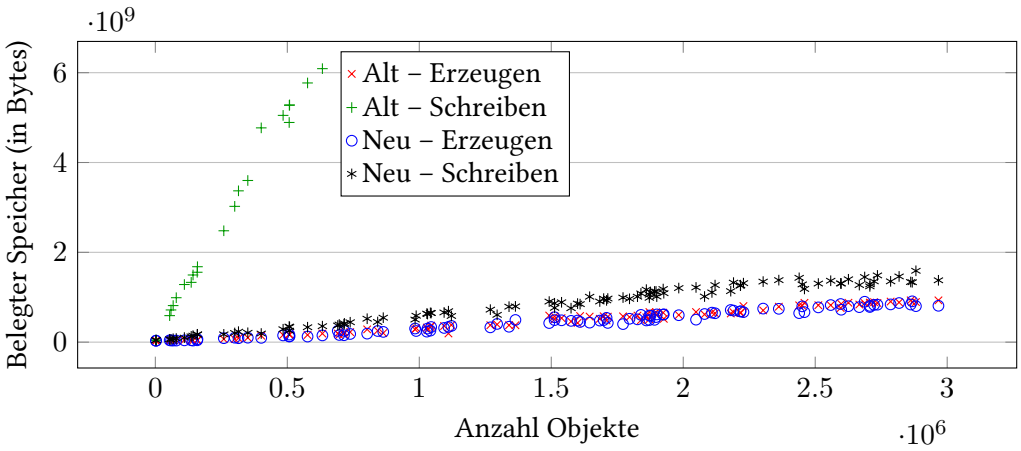

Abbildung 7.21.: Set Test 1 – Erzeugen und Schreiben von Objekten

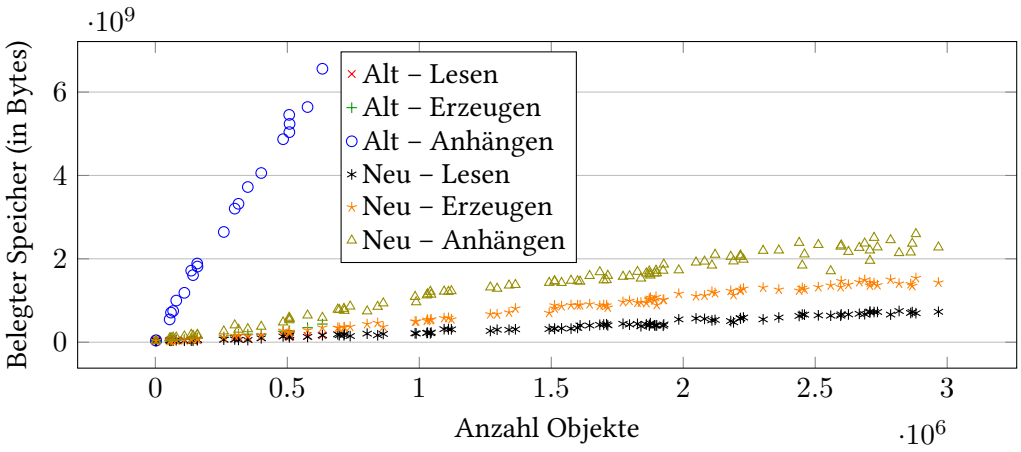

Abbildung 7.22.: Set Test 1 – Lesen, Erzeugen von weiteren Objekten und Anhängen

Wie die Diagramme 7.21 und 7.22 zeigen, hat die ursprüngliche Implementierung Probleme, wenn viele Mengen vorhanden sind. Auch hier ist das Problem wieder der Ausgabepuffer, gefüllt mit vielen kleinen Arrays (siehe Abschnitt 4.8). Da die Mengen selbst aber auch viel Arbeitsspeicher benötigen, tritt hier das Problem in verstärkter Form auf, so dass es schon für weniger als eine Million Objekte während der Schreibaktion zum Speicherüberlauf kommt, d. h. das Programm wegen eines `java.lang.OutOfMemoryError` abstürzt (fehlende Datenpunkte).

Im zweiten Test werden in `createElements` 10 Objekte erzeugt und mit den Zahlen 0 bis $n-1$ befüllt, in `createMoreElements` werden weitere n Zahlen, n bis $2n-1$, hinzugefügt.

Listing 7.17: Mengen – Elemente (Test 2)

```
def createElements(state: SkillState, n: Int) =
  for (i <- 0 until 10) {
    val data = new HashSet[Long]();
    for (j <- 0 until n) data += j
    state.ASet(data)
  }
def createMoreElements(state: SkillState, n: Int) =
  for (obj <- state.ASet.all) {
    val data = obj.aSet
    for (j <- 0 until n) data += j + n
  }
```

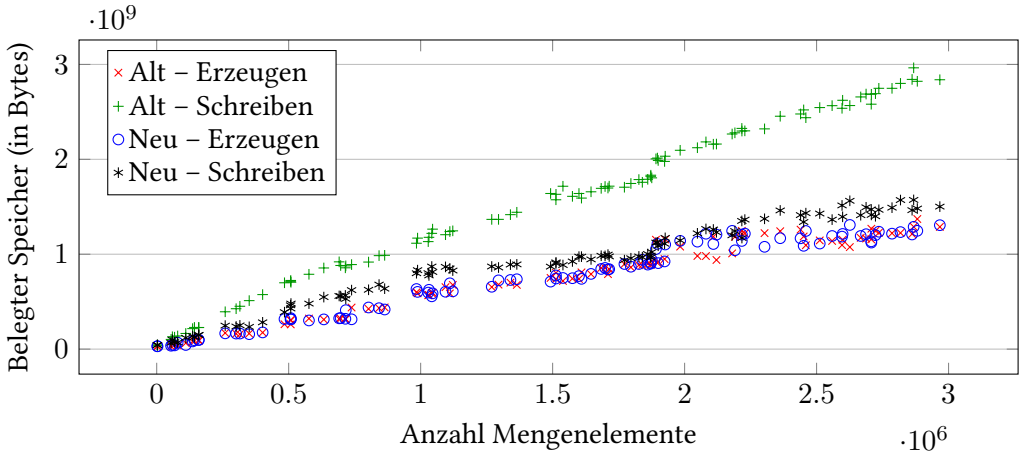

Abbildung 7.23.: Set Test 2 – Erzeugen und Schreiben von Objekten

In den Diagrammen 7.23 und 7.24 kann man sehen, dass sich die ursprüngliche und die optimierte Implementierung (bis auf das Schreiben) gleich verhalten. Das ist darauf zurückzuführen, dass die Mengen zwar an einer anderen Stelle, nämlich im Speicherpool, gespeichert sind, die Mengen selbst sich aber nicht verändert haben, da hier ein primitiver Elementtyp gewählt wurde. Der Unterschied beim Schreiben wird auch hier wieder durch den Ausgabepuffer ausgelöst.

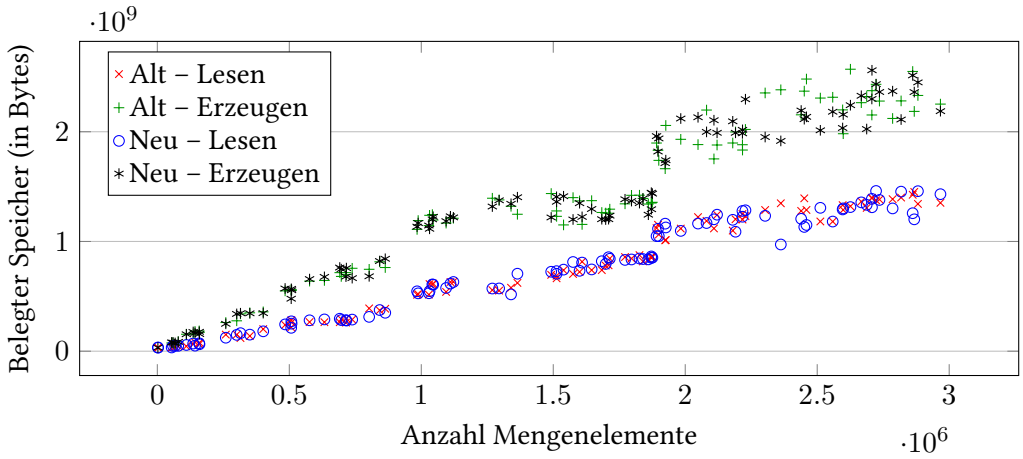

Abbildung 7.24.: Set Test 2 – Lesen und Erzeugen von weiteren Objekten

7.7. Laufzeit und Korrektheit

Zur Überprüfung der Laufzeit und Korrektheit wurde die SKilL-Scala-Testsuite [SKi14b] mit leichten Anpassungen an die veränderte Benutzerschnittstelle für die optimierte Implementierung ausgeführt. Dabei wurde festgestellt, dass die optimierte Implementierung in den Fällen langsamer als die ursprüngliche Implementierung war, in denen wenige Objekte erzeugt wurden, viele Feldzugriffe stattfanden oder viele Referenzen korrigiert werden mussten. Jedoch lag der zeitliche Mehraufwand meist unter 25% des Zeitaufwands für die ursprüngliche Implementierung, außer wenn Referenzen überprüft und eventuell korrigiert werden mussten. In diesen Fällen wurden zum Teil, vor allem beim `graph.WSR14Test` mit Referenzen in `HashSet`-Objekten, doppelte bis vierfache Laufzeiten festgestellt, allerdings war der Faktor für größere Objektanzahlen geringer als für kleinere. Der Test `date.WriteTest` zeigte dagegen, dass für große Datenmengen (1,6 Millionen Objekte bzw. 10 MB Daten aus einer Datei) die optimierte Implementierung sogar weniger Zeit benötigt. Auch bei allen hier durchgeführten Speichertests war die optimierte Implementierung deutlich schneller als die ursprüngliche. Insbesondere benötigten bei der optimierten Implementierung die Lesevorgänge selbst für große Dateien (etwa 30 Millionen Objekte) wenige Millisekunden. Die ursprüngliche Implementierung dagegen benötigte mehrere Sekunden.

Alle Tests der Testsuite wurden fehlerfrei ausgeführt und die erzeugten Dateien entsprachen den erwarteten Ergebnissen. Die ursprüngliche Implementierung erzeugte in zwei Teilen des Tests `toolchains.node.CoreTest` dagegen fehlerhafte Ergebnisse beim Schreiben eines Zustands in zwei verschiedene Dateien.

7.8. Zusammenfassung der Ergebnisse

Die Ergebnisse aller Tests zusammengenommen zeigen, dass für Felder mit einfachen Type, d. h. nicht zusammengesetzten Typen, die Optimierungen die gewünschte Wirkung gezeigt haben. Hier

wurde zum Teil deutlich weniger Speicher verbraucht, insbesondere bei Schreib- und Anhängoperationen durch die zusätzliche Wirkung des verbesserten Ausgabepuffers. Bei zusammengesetzten Typen dagegen wurde lediglich bei Arrays konstanter Länge durch die Verwendung von Arrays statt `ArrayBuffer`-Objekten Speicher eingespart, in allen anderen Fällen wird durch die Typlöschung nicht nur kein Speicher eingespart, sondern sogar beim Schreiben und Anhängen durch die Umordnungstabellen mehr Speicher verbraucht.

Weiterhin hat sich die Laufzeit beim Lesen und Schreiben vieler Daten deutlich verbessert, ausgenommen der Fälle, in denen viele Referenzen überprüft werden müssen, dafür ist der Zugriff auf Felddaten durch mehr Indirektion vor allem bei neu erzeugten Objekten deutlich langsamer.

8. Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurden einige SKilL-Spezifikationen genutzt, die einzelne Bereiche der Scala-Anbindung getrennt voneinander beleuchten. Die entsprechenden Tests wurden einfach gehalten, da es nur auf den Speicherverbrauch des aus den SKilL-Spezifikationen generierten Codes ankam. Diese werten den Speicherverbrauch in den dafür interessanten Phasen der Scala-Anbindung aus. Das beinhaltet das Erzeugen von Objekten sowie das Lesen, Schreiben und Anhängen von Objekten aus, in bzw. an SKilL-Dateien.

Bei diesen Tests wurde festgestellt, dass die ursprüngliche Implementierung kaum Speicherplatz verschwendet, d. h. für ein Scala-Programm übermäßig viel Speicher benötigt, mit Ausnahme des beim Schreiben und Anhängen verwendeten Ausgabepuffers. Da aber viele der verwendeten Objekte relativ klein sind, d. h. weniger Felder in der SKilL-Spezifikationen sind als Objekte existieren, bestand hier dennoch eine Verbesserungsmöglichkeit.

Um den Speicherverbrauch der Scala-Anbindung zu verringern, ist es aufgrund des Grundspeicherbedarfs von Objekten und der Architektur der JVM sinnvoll, die Anzahl langlebiger Objekte, vor allem von kleinen, zu reduzieren. Gleichzeitig sollte die Größe der verbleibenden langlebigen Objekte erhöht werden. Dazu können die Daten der SKilL-Objekte statt direkt im Objekt als Datensatz im zugehörigen Speicherpool aufbewahrt werden, d. h. lokale Felder durch verteilte Felder ersetzt werden. Zum Zugriff auf die Daten eines Datensatzes kann ein Proxyobjekt verwendet werden, so dass sich für den Benutzer die Schnittstelle kaum ändert.

Gelesene und bereits geschriebene Daten werden dabei analog zur serialisierten Form gespeichert und mittels der selben SKilL-ID adressiert. Feldzugriffe dagegen benötigen mehr Indirektionen und sind daher langsamer. Ebenso geht die ursprüngliche Cache-Lokalität für kleinere Objekte verloren.

Neu erzeugte, bisher ungeschriebene Daten haben zudem das Problem, dass sie anders als gelesene bzw. geschriebene Daten nicht nach ihrem Typ in Blöcke sortiert sind, sondern in beliebiger Reihenfolge vorliegen können. Daher werden hier zusätzliche Verwaltungsstrukturen benötigt. Um die Iteration in Typreihenfolge zu unterstützen, werden weiterhin alle Daten aller Objekte eines Typs in einem Speicherpool gesammelt, statt wie bei gelesenen bzw. geschriebenen Daten im assoziierten Speicherpool des das Feld definierenden Typs.

Die veränderte Speicherung der Daten benötigt außerdem einen Auflösungsmechanismus, um aus den Adressen von Datensätzen, gespeichert in den SKilL-IDs, Objekte erzeugen zu können und umgekehrt von diesen Objekten Rückschlüsse auf die SKilL-ID zu ziehen, um auf den zugehörigen Datensatz zugreifen zu können. Der Nachteil ist, dass durch diesen Auflösungsmechanismus der Zugriff auf Felder mit Referenz- bzw. Annotations-Typ zusätzlichen Aufwand erfordert (Auflösen der Referenz, Erzeugen eines neuen Objekts) und daher langsamer ist als zuvor. Außerdem ist es nötig, in solchen Feldern die Referenzen anzupassen, wenn die referenzierten Objekte umgeordnet

wurden. Entsprechendes gilt für zusammengesetzte Typen mit Referenz- und Annotations-Typen als Inhalten.

Diese Verbesserungen bezüglich des Speichers wurden in den Codegenerator der Scala-Anbindung eingearbeitet. Dabei wurde im Vergleich zur ursprünglichen Implementierung die Behandlung von als gelöscht markierten Objekten verbessert. Allerdings wurde die Korrektur von Referenzen und Annotationen in unbekannten Felder noch nicht implementiert. Daher ist in der optimierten Implementierung die Behandlung unbekannter Felder nicht korrekt, falls dort Referenzen oder Annotationen korrigiert werden müssen. Das schränkt die Benutzung der optimierten Implementierung nur dann ein, wenn viel mit unbekannten Feldern gearbeitet wird. Für übliche Anwendungen ist dies aber eher irrelevant.

Wie beim Vergleichen der beiden Implementierungen festgestellt wurde, führt die reduzierte Speicherlast dazu, dass nur wenige teure große Garbage Collections durchgeführt werden, vor allem während der Schreib- und Anhängphasen. Zusätzlich führt der verbesserte Ausgabepuffer durch die Reduktion der Anzahl der gespeicherten Byte-Arrays nicht nur zu einer geringeren Speicherauslastung, sondern auch zu einem schnelleren Schreibvorgang, da größere Blöcke geschrieben werden können. Insgesamt sind daher in der optimierten Implementierung die Schreib- und Anhängphasen deutlich schneller als in der ursprünglichen Implementierung. Ebenso wird durch die Speicherung der Felder analog zum Dateiaufbau und dadurch, dass nur wenige Objekte erzeugt werden müssen, der Lesevorgang wesentlich beschleunigt.

8.1. Ausblick

Weitere Optimierungen können im Bereich der verwendeten Container für zusammengesetzte Typen vorgenommen werden. Da die Implementierung bis auf den Typ `String` ausschließlich primitive Typen zur Speicherung der Daten verwendet, bietet es sich an, statt der nicht-spezialisierten Container aus der Scala-Standardbibliothek spezialisierte Container zu verwenden. Durch diesen Schritt wird vermieden, dass primitive Typen in ein Objekt geschachtelt werden, da nicht-spezialisierte Container wegen der Typlöschung nur auf Objekten arbeiten. Die Standardbibliothek enthält zurzeit für keinen der verwendeten Container spezialisierte Varianten, diese müssen alle selbst geschrieben werden. Der Nutzen ist allerdings groß, denn für einen einzigen in einem solchen Container gespeicherten Wert entfallen 24 Byte, 8 Byte für eine Referenz, 16 Byte für den Grundbedarf eines Objekts. Unter Missachtung der Ausrichtung von Werten macht das bei einem `Long` 75% und bei einem `Byte` sogar 96% des ursprünglichen Verbrauchs aus.

Um die vollständig korrekte Behandlung von unbekannten Feldern in der optimierten Implementierung zu ermöglichen, ist es ratsam, deren Speicherung zu ändern. Die Daten unbekannter Felder werden zurzeit in einer `HashMap[FieldDeclaration, HashMap[Int, Any]]` gespeichert. Unter Umständen wäre es besser, die innere `HashMap` durch korrekt typisierte Arrays zu ersetzen, d. h. den Typ `HashMap[FieldDeclaration, Array[_]]` zu verwenden, und unbekannte Felder dann wie bekannte Felder zu behandeln. Da die Felddeklarationen der unbekannten Felder für jeden Typ in einer Liste im assoziierten Speicherpool gespeichert sind, besitzen sie einen Index. Daher kann sogar noch weitergegangen werden und ein `Array[Array[_]]` verwendet werden. Die Aktualisierung unbekannter Felder kann dann analog zur Aktualisierung bekannter Felder durchgeführt werden.

Eventuell bietet es sich an, spezialisierte Speicherpools zu erstellen, wie zum Beispiel für bekannte Basistypen ohne bekannte Untertypen, Singletons oder monotone Typen, d. h. Typen, von denen Instanzen erzeugt, aber nicht mehr gelöscht werden können. So kann für bekannte Basistypen ohne bekannte Untertypen die Verwaltung neu erzeugter Datensätze vereinfacht werden, da neue Instanzen unbekannter Typen nicht erzeugt werden können und somit alle neuen Datensätze auch ohne zusätzlichen Verwaltungsaufwand eindeutig zugeordnet werden können. Singletons, die eigene Felder definieren, müssen deren Daten nicht in Arrays speichern, da ja nur eine Instanz existiert. Zudem kann hier das einzige Objekt tatsächlich gespeichert werden und alle eigenen Felder direkt enthalten. Für monotone Typen kann der gesamte Mechanismus zum Löschen von Objekten und die zugehörigen Datenstrukturen entfallen.

A. Anhang

A.1. Testframework

Dieser Abschnitt beschreibt die Implementierung einiger Teile des Testframeworks und enthält Hinweise zur Implementierung eigener Ergebnisse und Aktionen.

A.1.1. Ergebnisse

Ergebnisse implementieren das versiegelte, d. h. als **`sealed`** gekennzeichnete, Trait `Result`. Dazu müssen sie von einem der Traits `SingleValueResult` (für einen Messwert pro Parameter) oder `MultiValueResult` (mehrere Messwerte pro Parameter) erben. Diese haben folgende Definitionen:

Listing A.1: Result-Schnittstellen

```
package common.storage

sealed trait Result {
  val name: String

  def +=(n: Int, s: Double): Unit
}

trait SingleValueResult extends Result {
  val storage: HashMap[Int, Double]
}

trait MultiValueResult {
  val storage: HashMap[Int, ArrayBuffer[Double]]
}
```

Die enthaltenen `HashMap`-Objekte speichern die Daten des Ergebnisses. Neue Daten werden mit der Methode `+=` hinzugefügt. `n` ist dabei der Parameter, `s` der Messwert. Die Verarbeitung mehrerer Messwerte zu einem Wert für implementierende Klassen des `SingleValueResult`-Traits ist vollständig dem Implementierer überlassen. `n` ist üblicherweise der Parameter eines Tests. `name` ist der Name des Ergebnisses, der von den `saveGraph`-Methoden als Beschriftung der entsprechenden Datenlinie verwendet wird.

Innerhalb des Testframeworks wird die Methode `+=` nur von der Klasse `ValueReporter` aufgerufen, die im nächsten Abschnitt beschrieben wird.

A.1.2. Drucker und die Klasse `ValueReporter`

Drucker steuern die prozessinterne Speichermessung und die Ausgabe der Messwerte. Es gibt genau drei verschiedene Drucker, zwei für die Verwendung nur eines Prozesses für Messung und Ausführung (siehe auch Abschnitt 6.2), einen für die externe Speichermessung:

`ConsolePrint` gibt die maximale Heapkapazität und den maximalen Speicherverbrauch innerhalb des Messzeitraums auf der Konsole aus. Die Ausgabe erfolgt in tabellarischer Form, einzelne Spalten sind durch Tab-Zeichen getrennt. In der ersten Spalte steht der Parameter der aktuellen Ausführung (n), in der zweiten Spalte die maximale Heapkapazität in Byte, in der dritten der maximale Speicherverbrauch, ebenfalls in Byte.

`ResultPrint` gibt die maximale Heapkapazität und den maximalen Speicherverbrauch innerhalb des Messzeitraums in Ergebnisse aus, die zu Beginn der Messung gesetzt werden. Die Ergebnisse sind optional. Wird eines der Ergebnisse oder sogar beide nicht gesetzt, werden die entsprechenden Messwerte ignoriert.

`StartStopPrint` wird für die externe Speichermessung verwendet. Dieser Printer gibt lediglich „`start`“ bei Beginn der Messung und „`finished`“ bei Ende der Messung auf der Konsole aus. Nach einer Ausgabe wartet er auf ein Signal zum Fortsetzen per Standardeingabe. Das interagierende Gegenstück zu diesem Drucker ist die `runProcess`-Methode der `StorageTestBase`-Klasse.

Die Methoden der Drucker-Klassen können nur von der Klasse `TaskBase` verwendet werden. Aktionen rufen sie indirekt über die `Task`-Instanz auf, die sie als Parameter bekommen. `Task` und `TaskBase` werden im nächsten Abschnitt genauer beschrieben.

Die eigentliche Speichermessung wird durch die `ValueReporter`-Klasse durchgeführt. Instanzen dieser Klasse registrieren sich selbst als Beobachter bei der zu überwachenden JVM. Dazu bekommt der Konstruktor die Prozess-ID der Ziel-JVM als Parameter. Das folgende Listing enthält alle wesentlichen Felder und Methoden dieser Klasse. Dabei wurden Hilfsdaten und -methoden weggelassen.

Listing A.2: Jvmstat-Schnittstelle

```
package common.storage

import sun.jvmstat.monitor.event._

class ValueReporter(pid: String, var n: Int) extends VmListener {
  private var capacityResult: Option[Result]
  private var usedResult: Option[Result]
  def switchResults(capacity: Option[Result], used: Option[Result]): Unit

  ...
  override def monitorStatusChanged(event: MonitorStatusChangeEvent): Unit
  override def monitorsUpdated(event: VmEvent): Unit
  override def disconnected(event: VmEvent): Unit
}
```

`capacityResult` nimmt die Messwerte der aktuell verfügbaren Speicherkapazität auf, `usedResult` die des aktuellen Speicherverbrauchs. `switchResults` tauscht beide Ergebnisse durch die übergebenen

Parameter aus. Die Ergebnisse sind optional: Werden Messwerte, z. B. der Kapazität, nicht benötigt, kann das entsprechende Ergebnis einfach auf `None` gesetzt werden.

Die letzten drei Methoden implementieren die `VmListener`-Schnittstelle [Jvm]. Dabei wird die Methode `monitorStatusChanged` nicht verwendet, d. h. die Implementierung ist leer. Diese Methode wird aufgerufen, wenn sich der Zustand der Überwachungsmonitore geändert hat. `monitorsUpdated` fügt neue Messwerte zu den Ergebnissen hinzu. Diese Methode wird aufgerufen, wenn sich die Werte der Überwachungsmonitore geändert haben. Bricht die Verbindung zur überwachten JVM ab, wird `disconnected` aufgerufen, welches die Datenaufzeichnung beendet. Danach kann die betroffene Instanz der `ValueReporter`-Klasse nicht mehr verwendet werden. Die Monitore werden alle 10 ms abgefragt, um eine relativ dichte Messwertmenge zu erhalten, ohne den Fortschritt des Programms unnötig zu behindern.

A.1.3. Aktionen und Testabläufe

Das folgende Listing zeigt die Definitionen der Traits `Action` sowie die beiden Spezialisierungen davon, `TypedAction` für typisierte Aktionen und `UntypedAction` für typlose Aktionen.

Listing A.3: Action-Schnittstellen

```
package common.storage

sealed trait Action[+StateType] {
  def apply(t: TaskBase, n: Int, f: Path): Unit

  def +>(b: UntypedAction): Action[StateType]
  def +>[StateType2 >: StateType](b: TypedAction[StateType2]): Action[StateType2]

  def name: String

  def results: Iterator[Option[Result]]
}
object Action {
  def fold[StateType](seq: Iterable[Action[StateType]]): Action[StateType]
}

trait TypedAction[StateType] extends Action[StateType] {
  final override def apply(t: TaskBase, n: Int, f: Path) =
    apply(t.asInstanceOf[Task[StateType]], n, f)
  def apply(t: Task[StateType], n: Int, f: Path)

  final override def +>(b: UntypedAction): TypedAction[StateType]
  final override def +>[StateType2 >: StateType](b: TypedAction[StateType2]):
    TypedAction[StateType2]
}

trait UntypedAction extends Action[Nothing] {
  override def +>(b: UntypedAction): UntypedAction
  override def +>[StateType](b: TypedAction[StateType]): TypedAction[StateType]
}
```

Die wichtigste Methode einer Aktion ist `apply`. Diese führt die Aktion für den gegebenen Testablauf `t`, den Durchlaufparameter `n` und die für Dateiein- und -ausgaben zu verwendende SKilL-Datei `f` aus. Die einzige von `TaskBase` erbende Klasse ist `Task`, welche garantiert, dass für eine zugehörige Aktion der Typparameter übereinstimmt. Daher können typisierte Aktionen auf den gesamten Testablauf zugreifen, insbesondere den enthaltenen SKilL-Zustand, typlose dagegen nur auf `TaskBase`. Gäbe es für alle `SkillState`-Traits ein gemeinsames Basis-Trait, könnte die `TaskBase`-Klasse komplett eingespart werden.

Sollen Aktionen in einem externen Prozess ausgeführt werden, werden sie durch ihre `name`-Methode serialisiert. Die entstehende Zeichenfolge wird an den externen Prozess übergeben und dort mithilfe der Methode `stringToAction` in `StorageTestBase` wieder zu Aktionen deserialisiert. Möchte man eigene Aktionen zur Ausführung in einem externen Prozess definieren, müssen diesen eindeutige Namen ohne Leerzeichen zugewiesen werden und die Methode `stringToAction` in der eigenen Testklasse überschrieben werden. `results` ist ein Iterator über alle Ergebnisse, die von einer Aktion verwendet werden, d. h. in die die Aktion schreibt. Diese Methode wird ebenfalls für die externe Ausführung verwendet.

Die `Task`-Klasse für Testabläufe ist wie folgt definiert:

Listing A.4: Testablauf-Klassen

```
package common.storage

sealed abstract class TaskBase(val printer: TaskBase.Printer) {
  def reset: Unit

  final def startMeasuring(cap: Option[Result], used: Option[Result])
  final def stopMeasuring()

  protected final def setCount(n: Int)
}
object TaskBase {
  sealed trait Printer {
    protected[TaskBase] def print(): Unit
    protected[TaskBase] def setCount(n: Int): Unit
    protected[TaskBase] def setResults(cap: Option[Result], used: Option[Result]): Unit
  }

  class ConsolePrint extends Printer { ... }
  class ResultPrint extends Printer { ... }
  object StartStopPrint extends Printer { ... }
}

final class Task[StateType](printMemory: TaskBase.Printer, private val action: Action[StateType])
extends TaskBase(printMemory) {
  var state: Option[StateType]

  override def reset: Unit

  def apply(n: Int, f: Path)
}
```

`TaskBase` ist die Basisklasse aller Testabläufe und bietet eine vom SKilL-Zustand unabhängige Schnittstelle für typlose Aktionen. Dabei ist auch die Verwaltung der Ausgabe über die verschiedenen Drucker enthalten. Deren Methoden können nicht direkt aufgerufen werden. Stattdessen wird die Ausgabe mit den Methoden `startMeasuring` (Beginn der Speichermessung), `stopMeasuring` (Beenden der Speichermessung) und `setCount` (Festlegen des Durchlaufparameters) gesteuert. `reset` erlaubt das Zurücksetzen des Zustands `state` auf den Wert `scala.None`, was auch dessen Initialwert ist.

Die `Task`-Klasse ergänzt lediglich den (optionalen) SKilL-Zustand `state` und die Methode `apply`, die die an den Konstruktor übergebene Aktion `action` ausführt. Genauer setzt sie den Durchlaufparameter `n` im zugehörigen Drucker mittels `setCount` und ruft anschließend die `apply`-Methode der Aktion auf.

Glossar

Annotation Eine Referenz, die auf einen beliebigen Benutzertyp verweisen kann. In der ursprünglichen Implementierung einfach eine `SkillType`-Referenz, in einer SKilL-Datei und der optimierten Implementierung eine reiche Referenz, bestehend aus dem Typ und der SKilL-ID des Ziels. 27–30, 36, 37, 57, 58, 69, 70

Benutzertyp oder kurz Typ. Nach [Fel13] ein in einer SKilL-Spezifikation deklarierter Typ. In dieser Arbeit wird der Begriff weitergehend auch für Typen verwendet, die aus einer SKilL-Datei gelesen wurden. 17–19, 23, 24, 27–30, 32, 79, 80

Generation Auf der JVM ist der Heap in mehrere sogenannte Generationen aufgeteilt. Die verwendete JVM teilt den Heap in eine junge Generation, eine alte Generation und eine permanente Generation. Die junge Generation enthält neu erzeugte Objekte und hat eine relativ kleine Größe, die alte enthält Objekte, die einige Garbage Collections in der jungen Generation überlebt haben sowie neu erzeugte Objekte, die zu groß für die junge Generation sind [GCT]. 20, 21, 40

Scala-Anbindung Die SKilL-Anbindung für die Programmiersprache Scala. 7–10, 18, 19, 23, 39, 69, 70

SKilL-Anbindung Ein Generator für SKilL-Spezifikationen, der Code für eine bestimmte Programmiersprache erzeugt. Dieser muss mindestens die in [Fel13] definierte Kernsprache unterstützen. Oft wird mit SKilL-Anbindung auch der generierte Code bezeichnet. Ist nicht aus dem Kontext ersichtlich, um welche Bedeutung es sich handelt, ist diese explizit angegeben. 7, 9, 10, 17, 18, 49, 79

SKilL-Datei Die serialisierte Form von Daten nach [Fel13], die von einer generierten SKilL-Anbindung geschrieben und gelesen werden. Es handelt sich hierbei um ein binäres Dateiformat, meist mit der Dateiendung `.sf`. 7, 9, 17, 19, 20, 23, 24, 26, 27, 33, 35, 38, 46, 47, 69, 70, 76, 79, 80

SKilL-ID Eine Zahl, die ein Objekt bzw. einen Datensatz innerhalb eines Basispools eindeutig identifiziert. Dabei werden zwei Typen unterschieden: SKilL-IDs in der binären SKilL-Datei und SKilL-IDs in der generierten SKilL-Anbindung. 23, 25–28, 31–37, 69, 79

SKilL-Spezifikation Eine der in [Fel13] beschriebenen Grammatik entsprechende Datei, meistens mit der Dateiendung `.skill`. 7, 17, 18, 23, 24, 26, 27, 29, 48, 49, 51, 69, 79

SKilL-Zustand Repräsentation einer SKilL-Datei im Code. Bietet Möglichkeiten, auf gelesene Daten zuzugreifen, neue Objekte im Zustand zu erzeugen und einen Zustand in eine SKilL-Datei zu serialisieren. 19, 23–26, 28, 30, 35, 43–47, 50, 76, 77

Typlöschung (type erasure) bedeutet, dass beim Übersetzen eines generischen Typs alle generischen Parameter durch konkrete Klassen ersetzt werden. Einerseits führt das dazu, dass eine generische Klasse wie eine nicht-generische Klasse nur in einer einzigen Variante kompiliert wird. Andererseits stehen keine Informationen über die eingesetzten Typparameter bei der Verwendung zur Verfügung. [O$^+$, § 3.7] [Ull11]. 29, 30, 38, 67, 70

unbekannter Typ Die Teilmenge der Benutzertypen, die ausschließlich aus einer SKilL-Datei gelesen wurden, d. h. für die kein spezialisierter Code generiert wurde. 17–19, 25, 71

zusammengesetzter Typ Einer der vordefinierten SKilL-Typen $X[n]$, $X[]$, ***list***<X>, ***set***<X> und ***map***<X, Y, …>, wobei X und Y vordefinierte, nicht zusammengesetzte SKilL-Typen oder Benutzertypen sind, n eine natürliche Zahl ist und ***map*** mindestens zwei Typargumente enthält [Fel13]. 17, 18, 28–31, 37, 38, 51, 57, 66, 67, 70

Abkürzungsverzeichnis

JVM Java Virtual Machine. 7, 8, 10, 11, 20, 25, 28, 30, 38–40, 44, 46, 51, 54, 69, 74, 75

SKilL Serialization Killer Language. 3, 7, 8, 10, 11, 17, 18, 29, 66

Literaturverzeichnis

[Fel13] T. Felden. The SKilL Language. Technischer Bericht, Universität Stuttgart, 2013. (Zitiert auf den Seiten 7, 11, 17, 18, 19, 20, 23, 24, 25, 27, 28, 29, 33, 79 und 80)

[Fel14] T. Felden. Efficient and Change-Tolerant Serialization for Program Analysis Tool-Chains. In *16. Workshop Software-Reengineering und -Evolution*. Bad Honnef, Deutschland, 2014. (Zitiert auf Seite 9)

[Fel15] T. Felden. javaCommon – Common implementation to all Java targets, 2015. Informationen aus unveröffentlichter privater Korrespondenz. (Zitiert auf Seite 38)

[Feu14] C. Feuersänger. pgfplots – Create normal/logarithmic plots in two and three dimensions, 2014. URL https://www.ctan.org/pkg/pgfplots. (Zitiert auf den Seiten 41 und 42)

[GCT] Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning. URL http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html. (Zitiert auf den Seiten 20, 21, 40 und 79)

[Har14] F. Harth. *Plattform- und sprachunabhängige Serialisierung mit SKilL*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2014. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3665&engl=. (Zitiert auf den Seiten 7 und 10)

[IEE08] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, S. 1–70, 2008. doi:10.1109/IEEESTD.2008.4610935. URL http://ieeexplore.ieee.org/servlet/opac?punumber=4610933. (Zitiert auf Seite 17)

[Jav] Oracle Java SE Documentation – java. URL http://docs.oracle.com/javase/7/docs/technotes/tools/windows/java.html. (Zitiert auf den Seiten 51 und 54)

[Jvm] *Jvmstat*. URL http://openjdk.java.net/groups/serviceability/jvmstat/. (Zitiert auf den Seiten 39 und 75)

[KW08] T. Kotzmann, C. Wimmer. Synchronization and Object Locking, 2008. URL https://wikis.oracle.com/display/HotSpotInternals/Synchronization. (Zitiert auf Seite 7)

[O$^+$] M. Odersky, et al. *Scala Language Specification – Version 2.11*. URL http://www.scala-lang.org/files/archive/spec/2.11/. (Zitiert auf den Seiten 7, 11, 12, 13, 14, 15, 16, 23 und 80)

[Ora14] Oracle. *Java™ Virtual Machine Technology*, 2014. URL http://docs.oracle.com/javase/7/docs/technotes/guides/vm/index.html. (Zitiert auf den Seiten 7 und 39)

[Prz14] D. Przytarski. *Performance-Evaluation einer sprach- und plattformunabhängigen Serialisierungssprache*. Bachelorarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2014. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=BCLR-0106&engl=. (Zitiert auf den Seiten 7 und 9)

[Sca] ScalaTest. URL http://www.scalatest.org. (Zitiert auf Seite 45)

[SKi14a] Cross platform, cross language, easy-to-use serialization interface generator, 2014. URL https://github.com/skill-lang/skill. (Zitiert auf den Seiten 7, 9, 23 und 51)

[SKi14b] The SKilL scala generator test suite, 2014. URL https://github.com/skill-lang/skillScalaTestSuite. (Zitiert auf den Seiten 45 und 66)

[Ull11] C. Ullenboom. *Java ist auch eine Insel – Das umfassende Handbuch*, Kapitel 9. Rheinwerk Verlag GmbH, 10. Auflage, 2011. URL http://openbook.rheinwerk-verlag.de/javainsel/javainsel_09_002.html. (Zitiert auf den Seiten 30 und 80)

[Ung14] W. Ungur. *Nutzbarkeitsevaluation einer sprach- und plattformunabhängigen Serialisierungssprache*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2014. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3603&engl=. (Zitiert auf den Seiten 7 und 10)

[Vis14] VisualVM 1.3.8 – All-in-One Java Troubleshooting Tool, 2014. URL http://visualvm.java.net. (Zitiert auf den Seiten 25 und 39)

Alle URLs wurden zuletzt am 25. 05. 2015 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift