Bachelorarbeit zur Erlangung des akademischen Grades
Bachelor of Science
Informatik

Prototypische Implementierung eines BETA nach Java Übersetzers

vorgelegt von

Inhaltsverzeichnis

1 Einleitung

Bei Anwendungen, die über einen Zeitraum von vielen Jahren hinweg verwendet werden, kommt es immer wieder vor, dass sie auf neuere Systeme portiert werden müssen. Ein solcher Fall gab den Anstoß für diese Arbeit. Das Content Management System (CMS), das die zentralen Aspekte der Internetpräsenz verwaltet, wurde in der Programmiersprache BETA[1] erstellt. Diese Sprache existiert inzwischen faktisch nicht mehr. Der einzige Anbieter hat die aktive Entwicklung der zugehörigen Plattform im Jahre 2002[2] eingestellt. Das CMS, der so genannte Trans[3], wird stetig an neue Anforderungen angepasst und soll noch weitere Jahre im Einsatz bleiben. Da jedoch keine Entwicklung der Sprache und ihres Umfeldes mehr stattfindet, stehen keine aktuellen Bibliotheken für die Verwendung neuer Technologien zur Verfügung; dadurch wird die Weiterentwicklung erschwert. Hinzukommt die Tatsache, dass der Compiler einige Fehler enthält und deswegen unter anderem nicht mehr ohne weiteres auf neuen 64-Bit-Linux Systemen eingesetzt werden kann. Deshalb wurde der Entschluss gefasst, den Trans in eine aktivere Sprache zu überführen.
Als neue Sprache wurde Java ausgewählt. Neben den allgemeinen Vorteilen von Java, wie der Plattformunabhängigkeit und den umfassenden Bibliotheken, ist dies durch die Ähnlichkeit beider Sprachen in zentralen Punkten begründet:

1. Beide sind rein objektorientiert.
2. Beide erlauben die Verwendung von verschachtelten und lokalen Klassen (bzw. Patterns wie sie in BETA genannt werden).
3. Beide verwenden einen Garbage Collector.
4. Beide sind streng typisiert und kennen keine Zeiger.

Da der Trans sehr umfangreich ist, sollte diese Konvertierung möglichst automatisch erfolgen. Dadurch soll zum einem die finale Umstellung schnell durchführbar sein und zum anderem sollen kleine (Tipp-) Fehler, die bei einer manuellen Übersetzung wahrscheinlich sind, vermieden werden. Das Erstellen eines Übersetzers, der diese Aufgabe übernehmen kann, ist das Ziel dieser Arbeit.

1.1 Ziele

Für den zu erstellenden Übersetzer erhält die Bezeichnung *„BETA2Java“*.
Es werden folgende Ziele für ihn definiert:

1. BETA-Code soll in Java-Code übersetzt werden, dabei soll die Übersetzung nur soweit gehen, wie es für den Trans nötig ist.
2. Die Übersetzung soll möglichst automatisch erfolgen. Es ist jedoch zulässig, dass spezielle und seltene Fälle, die nicht oder nur mit großen Aufwand übersetzbar sind, gesondert behandelt werden. Das heißt, dass sie entweder im erzeugten Code manuell nachbearbeitet oder dem Übersetzer zusätzliche Informationen gegeben werden.
3. Der erzeugte Java-Code ist so zu gestalten, dass er als Basis für die weitere Entwicklung des Programms dienen kann. Er soll für einen Programmierer verständlich sein und sich nach Möglichkeit an der üblichen Java-Programmierung orientieren. Es ist jedoch nicht nötig, dass der erzeugte Code „perfekt“ ist. Es ist angedacht, dass der Code im Laufe der Weiterentwicklung immer mehr angepasst wird und er sich mit der Zeit immer mehr von seinem BETA-Ursprung entfernt.

1 http://www.daimi.au.dk/~beta/
2 vgl. [MI]
3 Tatsächlich besteht das gesamte CMS noch aus weiteren Teilen, in diesem Kontext ist aber nur der Trans von Bedeutung.

4. Das erzeugte Java-Programm soll eine ähnliche Performance erzielen wie das bestehende BETA-Programm. Wie in 3. schon angesprochen gilt auch hierbei, dass eine Verbesserung der Performance im Laufe der Zeit zu erwarten ist, denn beim Weiterentwickeln können immer mehr Funktionen aus den BETA-Bibliotheken durch passende Teile der Java-API ersetzt werden. Im Idealfall sind nach einiger Zeit die BETA-Bibliotheken nicht mehr nötig.

Aus dem oben Aufgeführten folgen einige Punkte, die nicht Ziel dieser Arbeit sind:
- Ein vollständiger und leicht zu verwendender Übersetzer. Wie schon festgestellt soll nur eine Übersetzung des Trans erfolgen. Deshalb werden auch nur die vom Trans benötigten BETA-Bibliotheken behandelt.
- Umfassende Tests oder Beweise, dass die verwendeten Übersetzungen korrekt sind. Es genügt, wenn sich die übersetze Version des Trans in verschiedenen Situationen so verhält wie das Original.
- Eine hohe Übersetzungsgeschwindigkeit. Es ist für die vollständige Übersetzung des Trans durchaus akzeptabel, wenn der Übersetzungsvorgang mehrere Minuten benötigt.

1.2 Bestehende Ansätze

Die Recherche ergab, dass drei ähnliche Aufgabenstellung bearbeitende Projekte bereits existieren:
- Beta2Java ([LK00]) behandelt die Übersetzung von einfachem BETA-Code direkt in Java-Bytecode zur Ausführung mittels der JVM. Es ist lediglich die Dokumentation verfügbar und der übersetzbare BETA-Code ist stark eingeschränkt.
- The Loki Project ([Co]) hat die Übersetzung in Java-Code als Ziel; allerdings sind nur einige Entwurfsdokumente vorhanden.
- Object-Oriented Language Interoperability ([OOLI]) ist das am meisten fortgeschrittene Projekt. Ziel ist es, BETA und Java (bzw. BETA und .NET) nebeneinander zu verwenden, so dass aus Java heraus BETA-Patterns und aus BETA heraus Java-Klassen aufgerufen werden können.
  Um dies zu erreichen, wird der BETA-Code in Java-Bytecode übersetzt. Hierbei ist das Projekt fortgeschritten und ein entsprechender Übersetzer ist verfügbar, jedoch ohne Quellcodes oder exakte Dokumentationen.

Alle drei Projekte sind seit langem nicht mehr aktiv, somit sind zukünftig keine neuen Entwicklungen mehr zu erwarten. Keines der gefundenen Projekte erfüllt die gewünschten Anforderungen und kann verwendet werden. Trotzdem können einige Ideen als Ausgangspunkt genutzt werden.

2 Kurze BETA Einführung

Dieses Kapitel soll einen kurzen Überblick über die wichtigsten Sprachelemente von BETA geben. Eine komplette Einführung findet sich in der Beschreibung der Entwicklungsumgebung „The Mjølner System“ ([MI04d]) und die Sprach Referenz im „BETA Book“ ([MMN93]).

Das „Hello World“-Programm in BETA:

```
1 ORIGIN '~beta/basiclib/betaenv'
2 ---- program: descriptor ----
3 (* Dies ist ein Kommentar. *)
4 (#
5   do
6      'Hello World!' -> putline;
7 #)
```

Die ersten beiden Zeilen gehören zum Fragmentsystem. Es dient zur Verteilung des Codes auf mehrere Dateien, dem Einbinden von Bibliotheken und so weiter. Es ist nicht Bestandteil von BETA selbst und wird deshalb hier übergangen. Genauere Erläuterungen dazu finden sich im BETA Book ab Seite 255.

Grundelement von BETA ist das Pattern. Ein Pattern kombiniert verschiedene Konzepte. So kann es in Java einer Klasse, einer Funktion oder auch einer Konstante entsprechen. Das obige Beispiel stellt ein solches Pattern dar. Dabei handelt es sich in diesem Fall um ein anonymes Pattern. Es beginnt in Zeile 4 und endet in Zeile 7. Da ein Pattern eine Funktion darstellen kann, kann es auch ausgeführt werden. Alle auszuführenden Anweisungen folgen auf das Schlüsselwort `do`. Mit der Anweisung in Zeile 6 wird ein Pattern namens `putline` ausgeführt, wobei `'Hello World'` als Argument übergeben wird.
Alle Anweisungen werden in BETA von links nach rechts gelesen, der Pfeil `->` ist der Zuweisungsoperator. Jede Zuweisung ist zunächst ein Aufruf eines Objektes. Erweitern man das Beispiel um eine dynamische Referenzvariable (d.h. eine Variable, die zur Laufzeit auf verschiedene Objekte verweisen kann), so ergibt sich:

```
1 (#
2   t: ^text;
3   do
4      'Hello World!' -> T[]; (* BETA unterscheidet nicht t und T *)
5      t[] -> &putline;
6 #)
```

Um den Text in der Variablen `t` ablegen zu können, muss `[]` angehängt werden, ohne `[]` würde BETA das Objekt `t` ausführen. Da `t` an dieser Stelle noch nicht belegt ist, also nur den Wert `NONE` enthält, würde das Programm abstürzen.
Im Unterschied zur ersten Version wird `putline` mit einem `&` aufgerufen. Das `&` entspricht dem `new` Operator in Java. Er ist auch im ersten Beispiel enthalten, allerdings implizit; denn bei jedem direkten Aufrufen eines Patterns wird immer zuerst eine Instanz dieses Patterns erzeugt. Diese Instanz wird dann ausgeführt und anschließend wieder freigegeben. Weil BETA nicht zwischen Patterns, die als Klassen und Patterns, die als Funktionen agieren, unterscheidet, wird bei jeden Funktionsaufruf ein Objekt erzeugt. Um das zu verhindern, kann eine Instanz von `putline` angelegt werden:

```
1 (#
2   t: ^text;
3   put: @putline;
4   do
5      'Hello World!' -> T[];
6      t[] -> put; (* aufrufen der Instanz *)
7 #)
```

Die Variable `put` (Zeile 3) ist im Unterschied zu `t` mit einem `@` statt einem `^` deklariert, dies unterscheidet die dynamische Referenz von der statischen Referenz. Eine statische Referenz verweist immer genau auf ein Objekt, das beim Anlegen der Variable erzeugt wird.

Ein Pattern kann in BETA auch weitere Patterns enthalten und von einem anderen Pattern erben:

```
1 (#
2   put: @putline;
3
4   printText: (#
5      t: ^text;
6      enter t[]
7      do
8         t[] -> put;
9         inner;
10     exit t[]
11  #);
12
13  printMore: printText(#
14     t2: ^text;
15     enter t2[]
16     do
17        t2[] -> put;
18     exit t2[]
19  #)
20
21  do
22     ('Hello World!', 'How are you?') -> printMore ->
23        (printText, printText);
24#)
```

`PrintText` (Zeile 4) kann mit einem Parameter aufgerufen werden (Zeile 6). Der angegebene Text wird ausgegeben und anschließend wieder zurückgegeben (Zeile 10). Diese Funktionalität wird durch `printMore` (Zeile 13) erweitert, so dass nun 2 Parameter angegeben, ausgegeben und anschließend zurückgegeben werden können. In BETA ist es beim Erben immer nur möglich, das Vater-Pattern zu erweitern. Eigenschaften des Vaters können nicht überschrieben werden. Das Vater-Pattern bestimmt außerdem wann, ob und wie oft Anweisungen von sich ableitenden Patterns ausgeführt werden. Dazu dient die Anweisung in Zeile 9. Wird `printMore` aufgerufen beginnt die Ausführung nicht im Anweisungsteil von `printMore` (Zeile 16), sondern immer im Vater-Pattern, also in Zeile 7. Die Anweisungen von `printMore` werden erst mit dem Aufruf von `inner;` ausgeführt.
Zum Übergeben und Entnehmen von mehr als einem Wert verwendet BETA Listen, wie in Zeile 22 zu sehen ist. Da `printMore` zwei Rückgabewerte hat, können damit in einer Liste auch durchaus zwei Patterns mit je einem Parameter aufgerufen werden (Zeile 23).

2.1 Kontrollstrukturen

If

Die allgemeine Form des If's ähnelt einem Switch in Java: Es wird zunächst ein Anfangswert ermittelt. Anschließend werden die Ausdrücke von oben nach unten ausgewertet und mit dem Anfangswert verglichen. Bei der ersten Übereinstimmung (d.h. Gleichheit) wird die Auswertung abgebrochen und die Anweisungen des zugehörigen Then-Zweig werden ausgeführt.

```
(if <Start Ausdruck>
     // <Ausdruck 1> then <Anweisungen 1>
     // <Ausdruck 2A>
     // <Ausdruck 2B>
     ...
     // <Ausdruck 2X> then <Anweisungen 2>
     ...
     // <Ausdruck N> then <Anweisungen N>
     else <Anweisungen sonst>
if)
```

Alternativ kann das If auch auf die in Java übliche Form abgekürzt werden:

```
(if <Ausdruck> then <Anweisungen>
     else <Anweisungen>
if)
```

For

Die For-Schleife ist in BETA nur sehr begrenzt verfügbar. Sie ermöglicht es lediglich, von 1 bis zu einem berechneten Wert zu zählen:

```
(for <Name>:<Ausdruck> repeat
     <Anweisungen>
for)
```

Label

BETA bietet mit Labels die Möglichkeit, Blöcke zu verlassen oder zu wiederholen. Dabei kann ein Block ein Pattern, eine If-Anweisung oder eine For-Schleife sein:

```
<Name>: (#
     do
        ....
        <Name 2>: (if something then
           leave <Name>; (* verlässt den Block *)
        else
           restart <Name 2> (* Schleife über die If-Bedingung *)
        if);
        ....
#);
```

2.2 Vergleich mit Java

BETA hat Gemeinsamkeiten mit Java. Dazu gehören: strikte Objektorientierung (mit Ausnahme der primitiven Datentypen), Garbage Collection, verschachtelte Klassen und mehr. Dennoch sind beide Sprachen auch sehr unterschiedlich. Der Sprachumfang von BETA ist wesentlich kleiner als der von

Java. Viele Konstrukte, die in Java direkt als Syntaxelemente integriert sind, werden in BETA durch die Bibliothek realisiert. Die nachfolgende Tabelle gibt einen stark verkürzten Überblick darüber, welche Java-Elemente sich in BETA wiederfinden.

Java	BETA
Klassen, lokale Klassen, anonyme Klassen	Patterns
Methoden (final und static)	
Konstanten	
Methoden (mit später Bindung)	virtuelle Patterns
Generics	
Konstruktoren	nicht vorhanden
If	If
Switch	
For-Schleife	For (eingeschränkt)
Foreach-Schleife	Teil der Bibliothek
Do-, Do-while-Schleife	Teil der Bibliothek
Exception Handling	Teil der Bibliothek
Interfaces, Abstrakte Klassen/Methoden	nicht vorhanden
Cast	nicht nötig/nicht vorhanden

2.3 Begriffsdefinitionen

Die im Folgenden erläuterten Begriffe wurden im Verlaufe des Projekts eingeführt, um bestimmte Sachverhalte beschreiben zu können. Sie werden in der Beschreibung der Übersetzung und in der Implementierung verwendet.

Umgebendes Pattern: Gemeint ist immer das Pattern in dem das jeweilige Element enthalten ist. Die Bezeichnung wird analog für Klassen verwendet. z.B.:

```
a: (#
     ...
     p: (#
        ...
     #);
#);
```

In diesem Fall ist `a` das umgebende Pattern von `p`.

Inline-Pattern: Gemeint ist ein Pattern, das sich innerhalb eines Anweisungsblockes befindet. Die Bezeichnung wird analog für Klassen verwendet. z.B.:

```
a: (#
   do
      ...
```

```
        b(#
           ...
        #);
    ...
#);
```

In diesem Fall ist `b` ein Inline-Pattern.

Lokale Variable: Jede Variable, die zum aktuellen Pattern/Block/Klasse gehört. z.B.:

```
pat: (#
     ...
     b : @Integer;
     func: (#
        c : @Integer;
        ...
     #);
#);
```

Hier ist `b` eine lokale Variable von `pat` und `c` eine lokale Variable von `func`.

3 Übersetzung

Die Übersetzung eines BETA-Programms erfolgt immer als Ganzes. Das bedeutet, dass immer alle verwendeten BETA-Quellen übersetzt werden, dazu gehört auch die BetaLib. Dies ist nötig, da sich der tatsächliche Inhalt der BetaLib durch das Fragmentsystem[4] von Programm zu Programm unterscheiden kann. Die Übersetzung der BetaLib für zwei unterschiedliche Programme kann also durchaus zu unterschiedlichen Ergebnissen führen. Das Verwenden der BetaLib vereinfacht zunächst die Arbeit des Übersetzers. Fast alle Funktionen können weiterverwendet werden, es ist also nicht nötig, alle Aufrufe der Bibliothek durch passende Aufrufe der Java-API zu ersetzen.
Jedes übersetzte Programm wird durch eine Laufzeitbibliothek (Paket „de.teltarif.beta.bTypes“, siehe Anhang C) ergänzt. Sie enthält Klassen, die verwendetet werden, um BETA-Sprachelemente abzubilden, die in Java nicht oder nicht in dieser Form verfügbar sind. Die daraus resultierende Vervielfältigung des Codes wird akzeptiert, um die Verwendung des erzeugten Java-Codes zu erleichtern.

Im Folgenden werden die wichtigen Ideen und Konzepte der durchgeführten Übersetzung erläutert.

3.1 Namensänderungen

Vor der eigentlichen Übersetzung ist es nötig, alle BETA Bezeichner anzupassen. Zum einen unterscheidet BETA nicht zwischen Groß- und Kleinschreibung; zum anderen darf kein Bezeichner einem Java Schlüsselwort oder wichtigen Klasse/Funktion entsprechen[5]. Weiterhin verwendet der Übersetzer eine Reihe von Namen sowie Prä- und Postfixen im erzeugten Code. Um Kollisionen und Irritationen zu vermeiden, sollten diese ebenfalls nicht als BETA-Bezeichner auftauchen. Anhang D enthält eine Liste aller betroffenen Wörter. Findet sich ein solcher Bezeichner im BETA Quelltext wird ein „_B“ angehängt. Beginnt ein Bezeichner mit einem der verwendeten Präfixe wird ihm ein „b_“ vorangestellt. In der Annahme, dass Bezeichner dieser Art eher unüblich sind, führt der Übersetzer derzeit keine Prüfung durch, ob durch eine der Änderungen ein Namen doppelt vorhanden ist.

Weiterhin wird versucht, alle Namen an den üblichen Java-Stil anzupassen. Dazu werden alle Attribute mit einem Kleinbuchstaben und alle Patterns mit einem Großbuchstaben am Anfang versehen.

3.2 Kommentare

Alle Kommentare des BETA-Codes werden 1:1 übernommen, dabei verändert sich lediglich ihre Position im Quellcode ein wenig. Falls gewünscht kann jedes Kommentar, das direkt vor einem Pattern steht, in ein Javadoc-Kommentar für das zugehörige übersetzte Element gewandelt werden. Da es eine große Chance gibt, dadurch eine zum Pattern gehörende Beschreibung zu erhalten.

Zusätzlich zu den bereits vorhandenen Kommentaren kann der Übersetzer neue Kommentare für die erzeugten Klassen, Funktionen und Variablen anlegen. Diese enthalten dann eine Angabe darüber, aus welchen Teilen des BETA-Quellcodes sich das jeweilige Element ergeben hat sowie kurze

4 Die BetaLib gibt dem Programmierer mittels des Fragmentsystems an vielen Stellen die Möglichkeit zusätzliche Funktionen in Standard-Patterns wie Stream und Text einzufügen, ohne dass die Patterns dafür abgeleitet werden müssen.

5 vgl. [Co] Kapitel 3

Erläuterungen dazu, wie das Element bei der Weiterentwicklung des Java-Codes verwendet werden kann.

3.3 Patterns

Das Pattern ist das wichtigste Element in BETA und bietet die meisten Möglichkeiten. Die Übersetzung von Patterns ist daher von zentraler Bedeutung für den gesamten Übersetzungsvorgang.

Im Allgemeinen wird ein Pattern in eine Klasse übersetzt, Subpatterns als innere Klassen[6]:

```
PatternA: VaterPattern (#
     Sub: (#
     #);
#);
```

wird zu

```
public class PatternA extends VaterPattern {
     public class Sub extends Object_B {
     }
}
```

Die Vererbungshierarchie kann direkt von BETA übernommen werden. Alle Patterns, die nicht explizit erben, erben implizit vom Pattern `Object`. Um korrekte Vererbungsstrukturen zu gewährleisten, müssen in der Übersetzung alle Patterns, die bisher implizit von `Object` abgeleitet waren, explizit von der entsprechend übersetzten Klasse `Object_B` erben.

Eine Alternative ist in [AM02] Seite 22 beschrieben. Hier werden Subpatterns in Klassen übersetzt und eine `origin` Variable angelegt, um auf das umgebende Pattern zuzugreifen. Dies ist jedoch eine spezielle Notwendigkeit, um BETA direkt in Bytecode zu übersetzen. Der Java-Compiler tut genau dies für die angelegte innere Klasse.

3.3.1 Do-Teil und Inner-Anweisung

Die Übersetzung des Do-Teils eines Patterns mit einer Inner-Anweisung basiert auf der Idee von [AM02] (Seite 26). Angenommen es liegen folgende Patterns vor:

```
1 A: (#
2    do
3       inner;
4 #);
5 B: A (#
6    do
7       inner;
8 #);
```

Pattern A wird mit einer Funktion `do_F` übersetzt. Sie bildet den Startpunkt für alle Aufrufe dieses und abgeleiteter Patterns und enthält den Code aus dem Do-Teil von Pattern `A`. Weiterhin wird eine Funktion `inner_0` angelegt. Diese Funktion ist leer und ersetzt die `inner` Anweisung aus Zeile 3. In

6 vgl. [Co] Kapitel 4

der Übersetzung von Pattern `B` wird die Funktion `inner_0` mit dem Inhalt des Do-Teil von Pattern `B` überschrieben. Um den Inner-Aufruf aus Zeile 7 zu ersetzen, wird wiederum eine Funktion mit den Namen `inner_1` angelegt. Alle weiteren Ableitungen werden entsprechend behandelt. Das Ergebnis der Übersetzung sieht also wie folgt aus:

```
public class A {
     public A do_F() {
        inner_0();
        return this;
     }
     protected void inner_0() { }
}
public class B extends A {
     protected final void inner_0() {
        inner_1();
     }
     protected void inner_1() {}
}
```

Um zu verhindern, dass ein Programmierer beim Ändern des erzeugten Codes eine Inner-Funktion aufruft oder die falsche Inner-Funktion überschreibt, werden diese als `protected` und beim Überschreiben als `final` deklariert.
Die Funktion `do_F()` gibt die jeweilige Instanz von `A` zurück, um einen direkten Zugriff auf eine eventuelle Exit-Funktion zu ermöglichen (siehe nächstes Kapitel).

3.3.2 Enter- und Exit-Teil

Ein Pattern kann in bis zu 4 unterschiedlichen Kombinationen von Enter und Exit aufgerufen werden[7]. Sowohl der Enter- als auch der Exit-Teil können dabei komplexe Anweisungen enthalten, die Nebeneffekte verursachen z.B.:

```
1 (#
2    test: (#
3       t: ^Text;
4       enter t -> putline
5       do
6          (* Code *)
7    #);
8    varT: ^test;
9    do
10      &test[] -> varT;
11      'hello' -> varT;
12      varT;
13 #);
```

Beim ersten Aufruf (Zeile 11) wird dem Pattern ein Parameter mitgegeben; deshalb muss der Enter-Code in Zeile 4 ausgeführt werden und `'hello'` wird ausgegeben. Beim zweiten Aufruf in Zeile 12 gibt es hingegen keinen Parameter, deshalb braucht Zeile 4 nicht ausgeführt werden und der Nebeneffekt entfällt.
Um dieses Verhalten nach Java übersetzen zu können, ist eine Aufspaltung in mehrere Funktionen für die verschiedenen Aufrufvarianten nötig. Hierbei sind grundsätzlich zwei Varianten denkbar:

7 vgl. [LK00] Seite 3

1. Für jede der vier möglichen Kombination von Enter/Exit wird eine Funktion erstellt.
   Dabei ist zu beachten, dass abgeleitete Patterns den Enter/Exit-Teil erweitern könnten, woraus sich weitere Kombinationen ergeben. Daher sind viele Funktionen nötig, die möglicherweise einen doppelten Code enthalten.
2. Für jedes Enter/Exit wird eine eigene Funktion angelegt, die nur die jeweiligen Anweisungen enthält und der Aufruf des Patterns wird dann entsprechend der Verwendung aus diesen Funktionen kombiniert. Nachteil dieser Variante ist die hohe Anzahl von Funktionsaufrufen, die einerseits den Code vergrößern und andererseits die Performance reduzieren.

Mit den Nachteilen der zweiten Variante lässt sich wesentlich besser umgehen, deshalb wird diese vom Übersetzer verwendet. Um die Performanceeinbußen gering zu halten, werden möglichst viele der erstellten Funktionen als `final` deklariert[8]. Zudem kann der Übersetzer in einem optionalen Optimierungsschritt automatisiert die Refaktorisierung „Methode Extrahieren“[9] anwenden. Dadurch wird die am meisten genutzte Kombination von Instanzierung, Enter-Funktion, Do-Funktion und Exit-Funktion des Patterns in eine Funktion `use_<Name>` zusammengefasst.

In Anhang B.1 sind beide Varianten beispielhaft gegenübergestellt. Variante 1 ist im Aufruf deutlich kürzer (ab Zeile 63, in Variante 2 ab Zeile 51) benötigt aber insgesamt mehr Funktionen, so dass die gesamte Übersetzung länger ist (73 Zeilen gegenüber 62 Zeilen von Variante 2). Zudem müssten für die Funktionen je nach Enter/Exit Kombination andere Namen verwendet werden. Sichtbar ist auch, dass Teile des Codes wiederholt werden müssen (z.B. sind Zeile 11, 16, 33 und 43 gleich), weil nicht wie in Variante 2 die jeweils zugehörige Funktion der Vaterklasse aufgerufen werden kann. In diesem Beispiel ist das zwar noch unerheblich, aber es gibt auch Fälle, in denen ein Enter/Exit aus komplizierten Inline-Patterns besteht.

Mehrere Rückgabewerte

Um in Java mehrere Rückgabewerte aus einer Funktion zurückgeben zu können, müssen diese in ein zusätzliches Objekt verpackt werden. Dabei könnte dieses Objekt auch die aufgerufene Instanz selbst sein[10]. Nachteil dieser Lösung ist, dass die Klasse selbst größer wird und dauerhaft mehr Speicher belegt. Deshalb verwendet der Übersetzer stattdessen Rückgabeobjekte, wobei in Abhängigkeit von der Anzahl der Rückgabewerte eine Klasse `Exit<Anzahl>` benutzt wird.[11]
Die Verwendung ist im Anhang B.1, Variante 2 zu sehen: in Zeile 33 wird das Objekt mit den einzelnen Rückgabewerten belegt und in Zeile 55 werden diese wieder ausgepackt. Um die Gültigkeit der Hilfsvariablen, in der das Rückgabeobjekt abgelegt wird, zu verringern, ist das Auspacken in einem Block zusammengefasst.

3.3.3 Patterns mit Funktionscharakter

Patterns, die einen deutlichen Funktionscharakter aufweisen, werden möglichst nicht in Klassen, sondern in Funktionen übersetzt. Es müssen jedoch eine Reihe von Bedingungen eingehalten werden, um Fehler in anderen Teilen des Quellcodes zu vermeiden.
Um ein Pattern als Funktion übersetzen zu können, muss es folgende Bedingungen erfüllen:
– Es darf nicht als Typ verwendet sein.

8 vgl. [Se05] Seite 4 und siehe Test in Anhang E.2
9 siehe [Fo05] Seite 106f
10 vgl. [AM02] Seite 30
11 Der Übersetzer unterstützt maximal 10 Rückgabewerte.

- Es darf weder erben (außer von `Object_B`), noch als Vater-Pattern verwendet sein.
  Grund hierfür ist, dass die Ableitung eines als Funktion übersetzten Patterns nur als Überschreiben der Funktion nachgebildet werden könnte. Die Ableitung in BETA bietet aber mehr Möglichkeiten, diese könnten dadurch nicht übersetzt werden.
  Im Übersetzer wurden jedoch einige wenige Ausnahmen von dieser Regel vorgesehen. So ist eine Übersetzung in eine Funktion auch möglich, wenn von einigen speziellen Patterns geerbt wird. Implementiert ist diese Ausnahme für die `<primitiver Datentyp>Value` und `<primitiver Datentyp>Object` Patterns, da diese keinerlei Funktionalität, sondern nur eine lokale Variable sowie die Enter- und Exit-Liste bestimmen.
- Es darf immer nur mit Enter- und Exit-Liste aufgerufen werden oder diese dürfen keine Nebeneffekte enthalten. Als Nebeneffekt werden hierbei (vereinfachend) Anweisungen betrachtet, die nicht in eine Zuweisung übersetzt werden können.
- Keine der lokalen Variablen des Patterns darf in einem eventuellen Inline-Pattern verwendet sein, es sei denn, das Inline-Pattern muss nicht in eine Klasse übersetzt werden (siehe Kapitel 3.5). Diese Bedingung ist nötig, um keine Java-Syntaxfehler zu verursachen, denn die lokalen Variablen einer Funktion können nicht in einer lokalen Klasse verwendet werden.
  Eine Ausnahme wäre möglich, wenn die lokale Variable innerhalb des Inline-Pattern nur gelesen wird. In diesem Fall könnte eine als `final` deklarierte Kopie der Variablen angelegt und diese verwendet werden. Das ist im vorliegenden Übersetzer jedoch nicht implementiert.

Die Übersetzung als Funktion statt als Klasse ist deutlich von Vorteil sowohl aus Sicht der Performance als auch für die Lesbarkeit des Codes. Nimmt man z.B. folgendes einfaches Pattern an:

```
CASELESS: (# do value %Bor 1 -> value #);
```

Als Klasse übersetzt erhält man:

```
public class CASELESS extends Object_B {
     public CASELESS do_F() {
        value = value | 1;
        return this;
     }
}
```

Wohingegen die Funktionsübersetzung kürzer ist, ohne Funktionalität einzubüßen:

```
public final void CASELESS() {
     value = value | 1;
}
```

Die Funktion wird hierbei als `final` deklariert, um versehentliches Überschreiben zu vermeiden, weil im Gegensatz zu Java in BETA alle Patterns zunächst statisch gebunden sind.

3.3.4 Konstanten

Konstanten werden in BETA ebenfalls als Patterns realisiert z.B.:

```
pcre_CASELESS: (# exit 1 #);
```

Die allgemeine Übersetzung als Klasse ist in solchen Fällen unnötig. Stattdessen kann das Pattern direkt in seiner Bedeutung als Konstante übersetzt werden:

```
public static final int Pcre_CASELESS = 1;
```

Hat solch ein Pattern kein umgebendes Pattern, wird die Konstante in der Übersetzung in eine neu erzeugte Klasse verschoben, die den Namen des aktuellen Packages erhält. Gleiches gilt für die oben beschriebene Funktionsübersetzung.

3.3.5 Statische Patterns

Statische Patterns (oder Part-Objekte[12]) sind solche, die als `<Name>: @(# ...` deklariert wurden. Das bedeutet, dass es, wie bei statischen Referenzen, nur eine Instanz dieses Patterns gibt, die bei der Initialisierung des umgebenden Objektes erzeugt wird.
Übersetzt wird dies durch eine Aufspaltung in ein normales Pattern und eine Variable,; es sei denn, das Pattern kann als Funktion übersetzt werden. Das Pattern (bzw. die Klasse, die daraus entsteht) erhält das zusätzlich Namenspräfix „at_“. Die Variable wird als `final` deklariert und entsprechend mit einer Instanz der Klasse initialisiert. Die Klasse kann außerdem einen Konstruktor erhalten, der die Sichtbarkeit auf das Package beschränkt, um falsche Verwendungen bei späteren Codeänderungen soweit wie möglich zu vermeiden.

3.3.6 Das Pattern External

Dieses Pattern hat eine besondere Bedeutung in BETA. Es wird dazu verwendet, externe Bibliotheksfunktionen aufzurufen. Es ist nicht möglich, diese Aufrufe automatisch nach Java zu übersetzen. Eine Übersetzung dieses Pattern und der daraus abgeleiteten Patterns wird daher nicht bzw. nicht funktionsfähig vorgenommen. In der gleichen Weise werden die Helfer-Patterns behandelt, die zum Zugriff auf Strings, Strukturen oder Arrays aus C-Funktionen dienen. Weiterhin nicht übersetzt wird der Adressoperator `@@`, der im Zusammenhang mit Externals verwendet wird, um die Speicheradresse einer Variablen zu ermitteln.
Alle Externals müssen nach der Übersetzung nachträglich durch passende Aufrufe von Java Funktionen ersetzt werden. Zur Reduzierung des Aufwands der Nachbearbeitung verwendet der Übersetzer eine speziell angepasste Version der BetaLib, in der bereits die häufig verwendeten Externals mit Hilfe der in Kapitel 4.4.3 beschriebenen Java-Hints korrigiert sind.

3.4 Virtuelle Patterns

Virtuelle Patterns werden unterteilt in Virtual Procedure Patterns und Virtual Class Patterns[13]. Die Class Patterns haben große Ähnlichkeit mit den in Java vorhandenen Generics; während die Procedure Patterns das Prinzip der späten Bindung realisieren.
Eine Unterscheidung zwischen beiden Varianten ist anhand der Syntax nicht möglich, sondern nur unter Betrachtung ihrer Verwendung. Deshalb werden alle virtuellen Patterns zunächst in Klassen übersetzt. Nur durch den Nutzer speziell mit einem Java-Hint (siehe Kapitel 4.4.3) markierte Class Patterns werden als Generics übersetzt.
Die Übersetzung als Generic weist, sobald die Entscheidung durch den Benutzer getroffen ist, keine Besonderheiten auf. Markiert der Nutzer allerdings ein ungeeignetes virtuelles Pattern, sind Fehler bei der Übersetzung wahrscheinlich.
Die Übersetzung als Klasse folgt der üblichen Übersetzung jedes normalen Patterns, es wird lediglich zusätzlich eine Create-Funktion erstellt. Dies ist erforderlich, da bei der Erzeugung mit

12 Beschreibung siehe [MMN93] Seite 31
13 vgl. [MMN93] 105ff und 139ff

`new` die Klasse bereits zur Compilezeit feststeht und die nötige späte Bindung nicht erfolgen würde. Die Create-Funktion ersetzt also für diese Klasse den `new` Operator.
Bei virtuellen Patterns, die lediglich ein anderes Pattern verwenden aber keinen eigenen Code definieren wie:

```
nCS:< BooleanValue;
```

wird nur die Create-Funktion erstellt.

Der folgende Ausschnitt aus der ContainerLib zeigt die Übersetzung als Generic und als Klasse:

```
container: (#
     (*JAVAHINT:GENERIC*)
     element:< Object;
     (*ENDHINT:GENERIC*)
     equal:< BooleanValue(#
        left, right: ^element;
        enter (left[], right[])
        do
           (left[] = right[]) -> value;
           inner;
     #);
     ...
#);
```

```
1  public class Container<element extends Object_B> extends Object_B {
2
3     public class Equal extends BooleanValue {
4        protected void inner_1() {
5        }
6        public element left;
7        public element right;
8        public final Equal enter_Equal(element param_0,
9              element param_1) {
10          left = param_0;
11          right = param_1;
12          return this;
13       }
14       protected final void inner_0() {
15          value = (left == right);
16          inner_1();
17       }
18    }
19
20    public Equal create_Equal() {
21       return new Equal();
22    }
23    ...
```

Die Klasse `Equal` darf nur über die Create-Funktion in Zeile 20 instanziert werden. Um dies zu unterstützten, kann in der `Equal` Klasse ein die Sichtbarkeit einschränkender Konstruktor angelegt werden. Wird die Create-Funktion hingegen nicht genutzt, würde immer nur die simple Variante des `Equal` Patterns verwendet werden; spezifizierte Varianten, die in von `Container` abgeleiteten Patterns enthalten sind, kämen nicht zur Anwendung.

3.4.1 Virtuelle Patterns mit Funktionscharakter

Häufig haben die Virtual Procedure Patterns die Eigenschaften einer Funktion und können unter gewissen Voraussetzungen auch als Funktion übersetzt werden. Folgende Bedingungen müssen dafür erfüllt sein:

- Es darf innerhalb des Pattern und des Vater-Patterns (und dessen Vater-Patterns etc.) nur genau einen Inner-Aufruf geben. Dieser darf nur direkt am Ende oder Anfang des Do-Teils liegen.
  Dies ist zwingend nötig, um den Unterschied zwischen dem Aufruf von Inner im BETA-Code auf die Möglichkeit zum Aufrufen von `super.func()` im Java-Code abzubilden. Ist das Inner am Ende wird in der überschreibenden Funktion zuerst `super` aufgerufen, bevor der eigene Code ausgeführt wird. Ist Inner hingegen am Anfang, wird der Super-Aufruf hinter dem eigenen Code ausgeführt.
  Verändert sich die Position des Inner-Aufrufs innerhalb der Kette von Bindungen[14] eines virtuellen Patterns, muss die Übersetzung ab der Stelle des Unterschieds wieder als Klasse erfolgen.
- Wie auch bei normalen Patterns, die in Funktionen übersetzt werden, dürfen keine lokalen Variablen in Inline-Patterns verwendet sein. Zusätzlich dürfen keine lokalen Variablen, die nicht Bestandteil von Enter oder Exit sind, in den Bindungen verwendet werden, weil diese sonst als zusätzliche Parameter und Rückgabewerte in der Funktionsübersetzung durchgereicht werden müssten.
  Weiterhin dürfen, wenn der Inner-Aufruf am Anfang steht, die Exit-Werte nicht gelesen und, wenn der Inner-Aufruf am Ende steht, die Enter-Werte nicht geschrieben werden. Anhang B.4 zeigt an einem Beispiel auf, welches Problem sonst entstehen würde.
- Die Enter-Liste darf sich bei allen Bindungen nicht verändern, da beim Überschreiben einer Funktion das Ändern der Parameterliste nicht erlaubt ist, um nicht mit dem Überladen zu kollidieren.
- Zur Vereinfachung für den Übersetzer darf sich außerdem die Exit-Liste nicht ändern und nur ein einziger Rückgabewert verwendet werden.
- Enter und Exit dürfen keine Nebeneffekte aufweisen.
- Das Pattern, das als Vater für die erste Bindung dient, sowie die Enter- und Exit-Liste sollten keine Inline-Patterns aufweisen. Diese Bedingung ist mehr auf die Sinnhaftigkeit der Übersetzung bezogen. Beständen die Enter- oder Exit-Liste aus Inline-Patterns, so müsste dieses in jede der überschreibenden Funktionen kopiert werden. Das wiederum würde den Vorteil der Übersetzung als Funktion aufheben und zu doppeltem Code führen.
- Das virtuelle Pattern darf nicht als Typ verwendet werden.

Betrachten man das Beispiel in 3.4, so stellt man fest, dass die Bedingungen eingehalten werden und man das `Equal` Pattern deutlich kürzer und schöner in eine Funktion übersetzen kann:

```
public boolean Equal(element left, element right) {
     boolean value = (left == right);
     return value;
}
```

Der Inner-Aufruf ist in der Funktion nicht möglich, stattdessen wird er durch einen Super-Aufruf in der überschreibenden Funktion simuliert:

```
@Override
public boolean Equal(element left, element right) {
```

14 Bindung bezeichnet in BETA das spezifizieren eines virtuellen Patterns

```
        // Die Übersetzung des Inner Aufrufs am Ende
        boolean value = super.Equal(left, right);
        ...
        return value;
    }
```

3.5 Inline-Patterns

Im allgemeinen Falle werden Inline-Patterns genauso wie normale Patterns übersetzt mit dem Unterschied, dass zusätzlich sofort eine Instanz der Klasse erstellt und verwendet wird.
Leitet sich eine Inline-Klasse von einer anderen ab, so benötigt sie in aller Regel zur Initialisierung das zu verwendende umgebende Objekt. Dazu erhält die Inline-Klasse einen Konstruktor dem dieses Objekt übergeben wird. In BETA hingegen ist das umgebende Objekt bereits implizit in der Deklaration des Pattern vorhanden:

```
1 t: ^Text;
2 ...
3 t.scanAll(#
4    do
5       ...
6 #);
```

`T` ist in diesem Fall das umgebende Objekt für das Inline-Pattern, das sich von `Text.scanAll` ableitet. In der Übersetzung muss Zeile 3 in die zwei enthalten Angaben zerlegt werden: Erstens das umgebende Objekt `t` in Zeile 13 und zweitens das Vater-Pattern in Zeile 7:

```
7 class inlineClass_0 extends Text.ScanAll {
8    public inlineClass_0(Text init) {
9       init.super();
10   }
11   ...
12 }
13 inlineClass_0 resVar_1 = new inlineClass_0(t);
14 resVar1.do_F();
```

Ein Inline-Pattern, das im BETA-Code nur verwendet wird um einen Block von Anweisungen zu gruppieren oder mit einem Label zu versehen, kann anders behandelt werden. z.B.:

```
L: (#
     do
        ...
#);
```

Dieses Pattern kann einfach in einen `{}` Block übersetzt werden. Allerdings nur, wenn das Pattern keine weiteren Patterns enthält und nur lokale Variablen definiert.

3.6 Attribute

Attribute (also Variablen) können mit nur geringen Änderungen übernommen werden. Primitive Datentypen werden entsprechend in die Java-Typen überführt. Zu beachten ist lediglich der Typ

`char`. Der Java-Char ist mit 2 Byte größer als der BETA-Char. Es ist möglich, dass diese Übersetzung zu Fehlern führt, wenn der BETA-Code auf dem Überlaufen des Char-Werts aufbaut:

```
newline;
255 -> c;
c + 100 -> c;
c -> screen.put;
newline;
```

Übersetzt man diesen Code mit dem BETA-Compiler, so erhält man als Ausgabe „c“. In der Java Übersetzung erhält man aber ein „?“, da die Variable `c` nach der Rechnung nicht 99 sondern 355 enthält. Der Übersetzer ignoriert dies unter der Annahme, dass so etwas nur selten verwendet wird. Sollte es dennoch auftauchen, kann es leicht manuell korrigiert werden.

Bei Referenzdatentypen wird in BETA zwischen statischen und dynamischen Referenzen unterschieden (bei primitiven Datentypen im Grunde auch, aber dort sind nur statische erlaubt). Dieser Unterschied wird in der Java-Übersetzung derart realisiert, dass statische Attribute als `final` deklariert werden und mit einer Instanz des jeweiligen Typs initialisiert sind.

Array-Attribute werden in jedem Fall initialisiert. Die Unterscheidung zwischen statischer und dynamischer Referenz zeigt sich hier dadurch, dass bei einer statischen Referenz jedes Feld des Arrays mit einer Instanz des jeweiligen Objektes initialisiert wird. Der Übersetzer schränkt in einem solchen Fall die Möglichkeiten zur Angabe der Arraygröße auf einfache Literale ein.

3.6.1 Pattern-Variablen

Pattern-Variablen in BETA finden ihre Java Entsprechung in `Class` Objekten, die Möglichkeiten von BETA gehen jedoch etwas weiter. Die Pattern-Variable speichert in BETA zwei Informationen: Erstens den Typ des Pattern und zweitens ein umgebendes Objekt, in dem eine neue Instanz des Pattern erstellt werden kann. `Class` Objekte hingegen speichern nur den Typ. Die Übersetzung bedient sich deshalb der Klasse `BetaClass`, die die Eigenschaften von Pattern-Variablen mit Hilfe der Reflection-API implementiert. Der Übersetzer braucht bei Verwendung von Pattern-Variablen lediglich die Funktionen von `BetaClass` aufzurufen, der erzeugte Java-Code bleibt dadurch kompakt. Die Reflection-API ist jedoch zugleich Nachteil wie Vorteil. Sie ermöglicht es zwar zur Laufzeit für beliebige Objekte das zugehörige umgebende Objekt zu ermitteln, das für eine neue Instanz nötig ist. Dieses geht aber zu Lasten der Performance, denn die Reflection-API arbeitet im Allgemeinen langsam.[15]
Die Vergleiche von Pattern-Variablen werden durch Funktionen in der Klasse `BetaClass` realisiert. Hier wird der Performancenachteil deutlich, da vor jedem Vergleich zunächst das umgebende Objekt ermittelt werden muss.

Eine Alternative zur Verwendung der Reflection-API wäre es in jeder Klasse eine Funktion einzufügen die, ähnlich der Standardfunktion `getClass`, das umgebende Objekt zurückgibt. Da dies aber den erzeugten Code um ein gutes Stück verlängern würde, versucht der Übersetzer stattdessen an möglichst vielen Stellen das umgebende Objekt bereits zur Übersetzungszeit zu bestimmen.

15 vgl. [Bl01] Seite 158

3.6.2 Zugriffsbeschränkungen

BETA als Sprache selbst unterstützt keinerlei Zugriffsbeschränkungen wie Java es tut. Das wird erst möglich durch das Fragmentsystem, mit dem die Sichtbarkeit eines Attributs auf eine Datei eingeschränkt werden kann. Es gibt in Java keine direkt vergleichbare Zugriffsbeschränkung, deshalb werden grundsätzlich alle übersetzten Elemente `public`.[16]
Nachteil ist, dass die vorhandene Trennung von Interface und Implementierung dadurch verloren geht. Auf die Korrektheit des übersetzten Codes hat das aber keinen Einfluss. Allerdings sollten die Programmierer, die später am übersetzten Java-Code weiterarbeiten, sich dessen bewusst sein und vor der Verwendung überprüfen, ob ein Attribut nicht eigentlich `private/protected` ist. Das ist oft an der Benennung und/oder den Kommentaren im Quellcode erkennbar.

3.7 Arrays

Arrays in BETA unterscheiden sich stark von Java-Arrays. Einige spezifische Übersetzungsmaßnahmen sind nötig:

1. Im Gegensatz zu Java betrachtet BETA Arrays nicht als Referenztypen. Das bedeutet, dass Zuweisungen von Arrays immer eine Kopie des Arrays zur Folge haben; daher müssen alle Zuweisungen von Arrays in der Übersetzung durch das Aufrufen einer Kopierfunktion durchgeführt werden.
2. Neben dem direkten Zugriff auf die Länge eines Arrays mit `.range`, was äquivalent zu dem in Java verwendeten `.length` ist, bietet BETA noch weitere Funktionen, die direkt auf dem Array ausgeführt werden können. Diese werden, wie auch die Kopierfunktion, durch das Aufrufen einer Funktion der Laufzeitbibliothek ersetzt.
3. In BETA hat das erste Element den Index 1, in Java aber 0. Deshalb wird bei allen Array-Zugriffen eine `-1` angehängt. Bei Fällen in denen der Array-Zugriff mit einer direkten Zahl erfolgt, wird stattdessen die Zahl um eins erniedrigt.
4. Neben dem einfachen indizierten Zugriff auf ein Array-Element ist es möglich, einen Slice[17] (ein Subarray) zu erstellen. Dies muss in der Java-Übersetzung wiederum durch eine spezielle Funktion abgebildet werden.

3.8 Kontrollstrukturen

3.8.1 Label

Für die Übersetzung von Labels wird die Möglichkeit von Java genutzt, Schleifen mit einem Label zu versehen. So können bei verschachtelten Schleifen die Break- und Continue-Anweisungen auf mehr als nur die innerste Schleife angewendet werden. Zum Anspringen des jeweiligen `break` (für `leave`) oder `continue` (für `restart`) werden Exceptions verwendet[18]. Die folgende einfache If-Anweisung soll als Beispiel dienen:

```
L: (if something then
     leave L;
else
```

16 vgl. [Co] Kapitel 1
17 Beschreibung siehe [MMN93] Seite 49f
18 vgl. [AM02] Seite 31 und [LK00] Seite 9

```
     restart L;
if)
```

Die Übersetzung erfolgt als eine Endlosschleife mit einem Try-Catch-Block:

```
1  L_0: do {
2     try {
3        if (something) {
4           throw new BetaLeaveException("L_0");
5        } else {
6           throw new BetaRestartException("L_0");
7        }
8     } catch (BetaLeaveException e) {
9        if (e.isLabel("L_0")) {
10          break L_0;
11       } else {
12          throw e;
13       }
14    } catch (BetaRestartException e) {
15       if (e.isLabel("L_0")) {
16          continue L_0;
17       } else {
18          throw e;
19       }
20    }
21    break L_0;  //Fehler: unreachable
22 } while(true);
```

Zur Unterscheidung verschiedener Labels wird der Label-Name als Argument für die Exceptions verwendet (Zeile 4, 6, 9 und 15). Deshalb wird der Name um eine laufende Nummer (_0) erweitert, wodurch er im gesamten übersetzten Code eindeutig ist. Damit ein normaler Ablauf, wenn kein `leave/restart` auftaucht, gesichert ist, wird am Ende der Schleife ein `break` (Zeile 21) eingefügt. Im obigen Beispiel müsste dies aber wieder entfernt werden, da es niemals erreicht werden kann (der Übersetzer erledigt dies).
Diese Verwendung von Exceptions zur Flusskontrolle ist generell schlechter Java-Code, und zudem auch langsam[19]. In allen Fällen, in denen es möglich ist, wird deshalb die Verwendung von Exceptions übergangen und stattdessen direkt `break` bzw. `continue` eingesetzt. Das Beispiel kann dadurch reduziert werden auf:

```
L: do {
     if (something) {
        break L;
     } else {
        continue L;
     }
     break L; //Fehler: unreachable
} while(true);
```

Nicht möglich ist aber eine solche Reduzierung, wenn das Label in einem Inline-Pattern verwendet wird z.B.:

```
L: (if something then
     leave L;
else
     search -> list.find(#
        do
```

19 vgl. [Bl01] Seite 169f

```
                restart L;
        #)
    if)
```

Die `break L;` Anweisung würde in diesem Fall in einer Inline-Klasse liegen. Java kann an dieser Stelle das Label jedoch nicht verwenden, deshalb ist die Übersetzung mit Exceptions unumgänglich. Um in diesen Fällen zumindest die Geschwindigkeitseinbuße gering zu halten, wird für beide verwendeten Exceptions der hier nicht benötigte Stacktrace deaktiviert[20].

3.8.2 If

Die (allgemeine) If-Anweisung von BETA hat große Ähnlichkeit mit dem Switch-Konstrukt in Java. In Java kann ein Switch allerdings nur verwendet werden, wenn numerische Typen geprüft werden und alle Werte Konstanten sind. Alle anderen If-Anweisungen werden deshalb in if-then-else-if Konstruktionen überführt. Dabei ist darauf zu achten, dass der Anfangswert nur ein einziges Mal berechnet wird, um eventuelle Nebeneffekte zu vermeiden.
Ist die Unterscheidung zwischen Switch und if-then-else-if getroffen, ist der Rest der Übersetzung einfach. Anhang B.2 zeigt an verschiedenen Beispielen die Übersetzung.

3.8.3 For

Die For-Schleife in BETA ist in ihrer Funktionalität eingeschränkt. Sie kann lediglich von 1 bis zu einem maximalen Wert zählen, wobei die Schrittweite immer 1 ist. Die Übersetzung ist deshalb leicht. Es gibt lediglich zwei Dinge zu beachten:

- Der Endwert der Schleife wird nur beim Schleifeneintritt berechnet. Er muss deswegen in einer Hilfsvariablen zwischengespeichert werden.
- Die Laufvariable ist in Java lokal für den aktuellen Block. Es kann aber vorkommen, dass die Laufvariable in einem Inline-Pattern gelesen wird. Da Inline-Patterns in Inline-Klassen übersetzt werden, kann dort nicht mehr auf die Laufvariable zugegriffen werden. In solchen Fällen wird bei jedem Schleifendurchlauf eine Kopie der Laufvariable als `final` Wert angelegt.

Anhang B.3 zeigt ein Beispiel für die Übersetzung.

3.9 Koroutinen

Die so genannten Komponenten in BETA stellen Koroutinen dar[21]. Eine Koroutine ist das allgemeinere Konzept von normalen Routinen (Funktionen/Prozeduren/Methoden). „Normale“ Routinen beginnen ihre Ausführung immer an genau an der gleichen Stelle und geben nur ein einziges Mal ein Ergebnis zurück. Koroutinen hingegen können mehrfach Ergebnisse zurückgeben, Sie beginnen ihre Ausführung bei wiederholten Aufrufen nach der Position, an der das letzte Ergebnis zurückgegeben wurde. Abbildung 1 stellt diesen Zusammenhang dar; links eine normale Routine, in der Mitte eine Koroutine und rechts die Implementierung, die später beschrieben wird. Die durchgezogenen Linien zeigen den Programmfluss, die gestrichelten Linien wartende Programmteile. Zu sehen ist daran das Koroutinen, im Gegensatz zu normalen Routinen, auf den erneuten Aufruf warten, also ihren internen Zustand behalten.

20 vgl. [Se05] Seite 5 und Siehe Test im Anhang E.1
21 siehe [MMN93] 177ff

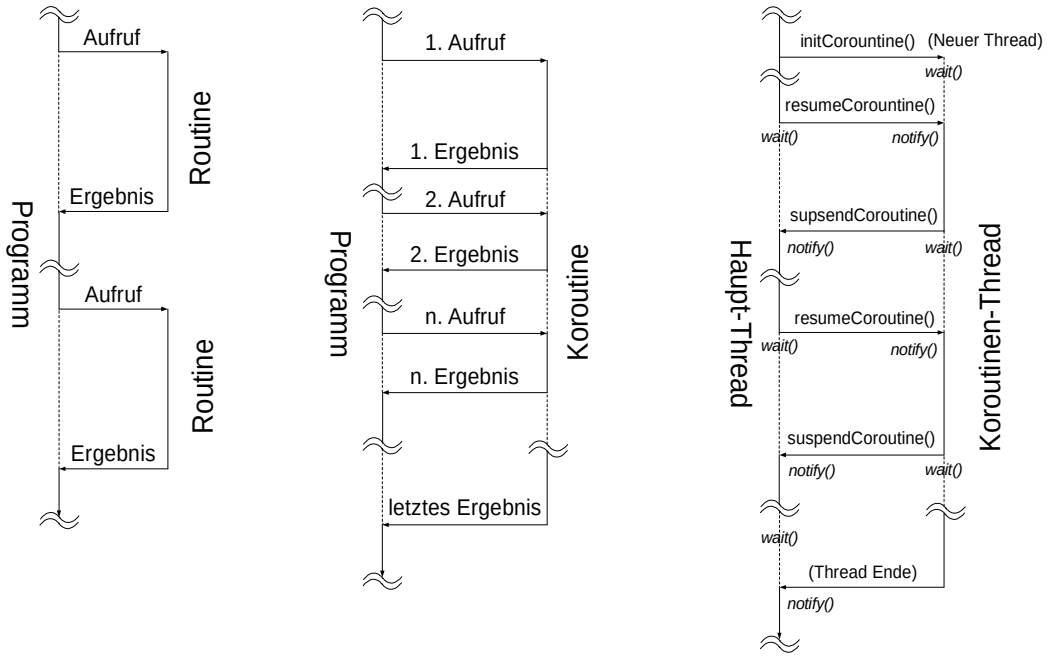

Abbildung 1: Koroutinen

Koroutinen sind also mit kooperativen Prozessen auf einem Einprozzesorsystem vergleichbar. Es können mehrere Koroutinen gleichzeitig vorhanden sein, wobei jede einen eigenen Instruction Pointer/Stack/Registersatz besitzt. Es wird jedoch immer nur eine Koroutine zu einem Zeitpunkt ausgeführt und jede Koroutine bestimmt selbst wann sie die Kontrolle an eine andere übergibt.
Um Koroutinen in BETA zu verstehen zunächst ein Beispiel:

```
1 (#
2   coroutine: (#
3      l, sum: @integer;
4      enter l -> putint
5      do
6         ' - Start ' -> puttext;
7         loop: (#
8            do
9               sum + l -> sum;
10              (if sum < 10 then
11                 suspend;
12                 '- Resume ' -> puttext;
13                 restart loop;
14              if);
15        #);
16        newline;
17        'Ende ' -> puttext;
18     exit sum
19  #);
20  myCoru: @|coroutine;
21  do
22     (for i: 20 repeat
23        i -> myCoru -> putint;
24        newline;
25     for);
26 #)
```

Durch die Verwendung des | in Zeile 20 wird myCoru als eine Koroutine vom Typ coroutine instanziert. Der Aufruf in Zeile 23 startet mehrfach dieselbe Routine. Man erhält folgende Ausgaben:

```
1 - Start 1
2 - Resume 3
3 - Resume 6
4 - Resume 10
5 - Resume
Ende 15
6
# Beta execution aborted: Executing terminated component.
```

Offenbar wird immer wieder die Anweisung aus dem Enter in Zeile 4 und dem Block in den Zeilen 7-15 wiederholt, bis die Summe den Wert 10 überschreitet und deshalb das suspend in Zeile 11 nicht mehr ausgeführt wird. Hat sich die Koroutine selbst beendet, ist ein erneutes Aufrufen nicht mehr möglich; daher die Fehlermeldung. Interessant ist aber, dass der Aufruf von Enter davon offenbar unabhängig ist.

Die Verwendung als Koroutine wird in BETA nur durch die Initialisierung bestimmt. Es ist problemlos möglich coroutine auch normal zu verwenden:

```
2 -> coroutine -> putint;
```

Das Schlüsselwort suspend bezieht sich immer auf die gerade aktive Koroutine. Ist keine Koroutine aktiv, beendet sich das gesamte Programm.

In der Übersetzung muss also dafür gesorgt werden, dass
- ein Pattern als Koroutine und trotzdem weiterhin normal ausgeführt werden kann,
- ein Suspend-Aufruf überall möglich ist,
- die Parameterübergabe unabhängig vom Zustand der Koroutine ist.
  Da die Enter/Exit/Do-Teile bereits in einzelne Funktionen getrennt sind (siehe Kapitel 3.3.2), ist dieser Punkt durch die normale Pattern-Übersetzung als Klasse bereits gesichert.

Anhang B.5 zeigt die vom Übersetzer erzeugte Übersetzung des Beispiels. Es wird das Pattern als Klasse übersetzt und diese Klasse dann um die Möglichkeit der Ausführung als Koroutine erweitert, wobei dies stets optional bleibt. Die Verwaltung der Koroutinenausführung selbst erfolgt durch die Klasse BetaCoroutine. Folgende Änderungen werden vorgenommen:
- Zum Initialisieren der Klasse als Koroutine wird die zusätzliche Funktion makeCoroutine angelegt (Zeile 47). Sie wird immer dort aufgerufen, wo eine Instanz konkret als Koroutine erzeugt werden soll (Zeile 52).
- Das Interface BetaCoroutineTask wird implementiert. Dazu wird eine Variable namens coroutineVar_ angelegt (ab Zeile 32) sowie die Methode run() erstellt (Zeile 12). Diese enthält die Anweisungen der Do-Funktion dieses Pattern.
- Die Do-Funktion der Klasse wird ersetzt durch eine Funktion die, falls als Koroutine initialisiert, die Koroutine startet/fortführt (Zeile 44) oder sonst die normale Ausführung mit der run() Methode vornimmt (Zeile 42).
- Alle Suspend-Aufrufe werden in den Aufruf der statischen Funktion suspendCoroutine() übersetzt (Zeile 18).

Wie schon dargelegt, können Koroutinen als Prozesse betrachtet werden. Daraus folgt, dass sie mit Hilfe von Threads implementierbar sind. Threads bieten jedoch mehr Möglichkeiten als gewünscht sind[22]: Java-Threads laufen (Quasi-) parallel ab. Es muss also dafür Sorge getragen werden, dass immer nur einer der Threads, die Koroutinen darstellen, tatsächlich aktiv abgearbeitet wird. Alle anderen Threads müssen solange in einem wartenden Zustand verbleiben. Ein Wechsel des aktiven Threads darf nur aufgrund des Aufrufs, der Unterbrechung oder der Beendigung einer Koroutine auftreten.

Die Klasse `BetaCoroutine` übernimmt die nötigen Aufgaben. Der rechte Teil der Abbildung 1 zeigt den tatsächlichen Ablauf einer Koroutine nach der Übersetzung.
Vor der eigentlichen Verwendung als Koroutine muss zunächst ein Thread erzeugt werden (`initCoroutine()`)[23]. Der neu erzeugte Thread (Koroutinen-Thread) verwendet einen Monitor, um sich zunächst in einen wartenden Zustand zu versetzen (`wait()`). Die Koroutine kann dann mit `resumeCoroutine()` gestartet werden. Dadurch wird der Koroutinen-Thread aufgeweckt (`notify()`) und der aufrufende Thread (Haupt-Thread) stattdessen in den wartenden Zustand gelegt. Der Thread startet nun die `run()` Methode der Koroutine und beginnt somit deren Abarbeitung. Führt die Koroutine im Folgenden `suspendCoroutine()` aus, versetzt sie sich selbst wieder in den anfänglichen wartenden Zustand und weckt den Haupt-Thread auf. Erst dann wird die `resumeCoroutine()` Methode verlassen. Anschließend kann der Haupt-Thread das aktuelle Ergebnis wie üblich mit der Exit-Funktion auslesen.
Das Zusammenspiel von Suspend und Resume wiederholt sich bis die `run()` Methode verlassen wird, die Koroutine also abgearbeitet ist. Passiert dies wird die Koroutine für weitere Aufrufe gesperrt und der verwendete Thread freigegeben. Durch einen abschließenden Aufruf von `notify()` wird die noch wartende Resume-Methode verlassen.

Die Resume-Methode, über die eine Koroutine gestartet wird, kann also erst wieder verlassen werden, wenn die Suspend-Methode aufgerufen wird oder die Koroutine sich selbst beendet. Dadurch wird die Bedingung, dass immer nur ein Thread zu einem Zeitpunkt aktiv ist, erfüllt.

3.10 Spezielle Übersetzungen

Wie eingangs erwähnt, wird die gesamte BetaLib mit übersetzt. Dadurch wird die Funktionalität, die häufig auch in der Java-API vorhanden ist, verdoppelt. Es ist daher wünschenswert, die BetaLib durch entsprechende Teile der Java-API zu ersetzen, um solche Verdopplung zu verhindern. Diese Ersetzung kann jedoch nur in geringem Umfang automatisiert werden und benötigt für jede zu übersetzende Funktionalität genaue Angaben darüber, was wie zu ersetzen ist. Der Implementierungsaufwand ist also hoch, weshalb sich der Übersetzer auf wenige, dafür häufig vorkommende Fälle beschränkt. Dabei wird von den Anforderungen für die Übersetzung des Trans ausgegangen. In anderen Programmen greift möglicherweise keine einzige der Ersetzungen.

3.10.1 Ersetzen von einzelnen Funktionsaufrufen

Spezielle Funktionen können direkt durch die jeweils passende Java-Funktion ersetzt werden. Dazu werden alle Aufrufe dieser Funktionen, die keine Besonderheiten enthalten, geändert. Die Klasse,

22 vgl. [Wo06]

23 Tatsächlich wird nicht für jede Koroutine ein Thread angelegt, stattdessen wird versucht Threads von bereits beendeten Koroutinen wiederzuverwenden.

die als Übersetzung der BetaLib an dieser Stelle vorher verwendet wurde, bleibt zwar erhalten, wird aber wesentlichen weniger verwendet und kann bei der späteren Weiterentwicklung des Codes gelöscht werden. Ideales Beispiel für diese Ersetzung sind die Ausgaben auf der Konsole:

```
new Putline().enter_Putline(new Text("Hello World")).do_F();
```

Sie können abgekürzt werden zu:

```
System.out.println("Hello World");
```

Die für die Übersetzung nötigen Informationen lädt der Übersetzer aus einer Datei. So kann sie der Nutzer nach seinen Bedürfnissen erweitern. In der Standarddatei sind nur die Funktionen zur Ausgabe auf der Konsole vorgesehen.

3.10.2 Ersetzen von Inline-Patterns durch Schleifen

Viele der Inline-Patterns, die im BETA-Code verwendet werden haben die Funktion von Schleifen, die über bestimmte Daten iterieren und für jedes gefundene Datum die Inner-Funktion aufrufen. Es ist daher möglich, solche Patterns statt mit Klassen mit den passenden Schleifen zu ersetzen. Dadurch müssen zwar einige Codeteile verdoppelt werden (der Schleifenkopf), dafür fallen aber die Inline-Klassen sowie die nötige Erzeugung von Instanzen dieser Klassen weg. In einigen Fällen ist es sogar möglich, dass durch die Reduktion auf eine Schleife das umgebende Pattern entsprechend Kapitel 3.3.3 oder 3.4.1 in eine Funktion gewandelt werden kann.
Es werden zwei Arten von Schleifen verwendet: Erstens eine gewöhnliche For-Schleife für Array-Zugriffe und zweitens eine Foreach-Schleife. Im Fall der Foreach-Schleife entfällt der Vorteil, dass kein Objekt erzeugt werden muss, da statt der Inline-Klasse ein neues Iterator-Objekt erzeugt wird. Trotzdem bleibt die Tatsache, dass sich im gesamten Code eine Klasse weniger befindet. Weiterhin wird meist durch die Verwendung der Foreach-Schleife die Funktion eines Code-Abschnitts leichter erkennbar. Folgendes Beispiel soll das deutlich machen:

```
eineListe.scan(#
     do
        current.irgendwas;
#);
```

Als Klasse würde es so übersetzt werden:

```
class inlineClass_0 extends Container.Scan {
     public inlineClass_0(List init) {
        init.super();
     }
     public void inner_0() {
        current.irgendwas();
     }
}
inlineClass_0 resVar_1 = new inlineClass_0(eineListe);
resVar_1.do_F();
```

Mit der beschriebenen Optimierung (und einer neuen Funktion in der Klasse List) kann die Übersetzung jedoch verkürzt werden:

```
for (typ current : eineListe.scanIter()) {
     current.irgendwas();
}
```

Für eine Ersetzung müssen dieselben Bedingungen eingehalten werden wie für Inline-Patterns, die nicht erben und potentiell in `{}` Blöcke übersetzt werden können. Zusätzlich dürfen die Inline-Patterns, die ersetzt werden, nicht verschachtelt sein, da es sonst zu Überschneidungen der in der Schleife verwendeten Laufvariablen kommt.
Es ist vorgesehen, dass die nötigen Ersetzungsinformationen in einer Datei gespeichert werden können. Dies ist aber leider noch nicht implementiert, stattdessen „lädt“ der Übersetzer drei fest codierte Optimierungen für `List.Scan`, `HashTable.Scan` und `Text.ScanAll`.

3.11 Nachfolgende Schritte

Einige Verbesserungen, die die Qualität des erzeugten Java-Codes erhöhen können, wurden bewusst nicht in den Übersetzer integriert. Sie werden von spezialisierten Tools, die auf den Code nochmals angewendet werden können, besser erledigt. Das betrifft u.a.:

- die Formatierung des Quellcodes,
- die verbesserte Namensgebung für Namen, die wegen Überschneidungsgefahr mit einen „_B“ ergänzt werden mussten,
- die Korrektur der Warnung des Java-Compilers dahingehend, dass auf statische Felder in statischer Weise zugegriffen werden sollte (diese Warnung entsteht, wenn Patterns in Konstanten übersetzt werden),
- das Entfernen von nie benutzten lokalen Variablen,
- das Löschen von nicht benötigten Imports (der Übersetzer geht großzügig beim Erstellen von Imports vor, um Fehler zu vermeiden),
- die Optimierung der Initialisierung lokaler Variablen.
  Diese sollte nur dann erfolgen, wenn es wirklich erforderlich ist. Der Übersetzer initialisiert jedoch grundsätzlich alle lokalen Variablen mit ihrem Standardwert, also 0/null/false. Es ist möglich, dieses Verhalten zu deaktivieren, wodurch keine lokalen Variablen mehr initialisiert werden. Anschließend können diese Fehler dann mit einem weiteren Tool (z.B. mit der Quick-Fix-Funktion von Eclipse[24]) automatisch oder halbautomatisch behoben werden, wodurch nur noch an den tatsächlich erforderlichen Stellen eine Initialisierung erfolgt.

24 IDE - http://www.eclipse.org/

4 Struktur des Übersetzers

Im Folgenden soll nun die Organisation des Übersetzungsvorgangs beschrieben werden; die Benutzung des Übersetzers ist im Anhang A dokumentiert.

Der Übersetzer besteht aus vier einzelnen Programmen und führt somit vier Läufe[25] durch. Diese Unterteilung folgt den üblichen Phasen eines Compilers. Ein fünftes Programm dient dazu, die Aufrufe der einzelnen Tools korrekt zu kombinieren.
Die Übersetzung lässt sich in einen quellsprachenabhängigen Teil und einen zielsprachenabhängigen Teil bzw. Front- und Back-End zerlegen. Der Übersetzer ist also so organisiert, dass es theoretisch möglich wäre, das Back-End auszutauschen und andere Ausgabesprachen zu erzeugen[25].
Das Zusammenspiel und die Zwischenergebnisse der einzelnen Schritte werden in Abbildung 2 dargestellt. Die einzelnen Programme sind mit durchgezogener Umrandung dargestellt, ohne Umrandung die verwendeten Eingaben und mit gestricheltem Rand die Zwischenergebnisse der Schritte. Rechts sind die korrespondierenden Compiler Phasen gemäß [ASU99] Seite 12ff abgebildet:

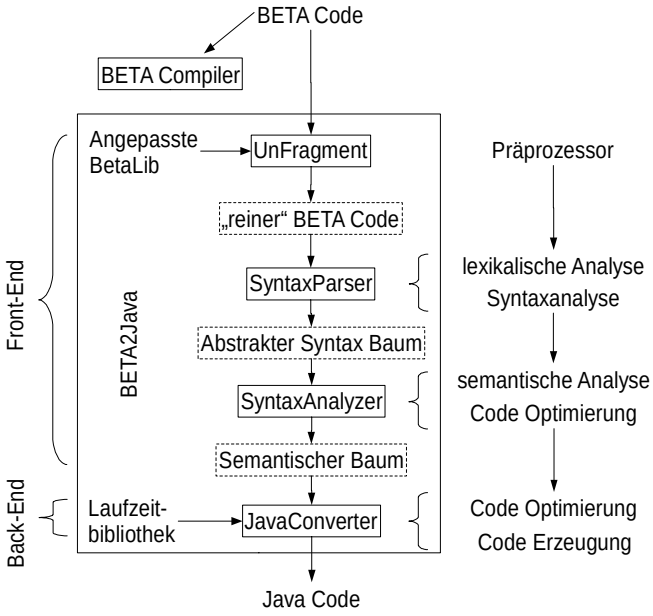

Abbildung 2: Struktur des Übersetzers

Der BETA-Compiler selbst ist nicht Bestandteil des Übersetzers. Der Übersetzer nimmt aber an, dass der eingegebene BETA-Code korrekt, ist also durch den BETA Compiler in ein funktionstüchtiges Programm übersetzt wurde. Auf Basis dieser Annahme kann sich der Übersetzer auf eine rudimentäre Fehlerbehandlung beschränken.

Der gesamte Übersetzer ist in Java geschrieben. Zum Erstellen der zwei verwendeten Parser wurde JavaCC[26] verwendet.

25 siehe [ASU99] Seite 25
26 Tool zum Automatischen erstellen von Parsern auf Basis einer Gramatik: https://javacc.dev.java.net/

Nahezu alle Operationen, die auf Bäumen arbeiten, sind mit dem Visitor-Pattern[27] implementiert. Somit enthalten die Knoten der Bäume selbst nur die Basisfunktionalitäten zum Traversieren, während die Funktionen bzw. Transformationen in separaten Klassen implementiert sind.

4.1 UnFragment

UnFragment dient dazu die BETA-Quellen vom Fragmentsystem zu bereinigen. Dabei wird das Fragmentsystem vereinfachend als eine Form von Includes betrachtet. Überall dort wo in den Ausgangsquellen Slots auftauchen, wird das entsprechende Fragment einkopiert. Ausnahmen bilden dabei die `lib` und `program` Fragmente. Jedes Fragment dieser Art bildet eine Ausgabedatei. Das Ergebnis des Laufs sind diese Dateien, bereinigt von allen Fragmentdefinitionen, sowie eine Liste der Dateien. Zusätzlich werden Informationen gespeichert, die angeben woher einkopierte Fragmentstücke ursprünglich stammen.

Zunächst parst UnFragment die Eingabedatei entsprechend der Definition in [MI04b], dabei wird eine Liste von weiteren zu verwendenden Dateien sowie eine Liste der in der Datei enthaltenen Fragmente erstellt. Alle gefunden Dateien werden ebenfalls bearbeitet. Dabei werden Verweise auf die BetaLib mittels einer speziellen für den Übersetzungsprozess angepassten Version die im UnFragment integriert ist, ersetzt. Sind alle gefundenen Dateien eingelesen, erfolgt die Ausgabe nach dem beschriebenen Prinzip.

4.2 SyntaxParser

Der SyntaxParser liest entsprechend der von UnFragment erstellten Liste alle BETA-Quellen ein und parst diese mittels der BETA-Grammatik[28] in einen abstrakten Syntaxbaum. Jeder Knoten des Baums erhält dabei eine Information über seine Quelle, damit auch bei späteren Fehlern korrekte Zeilenangaben verwendet werden können.
Um die Arbeit des SyntaxAnalyzers zu erleichtern, werden Kommentare die innerhalb von Anweisungen oder ähnlichem auftreten an günstigere Stellen verschoben z.B.:

```
3 + (* Kommentar *) 4 -> i;
```

wird vor die Anweisung gezogen:

```
(* Kommentar *)
3 + 4 -> i;
```

Durch diese Veränderung kann sichergestellt werden, dass Operationen im Baum immer nur zwei Kinder besitzen, wodurch Sonderfallbehandlungen beim Auftreten von Kommentaren in der Übersetzung nicht nötig sind.

Für jede geparste Datei gibt der SyntaxParser mittels Objekt-Serialisierung eine Datei aus, die den Syntaxbaum enthält.

27 Pattern ist in diesem Fall nicht das BETA-Pattern sondern ein Design-Pattern bzw. Entwurfsmuster, siehe [Ga95] Seite 269ff.

28 Verwendet wird die Grammatik wie sie beschrieben ist in [MI04c]

4.3 SyntaxAnalyzer

Mit dem SyntaxAnalyzer werden alle erstellten abstrakten Syntaxbäume in einem einzigen Baum zusammengefasst, der zusätzlich die Information über die Dateistruktur mit abbildet. Während des Zusammenfassens wird der Baum transformiert und in den so genannten semantischen Baum gewandelt.
Der semantische Baum fasst logisch zusammengehörige Syntaxelemente in einzelnen Knoten, die BETA-Sprachelemente wie z.B. Patterns und Attributdeklarationen darstellen, zusammen. Anschließend werden weitere Schritte unternommen, um den Baum mit zusätzlichen Informationen zu versehen. Diese sind im folgendem beschrieben.

Die Ausgabe des SyntaxAnalyzers ist eine einzige Datei. Sie enthält den Baum inklusive aller Zusatzinformationen in serialisierter Form.

Abbildung 3 zeigt die Verarbeitungsschritte innerhalb des SyntaxAnalyzers sowie die zum jeweiligen Schritt gehörenden Teile des Quellcodes.

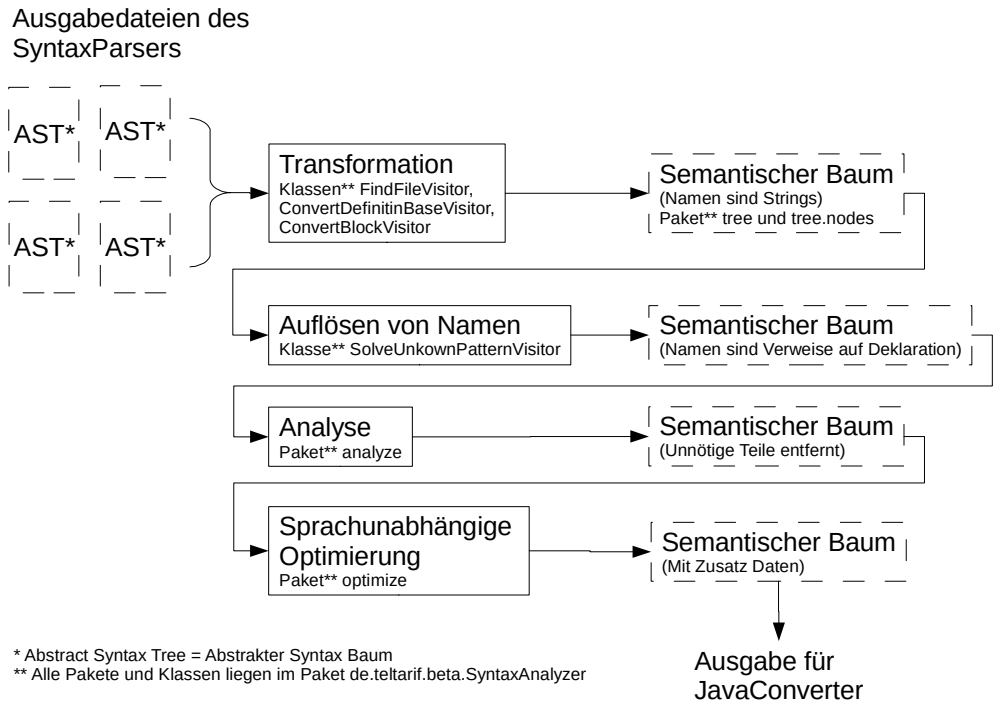

Abbildung 3: Vorgänge im SyntaxAnalyzer

4.3.1 Auflösen von Namen

Die wichtigste Aufgabe des SyntaxAnalyzers ist das Auflösen aller Namen. In einem iterativen Prozess wird der Baum durchsucht und alle Namen, die aus dem Syntaxbaum noch als Strings hinterlegt sind, werden durch Verweise auf die zugehörige Definition ersetzt.
Teil dieser Aufgabe ist es, den konkreten Typ von bestimmen Anweisungen zu ermitteln. So ist es z.B bei Anweisungen der Art:

```
(param -> pattern).attribut;
```

zunächst nötig zu bestimmen, welchen Typ `pattern` zurückliefert, um korrekt die Definition von `attribut` feststellen zu können. Ähnliches ist nötig bei allen Virtual Class Patterns:

```
1 List: (#
2   element:< Object;
3   ...
4   cur: ^element;
5   ...
6 #);
7 ...
8 myList: @List(#
9   element:: IntegerObject;
10#);
11...
12 10 -> myList.cur.value;
```

Um `value` (Zeile 12) auf die korrekte Definition in `IntegerObject` verweisen zu können, muss zunächst bestimmt werden welchen Typ `cur` (in Zeile 4 definiert als `element`) zu diesem Zeitpunkt tatsächlich hat.

4.3.2 Analyse

Ist das Auflösen der Namen erfolgreich, wird im nächsten Schritt eine Reihe von Analysen durchgeführt. Dabei werden keine neuen Informationen ermittelt, sondern lediglich im Baum bereits enthaltene Angaben in verwertbarer Form zusammengefasst und als Metadaten in den Knoten gespeichert.
Folgende Analysen werden durchgeführt:

- Bestimmen der Typliste für Enter- und Exit-Teile.
- Bestimmen ob ein Pattern eine Konstante ist.
- Bestimmen ob ein Inline-Pattern nur als Block verwendet wird, d.h.
  - es nicht erbt,
  - es keine Subpatterns benutzt,
  - es nur über lokale Variablen verfügt und
  - kein Enter- oder Exit-Liste vorhanden ist.
- Feststellen ob in einem If-Konstrukt alle Tests lediglich aus Literalen bestehen; es also einem Switch-Konstrukt von Java ähnelt.
- Sammeln von Verwendungsdaten für alle Patterns, um bestimmen zu können, ob es sich um eine Funktion oder Klasse handelt. Konkret heißt das:
  - Findet das Pattern Verwendung?
  - Finden Zugriffe auf Attribute des Patterns statt?
  - Wird das Pattern abgeleitet?
  - Anzahl der Verwendung als Typ,
  - Anzahl der Verwendung als Typ mit statischer Referenz,
  - Anzahl der Inner-Aufrufe innerhalb des Pattern,
  - Anzahl der Aufrufe des Pattern mit und ohne Enter/Exit und
  - Anzahl der Instanzierungen des Patterns oder Verwendung als Pattern-Variable.

4.3.3 Sprachunabhängige Optimierungen

Der letzte Teilschritt des SyntaxAnalyzer besteht in der Durchführung einiger Optimierungen, die unabhängig von Java als Zielsprache durchgeführt werden können:

- Alle unnötigen Klammern werden entfernt; z.B. wird `(x)` zu `x`.
- Nie verwendete Patterns und Attribute werden entfernt.
  Diese Optimierung kann jedoch zu Fehlern führen, wenn die Attribute nur „scheinbar“ nicht verwendet sind, weil sie z.B. in einem mit Hilfe von Java-Hints korrigiertem External benötigt werden.
- Inner-Aufrufe, die nicht verwendet wurden, weil es keine abgeleiteten Patterns gibt, werden entfernt.
  Möglicherweise können dadurch mehr Patterns in Funktionen übersetzt werden.
- Entfernen aller Verzeichnisse mit dem Namen „private“ aus der BetaLib, indem alle Inhalte in das darüber liegende Verzeichnis verschoben werden.
  Diese Verzeichnisse enthalten in der BetaLib meist nur Implementierungsfragmente, so dass sie nach der Anwendung von UnFragment nur noch wenig Inhalt haben. Dies dient also der „Verschönerung“ der später zu erzeugenden Ausgabeverzeichnisstruktur.

4.4 JavaConverter

Der JavaConverter verwendet den vom SyntaxAnalyzer erstellten semantischen Baum, den er in mehreren Schritten in den Java-Baum übersetzt. Nach erfolgreicher Übersetzung und der optionalen Optimierung wird der Java Baum durch den JavaWriter ausgegeben.
Abbildung 4 zeigt die internen Schritten des JavaConverters und in welchen Klassen bzw. Paketen sie implementiert sind:

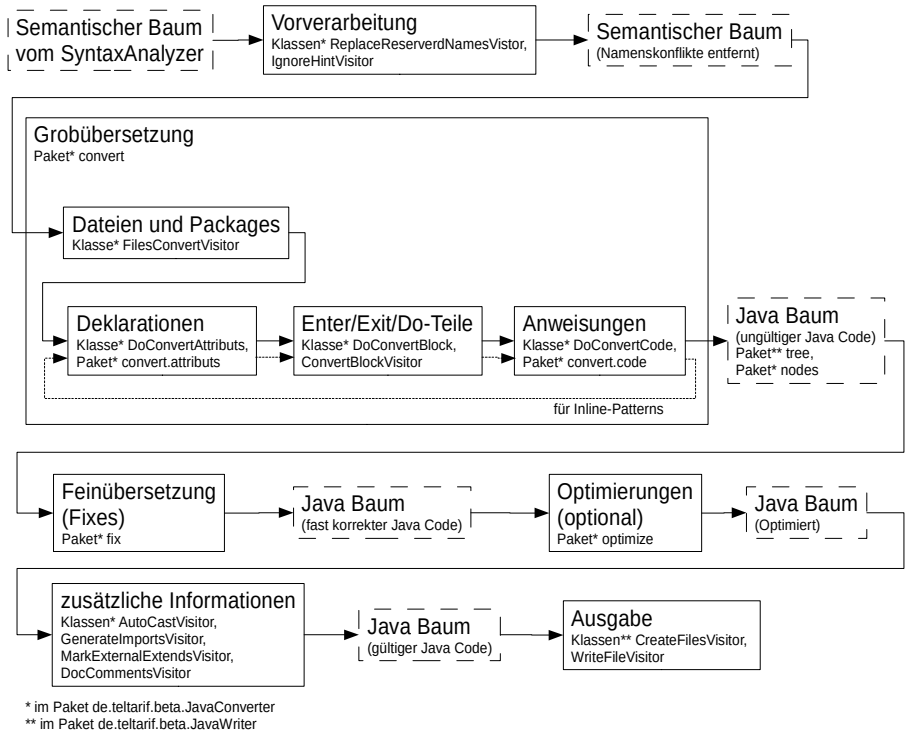

Abbildung 4: Vorgänge im JavaConverter

4.4.1 Übersetzung

Zunächst wird aus der Datei und der Verzeichnisstruktur eine Package-Struktur erzeugt. Jedes Verzeichnis wird zu einem Package, jedes Top Level Pattern (also ein Pattern das kein umgebendes Pattern kennt) zu einer Datei. Anschließend wird eine Grobübersetzung in drei Schritten durchgeführt:

1. Alle Pattern-Deklarationen werden auf Basis der vom SyntaxAnalyzer abgelegten Metadaten entweder in Klassen-, Konstanten- oder Funktionsdeklarationen umgewandelt. Ebenso werden alle Attributdeklarationen passend gewandelt. Virtuelle Patterns werden, falls mit einem Java-Hint markiert, in Generics, andernfalls zunächst komplett in Klassen übersetzt.
2. Für alle Enter-, Do- und Exit-Teile werden die zugehörigen Funktionen angelegt.
3. Sämtliche Anweisungen werden übersetzt. Besteht eine Anweisung aus einem Inline-Pattern, werden hierfür Schritt 1 und 2 wiederholt.

Nach Abschluss der Grobübersetzung ist der gesamte BETA-Code in Java-Code überführt. Die Grobübersetzung kann jedoch aufgrund ihrer Organisation nicht alles korrekt übersetzen. Deshalb werden anschließend eine Reihe von „Fixes“ durchgeführt. Diese korrigieren spezielle Fälle, die die Grobübersetzung zunächst ignoriert. Diese Organisation bietet für die Implementierung zwei Vorteile. Erstens kann sich die Übersetzung zunächst auf die häufigen Fälle konzentrieren, wodurch die Grobübersetzung einfacher wird. Zweitens können gewisse nötige Entscheidungen leichter getroffen werden, wenn davon ausgegangen werden kann, dass bereits alle Teile des BETA-Codes übersetzt wurden.
Die Fixes erledigen u.a. folgende Aufgaben:

- Korrigieren der Bindungen von Generics.
- Unterscheiden der BETA-String-Literale in Java-Char und String-Literale.
- Entfernen von unerreichbaren Anweisungen, die vom Übersetzer generiert wurden.
- Sicherstellen, dass alle Klasse aus `Object_B` abgeleitet sind.
- Ergänzen von unzureichenden Qualifikationen.
- Anpassen von Namen, die andere Namen verdecken.

Zuletzt müssen noch Informationen eingefügt werden, die im BETA-Code nicht vorhanden sind:

- BETA kennt keine expliziten Casts, ist dafür aber sehr viel großzügiger mit der Verwendung von impliziten Casts. Der `AutoCastVisitor` versucht diese, impliziten Casts durch explizite Casts zu ersetzen.
- Jede Java-Datei benötigt eine korrekte Importliste mit den in der Datei verwendeten Klassen; diese wird vom `GenerateImportsVisitor` erzeugt.
- Zum leichten Auffinden von Problemen und Externals im erzeugten Code werden an den entsprechenden Stellen Kommentare eingefügt, die das Wort „BETACONV“ beinhalten (`MarkExternalExtendsVisitor`).
- Der `DocCommentsVisitor` versucht, für alle aus Patterns erzeugten Klassen und Funktionen anhand der vorhandenen Kommentare, ein Javadoc-Kommentar zu erstellen.

4.4.2 Optimierungen

Der JavaConverter beherrscht eine Reihe von Optimierungen:

1. Ersetzen von einzelnen Zeilen durch passende Java-API-Aufrufe (siehe Kapitel 3.10.1). Alle nötigen Daten werden aus der Datei „replaceLineData.xml“ im Ausführungsverzeichnis

des Übersetzers geladen. Ist dort keine solche Datei vorhanden, wird eine interne Datei verwendet. Aus dieser Datei wird ein Suchbaum generiert. Dieser Baum wird dann verwendet, um mit allen Zeilen einen Mustervergleich durchzuführen. Trifft ein Muster zu, wird der zugehörige Austausch vorgenommen.

2. Austauschen von Inline-Klassen durch geeignete Java Strukturen (siehe Kapitel 3.10.2).
   Diese Optimierung besteht aus zwei Teilen. Die Analyse wird als zusätzliche Funktion bereits vom SyntaxAnalyzer ausgeführt. Er erhält als Argument eine zusätzlich auszuführende Klasse. Kann ein Inline-Pattern ersetzt werden, wird es im semantischen Baum markiert. Die weiteren Analysen, die vom SyntaxAnalyzer durchgeführt werden, betrachten dadurch dieses Pattern wie einen Block.
   Im zweiten Teil, der im JavaConverter ausgeführt wird, findet die eigentliche Ersetzung statt.
3. Wandeln von Klassen, die aus virtuellen Patterns erstellt wurden in Funktionen (siehe Kapitel 3.4.1).
4. Extrahieren von häufig vorkommenden Aufrufkombinationen (siehe Kapitel 3.3.2).
   Hier wird zunächst der gesamte Code nach allen auftretenden Verwendungen einer Klasse durchsucht. Für jede Klasse wird dann die häufigste Kombination ermittelt und eine Funktion erstellt, die diese Kombination enthält. Mit einem Suchbaum, wie er auch bei 1. verwendet wird, werden dann die Aufrufe durch die dafür erstellten Funktionen ersetzt.
5. Löschen von Java-Hints:
   Dabei handelt sich um keine Optimierung zu Verbesserung des Codes. Es werden lediglich die Kommentare, die nur Anweisungen für den Übersetzer enthalten entfernt, da sie nicht mehr benötigt werden.
6. Entfernen von Hilfsvariablen, die vom Übersetzer angelegt aber nicht benötigt wurden.
7. Reduzieren von Blöcken.
   Der Übersetzer legt einen Reihe von `{}`-Blöcken an, um Übersetzungsergebnisse zu gruppieren. Ein Teil dieser Blöcke ist jedoch nicht unbedingt nötig besonders wenn sie:
   - nur aus einer Zeile bestehen,
   - keinerlei Attributdeklaration beinhalten oder
   - in der übergeordneten Ebene das einzige Element sind.

   Reduzieren bedeutet dann, dass der Inhalt des Blocks direkt in die übergeordnete Ebene eingefügt wird.
8. Entfernen von leeren Klassen.
   Gelegentlich kommt es vor, dass ein statisches Pattern verwendet wird, nur um ein Virtual Class Pattern zu binden:

   ```
   myList: @List(#  element:: IntegerObject #);
   ```

   Wie in Kapitel 3.3.5 beschrieben ist, wird ein solches Pattern in eine Klasse und eine Variable mit dem Präfix „at_“ zerlegt. Da die Klasse in diesem Fall aber nur die Generic-Bindung enthält, kann dieser Fall nachträglich verkürzt werden auf:

   ```
   public final List<IntegerObject> myList = new List<IntegerObject>()
   ```

4.4.3 Java-Hints

Java-Hints sind spezielle Kommentare im BETA-Quellcode, um dem Übersetzer Hinweise zu geben. Sie werden für drei Zwecke verwendet:

- Korrektur von Übersetzungsfehlern, vor allem solche, die durch fehlende Casts verursacht werden.

- Anweisungen, dass andere Übersetzungen verwendet werden sollen, z.B. die in Kapitel 3.4 erwähnte Übersetzung als Generics.
- Einfügen von Java-Code, um Externals so zu verändern, dass sie Funktionen der Java-API verwenden.

Java-Hints folgen einem festen Format:

```
(#JAVAHINT:<Name>[|<parameter>[|<parameter>...]]#)
...
(#ENDHINT:<Name>#)
```

Der Hint ist in dem gesamten Bereich zwischen seiner Definition und dem `ENDHINT` für alle Übersetzungsschritte sichtbar. Werden Hints verschachtelt, so ist immer nur der zuletzt definierte Hint sichtbar. Zum Einfügen von Java-Code gibt es zwei Sonderformen:

```
(#JAVACODE: .... #)
(#JAVACODECLASS: ... #)
```

Die erste Variante fügt den Code an der Position des Hints ein, während die zweite Variante den Code am Anfang der Klassendatei einfügt, in der der Hint liegt.

Eine Beschreibung der verfügbaren Hints befindet sich in der Benutzerdokumentation im Anhang A.

5 Bekannte Probleme

Der Übersetzer arbeitet nicht fehlerfrei. Es gibt einige bekannte Probleme, die für die Übersetzung des Trans jedoch keine Bedeutung haben bzw. mit wenig Aufwand manuell oder mit Hilfe der Java-Hints korrigiert werden können.

5.1 Nicht übersetzbare Konstrukte

In der BETA-Grammatik sind zwei Konstrukte definiert, deren Bedeutung unbekannt ist und die deshalb nicht übersetzt werden können:

- `TOS<Name>`
  Möglicherweise als Abkürzung für „Top of Stack“ zum direkten Zugriff auf Werte aus dem Speicher. Die Anweisung wird in der BetaLib an wenigen Stellen im Zusammenhang mit dem direkte Zugriff auf Speicherelemente verwendet. Eine Übersetzung nach Java erscheint deshalb als unmöglich. Da die betroffenen Bereiche jedoch nicht vom Trans verwendet werden, wird diese Anweisung ignoriert.
- `(code ... code)`
  Die Bedeutung diese Konstruktes ist völlig unklar. Es findet auch keine Verwendung, deshalb kann der Übersetzer es ebenfalls problemlos ignorieren.

5.2 Nicht vollständig unterstützte Konstrukte

Spezielle Kombination von Konstrukten kann der Übersetzer nicht verarbeiten. Drei Fälle sind bisher aufgetreten.

1. Der größte Teil der sog. Low-Level-Primitiven[29] ist nicht verfügbar. Implementiert sind lediglich die Möglichkeiten zum Zugriff auf einzelne Bytes von Integer-Variablen.

2. In Enter- oder Exit-Teilen können keine virtuellen Patterns, bei denen sich die Parameter ändern, verwendet werden. Somit ist folgender Code nicht übersetzbar:

```
1  test: (#
2       i: @integer;
3       split:< (#
4          enter i
5          exit i
6       #)
7       enter split
8       exit split
9    #);
10   test2: test (#
11      j: @integer;
12      split::< (#
13         enter j
14         exit j
15      #)
16   #);
17   t: @test2;
18   do
19      (3, 2) -> test2 -> (putint, putint);
```

29 siehe [MI04a] Kapitel 13

Der Übersetzer legt die Enter-Funktion für `test` lediglich mit den Argumenten des dort bekannten `split` an. Eine Übersetzung des Aufrufs in Zeile 19 ist nicht möglich, weil dem Übersetzer nur die Enter-Funktion mit einem Parameter, wie sie sich aus Zeile 7 ergibt, bekannt ist. Um das Problem zu beheben, müsste in der Übersetzung von `test2` eine neue Enter-Funktion mit zwei Parametern angelegt werden. Die notwendigen Schritte wurden im Übersetzer nicht implementiert. Dieser Fall tritt im Trans nur ein einziges Mal auf und wurde an dieser Stelle manuell geändert, so dass nicht mehr das virtuelle Pattern verwendet wird.

3. Anwendungen von `this([..])` mit denen auf verdeckte Anteile eines Vater-Pattern zugegriffen, wird, sind nur für den direkten Vater reibungslos möglich. z.B.:

```
1 A1: (#
2    b: @Integer;
3    do inner;
4 #);
5 A2: A1(#
6    b: @Boolean;
7    do inner;
8 #);
9 A3: A2(#
10   b: @Char;
11   do
12      this(A1).b -> putint;
13      this(A2).b -> screen.putboolean;
14#);
```

Zeile 12 kann nicht nach Java übersetzt werden. Ein Zugriff auf das mehrfach verdeckte `b` aus Zeile 2 wäre nur möglich mit `A3.super.super.b`, diese Anweisung gibt es in Java aber nicht. Dagegen ist Zeile 13 möglich, da zum Erreichen des `b` aus Zeile 6 lediglich ein `A3.super.b` benötigt wird. Entdeckt der Übersetzer einen solchen Fall, so verwendet er die falsche Übersetzung mit nur einem `super` und gibt eine Meldung aus. Beim Übersetzen des Trans tritt das Problem nicht auf.

5.3 Übersetzungsfehler

Übersetzungsfehler in diesem Sinne sind Java-Fehler im ausgegebenen Code. Diese treten vorzugsweise im Zusammenhang mit Generics oder fehlenden Casts auf. Sie können durch kleine manuelle Änderungen des erzeugten Codes oder den Einsatz eines Hints behoben werden. Folgende Probleme können auftreten:

- Die aktuelle Bindung eines generischen Typs wird nicht richtig ermittelt. Vor allem dann, wenn mehrere generischen Typen mit dem gleiche Namen verschachtelt auftreten.
- Für eine Zuweisung ist ein Cast nötig, wobei der Typ der Zielvariablen generisch ist.
- Ein generischer Typ erhält für ein Attribut eine Bindung, obwohl dies gar nicht nötig ist.
- Integer-Zahlen im Quellcode werden nur bis maximal bis 2^{31}-1 eingelesen. Da nur positive Zahlen gelesen werden und ein eventuelles Vorzeichen davor als einstelliger Operator erkannt wird, kann die kleinste mögliche Integer Zahl (-2^{31}) nicht im BETA-Quellcode verwendet werden.
- Unnötige Angaben führen zu Fehlern.
- Patterns, insbesondere virtuelle Patterns, die zwar benutzt werden aber keinerlei Funktion haben, sind falsch übersetzt.

Allgemein ist zu sagen: Es ist leicht komplizierte bzw. umständliche Anweisungen oder Konstrukte zu schreiben, die nicht korrekt übersetzt werden können. Die meisten dieser Fälle tauchen jedoch bei der gewöhnlichen Programmierung nicht auf.

Weiterhin ist noch ein Fehler bekannt bei dem zwar ein korrekter Java-Code erzeugt wird, aber eine `NullPointerException` in der Ausführung auftritt. Das Problem tritt auf im Zusammenhang mit Inline-Patterns die statische Subpatterns verwenden. Das List-Pattern aus der BetaLib hat dieses Problem:

```
1    List: (#
2       ...
3       iter: (# ...
4          where:< cellPredicate;
5          listIteratePrivate: @(#
6             cp: @where;
7             ...
8             #);
9          ...
10      #);
11   #);
```

Die Übersetzung des Patterns `iter` könnte im Java-Code beispielsweise derart Verwendung finden:

```
12 ...
13 class iter ...
14      public final at_ListIteratePrivate ListIteratePrivate =
15         new at_ListIteratePrivate();
16      class at_listIteratePrivate {
17         public final CellPredicate cp = create_where();
18      }
19 ...
20 class inlineClass_0 extends List.Iter {
21            List inlineClass_0_init;
22            public  inlineClass_0(List _init) {
23               _init.super();
24               this.inlineClass_0_init = _init;
25            }
26            class where extends Main.MyList.CellPredicate {
27               protected  where() {
28                  inlineClass_0_init.super();
29               }
30            }
31            public where create_where() {
32               return new where();
33            }
34            ...
35         }
```

Die Klasse `where` benötigt ein umgebendes Objekt vom Typ `List`, um korrekt initialisiert zu werden. Dazu legt der Konstruktor des Inline-Patterns den nötigen Wert in einer Variablen ab (Zeile 24), so dass dieser durch Konstruktor von `where` verwendet werden kann. Da jedoch `cp` eine statische Referenz ist (Zeile 6), ergibt sich die Übersetzung in Zeile 14/15 wo die Variable direkt beim Anlegen initialisiert wird. Das heißt, dass hier der Konstruktor von `where` aufgerufen wird. Da das Anlegen der Instanzvariablen aber noch vor der Ausführung von Zeile 24 erfolgt, ist die vom Konstruktor verwendete Variable noch gar nicht belegt und es kommt zu der besagten `NullPointerException`.

Zur Vermeidung dieses Problem muss der BETA-Code so geändert werden, dass an dieser Stelle eine dynamische Referenz verwendet wird, die beim Aufruf des Do-Teils initialisiert wird.

5.4 Inkompatibilität des erzeugten Code

Der erzeugte Java-Code kann nicht mit dem Standard Java-Compiler von Sun (javac) compiliert werden. Es ist derzeit leider nur mit Eclipse oder dem gcj[30] möglich.
Es scheint sich dabei, zumindest teilweise, um Fehler im javac zu handeln. Deshalb wurde der Übersetzer nicht derart angepasst, dass auch ein Compilieren mit dem javac möglich wird. Es ist jedoch mit vertretbaren Aufwand möglich, den erzeugten Code entsprechend anzupassen; die Benutzerdokumentation im Anhang A enthält entsprechende Hinweise.

Einer der Fehler betrifft die Bindung von generischen Typen:

```
class A<T>  {
     class Sub1<T2 extends A<T>> {
     }
}

class B extends A<Number> {
     class Sub2 extends Sub1<B> {
     }
}
```

Dieser Code ist für den Eclipse-Compiler korrekt, der javac meldet aber diesen Fehler:

```
type parameter A<java.lang.Number> is not within its bound
        class Sub extends A<Number>.Sub<B> {
                                       ^
```

Offenbar ist es an dieser Stelle aber korrekt, dass `T2 extends A<T>` (also `A<Object>`) mit `B` gebunden werden kann. Ein ähnlicher Code wird auch vom javac akzeptiert:

```
class Y<T, T2 extends Y<T, T2>> {

}

class Z extends Y<Number, Z> {

}
```

Dies führt zu der erwähnten Annahme, dass der Fehler an dieser Stelle im javac liegt.

30 GNU Java Compiler, Teil der GCC: http://gcc.gnu.org/

6 Zusammenfassung

Trotz der angesprochenen Probleme erfüllt der entstandene Übersetzer die gesetzten Ziele. Natürlich ist es störend, dass dem Übersetzer vom Nutzer zusätzliche Hinweise gegeben und spezielle Fälle im Code verändert werden müssen. Es war aber notwendig Grenzen festzulegen, um den Aufwand zu reduzieren. Sicherlich wäre es möglich, diese Probleme zu beheben, aber für den zunächst erwünschten Zweck, ist dies übertrieben. Denn es soll zunächst nur ein einziges Programm, der Trans, nur ein einziges Mal übersetzt werden. Trotzdem wäre es auch für diesen Zweck noch wünschenswert, die Lesbarkeit des erzeugten Codes weiter zu verbessern. Störend fällt vor allem die große Anzahl von Klassen auf, insbesondere von inneren und lokalen (bzw. Inline).

6.1 Die Übersetzung des Trans

Der Übersetzer wurde erfolgreich für die Übersetzung der vorliegenden Version des Trans getestet, dabei traten lediglich Rundungsunterschiede[31] in der Ausgabe auf. Bevor die Java-Version des Trans jedoch die BETA-Version im produktiven System ersetzen kann, sind noch weitere Schritte nötig. Erstens muss geprüft werden, ob die Änderungen, die am Trans seit Beginn des Projektes vorgenommen wurden, übersetzt werden können. Zweitens sollten detaillierte Vergleichstests für alle Möglichkeiten, die der Trans bietet, erfolgen.

Für die Übersetzung mussten in den knapp 33.000 Zeilen des Trans insgesamt 40 Hints eingefügt und drei Konstruktionen verändert werden. Zusammen mit der nötigen BetaLib werden fast 48.000 Zeilen BETA-Quellcode übersetzt. Erzeugt werden daraus gut 96.000 Zeilen (72.000 ohne erklärende Kommentare) Java-Code in 269 Dateien mit ca. 2800 Klassen.
Die Anzahl der Klassen im Verhältnis zur Anzahl der Dateien zeigt das angesprochene Problem. Die Bemühungen Patterns in Funktionen statt in Klassen zu übersetzen hat hierbei aber schon deutliche Wirkung gezeigt, ohne diese würden sich fast 3900 Klassen[32] ergeben.

Die Frage der Performance wurde ebenfalls erfolgreich durch die Optimierungen verbessert. Den größten Einfluss auf die Geschwindigkeit hatte die Unterscheidung von Labels deren Übersetzung, je nach Möglichkeit, mit oder ohne Exceptions erfolgt. Insgesamt reduzierte sich die Ausführungszeit[33] von fast 4 Stunden auf 3-4 Minuten, wenn alle Spezialfallübersetzungen aktiv sind. Damit liegt das Java-Programm im gleichen zeitlichen Rahmen wie das originale BETA-Programm. Sehr unterschiedlich ist jedoch der Speicherumfang, der Java-Trans benötigt mit 900 MB fast 300 MB mehr Arbeitsspeicher als die BETA-Version.

6.2 Mögliche Verbesserungen des Übersetzers

Auch wenn die Übersetzung bereits erfolgreich war und damit der eigentliche Zweck des Übersetzers erfüllt ist, wären für eine weitere Verwendung des Übersetzers zu anderen Zwecken noch einige Verbesserungen möglich:

31 Es treten minimale Unterschiede bei Rechnungen mit Fließkommazahlen auf. Ein Unterschied von z.B. 1,785 und 1,7850001 führt dann zu unterschiedlichen Rundungsergebnissen.

32 Bei der Reduktion um ca. 1100 Klassen entfallen 484 auf Patterns die direkt in Funktionen übersetzt werden, 121 auf die Inline-Klassen-Ersetzung und 152 auf Ketten von virtuellen Patterns die in Funktionen gewandelt wurden.

33 Getestet wurde mit dem „fast“ Modus, einer von vielen verfügbaren Modi des Trans. Als System diente ein virtueller PC mit einem Intel Core 2 Duo 2.4 GHZ (nur ein Kern bei Emulation verwendet), 1536 MB RAM sowie OpenSuSE 11 (32 Bit).

- Überarbeitung des im Übersetzer verwendeten Systems zur Behandlung von Generics. Viele der Probleme, die mit Generics im Code bestehen, könnten dadurch möglicherweise behoben und somit auch auf einige Hints verzichtet werden.
- Beheben der Probleme aus Kapitel 5.3 und 5.2 (nur Punkt 2); eventuell auch Einführen eines Kompatibilitätsmodus für den javac.
- Casts in Funktionsaufrufen reduzieren indem zusätzliche Enter-Funktionen erstellt werden, die den erforderlichen Rückgabetyp besitzen. Gemeint sind Fälle wie z.B.:

  ```
  ((B)new B().enter_A([..])).do_F().exit_B()
  ```

- Bei virtuellen Patterns, die in Funktionen übersetzt werden, wird immer `super.func()` aufgerufen. Bei der ersten Bindung ist dies aber meist nicht notwendig, da in der aufgerufenen Funktion oft gar kein sinnvoller Code enthalten ist. Solche Anweisungen sollten deshalb entfernt werden.
- Da für jede Funktionsausführung in BETA ein Objekt erzeugt werden muss, wurde der BETA-Code häufig dafür optimiert. So wurden von Patterns, die eigentlichen reinen Funktionscharakter haben, Instanzen in Variablen abgelegt, um diese mehrfach zu verwenden. Bei der Übersetzung ist diese Optimierung jedoch störend, weil sie die eigentlich mögliche Übersetzung in eine Funktion verhindert. Wenn der Übersetzer solche Fälle erkennen würde (eventuell mit Hilfe eines Hints), könnte er die Optimierung entfernen und diese Patterns doch in Funktionen übersetzen. Zum Entfernen der Optimierung müsste lediglich die Variable gelöscht und ihre Verwendungen durch direkte Pattern-Aufrufe ersetzt werden.

Anhang

A Benutzerdokumentation

1 Aufruf

Der Aufruf hat in der Regel folgende Form:

```
java -jar BETA2Java.jar [<Optionen>] <Eingabedatei>
```

Eingabedatei ist dabei die Datei, die das `program:Descriptor` Fragment des zu übersetzenden Programms enthält.
Alle Ausgaben werden in das Verzeichnis ./JavaOut/ geschrieben.

Wichtig: Es werden im aktuellen Verzeichnis drei temporäre Verzeichnisse (UnFragmentOut, SyntaxParserOut, SyntaxAnalyzerOut) angelegt. Nach einer erfolgreichen Übersetzung werden diese wieder gelöscht unabhängig davon, ob sich auch andere Inhalte in diesen Verzeichnissen befinden.

Hinweise

- Es wird Java 6 zum Ausführen benötigt.
- Der erzeugte Code kann vorerst nur mit Eclipse oder dem gcj compiliert werden. Zur Compilierung mit dem javac müssen zunächst einige Änderungen vorgenommen werden. Hinweise dazu finden sich in Abschnitt 6.
- Es ist normal, dass der erzeugte Quellcode sehr viel Warnungen verursacht besonders dort, wo Generics verwendet werden.
- Wird die Übersetzung mit unterschiedlichen Eingaben oder Optionen mehrfach in das gleiche Verzeichnis ausgeführt, sollte der Inhalt vorher gelöscht werden. Erfolgt dies nicht entstehen möglicherweise Java-Fehler in alten Dateien, die nicht überschrieben wurden.
- Die verfügbaren Optionen bilden eine Zusammenfassung der Einstellungen der einzelnen Teilschritte. Für detailliertere Einstellungsmöglichkeiten könnten die Schritte auch einzeln ausgeführt werden. Die Dokumentationen dazu liegen den Quellen des Übersetzers bei.

2 Optionen

Allgemein

`-o <Verzeichnis>`
Kann zum Angeben eines anderen Ausgabeverzeichnisses verwendet werden.
Der Standardwert ist „./JavaOut/“

`-l <0-4>`
Stellt die Anzahl der ausgegebenen Meldungen ein:
0 = alle Meldungen
1 = unterdrückt detaillierte Fortschrittsmeldungen
2 = unterdrückt Verarbeitungsinformationen
3 = unterdrückt Warnungen **(Standard)**
4 = keine Meldungen

`-s <Datei>`
Ermöglicht die Angabe einer Liste mit Namen von Dateien oder Paketen, die bei der Ausgabe nicht überschrieben werden sollen. Jede Zeile in der angegebenen Datei wird dabei als ein Name betrachtet.

`-p[=<Package Name>]`
Ermöglicht es das Package zu spezifizieren, in dem alle übersetzen Java-Klassen erzeugt werden sollen. Standardmäßig wird der Name der Eingabedatei verwendet; eine Verwendung ohne Package kann erzwungen werden indem nur `-p` angegeben wird.

Übersetzung

`-O <0-2>`
Bestimmt die Stufe der Optimierung, die der Übersetzer verwenden soll.
0 = Keine Optimierungen durchführen; verwendet für alle Elemente die allgemeine Übersetzung. Diese Einstellung ist nur zum Vergleichen gedacht. Der erzeugte Code wird dadurch falsch. Vor allem weil die External-Implementierung auf Stufe 1 aufbauen (gelegentlich auch Stufe 2).
1 = Basisoptimierungen, die immer verwendet werden sollten. Enthält u.a. das Ersetzen von einzelnen Funktionen und die automatische Zusammenfassung häufiger Aufrufe.
2 = Alle verfügbaren Optimierungen durchführen. Kann meistens verwendet werden, in einigen speziellen Fällen ist es aber möglich, dass Fehler entstehen. Dann sollte Stufe 1 verwendet werden. **(Standard)**

`-S <0-3>`
Bestimmt die Aggressivität des Übersetzers beim Entfernen von nicht verwendeten BETA-Elementen.
0 = Nichts entfernen. **(Standard)**
1 = Alle Inner-Aufrufe entfernen, die nicht verwendet sind, da es keinen Do-Teil gibt, der ihre Funktion erweitert. Durch diese Option können möglicherweise mehr Patterns in Funktionen übersetzt werden.
2 = Entfernen von unbenutzten Patterns/Attributen im Bereich der BetaLib.
3 = Entfernen von unbenutzten Patterns/Attributen im gesamten Code.

`-B <0-4>`
Bestimmt wie viel zusätzlicher Code erzeugt wird, der für die korrekte Funktion nicht nötig ist aber den Stil verbessert.
0 = Nichts.
1 = Die Annotation `@Override` wird bei allen entsprechenden Funktionen eingefügt. Weiterhin werden – überall wo möglich – die `{}` - Blöcke reduziert. **(Standard)**
2 = Es werden zusätzlich Konstruktoren erzeugt, die die Sichtbarkeit einschränken, um versehentliche spätere Verwendung zu verhindern.
3 = Die Annotation `@SuppressWarnings("unchecked")` wird bei der Verwendung von nicht typisierten Generics eingefügt, wenn diese durch den Übersetzungsprozess als sicher betrachtet werden können. **(Empfohlen)**
4 = Die Annotation `@SuppressWarnings("unused")` wird für alle nicht verwendeten lokalen Variablen erzeugt. Dabei kann es jedoch zu unnötigen/falschen Markierungen kommen.

`-C <0-2>`
Gibt an, ob zusätzliche Kommentare für übersetzte Elemente erzeugt werden sollen. Diese enthalten Angaben über die BETA-Quelle und den Zweck oder Hinweise zur Verwendung.
0 = Keine. **(Standard)**
1 = Wichtige Zweckinformationen und Quellen für alle Pattern-Übersetzungen.

2 = Alle verfügbaren Quellen und Zweckinformationen. Dies ist nicht zu empfehlen, da Unmengen von sehr ähnlichen Kommentaren erstellt werden, deren Informationsgehalt gering ist.

`-c`
Aktiviert das „Schätzen“ von Javadoc-Kommentaren: jeder Kommentar, der direkt vor einem Pattern steht, wird als Javadoc-Kommentar verwendet.

`-f`
Schaltet die Initialisierung von lokalen Variablen ab, führt zu falschem Code! Kann verwendet werden, um manuell nur die Variablen zu initialisieren, die wirklich eine Initialisierung benötigen. (Der Übersetzer prüft nicht, ob die Initialisierung nötig ist.)

Entwicklung

`-d`
Aktiviert alle Debug-Ausgaben und verhindert das Löschen der temporären Verzeichnisse.

`-x <1-3>`
Deaktiviert das Löschen der temporären Ergebnisse und führt – falls möglich – nicht alle Teilschritte aus.
1 = UnFragment nicht ausführen.
2 = SyntaxParser nicht ausführen.
3 = SyntaxAnalyzer nicht ausführen.

3 Optionale Bibliothek

Die übersetzte BetaLib benutzt in einigen Fällen eine native Bibliothek. Die Bibliothek ist für das Ausführung des übersetzten Programms nicht zwingend nötig. Die Verwendung ist aber zu empfehlen, wenn korrekte Funktionalität gewünscht ist. Falls nicht verwendet, werden Warnmeldungen auf der Konsole ausgegeben und die Funktionen, soweit wie möglich, mit Java-API-Aufrufen ersetzt.
Einem übersetzten Java-Programm kann daher optional diese Bibliothek zur Verfügung gestellt werden. Der Code befindet sich in lib/ und 3 compilierte Versionen für Windows32, Linux32 und Linux64 ebenfalls im Verzeichnis lib/.

Behandelt durch die Bibliothek werden derzeit:
- `Stream.putreal`
- `ufile.hardLinkTo/From` (Teil des Trans-Codes, die Ersatzimplementierung ohne Bibliothek bzw. die Windows-Bibliothek besteht aus der Rückgabe, dass der Link nicht erzeugt werden konnte)

4 Hints

Hints sind spezielle Kommentare im BETA-Code, die vom JavaConverter ausgewertet werden und die Übersetzung beeinflussen. Ihr Zweck liegt vor allem in der Korrektur von Übersetzungen, die der JavaConverter nicht allein vornehmen kann.

Aufbau

Der Aufbau ist sehr streng. Abweichungen jeder Art vom exakten Aufbau führen zu Abstürzen bei der Übersetzung oder dazu, dass der Hint nicht erkannt wird.
Grundsätzlich hat ein Hint folgenden Aufbau:

```
(*JAVAHINT:Name|param1|param2|param3| ... |paramX*)
...
(*ENDHINT:Name*)
```

Ein Hint ist also ein Kommentar, damit wird die Bedeutung des Codes für den BETA Compiler nicht verändert. Der erste Teil `(*JAVAHINT:` dient der Erkennung im Code, danach folgt mit `Name` die Bezeichnung des gewünschten Hints. Es wird zwischen Groß- und Kleinschreibung unterschieden.

Abhängig vom Typ kann es verschiedene Parameter geben. Vor jedem Parameter steht ein `|`. Der Wert des Parameters besteht aus allen Zeichen zwischen seinem `|` und dem nächsten Parameter oder dem Ende des Kommentars. Innerhalb eines Parameters sind alle Zeichen, außer Zeilenumbrüche und den verwendeten Symbolen `|` und `*)` erlaubt.

Hints gelten immer für einen bestimmten Bereich, das Kommentar `ENDHINT` markiert das Ende dieses Bereiches. Tritt innerhalb des Bereiches eines Hint ein weiterer Hint auf, so überschreibt dieser die Gültigkeit des äußeren Hints. Der Übersetzer kann also an einer Stelle des Quelltextes maximal einen Hint als aktiv betrachten.

Hinweise

- Aufgrund des Verhaltens der Hints sollten diese immer nur für einzelne Zeilen verwendet werden z.B.

  ```
  (*JAVAHINT:WITHOUTGENERIC*)
  currentflist: ^btree;
  (*ENDHINT:WITHOUTGENERIC*)
  ```

- Beim Einsatz von Hints ist zu beachten, dass sich beim Parsen der Quelltexte die Position von Kommentaren ändern kann. Bei ungünstiger Positionierung kann dies dazu führen, dass ein Hint wirkungslos oder fehlerhaft wird. Das häufigste Problem ist:

  ```
  (if t##<=ws.todoHtaccess## then
  (*JAVAHINT:AUTOCASTFUNC|0|Insert|Websystem.TodoHtaccess*)
  t[]->ws.htl.insert
  (*ENDHINT:AUTOCASTFUNC*)
  else t if);
  ```

  Hier verschiebt sich der Hint wegen des fehlenden – da nicht nötigem – „;“ über die Zeile:

  ```
  (if t##<=ws.todoHtaccess## then
  (*JAVAHINT:AUTOCASTFUNC|0|Insert|Websystem.TodoHtaccess*)
  (*ENDHINT:AUTOCASTFUNC*)
  t[]->ws.htl.insert
  else t if);
  ```

  Dadurch hat der Hint keine Funktion mehr.
- Hints sollten gründlich geprüft werden. Ein falscher Einsatz kann leicht zu Abstürzen des Übersetzers führen. Der Übersetzer versucht Hinweise zu geben, wenn ein Hint-Fehler auftritt. Dies ist aber nicht immer möglich und meistens ungenau. Der Übersetzer prüft außerdem nicht die Vollständigkeit oder Korrektheit der Parameter. Fehler dieser Art führen mit großer Sicherheit zu Abstürzen mit unklaren Fehlermeldungen.

Tipp: Ideal ist es, nur wenige Hints einzufügen und die Übersetzung mit diesen Änderungen zu testen. Auf diese Weise behält man mögliche Fehlerquelle im Auge.

Verfügbare Hints

`WITHOUTGENERIC`
keine Parameter
Betrifft Attribute und sorgt dafür, dass Attribute, die innerhalb des Hints stehen, keinerlei Generic Bindungen erhalten, also nicht typisiert sind.
(Wird vom FixGenericsExtendsVisitor verarbeitet.)

`AUTOCASTFUNC`
Parameter: `Index`, `Name`, `Klasse`
Erzwingt einen bestimmten Cast bei einem Funktionsaufruf.
Es wird nach dem Funktionsaufruf gesucht, der auf `Name` endet. Der `Index`-te Parameter in diesem Aufruf wird auf `Klasse` gecastet, der Name von `Klasse` kann vollständig qualifiziert sein.
(Wird vom AutoCastVisitor verarbeitet.)

`AUTOCASTCASTASGENRIC`
Parameter: `Typ`
Die Erzeugung von in BETA impliziten Casts übergeht alle Stellen, an denen der Typ auf den zu Casten wäre ein Generic ist. Durch diesen Hint kann ein solcher Cast auf `Typ` erzwungen werden.
(Wird vom AutoCastVisitor verarbeitet.)

`TYPEREDEF`
Parameter: `Var`, `Typ`
Die Variablenverwendung namens `Var` wird nicht als der Typ bearbeitet, der automatisch erkannt wurde, sondern als der im Parameter angegebene `Typ`.
Sinnvoll ist dies dann, wenn aufgrund der Verwendung von Generics der automatisch erkannte Typ an dieser Stelle falsch ist.
(Wird bei der Anweisungskonvertierung verarbeitet.)

`IGNORE`
Parameter: keine
Alle Teile des BETA-Codes, die von diesem Hint eingeschlossen sind, werden in der Übersetzung vollständig ignoriert. Falls nur der Anfang eines Blockes im `IGNORE` liegt, sein Ende aber nicht, wird trotzdem der gesamte Block ignoriert. Werden durch diesen Hint-Deklarationen versteckt, die verwendet sind, führt das zu einem Fehler in der Übersetzung.
(Wird vom IngoreHintVisitor verarbeitet.)

`IGNORESUPER`
Parameter: keine
Ignoriert den Vater eines Patterns bei der Übersetzung. Sinnvoll dann, wenn das Vater-Pattern nach der Übersetzung keine Funktion mehr hat, z.B. `External`. Teile des Vaters, die im Pattern verwendet werden, müssen mit `IGNORE` entfernt werden. Ansonsten entsteht bei der Übersetzung ein Fehler. Das Pattern muss mit dem Hint immer vollständig umschlossen sein, sonst kann der Übersetzer es nicht korrekt verarbeiten:

```
(*JAVAHINT:IGNORESUPER*)
pat: external (#
     ...
#);
(*ENDHINT:IGNORESUPER*)
```

Aufgrund des allgemeinen Verhaltens von Hints kann es in diesem Fall aber vorkommen, dass Subpatterns betroffen sind. Um dies zu verhindern, kann ein bedeutungsloser Hint eingefügt werden:

```
(*JAVAHINT:IGNORESUPER*)
pat: external (#
     (*JAVAHINT:NIX*)
     other: dontignore (#
        ...
     #);
     (*ENDHINT:NIX*)
     ...
#);
(*ENDHINT:IGNORESUPER*)
```

(Wird vom ConvertPatternVisitor verarbeitet.)

`IMPORT`
Parameter: `Name`
Sucht nach einer konvertierten Klasse namens `Name` im gesamten Baum und fügt diese in der aktuellen Datei als Import ein. `Name` kann dabei nur der Name der Klasse sein, keine Qualifikation. Dies ist nützlich, wenn in einem `JAVACODE` Block konvertierte Klassen verwendet werden.
(Wird vom GenerateImportsVisitor verarbeitet.)

`GENERIC`
Parameter: `Name`
Veranlasst den Übersetzer das angegebene Element `Name` als Generic zu übersetzen; wird nur für virtuelle Patterns berücksichtigt.
z.B.

```
(*JAVAHINT:GENERIC|element*)
element :< Object;
(*ENDHINT:GENERIC*)
```

Alle nachfolgenden Bindungen des Elements werden dann ebenfalls als Generic übersetzt.
Dieser Hint ist vorsichtig einzusetzen. Eine ungünstige Anwendung des Hints kann dazu führen, dass der entstehende Java-Code Fehler enthält. Im Zweifelsfall sollte auf die Übersetzung als Generic verzichtet werden. Der Übersetzer prüft nicht, ob das virtuelle Pattern tatsächlich nur wie ein Generic verwendet wird (also nur als Typ). Wird das Pattern auch anders verwendet, entsteht mit Sicherheit ein fehlerhafter Code.

`BINDGENERIC`
Parameter: `Generic`, `Typ`
Eine im Bereich des Hints auftauchende Generic-Verwendung namens `Generic` wird mit der Klasse `Typ` typisiert, anstatt mit der vom Übersetzer erkannten Klasse. Typ darf hier nur ein einfacher Name einer Klasse sein, keine Qualifikation.
(Wird vom FixGenericsExtendsVisitor verarbeitet.)

Spezielle Hints

`JAVACODE`
Die Syntax diese Hints unterscheidet sich von den anderen:

```
(*JAVACODE: ... *)
oder
(*JAVACODE: ... :JAVACODE*)
```

Diese Angabe führt dazu, dass an genau dieser Stelle in der erzeugten *.java Ausgabe der Text eingefügt wird, der nach dem ':' steht. Es finden dabei keine weiteren Bearbeitungen oder Prüfungen des eingefügten Textes statt. Zulässig sind in diesem Fall auch Zeilenumbrüche innerhalb des Arguments.
(Wird vom JavaHintsVisitor verarbeitet.)

`JAVACLASSCODE`
Spezialform von `JAVACODE`. Der Code wird nicht an der Stelle des Kommentars eingefügt, sondern in den Kopf der Datei, damit ist es möglich spezielle Imports oder private Klassen einzufügen.
Mehrere Hints dieser Art sollten nicht in einem Pattern verwendet werden, da ihre Reihenfolge in der Ausgabe nicht klar definiert werden kann.

5 Zeilenersetzung

In der Optimierungsoption `-O 1` (oder höher) ist die Ersetzung von einzelnen Zeilen durch Java Anweisungen enthalten. Die zu ersetzenden Zeilen können dabei über die Datei „replaceLineData.xml“, die im aktuellen Verzeichnis gesucht wird, bestimmt werden. Ist die Datei nicht vorhanden, wird eine interne Standarddatei verwendet. Die Datei hat dabei folgendes Format:

```
<replaceLineData>
     <search name="name1.name2.name3">
        <search name="name1.name2.name3.$enter_name3">
           <replace name="name1.name2.name3.$do_F"
              with="funktion"
           />
        </search>
        <search ...
     </search>
     <search ...
</replaceLineData>
```

Jeder `<search>` Knoten gibt dabei mit einem vollständig qualifizierten Namen ein zu suchendes Element an. Dabei darf das Package, in das alle Klassen geschoben werden, nicht angegeben werden! Das obige Beispiel würde die Anweisung:

```
name3.enter_name3().do_F();
oder
new name3.enter_name3().do_F();
```

ersetzen mit:

```
funktion()
```

Tag Details

`<search name=“name.name“ [assub=“yes“]>:`
`Name` gibt wie schon beschrieben den vollständig qualifizierten Namen an, der zu suchen ist. Innerhalb des Attributs `name` können besondere Wörter auftauchen: „$enter_“, „$exit_“, „$do_F“ und „$create_“ diese werden durch die vom Übersetzer verwendeten Präfixe/Namen ersetzt:

„enter_“, „exit_“, „do_F“ und „create_“. Ziel ist es, dass bei einer eventuellen Änderung dieser Namen die Datei nicht verändert werden muss.
Ist das optionale Attribut `assub` angegeben, verändert sich die Bedeutung von `name`. Statt eines vollständigen Namens kann dann ein Name relativ zum `<search>` Tag verwendet werden, das auf der nächst höheren Ebene liegt.

`<replace name=“name.name“ with=“funktion“ [assub=“yes“] [unRollText=“yes“] [replaceIfMore=“yes“]>:`
`Name` und `assub` verhalten sich so wie bei `<search>`. `With` gibt einen beliebigen Text an, der als Übersetzung verwendet wird. Es wird immer ein `()` angehängt, in das alle von der bisherigen Anweisung verwendeten Parameter eingefügt werden.
Mit `unRollText` kann angegeben werden, dass alle Literale vom Typ Text, die normalerweise als `new Text(“text“)` übersetzt sind, reduziert werden auf `“text“`.
Ist `replaceIfMore` angegeben wird eine Ersetzung auch dann durchgeführt, wenn die Anweisung nach dem Fund von `<replace>` noch weitere `.irgendwas` enthält.

`<param pos=“Zahl“>:`
Subknoten für `<replace>`, kann verwendet werden, um die Reihenfolge von Parametern zu verändern. z.B:

```
<replace ...>
     <param pos=“3“/>
     <param pos=“2“/>
     <param pos=“1“/>
</replace>
```

kehrt die Reihenfolge der Parameter um.

6 Anpassungen für den javac

Der javac weist scheinbar einige Fehler auf, er kann den von BETA2Java erzeugten Code nicht compilieren. Es ist aber möglich, den erzeugten Code manuell zu verändern, um dieses Probleme zu umgehen. Im Folgenden sind Hinweise zur Korrektur der bekannten Fälle aufgelistet.

1. *cannot select from a type variable*
   Tritt auf, wenn über eine Generic auf einen Subklasse zugegriffen wird:

   ```
   class A<T extends Typ> {
      T t = new Typ();
      T.SubTyp s = t.new SubTyp();
   }
   ```

   Kann korrigiert werden, indem stattdessen der Typ verwendet wird, der nach `extends` steht:

   ```
   Typ.SubTyp = ((Typ)t).new SubTyp();
   ```

2. *cannot find symbol*
   symbol : constructor BetaClass([..], [..])
   Tritt auf, wenn ein BetaClass-Objekt bei der Initialisierung bereits die Information über das zugehörige umgebende Objekt erhält und dieses umgebende Objekt aus einer generischen Klasse stammt:

   ```
   class A<T> {
      class Sub { }
   ```

```
   ... new BetaClass(Sub.class, this.A) ...
}
```

Das kann korrigiert werden indem man als Ersatz den Reflection-Konstruktor verwendet, also die Bestimmung des umgebenden Objektes auf die Laufzeit verschiebt:

```
... new BetaClass(new Sub()) ...
```

3. *type parameter [..] is not within its bound*
   Tritt in zwei verschiedenen Fällen auf, kann jeweils dadurch behoben werden, dass die Generic-Bindung gelockert wird (mit einem Wildcard). Es sind dann aber meist noch weitere Casts nötig. Der erste Fall stammt aus der BetaLib:

```
public class DoEnter<containerType extends Container<element>>
   extends Object_B { [..] }
//Fehler: type parameter containerType is not within its bound
public class DoEnter<containerType extends ArrayContainer<element>>
   extends Container<element>.DoEnter<containerType> { [..] }
```

   Das Problem kann, unter Verlust der Typsicherheit, korrigiert werden durch:

```
public class DoEnter<containerType extends Container<?>>
   extends Object_B { [..] }
```

4. Die zweite Variante von 3. tritt auf, wenn virtuelle Patterns in Generics übersetzt werden. Allerdings nur, wenn sie auf eine spezielle Weise deklariert wurden:

```
pat: (#
   p:< (# ... #);
   (* oder  *)
   p :< typ;
   typ: (# ... #);
#);
```

   In der Übersetzung erhält man dann:

```
public class pat<p extends pat<p>.typ> extends Object_B { [..] }
```

   Dies ist nicht falsch, das Problem entsteht erst bei der Verwendung:

```
pat my = new pat<p.typ>();
```

   Es ist hier natürlich unmöglich `p.typ` vollständig zu typisieren, da man eine Endlosschleife erhält mit `p<p.typ>.typ`. Durch Ändern der Deklaration auf `<p extends pat.typ>` kann das Problem im javac umgangen werden.

B Übersetzungsbeispiele

B.1 Zwei Varianten zur Übersetzung von Enter/Exit

Als Grundlage dient der folgende BETA-Code der zwei einfache Patterns definiert und diese in verschiedenen Varianten verwendet.

```
1  main: (#
2
3     a: (#
4        i : @integer;
5        enter i
6        do
7           i + 1 -> i;
8           inner;
9        exit i
10    #);
11
12    b: a(#
13       j: @integer;
14       enter j
15       do
16          j + 1 -> j;
17          inner;
18       exit j
19    #);
20
21    x, y: @integer;
22    myA: ^A;
23
24    do
25       (5, 4) -> b;
26       b -> (x, y);
27       x -> a -> y;
28       &b[] -> myA[];
29       y -> myA;
30 #);
```

Zu beachten ist in Zeile 28, dass eine Instanz des Pattern `b` erstellt wird. Diese erwartet eigentlich zwei Parameter aber da der Typ zur Compilezeit nur als `a` bekannt ist, kann lediglich ein Parameter angegeben werden.

Wie in Kapitel 3.3.2 beschrieben können zwei Varianten verwendet werden. Nachfolgend ein Ansatz zur Übersetzung nach Variante 1, dieser Code ist jedoch nicht korrekt!

```
1  public static class Main {
2     public static class A  {
3        protected void inner_0() {
4        }
5        protected void code() {
6           i = i + 1;
7           inner_0();
8        }
9        public int i;
10       public final int do_F(int param_0) {
11          i = param_0;
12          code();
```

```
13          return i;
14       }
15       public final void do_F(int param_0) {
16          i = param_0;
17          code();
18       }
19       public final int do_F() {
20          code();
21          return i;
22       }
23       public final void do_F() {
24          code();
25       }
26    }
27    public static class B extends A {
28       protected void inner_1() {
29       }
30       public int j;
31       public final Exit2<Integer, Integer> do_F(int param_0,
32             int param_1) {
33          i = param_0;
34          j = param_1;
35          code();
36          Exit2<Integer, Integer> exit =
37             new Exit2<Integer, Integer>();
38          exit.v1 = i;
39          exit.v2 = j;
40          return exit;
41       }
42       public final void do_F(int param_0, int param_1) {
43          i = param_0;
44          j = param_1;
45          code();
46       }
47       public final Exit2<Integer, Integer> do_F() {
48          code();
49          Exit2<Integer, Integer> exit =
50             new Exit2<Integer, Integer>();
51          exit.v1 = i;
52          exit.v2 = j;
53          return exit;
54       }
55       public final void do_F() {
56          code();
57       }
58    }
59    public int x;
60    public int y;
61    public A myA;
62    public void do_F() {
63       new B().do_F(5, 4);
64       {
65          Exit2<Integer, Integer> resValue_0 = new B().do_F();
66          x = resValue_0.v1;
67          y = resValue_0.v2;
68       }
69       y = new A().do_F(x);
70       myA = new B();
71       myA.do_F(y);
72    }
73 }
```

Der Übersetzer verwendet die Variante 2 wie folgt:

```
1  public class Main extends Object_B {
2    public class A extends Object_B {
3       protected void inner_0() {
4       }
5       public int i;
6       public final A enter_A(int param_0) {
7          i = param_0;
8          return this;
9       }
10      public A do_F() {
11         i = i + 1;
12         inner_0();
13         return this;
14      }
15      public final int exit_A() {
16         return i;
17      }
18   }
19   public class B extends A {
20      protected void inner_1() {
21      }
22      public int j;
23      public final B enter_B(int param_0, int param_1) {
24         super.enter_A(param_0);
25         j = param_1;
26         return this;
27      }
28      protected final void inner_0() {
29         j = j + 1;
30         inner_1();
31      }
32      protected final Exit2<Integer, Integer> exit_B(
33            Exit2<Integer, Integer> exit) {
34         exit.v1 = super.exit_A();
35         exit.v2 = j;
36         return exit;
37      }
38      public B do_F() {
39         super.do_F();
40         return this;
41      }
42      public final Exit2<Integer, Integer> exit_B() {
43         return exit_B(new Exit2<Integer, Integer>());
44      }
45   }
46   public int x;
47   public int y;
48   public A myA;
49   public Main do_F() {
50      new B().enter_B(5, 4).do_F();
51      {
52         Exit2<Integer, Integer> resValue_0 =
53            new B().do_F().exit_B();
54         x = resValue_0.v1;
55         y = resValue_0.v2;
56      }
57      y = new A().enter_A(x).do_F().exit_A();
58      myA = new A();
59      myA.enter_A(y).do_F();
```

```
60      return this;
61   }
62 }
```

B.2 If

BETA-Quelle:

```
1  (#
2    c: @Char;
3    do
4       keyboard.get -> c;
5
6       (*Erweitertes If, das als Switch übersetzt werden kann.*)
7       (if c
8          // 'a'
9          // 'e'
10         // 'u'
11         // 'i'
12         // 'o' then 'Vokal' -> putline;
13         // '?' then 'Frage' -> putline;
14         // '@' then 'At' -> putline;
15         else 'Unbekannt' -> putline;
16      if);
17
18      (*einfaches If*)
19      (if c = 'e'
20         then
21            'Ist ein e' -> putline;
22         else
23            'Ist kein e' -> putline;
24      if);
25
26      (*Erweitertes If, das nicht als Switch übersetzt werden kann.*)
27      (if true
28         // c < 33 then 'Steuerzeichen' -> putline;
29         // (c > 57) and (c < 64)
30         // c < 48 then 'Zeichen' -> putline;
31         // (c > 64) and (c < 91) then 'Großbuchstabe' -> putline;
32         else 'Nicht behandelt' -> putline;
33      if);
34 #)
```

Übersetzung:

```
1  public class Main extends Object_B {
2    public char c;
3    public Main do_F() {
4       // Erweitertes If, das als Switch übersetzt werden kann.
5       c = Betaenv.keyboard.create_Get().use_Get();
6       switch (c) {
7       case 'a':
8       case 'e':
9       case 'u':
10      case 'i':
11      case 'o':
12         System.out.println("Vokal");
13         break;
```

```
14      case '?':
15         System.out.println("Frage");
16         break;
17      case '@':
18         System.out.println("At");
19         break;
20      default:
21         System.out.println("Unbekannt");
22         break;
23      }
24      // einfaches If
25      if (c == 'e') {
26         System.out.println("Ist ein e");
27      } else {
28         System.out.println("Ist kein e");
29      }
30      // Erweitertes If, das nicht als Switch übersetzt werden kann.
31      {
32         boolean resValue_0 = true;
33         if (resValue_0 == (c < 33)) {
34            System.out.println("Steuerzeichen");
35         } else if ((resValue_0 == ((c > 57) && (c < 64)))
36                || (resValue_0 == (c < 48))) {
37            System.out.println("Zeichen");
38         } else if (resValue_0 == ((c > 64) && (c < 91))) {
39            System.out.println("Gro\u00DFbuchstabe");
40         } else {
41            System.out.println("Nicht behandelt");
42         }
43      }
44      return this;
45   }
46}
```

B.3 For

BETA-Quelle:

```
1 (#
2    do
3       (* Schleife ohne Namen für die Laufvariable. *)
4       (for 20 repeat
5          newline;
6       for);
7
8       (* Normale Schleife, benötigt die Laufvariable in einem
9          Inline. *)
10      (for i:20 repeat
11         'z' -> (&Text[]).find(#
12            do
13               i -> putint;
14         #);
15      for);
16
17      (* Schleife mit Wert, der vorher berechnet werden muss. *)
18      (for i:10 * 3 repeat
19         i -> putint;
20      for);
21#)
```

Übersetzung:

```
1 public class Main extends Object_B {
2   public Main do_F() {
3      // Schleife ohne Namen für die Laufvariable.
4      for (int resValue_0 = 1; resValue_0 <= 20; resValue_0++) {
5         System.out.println();
6      }
7      // Normale Schleife, benötigt die Laufvariable in einem Inline.
8      for (int at_i = 1; at_i <= 20; at_i++) {
9         final int i = at_i;
10        {
11           class inlineClass_2 extends Text.Find {
12              public  inlineClass_2(Text _init) {
13                 // Konstruktor da umliegendes Objekt info nötig!
14                 _init.super();
15              }
16              protected final void inner_0() {
17                 System.out.print(i);
18              }
19           }
20           inlineClass_2 resValue_3 = new inlineClass_2(new Text());
21           resValue_3.use_Find('z');
22        }
23     }
24     // Schleife mit Wert, der vorher berechnet werden muss.
25     for (int i = 1 , maxValue_1 = 10 * 3; i <= maxValue_1; i++) {
26        System.out.print(i);
27     }
28     return this;
29  }
30}
```

B.4 Problem bei Inner-Position

Angenommen wird folgende Bindung einer virtuellen Funktion, in der der Enter-Wert vor dem Inner- Aufruf verändert wird:

```
func:< (#
     in : @integer;
     enter in;
     do
        5 -> in;
        inner;
#);

...
func::< (#
     do
        (if in = 5 then
           'Ist 5' -> putline;
        if)
        inner;
#);
```

Compiliert man diesen Code mit BETA, erhält man erwartungsgemäß immer als Ausgabe `'Ist 5'`, da der Enter-Wert `in` verändert wird. Übersetzt man dieses virtuelle Pattern in eine Funktion, würde man folgendes erhalten:

```
1 void func(int in) {
2   in = 5;
3 }
4 ...
5 @Override
6 void func(int in) {
7   super.func(in);
8   if (in == 5) {
9      new Putline().enter_Putline(new Text(„Ist 5“)).do_F();
10  }
11}
```

Hierbei erhält man die Ausgabe `'Ist 5'` nur, wenn als Parameter eine 5 übergeben wird. Der Parameter wird zwar durch die Funktion in Zeile 1 verändert. Aber die Änderung geht verloren, da die Parameter von Zeile 1 und Zeile 6 zwei voneinander unabhängige lokale Variablen sind (im Gegensatz zu `in` in Zeile 2 des BETA-Code). Der Code verhält sich also anders als das BETA-Original. Daher ist es nicht möglich Patterns, bei denen die Enter-Werte vor dem Inner-Aufruf geschrieben werden, in Funktionen zu übersetzen.

B.5 Pattern das als Koroutine verwendet wird

BETA-Quelle siehe Kapitel 3.9.
Übersetzung:

```
1 public class Main extends Object_B {
2
3   public static class Coroutine extends Object_B
4      implements BetaCoroutineTask {
5
6      public int l;
7      public int sum;
8      public final Coroutine enter_Coroutine(int param_0) {
9         System.out.print(l = param_0);
10        return this;
11     }
12     public Coroutine run() {
13        System.out.print(" - Start ");
14        loop : do {
15           {
16              sum = sum + l;
17              if (sum <= 10) {
18                 BetaCoroutine.suspendCoroutine();
19                 System.out.print(" - Resume ");
20                 continue loop;
21              }
22           }
23           break loop;
24        } while (true);
25        System.out.println();
26        System.out.print("Ende ");
27        return this;
28     }
```

```
29      public final int exit_Coroutine() {
30         return sum;
31      }
32      private BetaCoroutine coroutineVar_;
33      public BetaCoroutine getCoroutine() {
34         return coroutineVar_;
35      }
36      public void setCoroutine(BetaCoroutine arg) {
37         coroutineVar_ = arg;
38      }
39      @Override
40      public Coroutine do_F() {
41         if (coroutineVar_ == null) {
42            return run();
43         }
44         coroutineVar_.resumeCoroutine();
45         return this;
46      }
47      public Coroutine makeCoroutine() {
48         BetaCoroutine.initCoroutine(this);
49         return this;
50      }
51   }
52   public final Coroutine myCoru = new Coroutine().makeCoroutine();
53   public boolean isEnd;
54   @Override
55   public Main do_F() {
56      for (int i = 1; i <= 20; i++) {
57         System.out.print(myCoru.enter_Coroutine(i).do_F().
58            exit_Coroutine());
59         System.out.println();
60      }
61      return this;
62   }
63   public static void main(String[] args) {
64      Betaenv.InitEnv();
65      Betaenv.noOfArguments = args.length + 1;
66      Betaenv.argumentsList = new Text[Betaenv.noOfArguments];
67      Betaenv.argumentsList[0] = new Text("ProgramName");
68      for (int i = 0; i < args.length; i++) {
69         Betaenv.argumentsList[i + 1] = new Text(args[i]);
70      }
71      new Main().do_F();
72      Betaenv.ClearEnv();
73   }
74}
```

C Inhalt der Laufzeitbibliothek im Paket de.teltarif.beta.bTypes

Klasse	Funktion
Exit2 bis Exit10	Container-Objekte um Patterns mit mehreren Rückgabewerten zu realisieren.
BetaArray	Statische Implementierungen der Array Funktionen `.range`, `.new` und `.extend`.
BetaClass	Typ in den alle Pattern-Variablen übersetzt werden; weiterhin Implementierungen der Funktionen zum Vergleich von Pattern-Variablen.
BetaCoroutine	Verwaltet die Ausführung von Koroutinen in Threads.
BetaCoroutineTask	Interface, das jede Klasse, die als Koroutine ausgeführt werden soll, implementieren muss.
BetaLabel-, BetaLeave-, BetaRestarException	Spezielle Exceptions die verwendet werden, um die Programmflusssteuerung mit `leave` und `restart` abzubilden.
BetaLowLevel	Realisiert Low Level-Funktionen zum Lesen aus Integer-Werten.
BetaNative	Wird verwendet um External-Funktionen zu realisieren, die nicht mit Java-API-Funktionen ersetzt werden können; verwendet dazu JINI-Aufrufe einer C++ Bibliothek. Dabei sind die Funktionen allerdings derart implementiert, dass wenn die Bibliothek nicht geladen werden kann, sie soweit wie möglich mit Java simuliert werden.
BetaRuntimeException und BetaNotImplementedException	Fehler innerhalb der Laufzeitbibliothek.

D Liste von nicht zulässigen BETA-Bezeichnern

D.1 Wörter

Folgende Wörter dürfen nicht als BETA-Bezeichner verwendet werden. Um Irritationen zu vermeiden, sollten alle Variationen von Groß- und Kleinbuchstaben ebenfalls nicht verwendet werden. Die Liste basiert auf [Co] Kapitel 3. Einige dieser Namen haben auch in BETA besondere Bedeutung und können deshalb nicht als Bezeichner auftauchen. Der Vollständigkeit halber sind sie trotzdem aufgeführt.

abstract	false	notfiyall	synchronized
boolean	final	null	system
break	finalize	number	thread
byte	finally	object	case
float	package	catch	getclasss
private	throw	char	goto
process	throwable	character	hashcode
protected	throws	class	implements
public	toString	classloader	import
return	transient	clone	instanceof
runnable	true	cloneable	int
runtime	try	integer	void
const	interface	short	volatile
continue	long	static	wait
double	equals	new	extends
notfiy	switch	Object	String
native	this	native	strictfp
for	while	do	if
short	assert		

Wort	Bedeutung
do_F	Funktion, die das Pattern ausführt (bzw. die Klasse in die es übersetzt wurde).
Main	Name des anonymen Patterns, das den Einstiegspunkt in das Programm darstellt.
_init	Name des Parameters von Konstruktoren.
makeCoroutine	Initialisierungsfunktion für Koroutinen.
coroutineVar_	Variable in der der Thread, in dem eine Koroutine ausgeführt wird, abgelegt ist.
exit	Temporäre Variable in Exit-Funktionen.

Sowie alle Klasse der Laufzeitbibliothek wie bereits in Anhang C aufgeführt.

D.2 Präfixe

Die folgenden Präfixe haben spezielle Bedeutung im erzeugten Code und werden vom Übersetzer zur Generierung von Namen verwendet. Zur Vermeidung von Verwirrung und Fehlern sollte kein BETA Bezeichner mit einem dieser Präfixe beginnen.

Präfix	Bedeutung
enter_<Pattern-Name>	Enter-Funktionen
param_<Zahl/Name>	Namen für Funktionsparameter
exit_<Pattern-Name>	Exit-Funktion
inner_<Zahl>	Funktion, die anstelle eines Inner-Aufrufs ausgeführt wird.
createFunc_<Pattern-Name>	Funktion zum Erstellen der Instanz einer Klasse die ein virtuelles Pattern übersetzt.
resValue_<lfd>	Hilfsvariable u.a. für ExitX Objekte und Instanzen von Inline-Klassen.
maxValue_<lfd>	Maximalwert einer For-Schleife
at_<Pattern-Name>	Klassen die nur als ein statisches Objekt verwendet werden (name : `@(# ...#)`)
s_<Name>	Wird für Namen von statischen Funktionen verwendet, wenn es sonst einen Namenskonflikt mit einer normalen Methode gäbe.
use_<Pattern-Name>	Funktion, die die häufigste Kombination von Enter/Do/Exit-Aufrufen eines Pattern enthält.

D.3 Postfixe

Postfix	Bedeutung
<Name>_H	Wird an einen Namen angehängt, wenn er sonst einen anderen lokalen Namen verdecken würde.
<Name>_unused	Wird an `inner_<Zahl>` Funktionen angehängt, wenn es nie zu einem Aufruf kommen kann da kein entsprechendes `INNER;` existiert.

E Performancetests

Zum Testen der Performance wird eine einfache Zeitmessung mittels `System.nanoTime()` durchgeführt und in einer Schleife die zu testende Funktion mehrfach ausgeführt. Dadurch wird ein Mittelwert für die Ausführungszeit berechnet. Verwendet wird für die Tests folgende Klasse:

```
1  public abstract class SimplePerformanceTest {
2
3     public abstract void init(int count);
4     public abstract void operation(int run);
5     public abstract void finish(int count);
6
7     public long runTest(int count, boolean output) {
8        init(count);
9        long start = System.nanoTime();
10       for (int i = 0; i < count; i++) {
11          operation(i);
12       }
13       long end = System.nanoTime();
14       finish(count);
15       long time = end - start;
16       if (output) {
17          System.out.println("Total Time: " + time);
18          System.out.println("Average Time: "+(time/(double)count));
19       }
20       return time;
21    }
22
23
24    public static void simpleTest(SimplePerformanceTest test,
25          int prepareCount, int runCount) {
26       test.runTest(prepareCount, false);
27       System.gc();
28       test.runTest(prepareCount, false);
29       System.gc();
30       test.runTest(runCount, true);
31    }
32 }
```

Um mögliche Optimierungseffekte während des Ablaufes der Testschleife zu vermeiden, wird der Test zunächst „vorgewärmt“ (Zeile 26 und 28). Der Test ist als ungenau zu bewerten. Lediglich deutliche Unterschiede zwischen zwei Tests können als Ergebnis verwendet werden.

E.1 Erzeugung von Exceptions mit und ohne Stacktrace

Verwendet wird folgender Test:

```
1  class ExceptionA extends SimplePerformanceTest {
2
3     private static class TestException extends Exception {
4        public Throwable fillInStackTrace() {
5           return this;
6        }
7     }
8     private Exception[] save;
9
```

```
10   public void finish(int count) {
11      save = null;
12   }
13
14   public void init(int count) {
15      save = new Exception[count];
16   }
17
18   public void operation(int run) {
19      save[run] = new TestException();
20   }
21}
22
23
24class ExceptionB extends SimplePerformanceTest {
25   private static class TestException extends Exception {
26   }
27   private Exception[] save;
28
29   public void finish(int count) {
30      save = null;
31   }
32
33   public void init(int count) {
34      save = new Exception[count];
35   }
36
37   public void operation(int run) {
38      save[run] = new TestException();
39   }
40}
41
42public class ExceptionTest {
43
44   public static void main(String[] args) {
45      System.out.println("Test ohne Stacktrace");
46      SimplePerformanceTest.simpleTest(new ExceptionA(), 100, 10000);
47      System.out.println("Test mit Stacktrace");
48      SimplePerformanceTest.simpleTest(new ExceptionB(), 100, 10000);
49   }
50}
```

Aus den Ergebnissen geht ein deutlicher Geschwindigkeitsgewinn beim Verzicht auf die Erzeugung des Stacktrace hervor:

Versuch	Ohne Stacktrace	Mit Stacktrace
1 (mit JIT)	183.8 ns	2499.5 ns
2 (mit JIT)	178.0 ns	2525.3 ns
3 (ohne JIT)	455.1 ns	2831.6 ns
4 (ohne JIT)	448.6 ns	2752.3 ns

E.2 Funktionsaufrufe mit und ohne final

Verwendet wird folgender Test:

```
1 class FinalA extends SimplePerformanceTest {
2
3    public void func() {
4       int var = 5;
5       var++;
6       if (var == 6) {
7          var = 7;
8       }
9    }
10
11   public void finish(int count) {}
12   public void init(int count) {}
13
14   public void operation(int run) {
15      func();
16   }
17}
18
19class FinalB extends SimplePerformanceTest {
20
21   public final void func() {
22      int var = 5;
23      var++;
24      if (var == 6) {
25         var = 7;
26      }
27   }
28
29   public void finish(int count) {}
30   public void init(int count) {}
31
32   public void operation(int run) {
33      func();
34   }
35}
36
37public class FinalTest {
38   public static void main(String[] args) {
39      System.out.println("Aufruf ohne Final");
40      SimplePerformanceTest.simpleTest(new FinalB(), 1000, 10000000);
41      System.out.println("Aufruf mit Final");
42      SimplePerformanceTest.simpleTest(new FinalB(), 1000, 10000000);
43   }
44}
```

Es lässt sich mit der Verwendeten Methode jedoch nur ein minimaler Unterschied festzustellen:

Versuch	Ohne Final	Mit Final
1 (mit JIT)	11.6 ns	9.6 ns
2 (mit JIT)	12.7 ns	9.6 ns
3 (ohne JIT)	72.2 ns	68.3 ns
4 (ohne JIT)	73.1 ns	68.3 ns

F Abbildungsverzeichnis

G Literaturverzeichnis

[AM02] Anderson P.; Madsen, O. L.: Early experience with language interoperability - porting the BETA language to .NET and Java
http://www.daimi.au.dk/~beta/ooli/JAOO-2002/OOLI.pdf am 18.09.08.

[ASU99] Aho, A. V.; Sethi, R.; Ullman, J. D.: Compilerbau Teil 1, 2. Auflage,
Oldenbourg, 1999.

[Bl01] Bloch, J.: Effective Java,
Addison-Wesley, 2001.

[Co] Cowan, J.: The Loki Project,
http://home.ccil.org/~cowan/loki/ am 18.09.08.

[Fo05] Fowler, M.: Refactoring,
Addison-Wesley, 2005.

[Ga95] Gamma, E.; Johnson, R.; Helm, R.; Vlissides, J: Entwurfsmuster,
Addison-Wesley, 1995.

[LK00] Lassen, H.M.; Falk, K.: OOVM Beta2Java
http://www.daimi.au.dk/~henryml/oovm/report.ps am 18.09.08.

[MI] Mjølner Informatics: Mjølner Informatics History,
http://www.mjolner.com/index.php?id=360&L=1 am 25.09.08.

[MI04a] Mjølner Informatics: BETA Language Modifications - Reference Manual
http://www.daimi.au.dk/~beta/doc/beta/beta-index.html am 23.09.08.

[MI04b] Mjølner Informatics: The Fragment System: Further Specification
http://www.daimi.au.dk/~beta/doc/beta/fragment-index.html am 23.09.08.

[MI04c] Mjølner Informatics: Beta Grammar
http://www.daimi.au.dk/~beta/doc/grammars/beta.html am 23.09.08.

[MI04d] Mjølner Informatics: BETA Language Introduction
http://www.daimi.au.dk/~beta/doc/beta-intro/index.html am 24.09.08.

[MMN93] Madsen, O. L.; Møller-Pedersen, B.; Nygaard, K.: Object-Oriented Programming in the BETA Programming Language,
Addison-Wesley, 1993.

[OOLI] Object-Oriented Language Interoperability,
http://www.daimi.au.dk/~beta/ooli/ am 18.09.08.

[Se05] Sestoft P.: Java performance Reducing time and space consumption
http://www.dina.dk/~sestoft/papers/performance.pdf am 19.09.08.

[Wo06] Wolf, M.: I want coroutines!,
http://blogs.infosupport.com/martinw/archive/2006/04/19/5345.aspx am 25.09.08.

H Inhalt der CD

Inhalt der beigelegten CD:

/BETA2Java – Enthält das Übersetzerprogramm.

/Literatur – Enthält alle elektronisch verfügbaren Dokumente, die im Literaturverzeichnis der Dokumentation angegeben sind. Es wird jeweils der Titel als Datei oder Ordnername verwendet.

/src – Enthält die kompletten Quellen des Übersetzers und zusätzliche Informationen. Alle Quellen enthalten viele Kommentare, die die einzelnen Vorgänge genauer erläutern. Bis auf wenige Ausnahmen handelt es sich dabei (wie auch bei den weiteren Textdokumenten) primär um die Niederschrift von Gedanken. Diese entstanden meist direkt bei der Implementierung und wurden nicht weiter bearbeitet; sie sind deshalb möglicherweise schwer zu lesen.
/src/BETA2Java – Hauptprogramm
/src/Beispiel – BETA-Beispielprogramm, das zum Testen des Übersetzers verwendet wurde.
/src/Global – Fremdbibliotheken und Hilfsfunktionen.
/src/JavaConverter, /src/SyntaxAnalyzer, /src/SyntaxParser, /src/UnFragment – Quellen der einzelnen Teilschritte.
/src/JavaWriter – Java-Baum und Java-Code Ausgabefunktionen.
/src/Notizen – Verschiedene (teils veraltete) detaillierte Anmerkungen zu Teilen des Übersetzers.
/src/Test – Enthält kleine Tests die verwendet wurden, um Probleme während der Entwicklung zu klären; sind unabhängig vom Übersetzer.

Dokumentation.pdf – Die Dokumentation zum Projekt.
inhalt.txt – Diese Datei.