Betreuer: Prof. Robert Giegerich, Dr. Peter Steffen
Zeitraum: 30.4.2007 - 30.10.2007

Diplomarbeit

Java-Backend für den ADP-Compiler

Georg Sauthoff*

11. September 2007

*gsauthof@TechFak.Uni-Bielefeld.DE

Zusammenfassung

Im Mittelpunkt der Arbeit steht das Problem der Übersetzung von Algebraic-Dynamic-Programming-Programmen nach Java. Dazu wird ein Java-Backend für den existierenden ADP-Compiler erstellt. Es wird auf die Motivation für diese Übersetzung, auf Probleme, Optimierungsmöglichkeiten und Designentscheidungen bei der konkreten Implementation des neuen Java-Backends für den existierenden ADP-Compiler eingegangen. Neben Benchmarks von nach Java bzw. nach C übersetzten Bioinformatik-ADP-Algorithmen zur Strukturvorhersage von RNA-Sekundärstrukturen wird auch ein exemplarischer Anwendungsfall aus dem Bereich der Bioinformatik mit Integration des erzeugten Java-Codes in größere Softwaresysteme beschrieben. Darüber hinaus wird das Design eines Testframeworks bzw. einer Testsuite entwickelt, welche das Java-Backend und den restlichen ADP-Compiler testet. Abschließend werden die Perspektiven für allgemeine Anwendungen und mögliche Weiterentwicklungen diskutiert.

Danksagung

Ich danke Prof. Robert Giegerich und Dr. Peter Steffen für die Themenstellung, die Betreuung meiner Diplomarbeit und ihre stete Gesprächsbereitschaft.

Die Diplomarbeit ist auf einem Linux-System mit LaTeX erstellt worden. Der Editor hat dabei die regelmäßige Textarbeit erleichtert. Aus der umfangreichen LaTeX-Distribution sind besonders die KOMA-Script-Dokumentklassen, Ti*k*Z, hyperref und pdfTeX von Vorteil gewesen. Die Abbildungen sind unter anderem mit Gnuplot und Graphviz erstellt worden. Den Autoren dieser hervorragenden Software und der ausführlichen Programmdokumentationen sei an dieser Stelle gedankt.

Inhaltsverzeichnis

1 Einleitung

1.1 Zielsetzung und Organisation der Arbeit

Das Ziel der Diplomarbeit ist die Erzeugung eines Systems zur automatischen Übersetzung von ADP[1]-Programmen nach Java-Code.

In diesem ersten Kapitel wird diskutiert, warum eine Übersetzung von ADP nach Java benötigt wird und welche Untersuchungen im Bereich der Übersetzung von ADP existieren. In Kapitel 2 wird eine kurze Einführung in ADP gegeben. Kapitel 3 beschäftigt sich mit dem Design der Implementation des Übersetzungssystems und den damit zusammenhängenen Problemen. Eine Auswahl von nach Java und C übersetzten ADP-Programmen für die RNA-Sekundärstrukturvorhersage wird in Kapitel 4 auf ihre Laufzeit und ihren Platzverbrauch untersucht. In Kapitel 5 wird ein Test-Framework vorgestellt, mit dem die implementierte Java-Codegenerierung auf ihre Korrektheit getestet werden kann. Kapitel 6 beschreibt einen exemplarischen Anwendungsfall aus dem Bereich der Bioinformatik für die Verwendung von einem nach Java übersetzten DP[2]-Algorithmus, und in Kapitel 7 wird ein Fazit der Diplomarbeit gezogen. Im abschließenden Kapitel 8 werden mögliche Verallgemeinerungen und Weiterentwicklungen der angesprochen Programme diskutiert.

1.2 Motivation

Für Java [19] als eine Zielsprache der ADP-Übersetzung spricht die weite Verbreitung von Java in Forschung und Lehre. Java ist eine objektorientierte imperative Sprache, welche an vielen Universitäten als erste imperative Programmiersprache in den algorithmischen Einführungsveranstaltungen gelehrt wird. Besonders bei einer Zielgruppe der ADP-Anwender, nämlich der Entwickler von DP-Algorithmen in der Bioinformatik, sind wahrscheinlich Java-Kenntnisse weiter verbreitet als Programmierkenntnisse einer anderen imperativen Programmiersprache, wie z. B. C.

Java Source-Code wird kompiliert, aber im Gegensatz zu klassischen imperativen Sprachen wird er von dem Referenz-Compiler von Sun nicht in die Maschinensprache der gegebenen Architektur übersetzt, sondern in eine Zwischensprache, den Java-Object-Code. Dieser Object-Code wird von einer virtuellen Maschine (JVM[3]) interpretiert und je nach Implementation der JVM kann diese zur Laufzeit des gegebenen Java-Byte-Code-Programms Teile des Programms von einem Just-in-Time-

[1] Algebraic Dynamic Programming
[2] Dynamic Programming
[3] Java Virtual Machine

Compiler (JIT) in optimierten Maschinencode übersetzen oder den vorliegenden Object-Code unter Berücksichtigung der Eingabe weiter optimieren.

Durch die Verwendung einer VM[4] ist der generierte Object-Code relativ portabel, d. h. auf jeder Architektur, für die eine JVM existiert, kann dieser Code ausgeführt werden, ohne dass er für diese Architektur neu kompiliert werden muss. Im Vergleich zu Java besitzt C-Code eine größere Portabilität, da für eine größere Anzahl von verschiedenen Architekturen C-Compiler existieren. Allerdings muss dann für jede weitere Architektur der C-Code neu kompiliert werden, was bei Java-Code nicht notwendig ist, da der einmal erzeugte Object-Code portabel zwischen den von JVMs unterstützten Architekturen ist.

Java ist außerdem eine „sichere“ Sprache, d. h. in Java sind einige unsichere imperative Sprachkonstrukte nicht möglich, oder sie werden dynamisch geprüft. Beispielsweise ist im Gegensatz zu C keine Pointerarithmetik möglich, und Array-Zugriffe werden auf Einhalten der Indexgrenzen zur Laufzeit geprüft. Die Verwendung von Java als Zielsprache hat den Vorteil, dass bestimmte Codeerzeugungsfehler direkt durch Exceptions der JVM angezeigt werden. Es ist z. B. nicht wie in C möglich, dass ein Bufferoverflow stattfindet, welcher potentiell unbeabsichtigte Seiteneffekte auslösen kann.

Die Verwendung einer VM zur Erhöhung der Portabilität des Kompilats und die Zugriffsüberprüfungen zur Laufzeit bedeuten natürlich einen erhöhten Aufwand während der Programmausführung im Vergleich zu einer nicht-interpretierten Sprache, welche nativ und ohne Zugriffsüberprüfungen kompiliert wird. Aber durch die Verwendung eines JIT-Compilers und anderer Optimierungstechniken innerhalb einer JVM kann dagegen die Verlängerung der Laufzeit auf ein gewisses Maß begrenzt werden, so dass der Trade-Off für je nach Anwendungsfall zwischen Sicherheit bzw. Portabilität auf der einen Seite und Performance auf der anderen Seite noch akzeptabel sein kann.

Eine weitere Motivation ist die Integration von DP-Algorithmen in Java-Programme. Sun hat nicht nur die Sprache Java spezifiziert, sondern auch die API[5] einer umfangreichen Klassenbibliothek. Teil dieser Klassenbibliothek sind auch Klassen zum Programmieren einer GUI[6]-Anwendung. Eine so programmierte GUI ist ebenso portabel wie eine normale Java-Anwendung. Dies ist bei der Verwendung von anderen GUI-Bibliotheken für andere Sprachen nicht der Fall, da diese Bibliotheken oft nicht so portabel wie die verwendete Sprache sind. Beispielsweise ist GTK[7], bei Verwendung von GTK als GUI-Bibliothek für C-Anwendungen nicht für die native Aqua-Oberfläche von Mac-OSX verfügbar im Gegensatz zu einer Aqua-JVM, welche für Mac-OSX existiert.

Wenn ADP-Programme nach Java übersetzt werden können, kann ein DP-Algorithmus einfach als Komponente in ein größeres Java-Projekt integriert werden, welches z. B. eine benutzerfreundliche graphische Schnittstelle für die Eingabeda-

[4] Virtual Machine
[5] Application Programming Interface
[6] graphical user interface
[7] GIMP-Toolkit

ten und weitere unabhängige Komponenten für die Visualisierung der Ergebnisse bereitstellt.

Im Gegensatz zu einer automatischen Übersetzung von ADP nach Java ist eine manuelle Implementation von einem DP-Algorithmus in Java aufwändig, fehleranfällig und schwer zu debuggen, da, im Gegensatz zu der Beschreibung in ADP, Matrix-Rekurrenzen manuell entwickelt werden und alle Indexangaben in den Rekurrenzen manuell korrekt übertragen werden müssen.

1.3 Bisheriger Stand

Eine Implementation von ADP bettet ADP als Domain-Specific-Language (DSL) in Haskell ein [16], d. h. ADP-Programme können in einer implementationsspezischen Notation aufgeschrieben werden und sind zugleich ausführbare Haskell-Programme. Da Haskell eine nicht-strikte Auswertungsstragie verfolgt, werden bei tabellierten Nichtterminalen nur die Tabelleneinträge verwendet, welche tatsächlich benötigt werden. Ein Problem bei der Einbettung in Haskell ist, dass der ADP-Programmierer bei Fehlern in ADP-Programmen mit den Fehlermeldung des Haskell-Typecheckers konfrontiert wird. Um diese Fehlermeldungen nachvollziehen zu können, muss der Anwender Haskell beherrschen und Implementationsdetails der DSL kennen. Außerdem muss er Teile der ADP-Grammatik manuell annotieren, wenn z. B. Nichtterminale tabelliert werden sollen oder nicht.

Ein ADP-Typechecker [30] existiert und kann als Präprozessor für in Haskell eingebettete ADP-Programme verwendet werden. Wenn ein Typ-Fehler im ADP-Programm existiert, kann der ADP-Typechecker eine ADP-spezifische Fehlermeldung ausgeben.

Des Weiteren existiert ein ADP-Compiler [32], welcher ADP-Programme nach C übersetzt. Bei Verwendung dieses Compilers muss die Grammatik nicht mehr annotiert werden, da in einer automatischen Analyse allgemeine ADP-Kombinatoren durch speziellere Varianten ersetzt werden. Die Annotierung des Tabellierungsstatus ist nicht notwendig, da ein Modul die optimale Tabellenkonfiguration berechnet [24]. Der Compiler enthält neben dem C-Backend auch zwei experimentelle Backends zur Erzeugung Pascal- und Fortran-Codes. Die Benchmarks in [32] zeigen, dass die Laufzeit von ADP-Programmen durch die Übersetzung in eine imperative Sprache stark verringert werden kann im Vergleich zu kompilierten Haskell-Programmen. Dies liegt daran, dass der Haskell-Compiler keine Informationen über ADP-spezifische Optimierungen enthält bzw. keine ADP-spezifischen Optimierungen vornehmen kann.

1.4 Quellcode

Der Quellcode der während der Diplomarbeit erstellten Programme und Module ist in dem Entwicklungszweig `adp2java` des ADP-Compilers verfügbar. Der Entwick-

lungsstand zur Abgabe ist mit dem Tag `adp2java-abgabe` versehen. Auf einem Rechnersystem der Technischen Fakultät kann man den Quelltext mit Subversion abrufen:

```
# mkdir workdir
# cd workdir
# svn co file:///vol/adpc/src/SUBVN/adpc/branches/adp2java
  bzw.
# svn co file:///vol/adpc/src/SUBVN/adpc/tags/adp2java-abgabe
```

Alternativ ist der Quellcode direkt von dem Autor zu erhalten oder als Archiv Techfak-lokal unter `/homes/gsauthof/pub/adp2java.abgabe.tar.gz` verfügbar.

Die von dem Autor erstellten Dateien und Dateibestandteile sind durch die Revisionsinformationen des Subversion-Repository[8] gekennzeichnet. Außerdem sind die Quellcode-Dateien und Quellcode-Dateiteile, welche während der Diplomarbeit erstellt worden sind, in der Diplomarbeit an den Stellen, an denen auf sie Bezug genommen wird, referenziert.

[8] abrufbar via `svn log -v [Dateiname]`

2 Algebraic Dynamic Programming

Dynamic Programming ist eine Programmiermethode, welche in den 1950ern von Bellman entwickelt wurde [10, 14]. Mit DP können Algorithmen für kombinatorische Optimierungsprobleme entwickelt werden, wenn die optimale Lösung eines Problems sich aus den optimalen Lösungen seiner Teilprobleme zusammensetzt (Bellman's Principle [16]). Bei einem exponentiellen Suchraum hat der DP-Algorithmus eine polynomielle Laufzeit und benötigt polynomiellen Speicherplatz.

Algebraic Dynamic Programming ist ein formales Framework, mit dem die einzelnen Aspekte von einem DP-Algorithmus über Sequenzen separat entwickelt bzw. spezifiziert werden. Im Gegensatz dazu werden klassische DP-Algorithmen durch Matrix-Rekurrenzen spezifiziert, in denen die Beschreibung des Suchraums der Kandidaten, die Auswertung der Kandidaten und weitere Aspekte eines DP-Algorithmus nur implizit spezifiziert sind. Dies führt dazu, das komplexere DP-Algorithmen nur an Hand von Matrix-Rekurrenzen schwer zu implementieren, debuggen und vermitteln sind.

Eine ausführliche Einführung in Algebraic Dynamic Programming (ADP) mit exemplarischen Beispielen aus verschiedenen Anwendungsbereichen gibt [16]. Im folgenden wird ADP zusammenfassend beschrieben.

Bei dem Design von einem DP-Algorithmus in ADP wird zuerst festgelegt, über welchem Alphabet $\mathcal{A}$ die Eingabesequenz gebildet wird. Als nächstes wird die Signatur Σ des ADP-Programms spezifiziert, welche eine Menge von Operatoren, aus denen ein Kandidaten-Term konstruiert werden kann, und eine Sorte S enthält. Jeder Operator o hat eine feste Stelligkeit $o : s_1 ... s_{k_o} \rightarrow S$, wobei jedes s_i entweder S oder $\mathcal{A}$ sein kann. Danach wird eine Auswertungs-Algebra $\mathcal{I}$ definiert, welche jedem Operator o eine Funktion $o_\mathcal{I}$ mit derselben Stelligkeit $o_\mathcal{I} : (s_1)_\mathcal{I} ... (s_{k_o})_\mathcal{I} \rightarrow S_\mathcal{I}$ und der Sorte S die Wertemenge $S_\mathcal{I}$ zuordnet. Die Auswahlfunktion $h : [S_\mathcal{I}] \rightarrow [S_\mathcal{I}]$ ist Teil der Auswertungsalgebra. Die ADP-Grammatik ist eine reguläre Baumgrammatik über $\mathcal{A}$ und Σ, die den Suchraum der gültigen Terme, welche aus den Operatoren aus Σ konstruiert werden können, beschreibt. Dabei wird die ADP-Grammatik definiert als Tupel aus einer Menge V von Nichtterminalen, einem Startsymbol $Ax \in V$ und der Menge P von Produktionen, welche die Form $v \rightarrow t$ haben, wobei $v \in V$ und $t \in T_\Sigma(V)$ ist. T_Σ bezeichnet die Term-Algebra, in der die Operatoren der Signatur als Konstruktoren interpretiert werden, und $T_\Sigma(V)$ ist die Termalgebra, welche Variablen aus V enthält.

Auf einer konzeptionellen Ebene kann man sich die Auswertung eines ADP-Programms zweigeteilt vorstellen. Zuerst werden zu einer Eingabe über dem Alphabet $\mathcal{A}$ alle Terme konstruiert, die eine gültige Ableitung (einen Parse) in der ADP-Grammatik besitzen. Danach werden die Terme unter einer Algebra $\mathcal{I}$ aus-

gerechnet und die Liste der Ergebnisse von dem Typ $S_{\mathcal{I}}$ der Auswahlfunktion h der Auswertungsalgebra nach einem Optimierungskriterium ausgewählt (z. B. das Maximum oder das Minimum der Eingabeliste). Da der durch die Grammatik beschriebene Suchraum exponentiell ist, wird in existierenden Implementationen von ADP die Auswertungsfunktion schon auf Zwischenergebnisse der Nichtterminalparser angewendet. Außerdem werden Ergebnisse von Nichtterminalparsern tabelliert, wenn eine wiederholte Neuberechnung sonst die asymptotisch optimale Laufzeit des DP-Algorithmus asymptotisch verschlechtern würde.

Die frühe Anwendung der Auswahlfunktion h auf Listen von Teilergebnissen ist möglich, da eine Auswertungsalgebra Bellmans-Prinzip genügen muss. Die algebraische Version von Bellmans-Prinzip ist in [16] definiert.

Im Gegensatz hierzu wird klassisch ein DP-Algorithmus durch Matrix-Rekurrenzen beschrieben. Dabei entspricht ein tabelliertes Nichtterminal in ADP einer Matrix. Die Beziehungen zwischen den zu berechnenden Matrix-Einträgen, werden nicht in einer Grammatik angegeben, sondern durch den Matrix-Namen und ein Indexpaar. Die Verwendung von Indexen wird als eine große Fehlerursache bei der Entwicklung von DP-Algorithmen angesehen [16]. Ein ADP-Programm enthält keine Indexe. Des Weiteren werden die Funktionen einer ADP-Algebra direkt in den Matrix-Rekurrenzen verwendet, und es findet keine Abstraktion über eine Signatur statt. Wenn z. B. in einem DP-Algorithmus nicht mehr das Ergebnis minimiert, sondern maximiert werden soll, muss ein neuer Algorithmus mit einer kompletten Menge von leicht veränderten Matrix-Rekurrenzen angegeben werden. In ADP ist nur die Erstellung einer neuen Algebra notwendig. Das Alphabet, die Signatur und die Grammatik bleiben unverändert.

2.1 Beispiel

Als Beispiel für die Spezifikation eines DP-Algorithmus in ADP und als klassische Matrix-Rekurrenz wird der El-Mamun Algorithmus [16] zur optimalen Klammerung von arithmetischen Ausdrücken vorgestellt.

Der El-Mamun-Algorithmus löst das Problem, einen arithmetischen Ausdruck nach einem bestimmten Optimalitätskriterium optimal zu klammern. Dabei sind nur wohlgeformte und vollständige Klammerungen erlaubt. Eine Klammerung ist wohlgeformt, wenn ein geklammerter Ausdruck gleich viele öffnende und schließende Klammern enthält, und jede schließende Klammer einer vorangegangenen öffnenden Klammer zugeordnet werden kann. Eine öffnende Klammer darf nicht direkt links vor einem arithmetischen Infix-Operator stehen bzw. eine schließende Klammer nicht direkt rechts hinter einem arithmetischen Infix-Operator. Wenn der längste ungeklammerte Teilausdruck eines Ausdrucks maximal ein Infix-Operator enthält, ist der Gesamtausdruck vollständig geklammert. Beispielsweise kann der Ausdruck

$$1 + 2 \cdot 3 \cdot 4 + 5 \tag{2.1}$$

auf 14 verschiedene Arten geklammert werden. Den maximalen bzw. minimalen

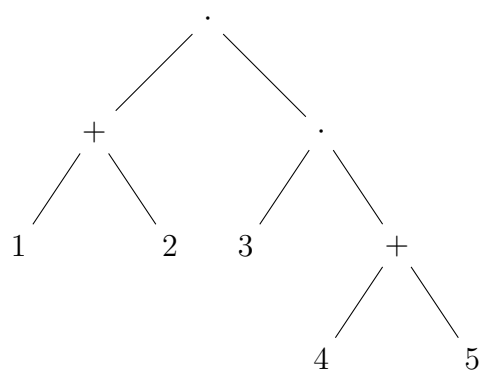

Abbildung 2.1: Ausdruck $(1 + 2) \cdot (3 \cdot (4 + 5))$ in Baumdarstellung.

Wert hat der Ausdruck 2.1 mit der Klammerung in 2.2 bzw. 2.3:

$$(1 + 2) \cdot (3 \cdot (4 + 5)) \tag{2.2}$$
$$1 + ((2 \cdot (3 \cdot 4)) + 5) \tag{2.3}$$

Für den Beispielausdruck 2.1 gibt es in Bezug auf die Maximierung bzw. Minimierung des Wertes nicht nur eine optimale Klammerung sondern zwei bzw. vier.

2.1.1 Modellierung in ADP

Ein geklammerter Term kann auch als Baum dargestellt werden. Abbildung 2.1 zeigt die Baumdarstellung für den Ausdruck 2.2. Die Topologie der Baumdarstellung ersetzt die explizite Klammerung des Ausdrucks. Die Darstellung von geklammerten Ausdrücken als Baumstrukturen ist umkehrbar eindeutig.

In der ADP-Methodik kann der El-Mamun Algorithmus nun wie folgt entwickelt werden. Das Alphabet der Eingabesequenz $\mathcal{A}_{el}$ ist eine Teilmenge des ASCII-Zeichensatzes:

$$\mathcal{A}_{el} = \{'0' \ldots '9', '+', '\cdot'\} \tag{2.4}$$

Die Signatur muss alle notwendigen Operatoren enthalten, mit denen die zu betrachtenden Kandidatenterme wie z. B. in Abbildung 2.1 konstruiert werden können. Die Signatur

$$\begin{aligned} \Sigma_{el} = (S_{el}, \{\sigma_{val}, \sigma_{ext}, \sigma_{add}, \sigma_{mult}\}) \\ \sigma_{val} : \mathcal{A}_{el} \to S_{el} \\ \sigma_{ext} : S_{el} \to \mathcal{A}_{el} \to S_{el} \\ \sigma_{add} : S_{el} \to \mathcal{A}_{el} \to S_{el} \to S_{el} \\ \sigma_{mult} : S_{el} \to \mathcal{A}_{el} \to S_{el} \to S_{el} \end{aligned} \tag{2.5}$$

enthält deswegen zwei dreistellige Operatoren σ_{mult} und σ_{add}, die die Multiplikation bzw. Addition mit zwei Argumenten für das Ergebnis der Teilausdrücke von dem

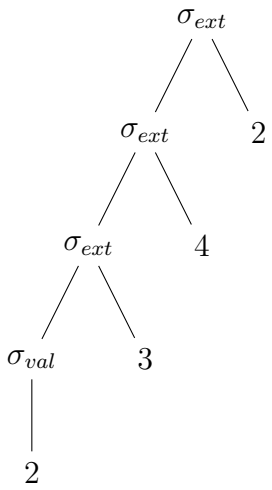

Abbildung 2.2: Konstruktion eines Kandidaten für die Eingabesequenz 2342 aus den Operatoren der Signatur Σ_{el}.

Typ der Sorte S_{el} und einem Argument für das Infix-Operator-Zeichen (Typ $\mathcal{A}_{el}$) modellieren. Eine einstellige Zahl in der Eingabe-Sequenz wird durch den einstelligen Operator σ_{val} konstruiert. Die Konstruktion einer mehrstellige Zahl kann durch den zweistelligen Operator σ_{ext} erreicht werden, der als ein Argument vom Typ S_{el} das Ergebnis einer weiteren Anwendung der Opertoren σ_{val} bzw. σ_{ext} zulässt. Ein Beispiel für die Konstruktion einer mehrstelligen Zahl ist in Abbildung 2.2 und für die Konstruktion des Ausdrucks 2.2 in Abbildung 2.3 angegeben.

Aus einem Kandidatenterm zu einer Eingabesequenz, konstruiert aus Operatoren der Signatur, kann wieder die Eingabesequenz bestimmt werden, indem der Baumkranz der Baumdarstellung abgelesen wird. Der Baumkranz ist die Konkatenation aller Blätterbeschriftungen, die in der Reihenfolge einer Depth-First-Traversion des Baumes betrachtet werden.

Nach der Spezifikation der Signatur Σ_{el} können Auswertungsalgebren definiert werden, unter denen die konstruierten Kandidaten interpretiert werden sollen. Wenn z.B. die Klammerung eines Ausdrucks, unter der der Wert des Ausdrucks maximal ist, ermittelt werden soll, kann der Sorte S_{el} die Menge der natürlichen Zahlen zugeordnet werden, die Auswahlfunktion muss über die Eingabeliste maximieren, und die Funktionen, welche die Operatoren ersetzen, die Argumente vom Typ S_{el} bei der Interpretation von σ_{mult} multiplizieren, von σ_{add} addieren und von σ_{val} die Integer-Darstellung zurückgeben. Die Funktion, welche σ_{ext} interpretiert, muss das Argument vom Typ S_{el} mit 10 multiplizieren und zu dem Wert des konvertierten Arguments vom Typ $\mathcal{A}_{el}$ addieren. Abbildung 2.4 stellt diese Maximierungsauswertungsalgebra in Haskell-Notation dar. Im Falle der Klammerung für den minimalen Wert eines Ausdrucks muss in der Auswertungsalgebra nur die Auswahlfunktion um eine Minimierung auf Listen ausgetauscht werden (Abbildung 2.5).

Des Weiteren ist eine Algebra sinnvoll, welche einen Kandidatenterm in Baum-

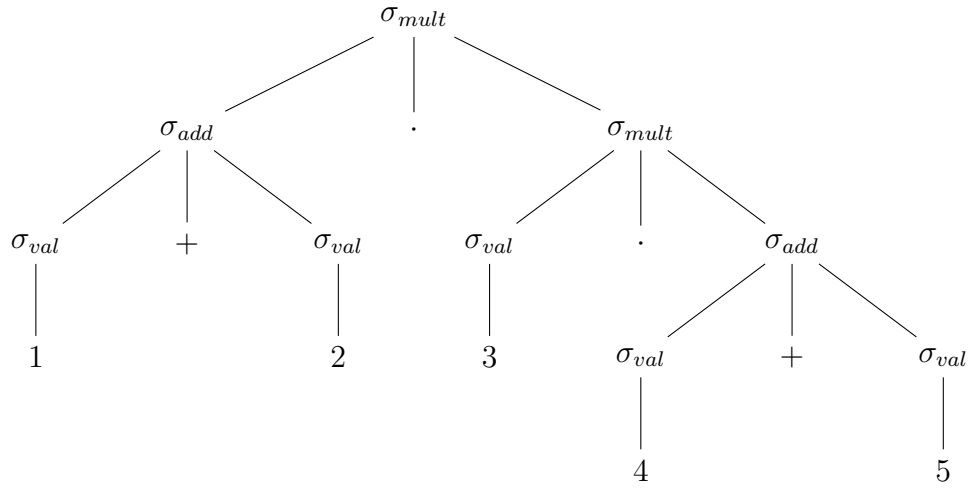

Abbildung 2.3: Ausdruck (1+2)·(3·(4+5)) konstruiert aus Operatoren der Signatur Σ_{el} in Baumdarstellung.

```
seller :: Bill_Algebra Char Int
seller = (val, ext, add, mult, h) where
  val c       = decode c
  ext n c     = 10*n + decode c
  add  x c y  = x + y
  mult x c y  = x * y
  h []        = []
  h l         = [maximum l]
```

Abbildung 2.4: El-Mamun-Maximierungsalgebra in Haskell-Notation.

darstellung als geklammerten String darstellt (Abbildung 2.6), denn in der Anwendung von vielen ADP-Algorithmen möchte man nicht nur wissen, wie der Wert von einem optimalen Kandidaten ist, sondern auch wie dieser Kandidat konstruiert wurde bzw. aussieht.

Da die alleinige Anwendung der Algebra `prettyprint` alle Kandidaten ohne den minimalen bzw. maximalen Wert ausgibt, kann die Algebra mit der Minimierungs- bzw. Maximierungsalgebra über den `***`-Operator kombiniert werden, welcher in [31] definiert ist. Diese Kombination führt dazu, dass nur die Kandidaten, welche die Auswertung unter einer Minimierungs- bzw. Maximierungsalgebra zurückliefert, von der `prettyprint`-Algebra ausgewertet werden. Das Ergebnis dieser kombinierten Auswertung ist eine Liste von Tupeln von den Typen der kombinierten Algebren.

Wie man leicht zeigen kann, erfüllen die vorgestellten Algebren und die Kombination der `prettyprint`-Algebra mit einer Minimierungs- bzw. Maximierungs-Algebra Bellmans Prinzip.

```
buyer :: Bill_Algebra Char Int
buyer = (val, ext, add, mult, h) where
  val c         = decode c
  ext n c       = 10*n + decode c
  add  x t y    = x + y
  mult x t y    = x * y
  h []          = []
  h l           = [minimum l]
```

Abbildung 2.5: El-Mamun-Minimierungsalgebra in Haskell-Notation.

```
prettyprint :: Bill_Algebra Char String
prettyprint = (val, ext, add, mult, h) where
  val   c     = [c]
  ext   n c   = n ++ [c]
  add   x c y = "(" ++ x ++ (c:y) ++ ")"
  mult  x c y = "(" ++ x ++ (c:y) ++ ")"
  h           = id
```

Abbildung 2.6: El-Mamun-Algebra in Haskell-Notation, welche die Kandidaten als Klammerstrings auswertet und keine Optimierung durchführt.

Im letzten Schritt wird die ADP-Grammatik angegeben, welche den Suchraum der gültigen Kandidatenterme beschreibt. Beispielsweise sollten die Terme $\sigma_{ext}(\sigma_{mult}2\cdot 3)4$ oder $\sigma_{add}2\cdot 3$ nicht als Kandidaten evaluiert werden, da sie nach der bisherigen informellen Modellierung unsinnig sind. Abbildung 2.7 enthält die ADP-Grammatik. Das Startsymbol ist das Nichtterminal `formula`. Eine ‚`formula`‘ bzw. ein geklammerter Ausdruck kann entweder eine Zahl sein (Nichtterminal `number`) oder aus zwei geklammerter Teilausdrücken bestehen, welche durch einen Infix-Operator getrennt sind. Das Nichtterminal `number` kann nach einer Ziffer oder einer Folge von Erweiterungen von Ziffern abgeleitet werden. Somit sind alle gültigen Kandidaten durch mögliche Ableitungen nach der ADP-Grammatik spezifiziert. Ein Beispiel für einen Ableitungsbaum für den Ausdruck 2.2 ist in Abbildung 2.8 dargestellt.

2.1.2 Modellierung in DP

Das Problem, eine optimale Klammerung für einen Eingabestring zu finden, ist für Ausdrücke, welche nur aus einer Zahl oder aus einem Infix-Operator mit zwei Argumenten bestehen, trivial zu lösen. Die optimale Klammerung eines längeren Ausdrucks kann nun als die Konkatenation der optimalen Klammerungen der zwei Teilausdrücke modelliert werden, welche durch einen Infix-Operator getrennt sind. Man muss natürlich beweisen, dass ein Ausdruck optimal geklammert ist, wenn

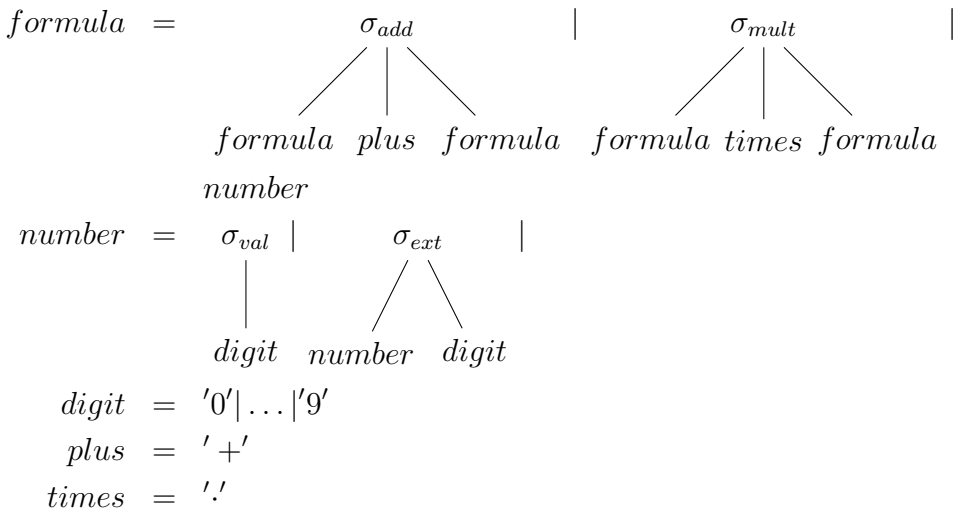

Abbildung 2.7: ADP-Baumgrammatik für den DP-Algorithmus El-Mamun.

seine Teilausdrücke optimal geklammert sind. Diese Einschränkung entspricht der Forderung, dass ein DP-Algorithmus Bellmans-Prinzip erfüllen muss.

Mit Hilfe der Indizierung des Eingabestrings können Teilstrings durch die Angabe eines Indexpaares referenziert werden. Ein Beispiel ist in Abbildung 2.9 angegeben. Die bisher entwickelte Modellierung kann nun z. B. für eine Minimierung des Wertes des geklammerten Ausdrucks in Matrixrekurrenzen übersetzt werden:

$$A_{i,j} = \min_{i<k<j} \begin{cases} \text{if } s[k] = '+' \text{ then } A_{i,j} + A_{k+1,j} \\ \qquad\qquad \text{else } \infty \\ \text{if } s[k] = '-' \text{ then } A_{i,k} \cdot A_{k+1,j} \\ \qquad\qquad \text{else } \infty \\ B_{i,j} \end{cases} \tag{2.6}$$

$$B_{i,j} = \text{if valueof}(s(i,j)) \in \mathbf{N} \text{ then valueof}(s(i,j)) \text{ else } \infty \tag{2.7}$$

Mit $s[x]$ wird auf das x-te Element der Eingabesequenz verwiesen, und $s(i,j)$ bezeichnet den Teilstring der Eingabe zwischen den Indexen i und j. Für einen Eingabestring der Länge n ist der minimale Wert eines optimal geklammerten Ausdrucks in dem Matrixelement $A_{0,n}$ enthalten. Der Wert des optimal geklammerten Teilstrings zwischen den Indexen i und j ist in $A_{i,j}$ abgespeichert. Zur Berechnung des optimalen Ergebnis werden rekurrent alle benötigten optimalen Ergebnisse der Teilausdrücke berechnet.

Um die Struktur der Kandidaten mit dem optimalen Wert und nicht nur den Wert auszugeben, muss in einer Implementierung des El-Mamun-Algorithmus auch ein Backtracing-Verfahren implementiert werden. Eine Möglichkeit ist, bei der zeilen- oder spaltenweisen Berechnung der Matrix A für jede Zelle in einer Hilfsmatrix die Indexpaare zu speichern, welche bei der Berechnung des minimalen Wertes der

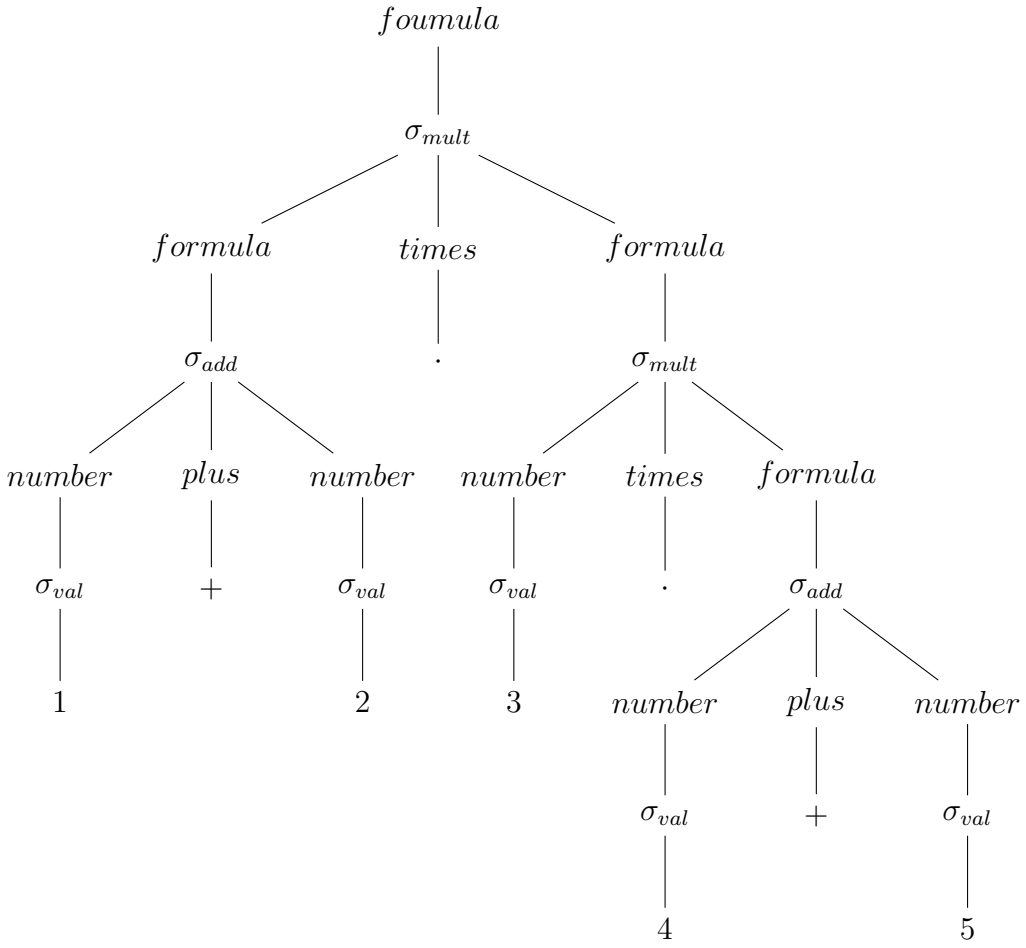

Abbildung 2.8: Ableitung des Klammerstring $(1+2)\cdot(3\cdot(4+5))$ über der El-Mamun ADP-Grammatik als Ableitungsbaum dargestellt. Die Eingabesequenz ist der Baumkranz des Ableitungsbaum.

$$_0 1 _1 + _2 2 _3 \cdot _4 3 _5 \cdot _6 4 _7 + _8 5 _9$$

Abbildung 2.9: Indexierung des Beispielstrings $s = 1 + 2 \cdot 3 \cdot 4 + 5$. Beispielsweise ist der Substring $s(2, 5)$ der Ausdruck $2 \cdot 3$. Der gesamte String wird durch $s(0, 9)$ bezeichnet.

Zelle herangezogen wurden. Nachdem die Matrix vollständig berechnet ist, kann beginnend mit dem Matrixeintrag $A_{0,n}$ rekursiv diesen Hilfsinformationen gefolgt und der geklammerte Ausdruck abgeleitet werden.

Der Platzverbrauch des El-Mamun-Algorithmus liegt in $O(n^2)$, da bei einer Sequenzlänge n eine quadratische $n \times n$ Matrix angelegt wird und die Anzahl der zu berechnenden Einträge in $O(n^2)$ liegt. Zur Berechnung jedes Matrix-Eintrags muss über $O(n)$ Einträge minimiert werden. Also liegt die Laufzeit des Algorithmus in $O(n^3)$.

2.1.3 Abschließende Bemerkungen

Der El-Mamun-Algorithmus ist als ein sehr kleines und einfaches Beispiel, welches für die erste Einführung in ADP geeignet ist. Reale DP-Algorithmen aus der Bioinformatik haben sehr viel mehr Nichtterminale. Diese Grammatiken werden manuell entwickelt (z. B. pknotsRG-enf [28] hat 40 Nichtterminale) oder automatisch generiert (Locomotif [27] erzeugt pro Baustein 5-6 Nichtterminale). Bei diesem Umfang der Grammatiken ist eine alternative manuelle Entwicklung basierend auf Matrix-Rekurrenzen aufgrund der Komplexität der vielfältigen Indexbeziehungen schwer vorstellbar.

Vereinfacht beschrieben übersetzt der ADP-Compiler [32] ein ADP-Programm intern in eine Menge von Matrix-Rekurrenzen. In einem weiteren Übersetzungsschritt wird imperativer Code erzeugt, der diese Rekurrenzen berechnet. Zusätzlich werden einige Optimierungen vorgenommen, wie z. B. die Bestimmung, welche Nichtterminale tabelliert werden müssen oder welche Tabellen nur linear tabelliert werden müssen. Außerdem erzeugt der Compiler Backtracing-Code, damit die prettyprint-Repräsentation der optimalen Kandidaten ausgegeben werden kann. Dieser Backtracing-Code erlaubt auch ein Backtracing von suboptimalen Kandidaten. In diesem Modus werden zusätzlich zu den Kandidaten mit dem optimalen Wert auch bis zu einer wählbaren Grenzen alle suboptimalen Kandidaten ausgegeben.

3 Implementation

3.1 Design

Es gibt mehrere Möglichkeiten ADP-Programme nach Java zu übersetzen. Auf die Vor- und Nachteile der einzelnen Alternativen wird in diesem Abschnitt eingegangen, und die in der Arbeit benutze Methodik wird vorgestellt.

3.1.1 JNI

Java enthält eine API[1], mit deren Hilfe man nativ compilierten Code aus einem Java-Programm aufrufen kann. Diese API heißt Java Native Interface (JNI [33]) und ist z. B. mit dem Foreign Function Interface (FFI [13]) von Haskell vergleichbar.

Mit Hilfe von JNI kann also der für eine gegebene Architektur kompilierte C-Code, welcher von dem existierenden ADP-Compiler erzeugt wird, aus einem Java-Programm aufgerufen werden. Die Zielsetzung wird jedoch von diesem Ansatz nicht erfüllt, da keine Übersetzung von ADP nach Java stattfindet. Allerdings ist eine Motivation erfüllt, nämlich die Integration von ADP-Algorithmen in Java-Programme.

Die Vorteile bei diesem Ansatz liegen in der Wiederverwendung von dem C-Codegenerator, welcher gut getesteten und performanten imperativen Code erzeugt [32]. Der ganze ADP-Compiler muss nicht verändert werden.

Damit der generierte Quellcode in einem Java-Programm verwendet werden kann, müsste ein Präprozessor passend zu jeder Ausgabe von dem ADP-Compiler C-Wrapper-Code erzeugen, welcher JNI-konforme Schnittstellen definiert und Java-Wrapper-Code, welcher eine Schnittstelle für den Java-Code bereitstellt, die die Implementationsdetails der per JNI importierten Methoden verbirgt.

Der Nachteil ist die nicht erreichte erhöhte Portabilität des kompilierten Java-Programms, welches einen ADP-Algorithmus verwendet, im Vergleich zu einem erzeugten C-Programm, da es von einem nativ kompilierten Code-Modul abhängt. Die einfache Distribution von kompilierten Java-Modulen auf jede Architektur, für die eine JVM existiert, ist nicht möglich, da für jede neue Architektur das native Modul, welches den ADP-Algorithmus enthält, neu kompiliert werden muss. Aus Sicherheitsgründen ist z. B. eine Integration in ein Java-Applet nicht möglich, da dort der native Code nicht von der JVM in einer Sandbox abgeschirmt werden kann.

Außerdem wird bei diesem Ansatz die Sicherheit der Sprache Java umgangen, da der native Code im selben Prozess von der JVM ausgeführt wird und die Zugriffs-

[1] Application Programming Interface

überprüfungen zur Laufzeit der JVM, wie z. B. die Prüfung der Überschreitung von Arraygrenzen, nicht auf nativ kompilierten Code angewendet werden können.

3.1.2 Umweg über MIPS-Code

Die abstrakte Maschine, welche von der JVM implementiert wird, hat große Ähnlichkeiten mit der real existierenden MIPS[2]-Architektur [8]. Aufgrund dieser Ähnlichkeiten sind Alliet und Megacz auf die Idee gekommen, den einfachen Übersetzer NestedVM zu schreiben, welcher MIPS-Code nach Java-Object-Code übersetzt [8]. So kann mit einem C-Cross-Compiler, wie z. B. dem gcc [1], C-Code nach MIPS-Code übersetzt werden, und über diesen Umweg erzeugt der NestedVM-Compiler aus der Ausgabe des MIPS-Cross-Compiler Java-Byte-Code. Dieser so erzeugte Byte-Code ist von Java-Programmen einbindbar und ist genauso portabel wie der von einem Java-Compiler erzeugte Byte-Code.

Auch bei diesem Ansatz kann der ADP-Compiler ohne Änderungen weiterverwendet werden, es muss nur ein Präprozessor Java-Wrapper-Code erzeugen, der eine einheitliche Schnittstelle zu den Implementationsdetails darstellt, wie z. B. die Übersetzung von Java-Strings nach C-Strings oder die Bekanntgabe der von dem ADP-Compiler gewählten Funktionsnamen.

Die Probleme bei diesem Ansatz liegen in dem unklaren Status des NestedVM-Compilers und in einer schwer abschätzbaren Verschlechterung der Performance im Vergleich zu dem nativ kompilierten C-Code. Der NestedVM-Compiler ist ein Open-Source-Projekt, welches aus der in [8] vorgestellten Implementation entstanden ist. Wie aktiv es weiterentwickelt wird und wie die zukünftige Supportperspektive bei diesem Projekt aussieht, ist unklar.

In den in [8] vorgestellten Benchmarks von nach MIPS/Byte-Code übersetzten C-Programmen wird von Verschlechterungen der Laufzeit um den Faktor zwei bis zehn berichtet. Wenn nun ein auf diesem Wege übersetzter ADP-Algorithmus, eine Laufzeitverschlechterung um den Faktor drei oder größer zeigt, ist der Spielraum für weitere Optimierungen gering. Denn der NestedVM-Compiler weiß nichts von ADP, und die Ergebnisse der Optimierungsstufen des ADP-Compilers sind an dieser Stelle der Übersetzerabfolge nicht mehr verfügbar.

3.1.3 Erweiterung des ADP-Compiler

Der vorhandene ADP-Compiler übersetzt ADP-Programme nach C-Code und enthält zwei experimentelle Backends, welche Pascal- und Fortran-Code erzeugen. Diese Situation impliziert, dass es möglich ist, den ADP-Compiler um Backends für weitere Ausgabesprachen zu erweitern. Also ist eine weitere Möglichkeit für die Übersetzung von ADP nach Java die Erweiterung des ADP-Compilers um ein Java-Backend.

[2] Microprocessor without interlocked pipeline stages

Die Schnittstelle zwischen dem Front- und Backend des ADP-Compilers ist die interne Zwischensprache TL (Target Language), d. h. die Ausgabe bzw. Eingabe des Frontend bzw. Backend ist TL. Die TL des ADP-Compilers ist sehr an den Gegebenheiten der Ausgabesprache C orientiert, d. h. sie enthält Konzepte, wie z. B. Funktionszeiger oder Pointer-Arithmetik, wie sie auch in C definiert sind. Die Verwendung von solchen Konstrukten in der Eingabesprache eines Java-Backends ist problematisch, da in Java diese Konzepte nicht existieren. In Java ist es nicht möglich, Funktionspointer zu verwenden oder Pointer-Arithmetik durchzuführen. Objekte werden an Hand von Referenzen identifiziert, die allerdings nicht arithmetisch verändert werden können.

Wie problematisch das C-zentrische Design der TL für das Design eines Java-Backends wirklich ist, hängt davon ab, in welcher Vielfalt und in welchen Kombinationen die einzelnen problematischen Konstrukte der TL von der TL-Codeerzeugung generiert werden. Die Analyse des TL-Codeerzeugungs-Moduls zeigt, dass z. B. Pointer-Arithmetik nur für die Stringverarbeitung verwendet wird. Die Verwendung von Funktionszeigern beschränkt sich auf Funktionen mit einem charakteristischen Präfix in dem Backtracing Code.

Der Vorteil bei der Erweiterung des ADP-Compilers ist der hohe Anteil an vorhandenem Compiler-Code, welcher wiederverwendet werden kann. Die Codegenerierung des ADP-Compilers ist gut getestet, und es werden einige umfangreiche Optimierungsläufe implementiert. Eine Implementierung eines eigenständigen ADP-nach-Java-Compilers ‚from scratch' müsste die im ADP-Compiler vorhandenen Merkmale neu implementieren, bis mit der Implementation von dem Java-Backend begonnen werden kann. Im Gegensatz zu der Implementierung von einem Backend für einen existierenden Compiler ist eine vollständige Implementierung von einem neuen Compiler in dem Zeitrahmen einer Diplomarbeit von sechs Monaten nicht möglich.

3.1.4 Auswahl

Aufgrund der Integrations- und Portabilitätsproblematik der JNI-API und der eingeschränkten Optimierungsmöglichkeiten bei der Verwendung des NestedVM-Compilers wird im Rahmen dieser Diplomarbeit die Erweiterung des ADP-Compilers durch ein Java-Backend zur Erreichung des gesetzten Zieles ausgewählt. Die Probleme der TL-Zwischensprache sind überschaubar und werden geprüft.

3.2 Codegenerierung

Der ADP-Compiler erzeugt Code-Module, in denen die Verwendung von einer Algebra durch Inlining erreicht wird. Das Java-Backend erzeugt für das Code-Modul eine Java-Klasse, welche die Schnittstelle zu dem ADP-Algorithmus implementiert. Die Klasse enthält mehrere innere Klassen, welche als Implementationsdetails für andere Java-Pakete nicht sichtbar sind. Die Implementierung des Java-Backends ist in

den Haskell-Modulen `TL.lhs` und `TLData.lhs` des `adp2java`-Entwicklungszweiges enthalten. Trotz der relativ umfangreichen Sonderbehandlungen von einzelnen Konstrukten der Target Language (TL) benötigt das Java-Backend nur einen Durchlauf, um die TL nach Java zu übersetzen. Das Java-Backend verwaltet keinen Status, wie z.B. eine Symboltabelle, also implementiert es das einfache Übersetzungsschema.

Da die Target Language (TL) des ADP-Compilers Sprachkonstrukte enthält, welche keine direkte Entsprechung in Java besitzen, muss das Java-Backend die problematischen Konstrukte erkennen und durch Java-Code ersetzen, welcher die Semantik des ADP-Programms nicht verändert. Beispiele für problematische Konstrukte sind Funktionszeiger und Pointer-Arithmetik.

Eine Alternative zu einem solchen aufwändigeren Java-Backend ist das Neu-Design der TL. Also der Entwurf einer neuen TL, welche problematische Konstrukte, wie Funktionszeiger, vermeidet und abstrakte Datentypen (ADT) enthält, die von jedem Backend optimal übersetzt werden können, z. B. ein Tupel-ADT, der in der aktuellen TL als record modelliert wird, und nur durch ein Namenspräfix von einem record unterschieden werden kann, welcher ein Element einer Liste ist. Der Austausch der TL hätte allerdings eine komplette Neu-Implementierung des TL-Codeerzeugungsmoduls zur Folge, in dem z. B. der Backtracing-Code erzeugt wird, was nicht in dem Zeitrahmen einer Diplomarbeit realisierbar ist. Außerdem ist das aktuelle Codeerzeugungsmodul gut getestet, so dass eine Neu-Implementation eine umfangreiche Testsuite, welche vielfältige Aspekte der Codeerzeugung abdeckt, voraussetzt.

In den folgenden Teilabschnitten werden die einzelnen problematischen Konstrukte und die gewählten Ableitungsschemata des Java-Backends beschrieben.

Beispiele des TL-Codes werden in der Ausgabe des C-Backends dargestellt, da die TL-Zwischensprache sehr stark an C angelehnt ist. Dies hat den Vorteil, dass nicht die Implementationsdetails der TL eingeführt werden müssen, welche aus mehreren algebraischen Datentypen besteht. Außerdem ist die C-Syntax aufgrund der weiten Verbreitung von C in Forschung und Lehre sehr bekannt und zumindest für Leser mit C-Kenntnissen einfacher zu lesen, als die Präfix-Notation der TL.

3.2.1 Stringverarbeitung

In den TL-Programmen des ADP-Compilers werden Strings während der Backtracing-Phase durch Pointer-Arithmetik manipuliert. Die Repräsentation der Strings entspricht der Repräsentation von C-Strings, d. h. es wird die Anfangsadresse von einem Speicherbereich in einer Zeigervariable gespeichert, und das Ende von eines Strings in dem referenzierten Speicherbereich ist durch den ASCII-Code 0 kodiert. Also wenn ein String s l Zeichen lang ist, dann ist an der Stelle l von s das Null-Zeichen gespeichert (wenn die Indizierung mit 0 beginnt). In dem von dem ADP-Compiler generierten Backtracing-Code wird am Anfang des Backtracing ein ausreichend großer Speicherbereich allokiert. Während der Konstruktion der String-Repräsentation eines optimalen bzw. suboptimalen Kandidaten werden immer nur Zeichen oder kurze Substrings an den existierenden String angehängt. In dem TL-

```
String a = new String("23");
d = new String("42");
StringBuilder c = new StringBuilder();
c.append(d);
c.append(a);
b = d.toString();
```

Abbildung 3.1: Zielcode in Pseudo-Code-Schreibweise, den der Java Compiler für die String-Konkatenation `String a = "23"; String b = "42"+ a;` erzeugt. Dies ist nachvollziehbar mit dem Java-Class-File-Disassembler `javap`.

Code wird dazu die Adresse des Null-Zeichens im Speicher als neue Anfangsadresse gesetzt, welche von der stringverarbeitenen Funktion dereferenziert werden kann. Bei einer vorherigen Stringlänge von l und m neugeschriebenen Zeichen wird das neue Ende des Strings durch ein neues Null-Zeichen an der Stelle $l + m$ markiert.

Strings werden in Java als Objekte repräsentiert. Java-Strings sind immutable, d. h. die Klasse `String` besitzt keine Methoden, mit denen der einmal erzeugte String verändert kann. Bei jeder Veränderung, wie z. B. das Anhängen von einem Zeichen, muss eine neue Instanz der String-Klasse angelegt werden. Besser geeignet für eine häufige Veränderung eines Strings in Java ist die Klasse `StringBuilder`[3], welche auch Methoden zum Anhängen von einzelnen Zeichen oder Strings enthält.

Ein vereinfachtes Beispiel für die Problematik bei Veränderungen von Java-Strings ist `String a = "23"; String b = "42" + a;`. Der Java-Compiler erzeugt dafür Code, der für die Erweiterung von 42 um 23 ein temporäres StringBuffer Objekte erzeugt, welches in ein neues String-Objekt umgewandelt wird. Das temporäre Objekt muss von der Garbage-Collection zu einem späteren Zeitpunkt wieder gelöscht werden. Der erzeugte Code entspricht dem Java-Code in Abbildung 3.1. Wenn im Java Code häufig diese skizzierte append-Operation durchgeführt wird, dann müssen entsprechend viele temporäre Objekte angelegt und kurze Zeit später wieder gelöscht werden, was natürlich sehr laufzeitaufwändig ist.

Ein Profiling des nach Java übersetzten RNAfold-ADP-Programms bei Verwendung von StringBuilder-Objekten zeigte eine signifikante Laufzeitverbesserung im Gegensatz zu der Verwendung von String-Objekten, was aus den genannten Gründen zu erwarten ist. Bei unterschiedlichen Eingaben, Eingabelängen und unterschiedlichen Schwellwerten für das suboptimale Backtracing ist eine Laufzeitverbesserung von RNAfold zwischen 11 und 27 Prozent beobachtet worden[4].

[3] Sie ist seit Java 6.0 Teil der Java-Distribution. Die API der Klasse ist identisch mit der `StringBuffer`-Klasse, sie ist aber nicht multithreading-sicher, was in Single-Threading-Anwendung einen Performance-Gewinn gegenüber der Verwendung von `StringBuffer` bedeutet.

[4] auf einer Sun-Fire-v20z, 2.4 GHz, 12 GB RAM, Solaris 10, Sun Java 1.6.2; Als Laufzeit wird die utime (user time) des Prozesses gemessen.

```
struct foo { int a, b; };
struct foo *x, *y;
...
*x = *y;
x = memcpy(x, y, sizeof(struct foo));
```

(a) foo-Structs x und y sind auf dem Heap angelegt.

```
struct foo { int a, b; };
struct foo x, y;
...
x = y;
memcpy(&x, &y, sizeof(struct foo));
```

(b) foo-Structs x und y sind auf dem Stack angelegt.

Abbildung 3.2: Beispiele für die Zuweisungen von Struct-Inhalten in C. Die jeweils letzten beiden Zeilen sind semantisch äquivalent.

Das Java-Backend erkennt die einzelnen Teilbäume der TL, welche die Stringverarbeitung während des Backtracing durchführen, mit Hilfe von Pattern-Matching und erzeugt dafür Java-Code, welcher für die Backtracing Phase ein `StringBuilder`-Objekt verwendet. Bei der Erzeugung des `StringBuilder`-Objekts wird eine ausreichende Mindestgröße des internen Zeichenbuffers angegeben, so dass unnötige teure Re-Allokationen von Speicher während des Backtracing vermieden werden. Nach dem Abschluss des Backtracing kann die Ausgabeschicht das `StringBuilder`-Objekt in einen Java-String umwandeln, um ihn z. B. mit den normalen Java-Ausgabefunktionen auszugeben.

3.2.2 Records

Die Target Language enthält Records oder Compound-Objects, welche Structs in C entsprechen. In der Codegenerierung wird Code erzeugt, welcher in bestimmten Funktionen Structs auf dem Heap und auf dem Stack anlegt. Es werden Pointer von Structs verwendet, aber auch nicht-referenzierte Structs direkt zugewiesen. In C ist eine direkte Zuweisung von Structs möglich und entspricht einer Kopie des von der Quell-Struct verwendeten Speicherbereichs (Beispiel 3.2). Des Weiteren können Structs in C geschachtelt sein, wobei bei einer Kopie der äußeren Struct die inneren Structs mitkopiert werden (Beispiel in Abbildung 3.3). Solche geschachtelten Structs werden auch in der Codegenerierung angelegt.

Das Java-Backend bildet Structs auf Java-Klassen ab. Für Structs, welche in C auf dem lokalen Stack-Frame erzeugt werden, erzeugt das Java-Backend extra Code, der die nicht initialisierten Java Objektrefernzvariablen initialisiert, d. h. die Garbage-Collection ist für die Speicherverwaltung zuständig und erzeugt wahrscheinlich alle Objekte - auch die, die nur innerhalb des aktiven Funktionsaufrufs benötigt werden - auf dem Heap.

```
struct foo {
  int b;
};

struct outer {
  int a;
  struct foo i;
};
```

Abbildung 3.3: Beispiel für geschachtelte Structs in C. Der Speicherbereich einer Struct-outer-Instanz schließt den Speicherbereich der enthaltenen Struct-foo-Instanz mit ein.

Die direkte Zuweisung von Objekten in Java ist nicht äquivalent zu der direkten Zuweisung von Structs in C möglich, denn in Java existieren von Objekten nur Referenzen, welche in Referenzvariablen gespeichert werden. Diese Referenzen auf Objekte können nicht de-referenziert werden bzw. es können keine Objekte explizit auf dem Stack angelegt werden. Das Java-Backend übersetzt diese Fälle, indem jede Klasse, deren Speicherplatz kopiert werden soll, das `Cloneable`-Interface[5] implementiert und eine Methode enthält, welche die `clone()`-Methode des `Cloneable`-Interface aufruft und das geklonte Objekt bzw. den Speicherbereich, den das Objekt belegt, kopiert. Die `clone()`-Methode wird von `Object` implementiert, welche die Basisklasse jeder Java-Klasse ist. Allerdings ist die `clone()`-Methode nur verwendbar, wenn die Klasse das Cloneable-Interface implementiert.

Geschachtelte Structs sind aus den gleichen Gründen nicht direkt in Java abbildbar, denn in Java kann innerhalb von einem Objekt nur eine Referenz auf ein weiteres Objekt gespeichert werden, d. h. der Speicherbereich, welcher die äußere Struct belegt, enthält nur die Referenz auf den Speicherbereich der inneren Struct und eben nicht den gesamten Speicherbereich. Also erzeugt das Java-Backend für die Verwendung von geschachtelten Structs zusätzlichen Code, der bei der Zuweisung von Structs nicht nur die die äußere Struct klont, sondern auch rekursiv die inneren Structs klont, also eine tiefe Kopie des Objekts durchführt.

3.2.3 Funktionszeiger

Das ADPC-Frontend erzeugt Backtracing-Code, welcher Funktionszeiger verwendet. In Java ist die Verwendung von Funktionszeigern nicht möglich. Das Java-Backend kapselt deswegen Funktionen, deren Adressen in dem TL-Code Zeigervariablen zugewiesen werden, in einer Klasse, die das Interface `Backtrace` implementiert. Das Interface `Backtrace` enthält nur eine Methodendeklaration. Die Funktionszeigervariablen der TL werden im Java-Code durch Objektreferenzvariablen von

[5] `java.lang.Cloneable`

```
public class Algebra {
  class bt_back_struct implements Backtrace {
    public str1 back(int i, int j, int diff) { .. }
  }
  bt_back_struct back_struct = new bt_back_struct();
  ...
  void do_stuff() {
    Backtrace f1;
    ...
    f1.back(...);
  }
}
```

Abbildung 3.4: Emulation von C-Funktionszeigern in Java durch Kapselung der Funktion und Implementierung von einem Interface, wodurch bei einer Funktionszeigerähnlichen Verwendung der Klasse von dem konkreten Klassennamen abstrahiert wird.

dem Typ `Backtrace` ersetzt und jeder Funktionsaufruf über einen Funktionszeiger in der TL durch den Aufruf der in dem Interface deklarierten Methode. Die Klassen der gekapselten Funktionen sind innere Klassen der Algebraklasse, da sie ein Implementationsdetail der Algebraklasse darstellen. Jeweils ein Objekt von jeder dieser Hilfsklassen wird bei Instantiierung der äußeren Klasse erzeugt und einer Objektreferenzvariable zugewiesen. Ein vereinfachtes Beispiel dieser Methodenkapselung und der Emulation von C-Funktionspointern in Java ist in Abbildung 3.4 dargestellt.

3.2.4 Zeiger auf Zeiger

Zeiger auf Zeiger werden in dem TL-Code verwendet, um von Namen von Pointer-Variablen zu abstrahieren, welche Teil von verschiedenen Structs sind. An einer Stelle des TL-Codes wird die Adresse von einem Pointer, welcher Teil einer Struct ist, gespeichert, damit an einer späteren Stelle in einer anderen Funktion dieser Zeiger auf einen Zeiger verwendet werden kann, um unabhängig von dem konkreten Namen der Struct-Komponente dieser Komponente einen neuen Wert zuweisen zu können. Eine Speicherung der Adresse der Struct ist nicht möglich, da die Struct mehrere Komponenten enthalten kann, deren Namen der spätere Zuweisungscode nicht kennt. Ein Beispiel von einem vereinfachten TL-Code in C-Notation ist in Abbildung 3.5 angegeben.

In Java können nur Referenzen von Objekten in Referenzvariablen gespeichert werden. Eine Verwendung von einer Referenz auf eine Referenz ist nicht möglich. Um für den betroffenen TL-Code Java-Code zu erzeugen, gibt es mehrere Möglichkeiten. Wenn die Codegenerierung des Frontend so verändert wird, dass die

```
struct foo {
  struct bar *a1;
  int a2; ...;
  struct bar *a5;
};
struct foo *x, *y;
...
struct bar **temp = & x->a1;
...
*temp = y->a1;
```

Abbildung 3.5: C-Beispiel für die Verwendung von Zeigern auf Zeiger, um von den Namen von Komponenten einer Struct zu abstrahieren.

Einzel-Komponenten von Structs, welchen später ein Wert über eine Zeiger-auf-Zeiger-Konstruktion zugewiesen wird, Teil eines der Struct zugehörigen Arrays sind, dann sind für den abstrakten Zugriff keine Zeiger auf Zeiger mehr notwendig. Für eine spätere Zuweisung muss dann nur ein Zeiger auf das Array und der Index des betroffenden Elements gespeichert werden. Dieser TL-Code könnte direkt in Java-Code übersetzt werden, da Arrays in Java Objekte sind, deren Referenz wie normale Objektreferenzen verwendet werden können. Der Nachteil bei dieser Alternative ist allerdings der große Implementierungsaufwand, da an einigen Stellen des Codegenerierungsmoduls umfangreiche Funktionen neugeschrieben werden müssen.

Java enthält einige dynamischen Sprachmerkmale, welche typisch für Scriptsprachen, wie z.B.Python [34] oder Ruby [5], sind, die den Zugriff auf vielfältige Objekt- und Klasseninformationen zur Laufzeit ermöglichen. Beispielsweise kann zur Laufzeit ermittelt werden, ob ein Objekt ein bestimmtes Interface implementiert, oder eine Methode eines Objektes kann an Hand von ihrem Methodennamen als String (der zur Laufzeit dynamisch konstruiert werden kann) aufgerufen werden. In Java wird auf diese Informationen über die Reflection-API [15] zugegriffen.

Das Java-Backend kann nun Code erzeugen, welcher die Reflection-API verwendet, um die Verwendung von Zeigern auf Zeiger in TL zu emulieren. Dazu wird an der Stelle, wo in dem TL-Code die Adresse des Zeigers gespeichert wird, ein Field-Objekt, welches Teil der Reflection-API ist, erzeugt, mit dessen Hilfe später dem Attribut oder Feld der Klasse unabhängig von dem Namen ein Wert zugewiesen werden kann. Abbildung 3.6 zeigt ein vereinfachtes Beispiel von einem Java-Code, der diese Möglichkeit verwendet.

Allerdings zeigt das Profiling des erzeugten Java-Codes von übersetzten ADP-Programmen, dass 40 Prozent der Laufzeit in Reflection-Routinen verbracht werden. Dies ist zu erwarten, da die Verwendung von Reflection-API erheblichen zusätzlichen Aufwand bedeutet, den die JVM durchführen muss. Beispielsweise muss die JVM in Symboltabellen nachschauen, ob überhaupt die Klasse ein Attribut

```
class foo { bar a1; };
foo x, y;
bar temp;
Object temp_obj;
Field temp_field;
...
temp = x.a1;
temp_obj = x;
temp_field = x.getClass().getField("a1");
...
temp_field.set(temp_obj, y.a1);
```

Abbildung 3.6: Vereinfachter Java-Code, der die Reflection-API verwendet, um Zeiger auf Zeiger Konstrukte der TL in Java abzubilden. Der notwendige Code, um die möglichen Java-Exceptions zu behandeln, ist ausgelassen.

mit dem angegebenen Namen enthält, ob die Sichtbarkeit ausreicht und ob die Sicherheitsregeln der VM den Zugriff erlauben.

Zeiger-auf-Zeiger-Konstrukte können in Java allerdings auch ohne die Verwendung der Reflection-API abgebildet werden. Dazu wird für jedes Attribut einer Klasse, dessen Adresse in der TL-Eingabe einem Zeiger auf einen Zeiger zugewiesen wird, eine innere Klasse erzeugt, welche den Zugriff auf das Attribut abstrahiert. Der Zugriff ist abstrahiert, weil die innere Klasse das allgemeine Interface `Set` implementiert. Die Referenz auf die innere Klasse wird als Typ von dem Interface `Set` gespeichert, und an der Stelle, wo in der TL-Eingabe der doppelte Zeiger auf der linken Seite einer Zuweisung einmal de-referenziert wird, wird in der Java-Ausgabe die `set()`-Methode aufgerufen (Beispiel in Abbildung 3.7).

Die letzte Möglichkeit der Codeerzeugung hat die Verwendung der Reflection-API im Java-Backend abgelöst, da die zu erwartende Optimierung der Laufzeit erreicht werden konnte. Das Profiling von erzeugten Java-Programmen zeigt eine Laufzeitverbesserung um 50 Prozent[6].

3.2.5 Tupel

ADP-Algebren können nicht nur elementare Datentypen wie Integer oder Double der Sorte zuordnen, sondern auch Tupel von elementaren Datentypen.

Bei der Erzeugung von ADP nach C können Tupel ohne großen Aufwand als Structs modelliert werden, denn Structs können wie elementare Datentypen auf dem Stack angelegt werden und wie elementare Datentypen durch eine einfache

[6] auf einer Sun-Fire-v20z, 2.4 GHz, 12 GB RAM, Solaris 10, Sun Java 1.6.2; Als Laufzeit wird die utime (user time) des Prozesses gemessen.

```
interface Set {
  void set(bar z);
}
class foo {
  bar a1;
  class Proxy_a1 implements Set {
    public void set(bar z) { a1 = z; }
  }
  Set proxy_a1 = new Proxy_a1();
}
foo x, y; bar temp; Set temp_set;
...
temp = x.a1;
temp_set = x.proxy_a1;
...
temp_set.set(y.a1);
```

Abbildung 3.7: Vereinfachter Java-Code, der die Verwendung von Zeigern auf Zeiger in Java emuliert.

Zuweisung kopiert werden. Die Erstellung von Arrays von Structs ist unproblematisch, da nach einer Allokation des gesamten Speicherbereiches die Komponenten einer Struct-Instanz nacheinander im Speicher liegen.

Für die Erzeugung von Java-Zielcode ist die Abbildung von Tupeln aufwändiger. Tupel können auf Klassen abgebildet werden. Objekte dieser Tupel-Klassen können aber nicht auf dem Stack angelegt werden, so dass die Objekte vor ihrer Verwendung explizit auf dem Heap angelegt und initialisiert werden müssen. Die Variablen in der TL vom Typ Struct werden in Java zum Typ Objekt-Referenz, so dass bei einfachen Zuweisungen nur die Objektreferenzen von existierenden Objekten zugewiesen werden. Wenn das Ziel der Zuweisung im folgenden Programmablauf nicht verändert wird, ist dies kein Problem. Die anderen Fälle müssen von dem Backend erkannt werden, damit in diesen Fällen extra Code erzeugt wird, der unter Behandlung von möglichen Exceptions die Objekte klont. Die Erstellung von Objekt-Arrays, in denen die Speicherbereiche der beteiligten Objekte in dem Speicherbereich des Arrays liegen, ist in Java nicht möglich. Stattdessen werden Arrays von Objektreferenzen angelegt. Die Tupel-Objekt-Instanzen werden einzeln auf dem Heap angelegt und ihre Referenzen in den Arrays gespeichert. Dies verschwendet Speicherplatz, da für jedes einzelne allokierte Objekt in der Speicherverwaltung Verwaltungsinformationen abgelegt werden müssen, und ein Array von Objektreferenzen erzeugt wird. Außerdem liegen die Objekte möglicherweise nicht benachbart im Speicher, sondern irgendwo im Heap, so dass unter Umständen nicht von der CPU-Cache bei sequentiellen Zugriffen profitiert werden kann.

3.2.6 Tabellierungsschemata

Die vom ADP-Compiler erzeugten Zielprogramme bestehen aus zwei Phasen. In der ersten Phase werden in einer doppelten For-Schleife die notwendigen Tabellen, welche die optimalen Teilergebnisse speichern, berechnet. Danach wird der optimale Score ausgegeben, und es beginnt die Backtracing-Phase.

Da bei einem $O(n^k)$-DP-Algorithmus $O(n^k)$ Tabellenzugriffe erfolgen, ist es wichtig, die Tabellen möglichst so zu repräsentieren, dass die Zugriffszeit minimiert wird, wodurch der konstante Faktor der asymptotisch optimalen Laufzeit verringert wird.

Die Tabellen sind im Allgemeinen zweidimensional und bei ADP auf Sequenzen ist nur das obere Dreieck der Matrix besetzt. Tabellierte Nichtterminale, bei deren Zugriff ein Index konstant bleibt, können mit linearem Speicherbedarf tabelliert werden [17].

Dynamische mehrdimensionale Arrays sind als eigenständiges Sprachkonstrukt nicht Teil von C. Als naive Lösung kann man in C ein zweidimensionales $n \times m$ Array A erzeugen, indem man zuerst ein Array von n Zeigern erzeugt, danach n m-elementige Zeilen erzeugt, deren Adressen im Zeiger-Array gespeichert werden. Auf $A_{i,j}$ kann man dann in C via `A[i][j]` zugreifen. Die Konstruktion von höherdimensionalen Arrays bzw. spaltenweisen Allokationen erfolgt analog zu diesem Schema. In diesem Abschnitt werden nur zweidimensionale Tabellierungsschemata betrachtet, da mehr Dimensionen für Single-Track-ADP auf Sequenzen nicht notwendig sind.

Das Problem dieses naiven Schemas ist zum einen, dass die Zeilen einzeln allokiert werden, womit Speicher durch Verwaltungsinformation des System-Allokators verschwendet wird und die Zeilen möglicherweise getrennt voneinander im Speicher liegen, so dass benachbarte Zeilen nicht von der CPU-Cache berücksichtigt werden können.

Um dieses Problem zu vermeiden, werden in einer Abwandlung von dem naiven Schema die Zeilen nacheinander in einem Speicherblock allokiert. In dem Zeiger-Array werden nun die Zeilenanfangsadressen, welche innerhalb des Speicherblocks liegen, gespeichert.

Zum anderen besteht bei beiden Schemata das Problem, dass ein Tabellenzugriff auf $A_{i,j}$ zwei Speicherzugriffe benötigt. Zuerst wird die Anfangsadresse x der i-ten Zeile geladen, und dann kann das Element von der Adresse $x + j$ de-refernziert werden.

Der erste Speicherzugriff und damit das Zeiger-Array kann eingespart werden, indem bei jeden Zugriff das Zeilenoffset berechnet wird. Also kann nach diesem Schema auf $A_{i,j}$ in C mit `A[i* j + i]` bei quadratischer Tabellierung bzw. mit `A[i*(i+1)/2]`, wenn nur das obere Dreieck der Matrix gespeichert wird, zugegriffen werden.

Die Überzeugung ist relativ weit verbreitet, dass das zweite Schema effizienter als das dritte Schema beim Array-Zugriff ist. Denn im zweiten Schema wird bei jedem Zugriff eine Multiplikation und eine Addition eingespart, indem für jede Zeile diese arithmetische Operation tabelliert wird. Eine Laufzeitersparnis kommt dadurch

allerdings nicht zu Stande, denn ein Speicherzugriff ist teurer als eine arithmetische Operation, bestehend aus Multiplikation und Addition bzw. Multiplikation, Addition und einem Shift. Auch wenn das Zeiger-Array gecacht ist, ist der Register-Zugriff bei der arithmetischen Operation günstiger. Für den GCC [1] existiert z. B. ein Optimierungsmodus, welcher Code erkennt, der mehrdimensionale Arrays nach dem naiven Schema allokiert, und durch Code ersetzt der die Arrays linearisiert und Offsetberechnungen durchführt [22]. Zusätzlich können nach einer Heuristik Transponierungen von Arrays durchgeführt werden, wenn bei einer zeilenweisen Allokation ein überwiegend spaltenweiser Zugriff bzw. bei einer spaltenweisen Allokation ein überwiegend zeilenweiser Zugriff erkannt wird. In [22] wird berichtet, dass die Linerarisierung von mehrdimensionalen Arrays bei zwei ausgewählten Programmen zu einer Laufzeitverringerung von 9 bzw. 39 Prozent führt.

Als Standard erzeugt der ADP-Compiler Code, in dem Offsetberechnungen tabelliert werden[7]. In seinem Dissertationsvortrag hatte Peter Steffen, auf Nachfrage nach dem experimentellen Fortran-Target-Code-Support, berichtet, dass bei ADP-Programmen, welche die paarweise Edit-Distanz berechnen, der kompilierte Fortran-Target-Code um bis zu 30 Prozent schneller als der kompilierte C-Target-Code ist. Dies lässt sich durch das im C-Code verwendete Tabellierungsschema erklären, da mehrdimensionale Arrays in Fortran Teil der Sprache sind, und somit Zugriffe (einfacher) von einem Fortran-Compiler optimiert werden können.

Um diese Hypothese zu untersuchen, ist ein Benchmark[8] des ADP-Programms `AffineLocSim`, welches für das paarweise lokale Alignment unter affinen Gap-Kosten berechnet, durchgeführt worden, indem die Tabellierung der Zeilenoffsets mit der Berechnung der Zeilenoffsets für jeden Zugriff verglichen wird. Bei einer zufällig generierten Sequenz von 20000 Basen, welche nach 15000 Zeichen durch ein Sentinel-Zeichen getrennt ist, führt die Nicht-Tabellierung der Zeilenoffsets zu einer Laufzeitverbesserung des C-Zielcodes von 26 Prozent. Der betrachtete DP-Algorithmus hat eine quadratische Laufzeit und verwendet drei Tabellen. Im imperativen Zielcode werden in der inneren For-Schleife nur wenige Vergleiche und Additionen von Gap-Kosten durchgeführt. Auf Grund des eingesparten Speicherzugriffs bei jedem Tabellenzugriff und der engen Haupt-Schleife, die wenig mehr Operationen als Speicherzugriffe enthält, ist die beobachtete Laufzeitverbesserung zu erwarten. Die Verbesserung entspricht der beobachteten Laufzeitverringerung des Fortran-Zielcodes, so dass die Verwendung des dritten Tabellierungsschemas im kompilierten Fortran-Code die wahrscheinliche Ursache für die beobachtete Differenz ist[9].

[7] Das Default wird wegen der genannten Gründe und den durchgeführten Benchmarks entsprechend dem dritten Tabellierungsschema geändert.

[8] durchgeführt auf einer Sun-Fire-v20z, 2.4 GHz, 12 GB RAM, Solaris 10, `-O2` Compileroption; Als Laufzeit wird die utime (user time) des Prozesses gemessen. Es wird kein Backtracing durchgeführt.

[9] Aus Zeitgründen ist auf die Wiederholung des Fortran Benchmarks und die Analyse des Assembler-Output eines Fortran-Compilers verzichtet worden, was die Hypothese zweifelsfrei belegen bzw. widerlegen würde.

Die Verwendung des dritten Tabellierungschemas, also die Berechnung der Offsets bei jedem Tabellenzugriff, im erzeugten Java-Code führt bei dem AffineLocSim-Programm zu einer geringen Laufzeitverbesserung von 2 Prozent im Vergleich zu der Tabellierung der Offsets[10]. Dies könnte unter anderem durch Zugriffsüberprüfungen der JVM zur Laufzeit erklärt werden. Die JVM prüft bei jedem Array-Zugriff, ob der Array-Index sich in den Grenzen des allokierten Arrays befindet. Die Ausführung der bedingten Sprunganweisungen überlagert möglicherweise in einer superskalaren CPU-Architektur den Speicherzugriff. Die Verwendung von zweidimensionalen Arrays, welche Teil der Java-Sprache sind, führt zu einer Laufzeitverschlechterung von über 1300 Prozent. Dieses Ergebnis ist schwer zu erklären. Es wird zwar mehr Speicher allokiert, da nicht nur die oberen Dreiecke der Tabellen allokiert werden, aber es werden auch von diesen Tabellen nur die Einträge der Diagonale und oberhalb berechnet. Der Einfluss der Garbage-Collection sollte gering sein, da in allen Benchmarks die Backtracing-Phase ausgelassen worden ist, so dass nur zu Programmstart Speicher allokiert wird, der nur zum Programmende freigegeben wird. Die zusätzlichen Zugriffsüberprüfungen der JVM für den zweiten Array-Index erklären auch nicht diesen krassen Laufzeitunterschied.

Bei anderen ADP-Programmen, wie zum Beispiel RNAfold, sind im Java- und C-Zielcode keine Laufzeitunterschiede bei den beiden untersuchten Tabellierungsschemata fesstellbar. Dies liegt daran, dass bei RNAfold in der inneren Schleife einige Funktionen Energiewerte aus Hilfstabellen nachschlagen, so dass die eingesparten Speicherzugriffe bei den DP-Tabellenzugriffen nicht mehr ins Gewicht fallen bzw. von anderen Operationen der CPU überlagert werden.

3.3 Schnittstellen

Bei der Kompilierung eines ADP-Programms mit dem ADP-Compiler muss eine Algebra angegeben werden, die in dem Ausgabe-Code eingesetzt sein soll. Für mehrere Algebren können mehrere Code-Module erzeugt werden, die den gleichen ADP-Algorithmus, aber in Verbindung mit verschiedenen Algebren implementieren. Das Java-Backend erzeugt für ein Code-Modul eine Klasse, welche als Namenspräfix den String `Algebra_` besitzt.

Damit die erzeugten Klassen einfach in Java-Programme integriert werden können, stellt jedes Java-Code-Modul zwei Schnittstellen bereit. Zum einen erweitert jede Code-Modul-Klasse eine abstrakte Klasse. In Abhängigkeit von dem ADP-Algorithmus ist die Basisklasse entweder `StdAlgebra` (Abbildung 3.9) oder RnaAlgebra (Abbildung 3.10). Diese beiden Klassen erweitern wiederum die abstrakte Basisklasse Algebra (Abbildung 3.8), welche die gemeinsamen Anteile beider Klassen konsolidiert. Das Code-Modul erweitert `StdAlgebra`, wenn es sich um einen ADP-Programm handelt, das auf allgemeinen Strings arbeitet. Wenn das ADP-

[10] Sun-Fire-v20z, 2.4 GHz, 12 GB RAM, Solaris 10, JDK 1.6.2, JVM Optionen: `-d64 -Xmx7128m -Xss64m`; Als Laufzeit wird die utime (user time) des Prozesses gemessen. Es wird kein Backtracing durchgeführt.

```
public abstract class Algebra {
  public abstract void start();
  public abstract void freeall();
  public abstract void set_traceback_diff(int i);
  public abstract void getTree(Node n);
  public abstract void setOutput(Output output);
}
```

Abbildung 3.8: Abstrakte Basisklasse, welche von allen übersetzten ADP-Programmen erweitert wird.

```
public abstract class StdAlgebra extends Algebra {
  public abstract void init(String s, StdOptions o);
}
```

Abbildung 3.9: Abstrakte Klasse, die von allgemeinen nach Java übersetzten ADP-Programmen erweitert wird. Diese ADP-Programme arbeiten auf allgemeinen Strings.

```
public abstract class RnaAlgebra extends Algebra {
  public abstract void init(Sequence s, Options o);
}
```

Abbildung 3.10: Abstrakte Klasse, die von erzeugten Java-Code-Modulen erweitert wird, welche auf bioinformatischen Sequenzen arbeiten.

Programm auf RNA- bzw. DNA-Sequenzdaten arbeitet, wird `RnaAlgebra` erweitert, weil diese Eingabedaten von dem generierten Java-Code in einer anderen Repräsentation erwartet und andere Optionen berücksichtigt werden.

Da die Code-Module eine abstrakte Klasse erweitern, kann ein Java-Programm die instantiierten Objekte von einem ADP-Algorithmus einer Referenzvariable zuweisen, die von dem Typ `StdAlgebra` oder `RnaAlgebra` ist. Die Java-Code-Modul-Klasse kann einfach ausgetauscht werden, da sich der Typ, der erweitert wird, nicht ändert und somit die von dem Java-Programm verwendete Schnittstelle weiterverwendet werden kann. Es wird von dem konkreten Algorithmus auf einen allgemeinen bzw. RNA-spezifischen ADP-Algorithmus abstrahiert.

Zum anderen enthält die abstrakte Basisklasse `Algebra` die Methode `setOutput()` mit der eine Klasse registriert werden kann, die für die Ausgabe eines ADP-Algorithmus zuständig ist und deswegen das Interface `Output` implementieren muss (siehe Abbildung 3.11). Von dem Interface existiert eine Standard-Implementation `DefaultOutput` für die Ausgabe auf ein Terminal nach `stdout`. Wenn eine Java-

```
public interface Output {
  public void prelude(String s);
  public void start(int diff);
  public void optimal(int score);
  public void optimal(double score);
  public void optimal(DoubleTupel score);
  public void optimal(IntTupel score);
  public void suboptimal(int score, StringBuilder s);
  public void end();
}
```

Abbildung 3.11: Das `Output`-Interface spezifiziert die Schnittstelle, die eine Klasse zur Behandlung der Ausgabe eines ADP-Algorithmus implementieren muss. Die Methode optimal ist überladen, da der der Typ der Auswertung des optimalen Kandidaten von der der Algebra eines ADP-Programm abhängt.

Anwendung die Ausgabe von einem ADP-Algorithmus speziell behandeln möchte, kann das Interface `Output` implementiert werden und bei dem ADP-Java-Code-Modul registriert werden. Beispielsweise kann so die Ausgabe sortiert, weiterverarbeitet oder in einem Ausgabe-Widget einer GUI dargestellt werden.

Die Schnittstellen Algebra, `StdAlgebra`, `RnaAlgebra` und `Output` sind Teil des ADPC Java-Pakets. Die Beziehung der Klassen bzw. des Interfaces untereinander ist in der Abbildung 3.12 dargestellt.

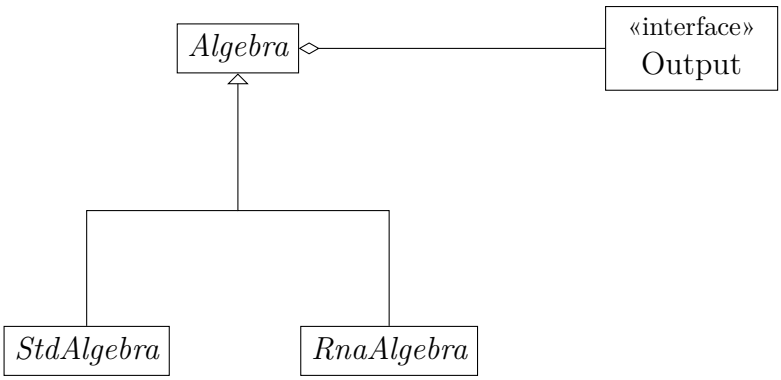

Abbildung 3.12: UML (Unified Modelling Language [12])-Diagramm der Schnittstellen die ein nach Java übersetztes ADP-Programm bereitstellt.

3.4 Metafrontend

Der ADP-Compiler wird über den Befehl `adpcompile` aufgerufen. Die Kommandozeilenoptionen sind zahlreich. Beispielsweise muss die Algebra angegeben werden, die für das Inlining verwendet wird, und es können eine Pretty-Print-Algebra, das Tabellierungschema, der Tabellenoptimierungsalgorithmus und viele weitere Detailoptionen angegeben werden. Die Erzeugung von Code, der einen Window-Mode unterstützt, ist optional möglich, wird aber nur bei RNA-spezifischen ADP-Programmen benötigt.

Außerdem fehlt dem ADP-Compiler das Spezialwissen für bestimmte Klassen von ADP-Programmen, welches notwendig ist, um an einigen Stellen im Zielcode spezialisierten Code zu erzeugen. Zur Zeit werden RNA-spezifische ADP-Programme und allgemeine ADP-Programme unterschieden. Die Unterschiede liegen z. B. darin, wie die Eingabesequenz repräsentiert wird oder dass RNA-ADP-Programme zusätzliche Bibliotheken benötigen.

Des Weiteren ist ein vom ADP-Compiler erzeugtes Code-Modul nicht direkt kompilier- und ausführbar. Der Compiler erzeugt keine zusätzlichen Module, die z. B. eine Benutzerschnittstelle für das direkte Testen der ADP-Programme bereitstellen.

Aus diesen Gründen ist die alleinige Bereitstellung der Benutzerschnittstelle adpcompile zur Kompilierung von ADP-Programmen für den ADP-Entwickler nicht ausreichend. Es wird ein Frontend für den ADP-Compiler benötigt. Da dieses Frontend vor dem Frontend des ADP-Compiler aufgerufen wird, wird es im Folgenden als Metafrontend bezeichnet.

Das Metafrontend soll nur mit dem ADP-Programm als Eingabe aufgerufen werden können und eine Menge von sinnvollen Standard-Optionen für die ADP-Programm-Klasse auswählen. Dazu muss das Metafrontend einen Algorithmus implementieren, der erkennt, ob es sich um ein RNA-spezifisches oder ein allgemeines ADP-Programm handelt. Auch muss die Pretty-Print-Algebra erkannt werden, falls das ADP-Programm eine enthält. Für alle weiteren Algebren, welche in der Eingabe enthalten sind, soll als Standard jeweils von dem ADP-Compiler ein Code-Modul erzeugt werden, wofür der ADP-Compiler von dem Metafrontend mit jeweils unterschiedlichen Optionen aufgerufen werden muss. In Abhängigkeit von der ADP-Programm-Klasse müssen im Zielcode einige spezifische Codeteile eingesetzt werden. Damit die erzeugeten Code-Module direkt über eine einheitliche Benutzerschnittstelle verwendet werden können, muss das Metafrontend ein extra Code-Modul erzeugen, welches diese Schnittstelle implementiert und von den für jede Algebra erzeugten Code-Modulen abhängt.

3.4.1 Bisheriger Stand

Die ADP-Compiler-Distribution enthält ein Metafrontend für die ADP-Übersetzung nach C. Dieses Frontend heißt XML-Interfacer und ist in dem Unterverzeichnis `interfacer` enthalten. Der XML-Interfacer besteht hauptsächlich aus dem Haskell-

Modul `Adpc.lhs`. Das kompilierte Metafrontend kann über die Kommandozeile mit `adpc` aufgerufen werden. Der Name XML-Interfacer resultiert aus der Implementierung, die XML als Konfigurationsdateiformat verwendet.

Das Metafrontend wurde nicht für die Verwendung von mehrere Zielsprachen entwickelt, sondern für die Verwendung von C als Zielsprache, denn es enthält keine Abstraktion, wie z. B. die durchgängige Trennung von sprachspezifischen Code in separate Funkionen, die eine einfache Erweiterung für weitere Zielsprachen zuließe.

Außerdem ist das Metafrontend nicht portabel. Zum einen ist es nicht portabel zwischen verschiedenen Unix-Varianten, da z.B. externe Systemaufrufe des `sed`-Kommando geschehen, die nicht portabel sind[11] oder externe Programme, wie `finger` aufgerufen werden, die nicht auf allen Unix-System vorhanden sind. Zum anderen ist es nicht portabel auf Windows-Systeme bzw. nicht-Unix-artige Systeme, da einige externe Unix-Befehle, wie `sh`, `sed`, `grep`, `cat` und `mv` aus dem XML-Interfacer heraus aufgerufen werden und für Pfadangaben statisch das Unix-Pfadtrennzeichen verwendet wird.

Die Codeerzeugung mit Hilfe des XML-Interfacer ist zusätzlich auf das Unix-Utility `make` angewiesen, da der XML-Interfacer ein Makefile erzeugt, das die eigentlichen Aufrufe des ADP-Compilers und weiterer Hilfsprogramme enthält.

Die Fehlerbehandlung des Metafrontend ist nicht ausreichend, da Fehler in externen Programmaufrufen, welche durch einen Exit-Code ungleich 0 signalisiert werden, nicht an allen Stellen berücksichtigt werden und die zugehörigen Fehlermeldungen nicht ausgegeben werden. Dies hat zur Folge, dass z. B. Fehler in dem ADP-Programm nicht nach dem Aufruf des ADP-Compilers gemeldet werden und der Vorgang abgebrochen wird, sondern erst Folgefehler in dem C-Compiler zu dem Abbruch der Übersetzung führen. Die Fehlermeldungen der Folgefehler führen zu für den ADP-Entwickler irreführenden Fehlermeldungen.

Wenn das ADP-Programm keine Pretty-Print-Algebra enthält, meldet der XML-Interfacer einen Fehler, obwohl der ADP-Compiler auch ADP-Programme übersetzen kann, die keine Pretty-Print-Algebra enthalten. In diesem Fall generiert der ADP-Compiler eine Pretty-Print-Algebra automatisch, die die Termdarstellung des Kandidaten in einer Präfixform darstellt.

Der XML-Interfacer enthält keinen Parser für die ADP-Syntax. Stattdessen werden System-Aufrufe aus `sed` und `grep` konstruiert, die benötigte Informationen, wie die Menge aller Algebranamen, aus dem ADP-Programm filtern. Auch die Erkennung, ob das ADP-Programm RNA-spezifisch ist oder nicht, wird durch einen einfachen grep-Aufruf realisiert. Das Problem dabei ist, dass einfach nur nach dem String ‚energy' in dem ADP-Programm gesucht wird. Wenn das Programm diesen String enthält, wird es als ein RNA-spezifisches Programm behandelt. Allerdings schlägt diese Strategie fehl, wenn energy in einem Kommentar oder als Teil von einem Bezeichner in einem allgemeinen ADP-Programm verwendet wird.

Die ADP-Syntax ist an Literate-Haskell orientiert, so dass alle Zeilen die nicht mit einem `>`-Zeichen beginnen, implizit Kommentare sind und die Abseitsregel [23]

[11] Verwendung von einem Newline in dem Muster

gilt. Die von dem Metafrontend verwendeten `sed`-Muster können die Abseitsregel nicht allgemeingültig berücksichtigen, und entweder wird auch auskommentierter Code berücksichtigt oder Code, der eine Einrücktiefe ungleich 1 hat, ignoriert. Dies trifft z. B. auch auf die `#algebra`-Direktive zu.

Da der XML-Interfacer keinen eigenen Parser enthält, wird die mögliche Fehlererkennung bis auf den Aufruf des ADP-Compilers verzögert, und für jede Algebra muss das ADP-Programm aufgrund der mehrmaligen Aufrufe des ADP-Compilers wiederholt geparst werden.

Die spezifische Code-Erzeugung für bestimmte Konstrukte in Abhängigkeit von der Klasse, aus dem ein ADP-Programm stammt, wird durchgeführt, indem das C-Backend jeweils zwei durch `@`-Zeichen eingeschlossene Marker an den entsprechenden Stellen in den C-Zielcode einfügt. Der XML-Interfacer erzeugt `sed`-Skripte, die diese Marker durch spezifischen Code ersetzen, und den entsprechenden Makefile-Code, der diese `sed`-Skripte als Post-Prozessor auf die Ausgabe des ADP-Compiler anwendet.

Allgemein werden die Aufrufe von externen Programmen, wie `sed`, über die Haskell-Funktion `system` implementiert, deren Semantik der C-Funktion gleichen Namens entspricht. Die Ausgabe wird in eine temporäre Datei geleitet, welche direkt wieder zur weitereren Bearbeitung eingelesen wird.

3.4.2 Lösung für Java

Aufgrund der eingeschränkten Portabilität, der schwierigen Erweiterbarkeit und zur Lösung der übrigen beschriebenen Probleme des existierenden XML-Interfacers wird für die Übersetzung von ADP nach Java ein neues Metafrontend entwickelt und implementiert.

Das neue Metafrontend ist komplett in Haskell geschrieben, es enthält keine System-Aufrufe von Helferprogrammen, wie `sed` oder grep. Es ist nicht auf ein Build-Werkzeug, wie make angewiesen. Zur Konstruktion der Systempfade wird die Haskell-API verwendet. Dadurch kann das Metafrontend auf jede Architektur portiert werden, für die ein Haskell-Compiler existiert. Außerdem ist die Erzeugung von zahlreichen temporären Dateien nicht mehr notwendig, was die Übersetzungsgeschwindigkeit verbessert.

Die Kommunikation mit dem ADP-Compiler findet nicht mehr über temporäre Dateien und `system`-Aufrufe statt, sondern das Metafrontend ruft direkt Funktionen des ADP-Compilers auf. Dazu ist der ADP-Compiler erweitert worden, damit der Parser und die restliche Übersetzungsfunktionalität separat aus einem Haskell-Modul heraus verwendet werden kann. Zu den Haskell-Modulen des Metafrontend werden die benötigten Haskell-Module des ADP-Compilers hinzugelinkt. Das Metafrontend parst das ADP-Programm und ruft bei Erfolg für jede Algebra die zentrale Compile-Funktion des ADP-Compiler mit dem Parse-Tree als Argument auf. So wird ein zeitaufwändiger redundanter Aufruf des ADP-Parsers vermieden. Im Fehlerfall werden die Fehlermeldungen des ADP-Compilers ausgegeben.

Ein Post-Prozessor, der die Text-Ausgabe des Backends neu einlesen muss, um die

```
%MARKER1%  {
Ersetzung
}

%MARKER2%
{
Ersetzung
}
```

Abbildung 3.13: Beispiel des Formats für die Konfiguration des neuen Metafrontends, in dem die Ersetzungen der TLMacro Konstruktoren spezifiziert werden. Der String zwischen den Prozentzeichen matcht das String-Argument des TLMacro-Konstruktors. Der TL-Konstruktor wird durch den Text ersetzt, der zwischen den geschweiften Klammern steht.

Marker zu ersetzen, ist vermieden worden. Die TL-Sprache ist um den Konstruktor TLMacro und die zentrale Compile-Funktion `cmainInt`, um eine Filter-Funktion als Argument erweitert worden. Die Filter-Funktion ist von dem Typ $[TL] \rightarrow [TL]$ und wird in dem ADP-Compiler auf die TL-Ausgabe des Frontends angewendet. So kann das Metafront cmainInt eine Funktion übergeben, welche im TL-Code die Vorkommen von TLMacro durch spezialisierten TL-Code ersetzt, bevor das Backend den TL-Code übersetzt. Diese Filterung durch eine von dem Metafrontend bereitgestelle Funktion ist notwendig, da nur im Metafront die Information vorhanden ist, ob das ADP-Programm RNA-spezifisch oder allgemein ist.

Da das Metafrontend Zugriff auf den Parse-Tree des ADP-Programms hat, geschieht die Erkennung der ADP-Programm-Klasse und die Ermittlung der Algebra-Namen zuverlässig auf dem Parse-Tree. Die Behandlung der Abseitsregel und von Kommentaren wird in dem existierenden Parser des ADP-Compilers erledigt. Wenn das ADP-Programm eine Pretty-Print-Algebra enthält, wird sie an Hand des Namens erkannt und entsprechend an den ADP-Compiler übergeben. Das ADP-Programm muss aber keine Pretty-Print-Algebra enthalten, da in diesem Fall der ADP-Compiler mit anderen Optionen aufgerufen, und die Pretty-Print-Algebra automatisch generiert wird.

Der spezifische Code, der in Abhängigkeit von der Zielsprache und der ADP-Programmklasse von der Filterfunktion eingesetzt wird, ist nicht in dem Metafrontend hart kodiert, sondern in externen Konfigurationsdateien spezifiziert. So muss das Metafrontend nicht neukompiliert werden, wenn sich der zu ersetzende Code ändert. Das Format dieser Konfigurationsdateien ist in Abbildung 3.13 beschrieben.

Generell ist das Metafrontend neben der Verwendung von externen Konfigurationsdateien so aufgebaut, dass es einfach um die Übersetzung weiterer Sprachen mit Hilfe dieses Metafrontends erweitert werden kann. Die Java-spezifischen Teile

-j Erzeugt Java Zielcode (default).
-c Erzeugt C Zielcode
-r Forciert die Behandlung des ADP-Programms als RNA-spezifisches Programm
-n Forciert die Behandlung des ADP-Programms als allgemeines ADP-Programm
-a Liste der zu übersetzenden Algebren (default: alle)
-t explizite Angabe der Konfigurationsdatei
-p Verzeichnis der Konfigurationsdateien
-f zusätzliche Optionen, die dem ADP-Compiler übergeben werden. Sie überschreiben die Default-Optionen, die Metafrontend ermittelt
-l listet alle Algebren auf, die das Metafrontend in dem ADP-Programm findet
-o Angabe des Ausgabeverzeichnis für den erzeugten Code (default: ./ADP-Programmname)
-x Java-Package-Name (default: ADP-Programmname)
-v ausführlichere Bildschirmausgabe
-h Hilfe

Abbildung 3.14: Optionale Parameter des neuen Metafrontend, die das Standardverhalten beeinflussen.

sind in eigene Funktionen ausgelagert, die die Sprache als Argument übergeben bekommen. Bei einer Erweiterung wird über Pattern-Matching die sprachspezifische Funktion ausgewählt.

Der Quellcode des Metafrontends ist in dem Verzeichnis `java` des `adp2java`-Entwicklungszweigs enthalten. Das Modul `Macro` enthält den Parser für das Konfigurationsdateiformat, das Modul `Helpers` einige Hilfsfunktionen und das Modul `FrontEnd` das eigentliche Metafrontend. Aufgerufen wird es über die Kommandozeile mit `fe`. Es kann mit dem ADP-Programm als alleinigen Parameter aufgerufen werden und erzeugt Code-Module mit sinnvollen Defaults. Ein Code-Modul, welches die Benutzerschnittstelle für ein bestimmtes ADP-Programm bereitstellt, wird ebenso erzeugt. Optional kann durch Highlevel-Optionen das Verhalten des Metafrontend beeinflusst werden. Beispielsweise kann das Ausgabeverzeichnis spezifiziert werden, wohin die Code-Module geschrieben werden sollen, oder der Package-Name, den die Java-Code-Module enthalten sollen. Eine Übersicht der Optionen des Metafrontend ist in Abbildung 3.14 dargestellt.

3.5 Benutzerschnittstelle

Damit das von dem Metafrontend übersetzte ADP-Programm von einem Anwender direkt ausgeführt und getestet werden kann, erzeugt das Metafrontend ein weiteres Code-Modul, welches eine Benutzerschnittstelle bereitstellt. Die Verwendung dieser Benutzerschnittstelle ist optional. Wenn z. B. der ADP-Algorithmus in ein Java-

Programm integriert werden soll, kann der Benutzerschnitstellen-Code ignoriert und entfernt werden.

Die Benutzerschnittstelle implementiert ein Command-Line-Interface (CLI) und ein Text-User-Interface (TUI). Der Benutzer interagiert mit dem CLI über Optionen, die er beim Programmstart auf der Kommandozeile in einem Terminal dem ADP-Programm übergibt. Das TUI besteht aus einer Eingabezeile, in der der Benutzer nach dem Programmstart iterativ Befehle eingeben kann.

Das CLI ermöglicht dem Benutzer bei der Ausführung des ADP-Algorithmus einzelne Algebren an- und abzuwählen. Dies ist sinnvoll, wenn das Metafrontend für jede Algebra des ADP-Programms ein Code-Modul erzeugt hat, aber für einen konkreten Programmaufruf nur das Ergebnis des ADP-Algorithmus unter einer bestimmten Algebra interessant ist. Die ADP-Programm-Eingabesequenz kann entweder in einer Datei stehen, welche über eine Option angegeben wird, oder direkt als Parameter auf der Kommandozeile übergeben werden. Bei der direkten Übergabe der Eingabe-Sequenz muss unter Umständen die Sequenz durch single-quote-Zeichen geschützt werden, wenn Zeichen der Sequenz eine Sonderbedeutung in der Eingabe-Shell des Terminals haben. Des Weiteren ermöglicht das CLI die Ausgabe von suboptimalen Kandidaten beim Backtracing zu beschränken. Dazu kann ein Wert x angegeben werden, so dass alle suboptimalen Kandidaten, deren Wert größer gleich dem optimalen Wert minus x ist, bei dem Backtracing ausgegeben werden.

Das TUI wird aufgerufen, wenn dem ADP-Programm über das CLI keine Eingabesequenz zur Verfügung gestellt wird. Über das TUI können alle Optionen des CLI zur Laufzeit wiederholt gesetzt werden. Die Eingabesequenz kann direkt in die Eingabezeile eingegeben oder als Option über eine Datei eingelesen werden. Diese Schnittstelle eignet sich für experimentelles Setzen von Parametern und Sequenzen, wenn nicht jedesmal das ADP-Programm, und damit die JVM, neu gestartet werden soll. Abbildung 3.15 zeigt eine Beispiel-Session mit dem TUI-Interface des ElMamun ADP-Programms.

Für ADP-Programme, welche RNA-spezifisch sind, erzeugt das Metafrontend Benutzerschnittstellen-Code, der zusätzliche Optionen zur Verfügung stellt. Das sind zum einen Optionen, die den Window-Mode betreffen, und zum anderen wird in RNA-spezifischen ADP-Programmen die Ausgabe von suboptimalen Kandidaten durch einen Prozentsatz in Bezug auf den optimalen Wert gesteuert. Im Window-Mode gleitet schrittweise ein Fenster über die Eingabesequenz. In jedem Schritt wird auf die Teilsequenz, die im Fenster liegt, der ADP-Algorithmus angewendet inklusive dem Backtracing. Über CLI und TUI lassen sich die Verwendung des Fenster-Modus, die Fenstergröße und die Schrittweite, um die das Fenster jedesmal verschoben werden soll, konfigurieren.

Der Window-Modus in dem Zielcode ist so implementiert, dass von einem Fenster nur die Spalten der verwendeteten Matrizen berechnet werden, die nicht Teil des vorherigen Fensters waren. Bei einer Fenstergröße von 20 und einer Schrittweite von 1 wird also in jedem Schritt eine Spalte neuberechnet.

Der generierte Schnittstellen-Code für das CLI und das TUI benötigt einen

```
$ java ElMamun.ElMamun
> 1+2*3*4+5
count
Optimal Score: 14
======================== Suboptimal Output ========================
===================================================================
buyer
Optimal Score: 30
======================== Suboptimal Output ========================
((1+((2*3)*4))+5) (Score: 30)
((1+(2*(3*4)))+5) (Score: 30)
(((1+(2*3))*4)+5) (Score: 33)
(1+(((2*3)*4)+5)) (Score: 30)
(1+((2*(3*4))+5)) (Score: 30)
(1+(2*((3*4)+5))) (Score: 35)
===================================================================
seller
Optimal Score: 81
======================== Suboptimal Output ========================
(((1+2)*3)*(4+5)) (Score: 81)
((1+2)*(3*(4+5))) (Score: 81)
===================================================================
> -h
-t, --diff     FOO                traceback diff
-f, --file     FOO                read from filename instead of argv
-a, --algebra  count|buyer|seller  toggle selected algebra off/on
-h, --help                        shows this nice help screen
> -a seller
[..]
```

Abbildung 3.15: Beispiel-Session mit dem TUI-Interface des ElMamun-ADP-Programms. Einge Leerzeichen und Gleichzeichen sind aus Gründen der Übersichtlichkeit entfernt.

Optionen-Parser. Dieser Parser soll eine einfach zu verwendende API besitzen, kurze und lange Optionen[12] und die automatische Generierung von einem Hilfetext unterstützen. Außerdem soll sein Code-Umfang möglichst gering sein, damit bei einer Distribution von einem ADP-Programm nicht Speicherplatz und Übertragungsressourcen verschwendet werden. Des Weiteren soll die Lizenz des Optionen-Parsers frei genug sein, so dass erzeugte ADP-Programme frei verteilt werden können und der Autor des ADP-Programms die Lizenz für sein Programm frei festlegen kann.

Als frei verfügbare Optionsparser sind Commons CLI 1.1 [7] und eine Implementation der libc-getopt-API [29] für Java evaluiert worden.

Commons CLI ist mit 36 kb komprimierten Java-Objekt-Code relativ gross. Es steht unter der Apache-Open-Source-Lizenz, welche die oben genannten Freiheiten erlaubt. Die API ist redundant, d. h. es existieren zwei verschiedene Schnittstellen, um Optionsspezifikationen dem Parser bekanntzugeben. Die automatische Generierung eines Hilfetexts wird unterstützt.

Java-getopt ist ein Port von GNU getopt. Er unterliegt der LGPL[13] und erfüllt somit die oben genannte Spezifikation. Die Library ist 7.6 kb gross, was für eine Distribution gut geeignet ist. Die API ist durch die Verwendung von kompakten Konfigurationsstrings, welche zur Laufzeit geparst werden, relativ kryptisch. Außerdem wird nicht die automatische Generierung von einem Hilfetext unterstützt.

Da die beiden untersuchten Optionen-Parser nicht der Spezifikation entsprechen, ist im Rahmen der Diplomarbeit ein allgemeiner Optionsparser entwickelt worden.

Der neue Optionen-Parser `optparse` verwendet als zentrales Konzept Java-Annotationen [2]. Java-Annotationen ist ein Sprachmerkmal, welches in der Java-Version 1.5 eingeführt wurde. Mit ihrer Hilfe können Methoden und Felder von Klassen annotiert werden. Je nach Definition können diese Annotationen statisch von anderen Programmen oder dynamisch von dem annotierten Programm zur Laufzeit ausgelesen werden. Eine verwendbare Annotation wird definiert durch eine spezielle deklarierte Java-Klasse.

Das optparse-Paket stellt die Annotation `Option` zur Verfügung, welche von dem Optionen-Parser OptionParser dynamisch zur Laufzeit eingelesen werden kann. Abbildung 3.16 enthält die Deklaration der Option-Annotation. Wenn auf Felder eines Objektes über das CLI bzw. TUI zugegriffen werden soll, dann werden sie in der Java-Klassendefinition mit der Annotation Option annotiert. Nach der Initialisierung von OptionParser werden alle annotierten Objekte dem Optionen-Parser bekannt gegeben, so dass er daraus einen internen Katalog der Optionenbeschreibungen aufbauen kann. Unter diesem Katalog werden die Optionen geparst, und in Abhängigkeit von Feld und Annotation werden die annotierten Objekte von dem OptionParser initialisiert. Außerdem wird ein Hilfetext aus den Annotationen generiert, der als Standard bei den üblichen Hilfeoptionen ausgegeben wird.

Im Gegensatz zu konventionellen Optionen-Parsern, wie z.B. Commons CLI, ma-

[12] Lange Optionen werden auch als GNU-Style-Optionen bezeichnet, z. B. `-help`. Ein Beispiel für eine kurze Option ist `-h`.

[13] Library General Public License

```
@Retention(RetentionPolicy.RUNTIME)
 public @interface Option {
    char opt() default ' ';
    String gnu() default  "";
    String desc() default  "";
    String arg() default  "FOO";
    boolean multiple() default false;
}
```

Abbildung 3.16: Deklaration der Option Annotation.

chen die optparse-API und die Verwendung von Annotationen das Schreiben von einigem Boilerplate-Code[14] überflüssig. So wird ein Feld einer Klasse durch seine Annotation mit seiner Deklaration als Option spezifiziert. Es ist keine zusätzliche Spezifikation an anderer Stelle für den Optionen-Parser notwendig, welche bei Aktualisierungen der Klasse mitaktualisiert werden müsste. Vor allem ist kein mechanistischer Initialisierungscode notwendig, der nach dem Durchlauf des Parsers alle möglicherweise eingelesenen Werte den betroffenen Objekten zuweisen muss. Dieser Initialisierungscode muss natürlich bei der Verwendung von konventionellen Optionen-Parsern auch separat von der Klassendeklaration gepflegt werden. Wie auch bei Commons CLI und im Gegensatz zu GNU Java getopt entfällt bei optparse die redundante fehleranfällige Pflege einer Bildschirmhilfe-Seite.

Ein vereinfachtes Beispiel der Verwendung der optparse-API zeigt Abbildung 3.17.

Die optparse-Bibliothek ist 11 kb gross, was für die Anzahl der Features in Ordnung ist. Der Source-Code ist in dem `adp2java`-Entwicklungszweig des ADP-Compiler im Verzeichnis `java/optparse` verfügbar. Die Lizenz steht noch nicht endgültig fest, aber zumindest soll die freie Verwendung für nicht-kommerzielle Zwecke zugesichert werden.

[14] Code-Teile, welche sich in wenig abgeänderter Form wiederholen und durch generische Programmiertechniken vermieden werden können.

```
class MyOptions {
    @Option(opt = 'w' , desc = "enables window mode")
      public boolean window_mode;
    @Option(opt = 'p', gnu = "prefix") public String prefix;
}
..
    MyOptions my = new MyOptions();
    MoreOptions mr = new MoreOptions();
    try {
      OptionParser op = new OptionParser(my);
      op.init(mr);
      List<String> files = op.parse(args);
..
```

Abbildung 3.17: Einfaches Beispiel für die Verwendung der optparse-API. Die notwendige Behandlung von Exceptions ist nicht in dem Beispiel enthalten. Nach dem erfolgreichen Aufruf von OptionParser.parse() sind alle registrierten Objekte initialisiert.

4 Benchmarks

Um das neue Java-Backend mit dem existierenden C-Backend zu vergleichen, sind im Rahmen der Diplomarbeit an Hand von ausgewählten übersetzten ADP-Programmen Benchmarks durchgeführt worden. Es wird die Laufzeit und der Speicherbedarf in Abhängigkeit von zufällig generierten Eingaben unterschiedlicher Länge gemessen. Neben der Länge der Eingabesequenz wird auch der Parameter variiert, der die Anzahl der auszugebenden suboptimalen Kandidaten bestimmt. Dadurch wird untersucht, welchen Einfluss das suboptimale Backtracing auf die Laufzeit des ADP-Programms hat. In [32] sind C-Benchmarks von RNAfold [20] und RNAshapes [18] enthalten, welche auch in diesem Kapitel untersucht werden.

Bei der Planung der Benchmarks ist die einfache Reproduzierbarkeit der Ergebnisse ein zentrales Ziel. Deswegen sind die Benchmarks automatisiert durchgeführt worden. Durch die Automatisierung wird auch die Wahrscheinlichkeit von Bedienungsfehlern verringert, da manuelle fehleranfällige und mühsame Tätigkeiten nicht durchgeführt werden müssen. Die Programme, mit denen die Benchmarks gesteuert worden sind, sind Teil der ADPC-Distribution und im Verzeichnis testsuite des `adp2java`-Entwicklungszweiges enthalten.

Der Vergleich des Speicherbedarfs des Java- und des C-Zielcodes ist schwierig, da grundlegend verschiedene Speicherverwaltungstechniken verwendet werden. Die Java-Programme verwenden zwangsläufig die in die JVM integrierte Garbage-Collection. C-Programme können Garbage-Collection-Algorithmen verwenden oder sie allokieren und geben Speicher explizit frei. Der C-Zielcode des C-Backends verwendet keine Garbage-Collection, stattdessen verwaltet er den Speicher explizit. Ein zentraler Parameter der JVM-Garbage-Collection ist die Größe de maximalen Speichers in MB, den die Garbage-Collection verwenden darf. Nun hängt der Speicherbedarf eines Java-Programms von dem Garbage-Collection-Algorithmus der JVM ab. Ein Algorithmus könnte z. B. die Strategie verfolgen, zeitaufwendige Speicherfreigaben zu vermeiden bis zu einem bestimmten Verhältnis von belegtem Speicher und maximal verwendbarem Speicher. Also benötigt das Java-Programm in diesem Fall nicht unbedingt den Speicherplatz entsprechend dem gemessenen Speicherbedarf. Wenn nämlich der maximale Speicherbedarf der JVM per Parameter eingeschränkt wird, würde der Garbage-Collection-Algorithmus früher von dem Programm nicht mehr referenzierten Speicher freigeben und die JVM, und somit das Java-Programm, würde weniger Speicher belegen.

Der gemessene Speicherbedarf des Java-Programms enthält auch den Speicherbedarf der JVM, die im Speicher Verwaltungsinformationen ablegt. Alternativ könnte auch nur die vom Java-Programm belegten Speichbereiche gemessen werden. Diese Bereiche zuverlässig zu messen, ist allerdings aufwändig und diese Messung spiegelt

nicht den tatsächlichen Speicherbedarf der Anwendung wieder.

Die Messung der Programmlaufzeit von Java-Programmen ist unterschiedlich durchführbar. Es kann entweder die Zeit von JVM-Start bis JVM-Ende oder die Zeit von der Ausführung der zentralen Java-Programmroutine bis zu ihrem Ende innerhalb der JVM gemessen werden. Die Benchmarks verwenden die erste Möglichkeit, da auch in einem zentralen Anwendungsfall, in dem der Anwender die nach Java übersetzten ADP-Programme für jede neue Eingabe neu aufruft, jedesmal die JVM neu gestartet wird. Der Anwendungsfall, in dem beispielsweise Java-Zielcode von einem Java-Applikationsserver aufgerufen wird, so dass die JVM nicht neu gestartet wird, ist auch realistisch, aber für den typischen ADPC-Benutzer weniger relevant. Im übrigen ist der Zeitaufwand, den die JVM zum Starten bzw. zum Beenden benötigt, konstant, so dass dieser Wert von den Messwerten nachträglich abgezogen werden kann.

4.1 Methoden

Soweit nicht anders angegeben, sind die Messungen auf zwei identischen Sun-Fire-V20z-Rechnern durchgeführt worden, welche jeweils mit zwei Opteron-CPUs mit jeweils 2.4 GHz und 12 GB RAM ausgestattet sind. Als Betriebssystem kommt Solaris 10 zum Einsatz. Da die erzeugten ADP-Programme single-threaded sind, verwenden sie während der Laufzeit nur einen Prozessor.

Als JDK wird das Sun JDK 1.6.2 verwendet. Optionen der JVM, wie die maximale Größe des Heaps, werden nicht beim Aufruf gesetzt, also gelten die Standardwerte[1]. Die generierten C-Programme werden mit dem GCC [1] C-Compiler 4.1 kompiliert. Die verwendete Optimierungsstufe ist `-O2`.

Die automatisierte Durchführung der Messungen ist mit Hilfe der ADPC-Testsuite (siehe Kapitel 5) implementiert. Zu diesem Zwecke ist sie um das Modul `perftest` erweitert worden, welches generisch die `Codegen`-Testfälle ableitet und ihre Programmerzeugungs- und Ausführungsfunktionalität wiederverwendet. Zusätzliche Attribute der `perftest`-Testfälle ermöglichen die Angabe von zulässigen Kombinationen von Eingabelänge, Suboptimalitätsparameter und weiteren Parametern bzw. der Anzahl der Wiederholungen. Bei der Erzeugung der Tests erzeugen die `perftest`-Testfälle für jede spezifizierte Kombination eine Instanz, welche einer Testsuite hinzugefügt wird. Bei dem Aufruf von externen ADP-Programmen wird ein konfigurierbarer Zeitrahmen gesetzt, nach dessen Ende das Programm abgebrochen wird. Für die Implementierung dieser beiden Merkmale ist das Testframework erweitert worden. Wenn eine Messung aufgrund zu langer Ausführungszeit fehlschlägt, dann werden alle folgenden Messungen, welche den gleichen Suboptimalitätsparameter und längere Eingaben verwenden, übersprungen, da sie keine geringere Ausführungszeit haben können. Dieses Merkmal ist wichtig für die Messung bei der Ausgabe von suboptimalen Kandidaten, da die Anzahl der suboptimalen Kandidaten und somit die Laufzeit exponentiell ansteigt.

[1] laut Dokumentation entspricht dies `-server -d32 -Xmx1024m`

Für die konkrete Ermittlung der Laufzeit und des Speicherbedarfs wird das Programm `memtime` 1.3[11] verwendet. Der zu beobachtende Prozess wird von memtime gestartet, und die Laufzeit wird durch die POSIX-Funktion `wait4()` von dem Betriebssystem am Prozessende bereitgestellt. Der maximale Speicherbedarf wird ermittelt, indem 10-mal in der Sekunde die Informationen von dem Betriebssystem abgefragt werden. Für sehr kurz laufende Prozesse ist dieses Sampling zu ungenau.

`memtime` ist für die Verwendung in der Testsuite angepasst worden. Die erweiterte Version ist komplett portabel auf 64Bit-Systeme, gibt den Exit-Status des beobachteten Prozesses weiter zurück und beendet beim vorzeitigen Beenden auch den beobachteten Prozess.

Bei der Auswertung der gesammelten Daten ist die 'resident size' (rss) und die 'virtual size' (vsize) der ADP-Prozesse berücksichtigt worden. Die 'resident size' bezeichnet den Speicherbedarf, den der Prozess tatsächlich verwendet. 'vsize' enthält außerdem noch Speicherbereiche, die beispielsweise in den Adressbereich des Prozesses abgebildet sind, auf die allerdings nicht zugegriffen wurde, und Bereiche, welche ausgelagert sind. Die rss wird im folgenden auch als echter Speicher und vsize als virtueller Speicher bezeichnet. Die Laufzeit utime bezeichnet die 'user time' und ist die vom Betriebssystem ermittelte Zeit, in der der Prozess im User-Space ausgeführt wurde. Beispielsweise werden Zeiten in Systemaufrufen, Zeiträume, in denen der Prozess gestoppt war oder pausiert, nicht mitgezählt.

Die zufälligen Eingabesequenzen werden durch das für diesen Zweck erstellte Haskell-Programm `randseq`[2] erzeugt, welches in dem Verzeichnis `addons` des `adp2java`-Entwicklungszweiges enthalten ist. Die vier RNA-Basen A, C, G und U sind gleichverteilt in der generierten Sequenz enthalten.

Die Ausgabe der Scores und der Kandidatenstrukturen wird nach `/dev/null` umgeleitet, damit nicht die Geschwindigkeit des Terminals oder des Dateisystems gemessen wird.

Die Messungen des Java-Zielcodes für RNAfold bzw. für RNAshapes sind bis zur einer Eingabesequenz von 600 bzw. 1200 Basen 10 mal wiederholt worden. Die restlichen Messungen sind 5 mal wiederholt worden. In den Diagrammen sind alle Wiederholungen eingetragen, so dass sich Messpunkte überlagern.

Die von den `perftest`-Testfällen generierten Daten werden mit dem dafür entwickelten `perf2plot`-Programm, welches in dem testsuite-Verzeichnis des `adp2java`-Entwicklungszweiges enthalten ist, ausgewertet. Die Diagramme werden aus den ausgewerteten Daten mit Gnuplot [6] generiert.

4.2 Ergebnisse

4.2.1 RNAfold

RNAfold [20] ist ein Algorithmus, der die Sekundärstruktur der Eingabesequenz berechnet, welche unter der der RNAfold-ADP-Grammatik und der minimum-free-

[2] für andere Zwecke können neben dem RNA-Alphabet beliebige Alphabete angegeben werden

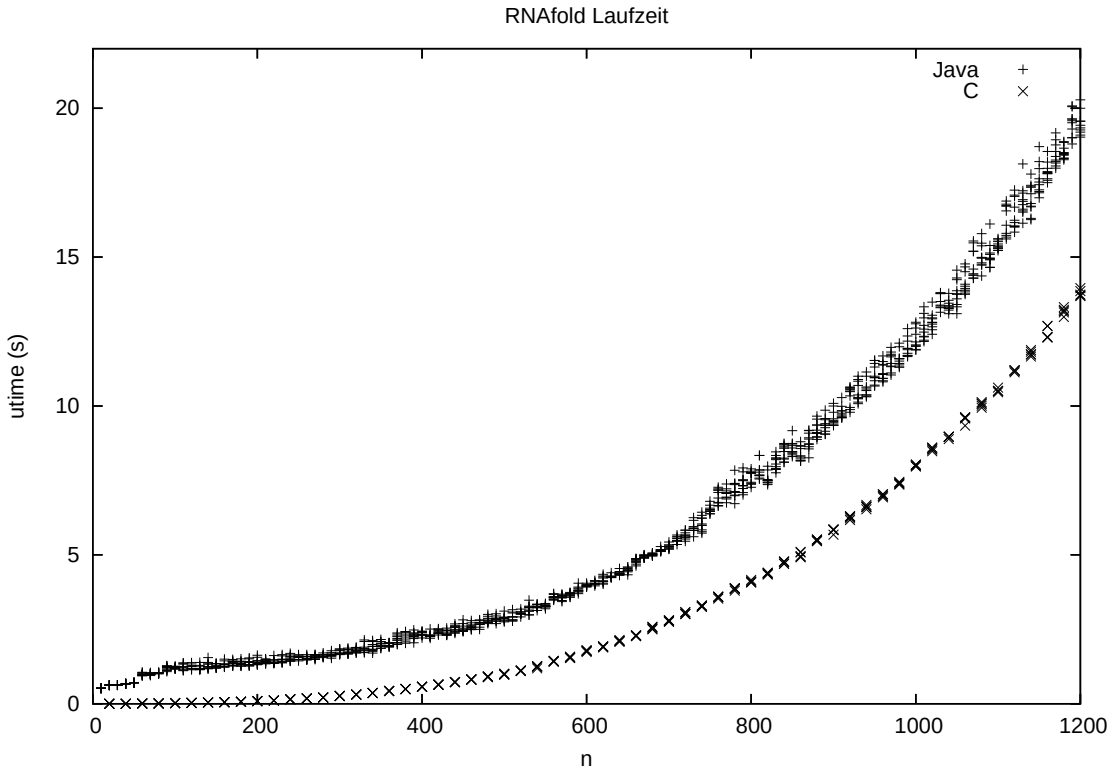

Abbildung 4.1: Laufzeit utime des nach Java bzw. C übersetzten RNAfold-Programms in Abhängigkeit von der Sequenzlänge n. Es werden keine suboptimalen Kandidaten ausgegeben.

energy-Algebra die minimale freie Energie besitzt. Die Laufzeit des Algorithmus liegt in $O(n^3)$, und der Speicherbedarf für die benötigten Tabellen liegt in $O(n^2)$.

Abbildung 4.1 stellt die Laufzeit des generierten C- und des Java-Zielcodes des RNAfold-ADP-Programms gegenüber. In dem Bereich der Sequenzlänge von 400 Basen ist der Java-Code um den Faktor 4 langsamer als der C-Code. Mit zunehmender Sequenzlänge verringert sich der Faktor bis zu einem Verhältnis von 1.5 bei einer Sequenzlänge von 1000 bis 1200 Basen. Der Laufzeitunterschied ist zu erwarten, und lässt sich durch Sicherheitsüberprüfungen der JVM während der Laufzeit, der Verwaltung der Garbage-Collection und der Interpretation bzw. JIT-Kompilierung von Java-Byte-Code erklären. Der Aufwand für den Start und das Beenden der JVM beträgt näherungsweise 0.5 Sekunden entsprechend dem Wert bei einer Sequenzlänge von 10 Basen.

Die Verwendung des suboptimalen Backtracing lässt die Laufzeit exponentiell ansteigen (Abbildung 4.2 bzw. 4.3), da die Anzahl der von dem Backtracing-Algorithmus auszugebenden Kandidatenstrukturen mit zunehmender Sequenzlänger exponentiell ansteigt im Vergleich zu dem optimalen Backtracing. Dass die Laufzeit des Java-Zielcodes im suboptimalen Fall noch schneller als die des C-

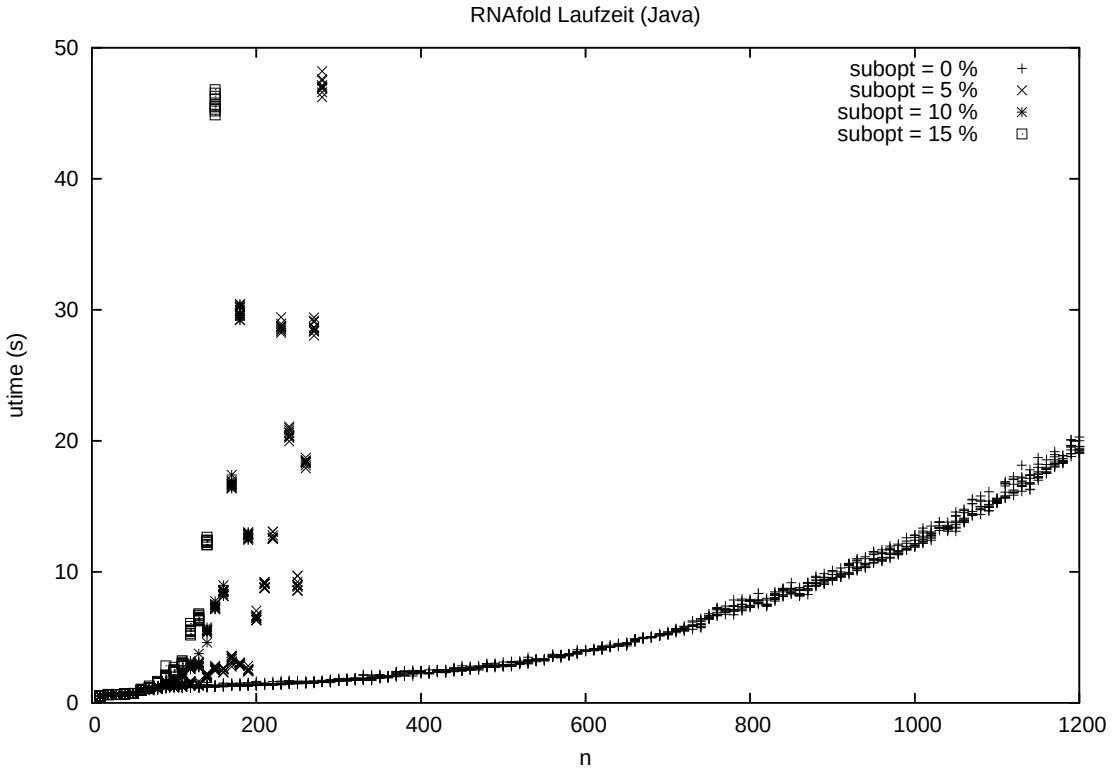

Abbildung 4.2: Laufzeit utime des nach Java übersetzten RNAfold-ADP-Programms bei verschieden gewählten Schwellenwerten subopt bei der Ausgabe von Kandidaten mit suboptimalen Scores in Abhängigkeit von der Sequenzlänge n.

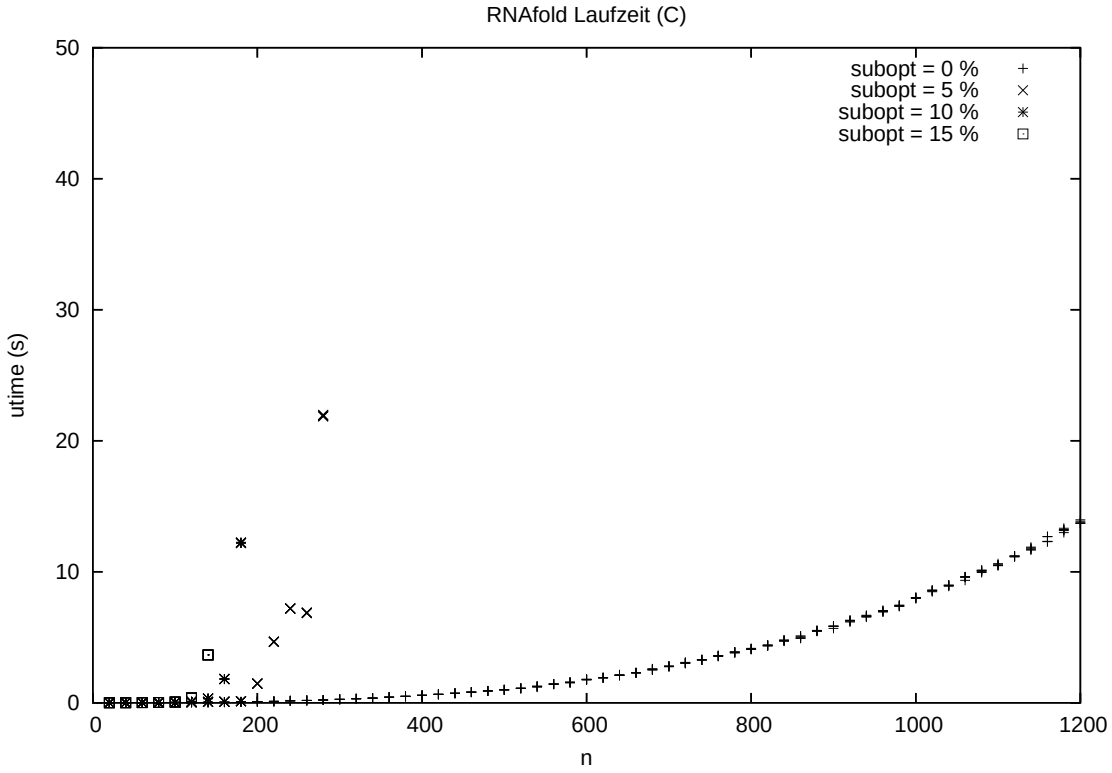

Abbildung 4.3: Laufzeit utime des nach C übersetzten RNAfold-ADP-Programms bei verschieden gewählten Schwellenwerten subopt bei der Ausgabe von Kandidaten mit suboptimalen Scores in Abhängigkeit von der Sequenzlänge n.

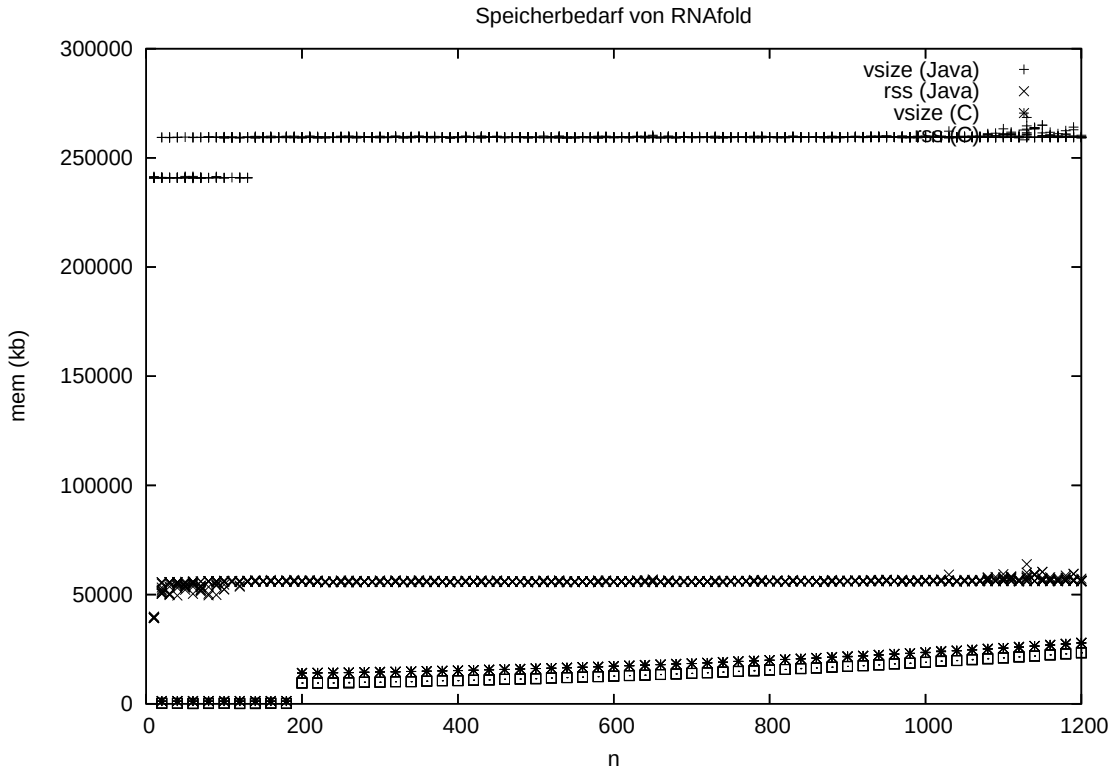

Abbildung 4.4: Bedarf an echtem Speicher rss und virtuellem Speicher vsize des nach C bzw. Java übersetzten RNAfold-ADP-Programms in Abhängigkeit von der Sequenzlänge n.

Zielcodes wächst, ist durch den höheren Aufwand der Java-Speicherverwaltung zu erklären, da der Backtracing-Algorithmus pro Kandidat einige temporäre Objekte benötigt.

Abbildung 4.4 zeigt deutlich den Unterschied zwischen der Verwendung einer Garbage-Collection im Java-Zielcode und einer expliziten Speicherverwaltung im C-Zielcode. Der Java-Zielcode belegt bis zu einer Sequenzlänge von 1200 Basen einheitlich etwas mehr als 50 MB echten Speicher (rss). Dies ist offensichtlich bei der gegebenen Eingabe der JVM eine berechnete Schwelle, bis zu der die JVM-Garbage-Collection auf die Freigabe nicht mehr referenzierter Speicherbereiche aus Performancegründen verzichtet. Bei der direkten Speicherverwaltung im C-Zielcode ist dagegen eine kontinuierliche Steigerung des Speicherbedarfs zu beobachten. Die Verhältnis von vsize zu rss des Java-Codes ist im Vergleich zum C-Code groß. Die vsize ist um den Faktor 5 größer. Es wurden keine Speicherbereiche ausgelagert; das Testsystem hat auch ausreichend freien Arbeitsspeicher zur Verfügung. Möglicherweise werden eine große Anzahl von shared-Libraries und Java-Byte-Code der Java-Standardbibliothek in den Speicher abgebildet. Auch kann die Speicherverwaltung Speicherseiten via `mmap()` in den Adressbereich des Prozesses abbilden,

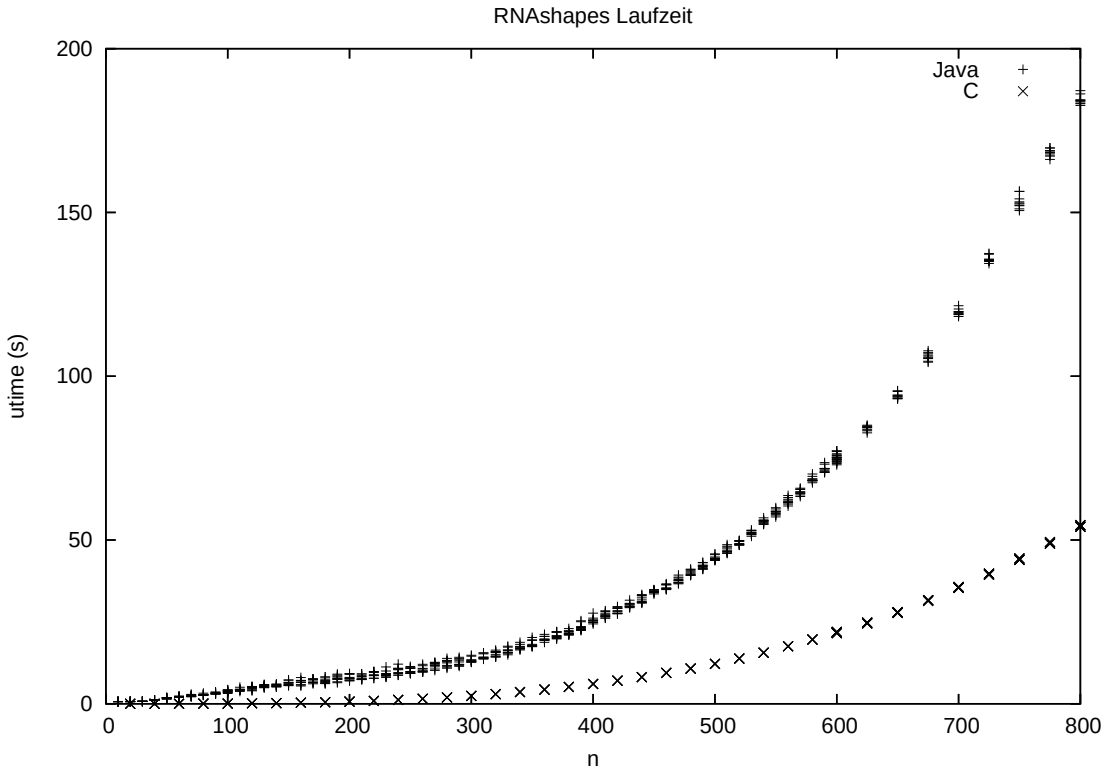

Abbildung 4.5: Laufzeit utime des nach Java bzw. C übersetzten RNAshapes-Programms in Abhängigkeit von der Sequenzlänge n. Es werden keine suboptimalen Kandidaten ausgegeben.

auf die allerdings nicht zugegriffen wurde.

4.2.2 RNAshapes

RNAshapes [18] berechnet die Sekundärstruktur der Eingabesequenz, welche die minimale freie Energie unter dem verwendeten Abstract-Shapes-Modell besitzt. Die Laufzeit des Algorithmus liegt in $O(n^3\alpha^n)$ und der Speicherbedarf für die Tabellen in $O(n^2\alpha^n)$. α ist ein Faktor, der von der Shape-Abstraktionsstufe und weiteren Parametern abhängt. Für typische Anwendungen ist er nur wenig größer als 1[32].

In Abbildung 4.5 wird die Laufzeit des nach Java übersetzten RNAshapes-Algorithmus der Laufzeit des kompilierten C-Zielcode gegenübergestellt. Bei einer Sequenzlänge von 400 Basen ist der generierte Java-Code um den Faktor 5 langsamer. Bis zu einer Sequenzlänge von 800 Basen verringert sich das gemessene Verhältnis der Laufzeiten bis zu einen Faktor von 3.3. Die größeren Laufzeitunterschiede im Vergleich zu den gemessenen RNAfold-Laufzeiten liegen darin begründet, dass die RNAshapes-Algebra Tupel als Antwortdatentyp verwendet. Im Java-Zielcode werden sie als Java-Objekte abgebildet (siehe Abschnitt 3.2.5). Dies erhöht die Anzahl

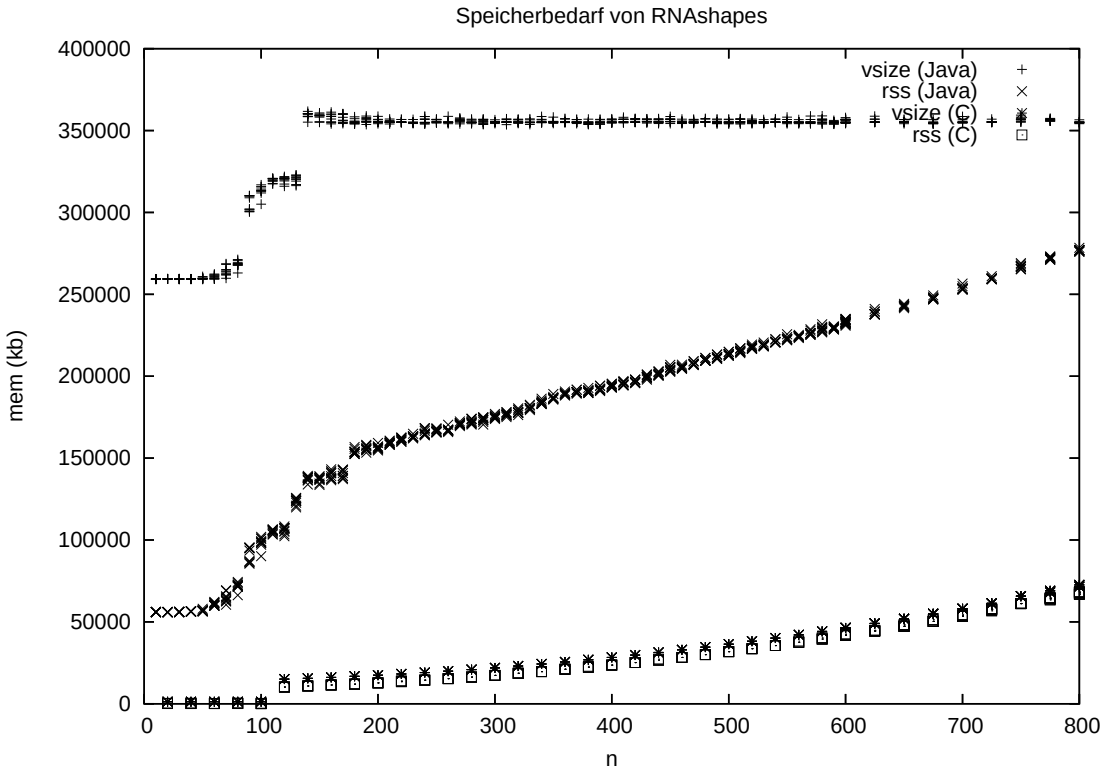

Abbildung 4.6: Bedarf an echtem Speicher rss und virutellem Speicher vsize, des nach C bzw. Java übersetzten RNAshapes-ADP-Programms in Abhängigkeit von der Sequenzlänge n.

der temporären Objekte, die auf dem Heap von der JVM-Speicherverwaltung angelegt und wieder freigegeben werden müssen. Dieser zusätzliche Aufwand entfällt bei der Abbildung von Tupeln auf Structs in dem C-Backend.

Der Speicherbedarf des RNAshapes Programms in Abhängigkeit von der Eingabelänge wird in Abbildung 4.6 dargestellt. Bei einer Sequenzlänge von 300 Basen benötigt der Java-Zielcode etwas 7 mal so viel realen Speicher wie der C-Zielcode. Dieser Faktor verringert sich mit zunehmender Sequenzlänge auf den Wert von 3.9 bei einer Sequenzlänge von 800 Basen. Im Gegensatz zu RNAfold steigt der reale Speicherbedarf des RNAshapes-Java-Zielcode mit zunehmender Sequenzlänge an, was durch die größere Anzahl von zu verwaltenden Objekte zu erklären ist, worauf der Garbage-Collection Algorithmus der JVM anders reagiert. Denn das RNAshapes-Programm verwendet 5 quadratische Tabellen und die ADP-Grammatik besteht aus 27 Nichtterminalen. RNAfold verwendet 3 quadratische Tabellen und besteht aus 12 Nichtterminalen. Außerdem wird pro Tabelleneintrag jeweils ein Tupel-Objekt bestehend aus zwei Integer-Werten angelegt. Hinzu kommen weitere temporäre Tupel-Objekte, die zur Berechnung jedes Tabelleneintrags berechnet werden müssen.

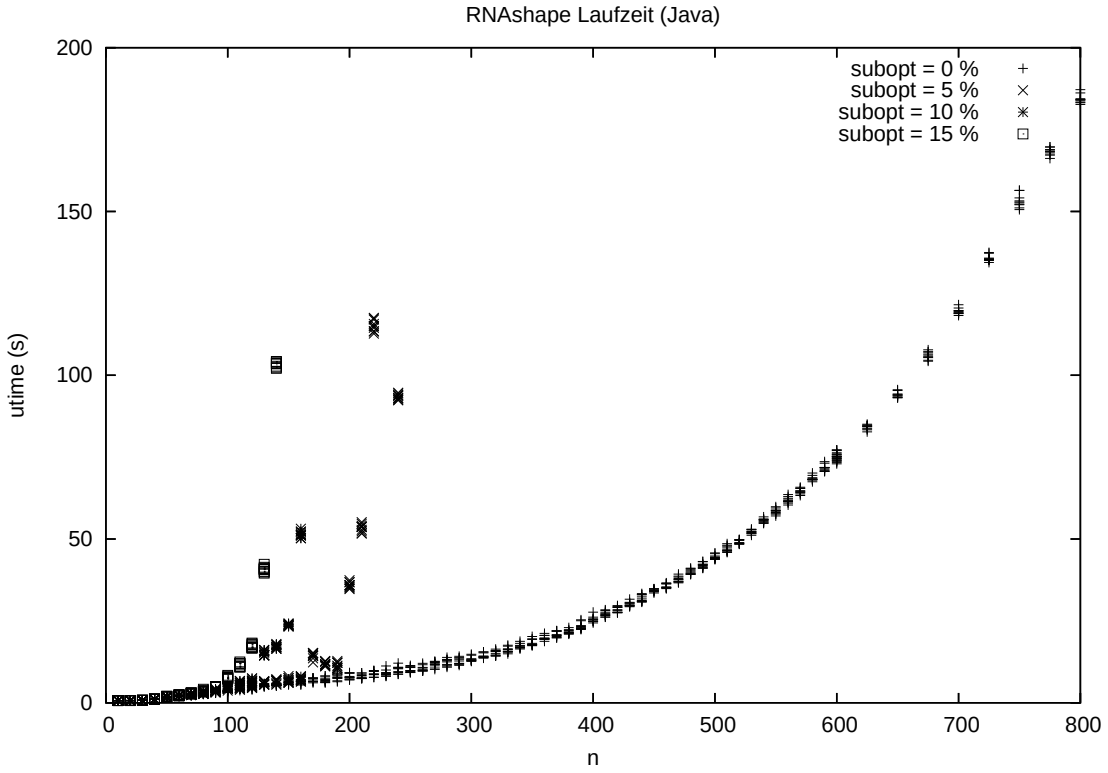

Abbildung 4.7: Laufzeit utime des nach Java übersetzten RNAshapes-ADP-Programms in Abhängigkeit von der Sequenzlänge n bei verschieden gewählten Schwellenwerten subopt bei der Ausgabe von Kandidaten mit suboptimalen Scores.

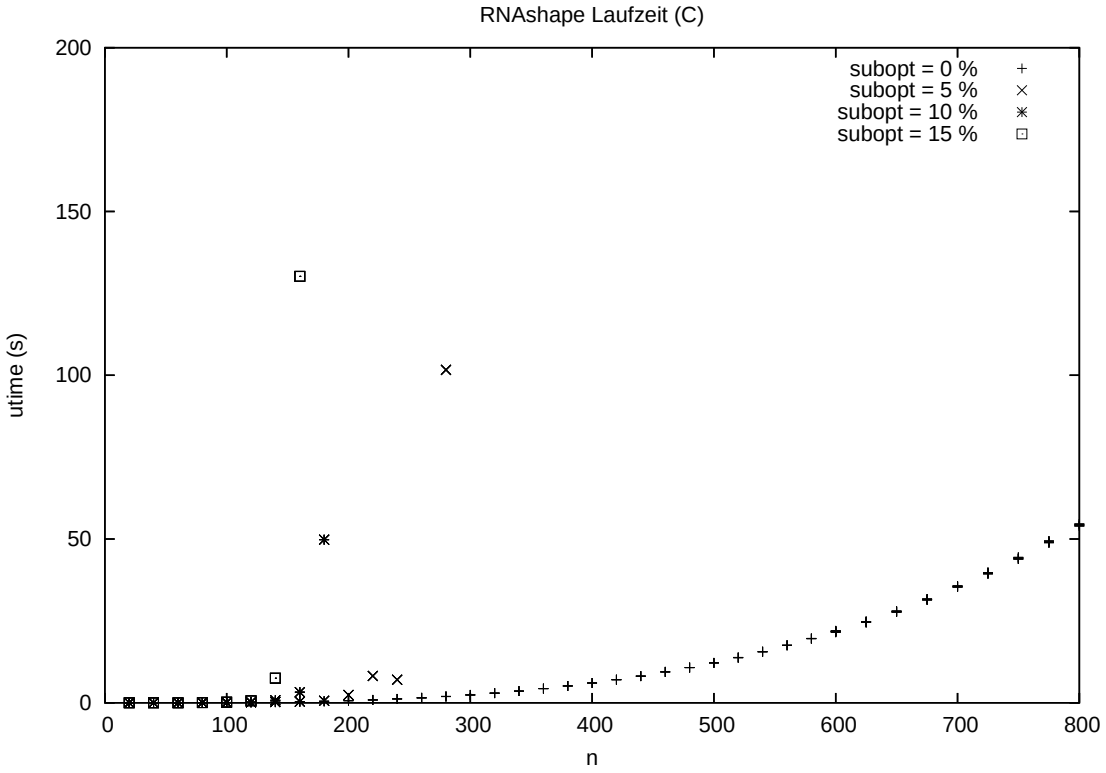

Abbildung 4.8: Laufzeit utime des nach C übersetzten RNAshapes-ADP-Programms in Abhängigkeit von der Sequenzlänge n bei verschieden gewählten Schwellenwerten subopt bei der Ausgabe von Kandidaten mit suboptimalen Scores.

In Abbildung 4.7 bzw. 4.8 ist die Laufzeit des RNAshapes-Java- bzw. C-Zielcodes bei verschiedenen Suboptimalitäts-Schwellenwerten aufgetragen. Die Ergebnisse sind qualitativ mit den von RNAfold vergleichbar (Abbildung 4.2 bzw. 4.3). Mit einem Suboptimalitätsschwellwert größer 0 steigt die Laufzeit exponentiell an. Aufgrund des erhöhten Verwaltungsaufwands der JVM bei steigender Anzahl von temporären Objekten steigt die Laufzeit des Java-Zielcode bei gleicher Suboptimalitätsschwelle im Vergleich zum C-Zielcode noch schneller an.

4.2.3 TDM

Für den Test des Window-Modes wird ein ADP-Programm verwendet, welches von Jens Reeder zu Test-Zwecken mit dem Locomotif [27, 26]-Programm erstellt wurde. Mit Locomotif kann der Benutzer über eine graphische Benutzerschnitstelle einen auf ein bestimmtes RNA-Motif spezialisierten thermodynamischen Matcher erzeugen. Dazu können vorgefertigte generische Elemente, wie Hairpins und Internal-Loops, auf einer Konstruktionsfläche interaktiv kombiniert werden.

Das generierte ADP-Programm verwendet eine minimum-free-energy-Algebra. Die Laufzeit liegt in $O(n^2)$, und der Speicherbedarf für die Tabellen liegt in $O(n^2)$.

In Tests ohne Window-Mode benötigt das nach Java übersetzte ADP-Programm weniger als 400 MB virtuellen bzw. 200 MB echten Speicher bis zu einer Sequenzlänge von 110 Basen (Abbildung 4.9). Der C-Zielcode benötigt zwischen 3 und 3.5 GB virtuellen bzw. reellen Speicher bei einer Eingabesequenz von 90 Basen (Abbildung 4.9). Der Grund für diesen unerwarteten Unterschied um den Faktor 10 ist noch nicht unklar. Es ist von einem Bug in der C-Codeerzeugung auszugehen.

Wenn nun bei nicht weitersteigendem Speicherbedarf längere Sequenzen auf das gesuchte Motif untersucht werden sollen, dann ist der Window-Mode eine Möglichkeit, ein Fenster über die Eingabesequenz gleiten und in dem Fenster jeweils nach dem Motif suchen zu lassen. Bei einer Fensterverschiebung müssen nur die Spalten der Tabellen neu berechnet werden, auf deren Index zuvor noch nicht zugegriffen worden ist.

Die Messungen am TDM-Programm im Window-Modes sind mit einem Fenster der Länge 70 und einer Schrittweite von 10 durchgeführt worden. Als Rechnersystem ist eine Sun Fire v20z, mit 1.8 GHz und 12 GB RAM unter Solaris 10 Beta 70[3] verwendet worden.

Bei Verwendung des Window-Modes können Sequenzen von bis zu 600 Basen in einem Zeitraum von 10 Minuten nach dem RNA-Motif durchsucht werden (Abbildung 4.10). Im Gegensatz dazu ist ohne Window-Mode bei Verwendung des Java-Zielcodes bzw. des C-Zielcodes nur die Verarbeitung von Sequenzen der Länge 120 bzw. 100 in einem Zeitraum von 10 Minuten möglich (Abbildung 4.9). In Abbildung 4.10 wird die Laufzeit des TDM-ADP-Programms bei aktiviertem Window-Mode dargestellt. Die Laufzeit des Java-Zielcodes ist um den Faktor 2 länger im Vergleich zum C-Zielcode.

[3] Build-Datum Dezember 2004

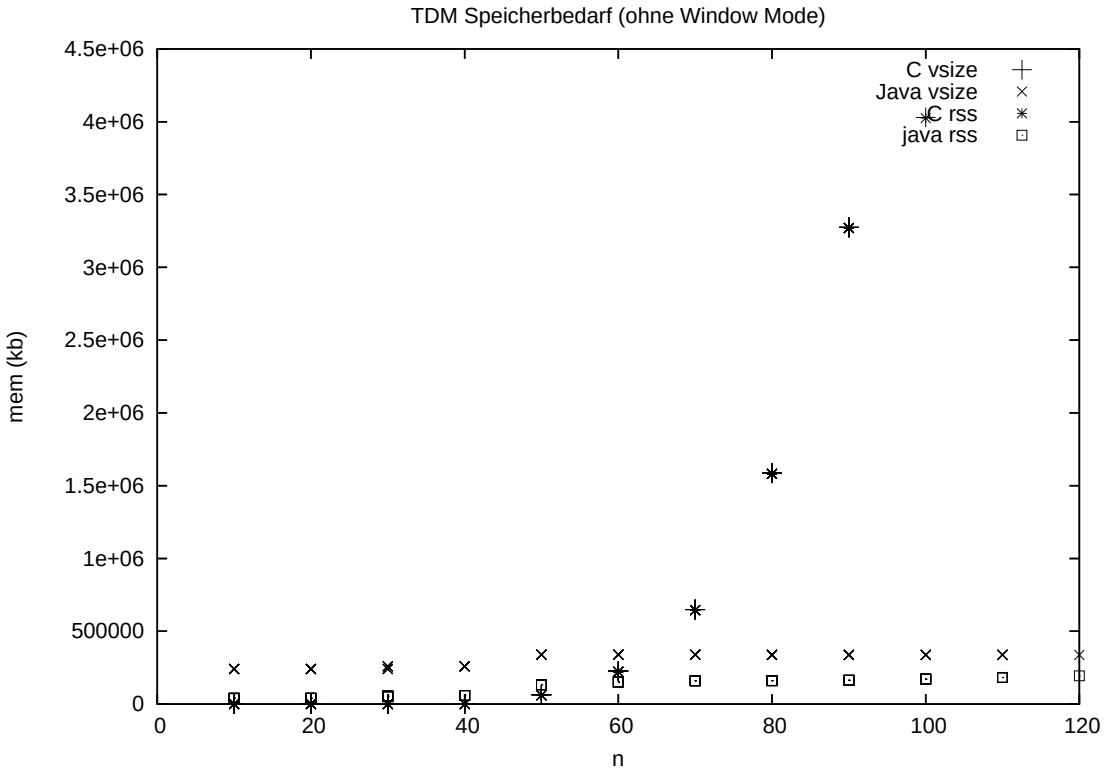

Abbildung 4.9: Bedarf an echtem Speicher rss und virutellem Speicher vsize eines mit Locomotif erzeugten ADP-Programms in Abhängigkeit von der Sequenzlänge n.

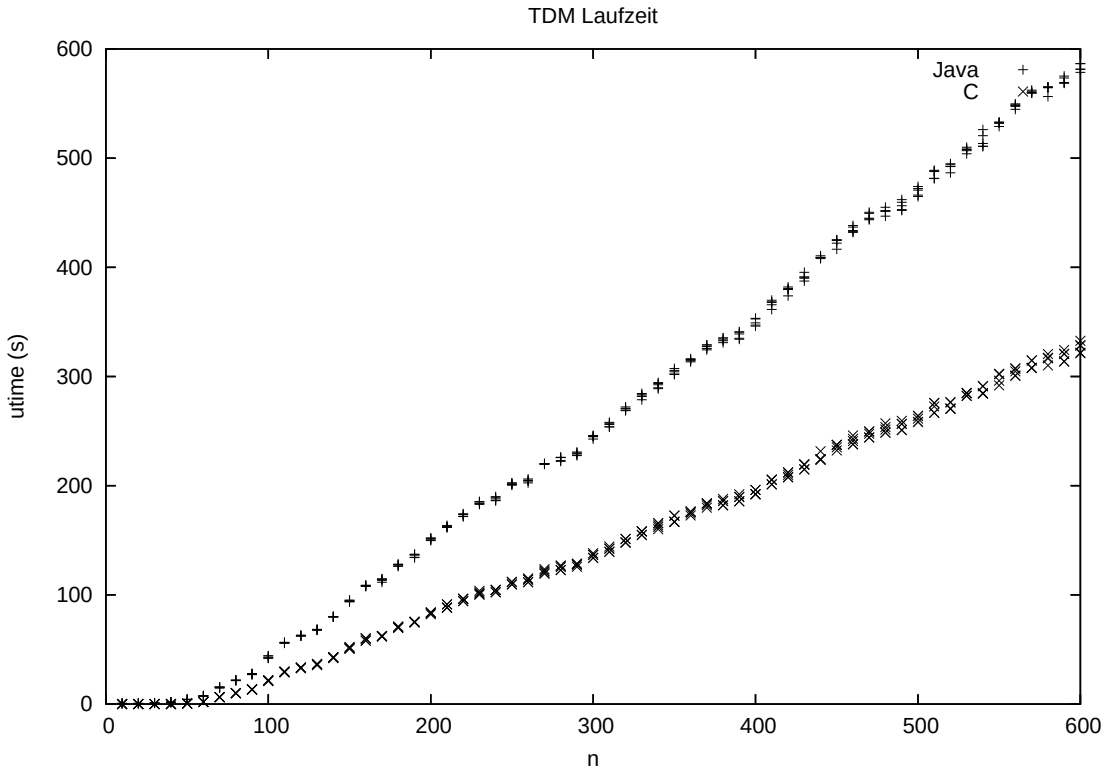

Abbildung 4.10: Laufzeit des TDM-Programms im Window-Mode in Abhängigkeit von der Sequenzlänge n. Es werden keine Kandidaten mit einem suboptimalen Score ausgegeben.

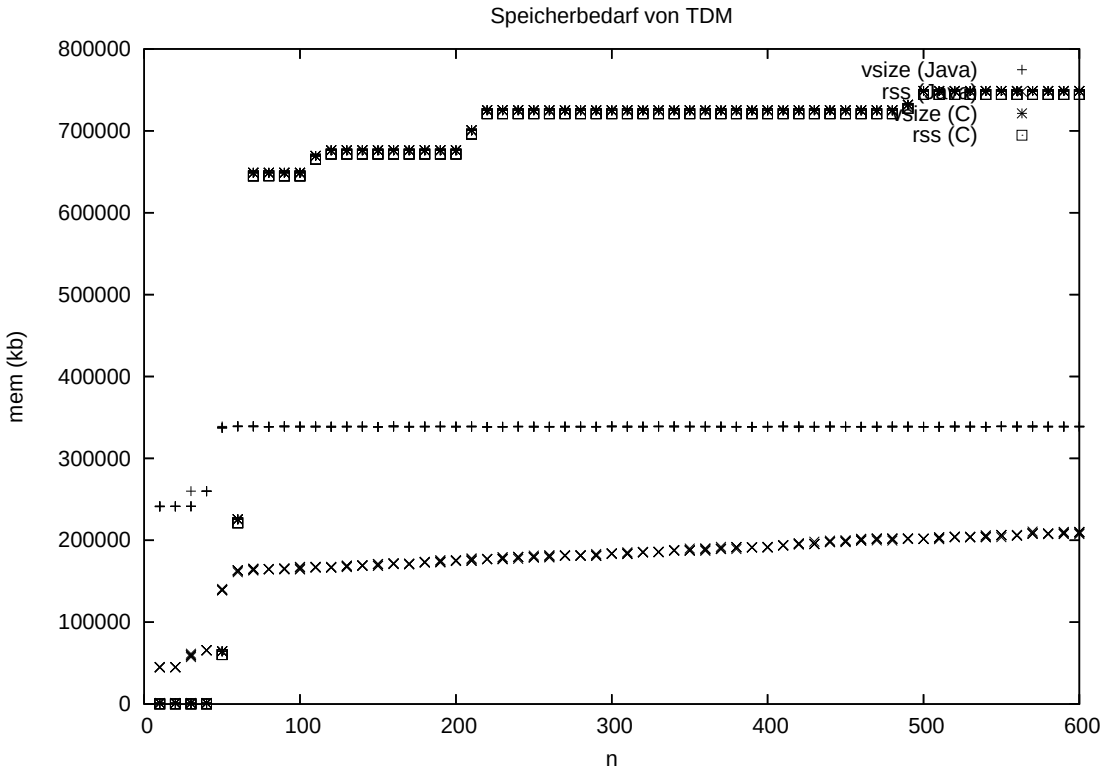

Abbildung 4.11: Bedarf an echtem Speicher rss und virutellem Speicher vsize des TDM-Programms im Window-Mode in Abhängigkeit von der Sequenzlänge n.

Abbildung 4.11 stellt den Speicherbedarf des nach Java und C übersetzten TDM-Programms dar. Der steigende Speicherbedarf des Java-Zielcodes ab einer Länge von 70 Basen ist durch die Garbage-Collection der JVM zu erklären, da mit jedem weiteren Schritt, um den das Fenster verschoben wird, zwar keine Tabellen erweitert werden, allerdings findet eine weitere Backtracing-Phase statt. In jeder Backtracing-Phase legt der Backtracing-Algorithmus einige temporäre Objekte an, welche von der Garbage-Collection nach dem Überschreiben der letzten Referenz nicht direkt freigegeben werden müssen, was zu einer Laufzeitersparnis führt. In einer Übersetzung, welche eine explizite Speicherverwaltung verwendet, steigt der Speicherbedarf im Window-Mode ab einer Sequenzlänge, welche größer als die Fenstergröße ist, nicht mehr. Der Speicherbedarf des C-Zielcodes in Abbildung 4.11 ist wie in Abbildung 4.9 sehr untypisch, da zum einen im Gegensatz zu RNAfold und RNAshapes der Speicherbedarf des C-Zielcodes weit über dem Speicherbedarf des Java-Zielcodes liegt und zum anderen der Speicherbedarf auch bei einer Eingabelänge, welche die Fenstergröße übersteigt, noch weiter zunimmt. Die Ursachen für diese Beobachtungen liegen wahrscheinlich in einem Bug der Codegenerierung, welcher durch dieses spezielle ADP-Programm ausgelöst wird.

4.3 Fazit

Die Performance des von dem Java-Backend generierten Java-Zielcode ist sehr gut. Eine gemessene Laufzeitverschlechterung des Java-Zielcode der exemplarisch ausgesuchten ADP-Programme nur um den Faktor 1.5 bis 4 im Vergleich zu dem C-Zielcode ist sehr gut, da der Java-Bytecode von der JVM interpretiert wird, der JIT-Compiler von der JVM während der Laufzeit aufgerufen wird, zusätzlicher Verwaltungsaufwand durch die Garbage-Collection der JVM entsteht und einige Sicherheitsüberprüfungen von der JVM während der Laufzeit durchgeführt werden. In [8] wird eine Laufzeitverschlechterung von C-Code, welcher automatisch nach Java-Bytecode übersetzt wird, um den Faktor 2 bis 10 bei verschiedenen Anwendungen angegeben. Der höhere Speicherverbrauch des Java-Zielcodes im Vergleich zum C-Zielcode ist durch die Verwendung der JVM-Garbage-Collection bedingt.

5 Testsuite

Im Rahmen dieser Diplomarbeit[1] ist ein modulares Testframework bzw. eine Testsuite entwickelt worden, welche verschiedene Bereiche des ADP-Compilers testet. Ein Schwerpunkt der Tests liegt in dem Bereich Codeerzeugung. Insbesondere wird die Codeerzeugung des bisherigen C-Backends mit der Codeerzeugung des neuen Java-Backends verglichen. Das Testframework ist im Verzeichnis `testsuite` des `adp2java`-Entwicklungszweiges des ADP-Compilers enthalten. In den folgenden Abschnitten dieses Kapitels wird darauf eingegangen, warum eine Testsuite notwendig ist und wie die konkrete Architektur und Implementation aussieht.

Der Begriff Testframework bezeichnet hier ein Softwaresystem, welches die Entwicklung von Tests für ein Softwareprojekt vereinfacht, indem eine wiederverwendbare projektunabhängige Infrastruktur bereitgestellt wird. Eine Testsuite ist eine Sammlung von Tests einschließlich ihrer Normausgaben. Bei Ausführung einer Testsuite werden die Ausgaben der Tests automatisch mit den zugeordneten Normausgaben verglichen. Eine Testsuite kann mit Hilfe eines Testframeworks entwickelt werden, muss es aber nicht. Ein Test der Codegenerierung wird definiert als Tupel aus Compiler-Eingabe, Programm-Eingabe des erzeugten Programms und Code, welcher nach bestimmten Kriterien die Norm-Programmausgabe und die aktuelle Programmausgabe vergleicht. Tests, die andere Teile des Compilers testen, bestehen nur aus der Eingabe des zu testenen Moduls, seiner Normausgabe und dem Vergleichscode.

5.1 Motivation

Die Entwicklung eines Compilers ist ein umfangreiches und komplexes Softwareprojekt. Wie jedes nicht triviale Softwareprojekt enthalten auch Compiler Fehler. Deswegen liegt es nahe, dass die Softwareentwickler während der Entwicklung von neuen Programmmerkmalen diese mit Beispieleingaben auf ihre Korrektheit testen. Bei einem Compiler besteht die Eingabe aus einem Programm der zu übersetzenden Sprache und einer Kombination von Optionen. Um die Korrektheit der Ausgabe des ADP-Compilers zu beurteilen, muss der erzeugte Code kompiliert werden und mit verschiedenen Test-Eingaben und Kombinationen von Optionen ausgeführt werden. Die Programmausgabe des kompilierten Programms muss nun von dem Entwickler auf ihre Korrektheit beurteilt werden, indem sie z. B. mit der einer existierenden

[1] Basis ist ein Prototyp, der vom Autor während seiner Hilfskrafttätigkeit am ADP-Compiler entwickelt wurde.

Implementation[2] verglichen oder auf Plausibilität geprüft wird. Der einmalige Test eines Programmmerkmals nach seiner erstmaligen Implementation reicht nicht aus, da zum einen weitere Programmmerkmale vorhandene Programmmerkmale beeinflussen können und zum anderen Optimierungen und Refaktorisierungen unbeabsichtigt neue Fehler in bereits getestete Programmmerkmale einführen können.

Mit steigender Anzahl von Programmmerkmalen nimmt auch die Anzahl von verschiedenen Tests zu, bestehend aus verschiedenen Testprogrammen, Programmeingaben und Optionskombinationen. Die Tests manuell durchzuführen, ist aus zwei Gründen problematisch. Zum einen ist die manuelle Durchführung mühsam und fehleranfällig. Zum anderen ist die regelmäßige Durchführung lästig, da der Tester diszipliniert die Bestandteile der existierenden Tests und deren Ausgaben aufbewahren muss, um die neuen Ausgaben mit den alten vergleichen zu können. Die Gefahr steigt, dass aus Gründen der Bequemlichkeit Tests ausgelassen werden bzw. die Tests eben nicht regelmäßig nach Änderungen am Projektcode durchgeführt werden.

Besonders ärgerlich sind Fehler, die schon einmal korrigiert wurden, aber nach einiger Zeit durch neue Änderungen am Projektcode erneut unbemerkt eingeführt werden, weil nach der ursprünglichen Korrektur kein Test erstellt wurde, der auf Vorhandensein des korrigierten Fehlers testet bzw. weil dieser Test nicht wiederholt wurde.

Aus diesen Gründen drängt sich die Erstellung eines automatisierten Testframeworks auf, welchem einfach neue Tests hinzugefügt werden können und welches ein Verzeichnis der zugehörigen korrekten Normausgaben der Tests zum automatischen Vergleich bei Wiederholungen der Tests verwaltet.

Im Bereich des Software-Engineering ist die Verwendung von automatischen Tests in Testframeworks bzw. Testsuites sehr verbreitet. [9] beschreibt ein allgemeines Testframework für Smalltalk und fordert eine Kultur des Testens bei der Smalltalk-Entwicklung. Testframeworks für andere Sprachen, wie JUnit für Java [3] und unittest für Python [25], sind durch den Aufbau dieses Smalltalk-Testframeworks beeinflusst. Beispiele für Compiler, welche umfangreiche Testsuites verwenden, sind der GCC [1], der GHC (Glasgow Haskell Compiler) und der Sun-Java-Compiler [4]. Andere umfangreiche Softwareprojekte, welche ausgereifte Testsuites enthalten, sind z. B. die glibc (eine Implementation der Standard C Bibliothek), sed (ein streamorientierter nicht interaktiver Editor), Eclipse (eine Java-Entwicklungsumgebung) und MoinMoin (eine Wiki-Implementierung in Python).

Eine Alternative zur umfangreichen Verwendung von Tests, die möglichst große Bereiche der Codeerzeugung und deren Randfälle abdecken, um Fehler zu finden und Regressionen zu vermeiden, ist der Korrektheitsbeweis des Compiler-Codes. Da Programmbeweise allerdings schon für kleine Programmfragmente sehr umfangreich ausfallen, ist der Beweis eines umfangreichen Compiler-Projekts nicht praktikabel. Im Gegensatz zu Tests, welche nur neugeschrieben werden müssen,

[2] Beim ADP-Compiler z. B. mit der Programmausgabe eines ADP-Programms, welches die DSL-Implementation von ADP benutzt.

wenn die verwendete Ausgabe-Schnittstelle sich ändert, muss der Beweis jedesmal erneut durchgeführt werden, wenn der Code aufgrund von neuen Merkmalen, Refaktorisierungen oder Optimierungen verändert wird. Genauso wie die Erstellung von Tests ist auch die Erstellung von Programmbeweisen selbst fehleranfällig.

Natürlich ist das erfolgreiche Durchlaufen einer umfangreichen Testsuite kein Beweis, dass der getestete Compiler im Allgemeinen korrekten Code generiert. Aber wenn die Testsuite einige Randfälle, typische Eingaben, bekannte korrigierte Fehlerfälle enthält und wenn möglichst viele verschiedene Codepfade im Compiler bei den verschiedenen Tests durchlaufen werden, können zumindest Regressionen vermieden und ein umfangreicher Anwendungsbereich gegen Fehler abgesichert werden.

5.2 Design

Literatur über Verwendung von Tests in Software-Engineering-Prozessen unterscheidet verschiedene Kategorieren von Tests. Zwei gängige Kategorien sind Unittests und Integrationtests (z. B. in [9]). Ein Unittest testet eine Einheit von einem Software-Projekt und ein Integrationtest ist ein Testfall, welcher mehrere Einheiten, die miteinander interagieren, des Projektes betrifft. Als Einheiten für den Unittest können einzelne Funktionen, Klassen oder Programmmodule gewählt werden.

Die Testsuite des ADP-Compilers enthält Unittests und Integrationtests. Die Unittests testen die beiden Tabellierungsoptimierungsmodule und die Ausgabe des Frontends, welche den aus der ADP-Grammatik abgeleiteten annotierten Dependency-Graph enthält und welche die Eingabe der Tabellierungsoptimierungsmodule ist. Die Codegenerierung des C- und des Java-Backends wird durch Integrationtests getestet, da alle Module des ADP-Compilers an der Codegenerierung bzw. Codeausgabe beteiligt sind.

Die Codegenerierungstests verwenden die Programmausgabe der kompilierten Compiler-Ausgabe. In [9] wird dies mit 'interface-based tests' bezeichnet und kritisiert, da solche Tests bei jeder Änderung am Interface fehlschlagen könnten und somit false-negatives produzierten. Dieser Einwand ist bei dem Test der ADPC-Codegenerierung nicht relevant, da zum einen die Standard-Ausgabe der generierten ADP-Programme sehr stabil ist. Zum anderen enthalten die Codegenerierungs-Tests einen flexiblen Parser, der Änderungen in unwesentlichen Teilen der Ausgabe, wie z.B. whitespaces, ignoriert. Außerdem werden so auch Fehler in dem Default-Ausgabe-Code gefunden, und die erzeugten Programme werden genauso kompiliert, wie dies auch typischerweise der ADP-Endbenutzer tut. Wenn nur zentrale Funktionen des generierten Codes getestet würden, würde der Default-Code gar nicht getestet, und es müsste extra Code erzeugt werden, der die Funktionsaufrufe zu den zentralen Funktionen durchführt.

5.2.1 Testfälle

Die Grundlage der ADPC-Testsuite sind verschiedene Testfälle. Ein Testfall bezeichnet hierbei eine Klasse, welche die Gemeinsamkeiten einer Menge von Tests zusammenfasst. Konkrete Tests dieser Menge sind Instanzen einer bestimmten Testfall-Klasse, wobei Attribute der Klasse die Modul-Eingabe bzw. die Programm-Eingabe bestimmen. Die Gemeinsamkeiten, die in einem Testfall kodiert sind, sind z. B. der Code, der die Normausgabe mit der aktuellen Ausgabe vergleicht, die Konstruktionen von externen Programmaufrufen und die Plausibilitätsprüfung der aktuellen Ausgabe.

Die Codegenerierung wird durch mehrere Testfälle abgedeckt. Allgemeine ADP-Programme werden durch Instanzen des CodegenTest-Testfalls getestet. Die Test-Instanzen der Codgenerierungstestfälle unterscheiden sich durch die Zielsprache, die verwendete Algebra, das verwendete ADP-Programm, die Programmeingabe und optionale Parameter für das erzeugte ADP-Programm. Da für RNA/DNA-spezifische ADP-Programme Code erzeugt wird, der weitere Optionen akzeptiert und eine andere Default-Ausgabe ausführt, ist für diese Klasse von ADP-Programmen der Testfall `RnaTest` zuständig. Der Testfall `TupleTest` ist notwendig, da das C-Backend für ADP-Algebren, welche Tupel-Datentypen enthalten, keinen Code erzeugt, der direkt kompilierbar ist. Der C-Code muss nämlich vom Endbenutzer gepatcht werden. Im Falle der C-Codegenerierung übernimmt der `TupleTest`-Testfall automatisch das Patchen des C-Codes, und im Java-Fall wird der Rna-Testfall normal aufgerufen. Der WindowTest-Testfall testet den Window-Mode-Code, den der ADP-Compiler optional für ADP-Programme erzeugt. Der Window-Mode hat eine abweichende Standard-Ausgabe, da für jede Bewegung des Fensters über die Eingabesequenz die Subsequenz, die das Fenster abdeckt, und deren (sub-)-optimale Scores inklusive Kandidaten ausgeben wird. Alle Codegen-Testfälle enthalten einen signifikanten Anteil von gemeinsamem Code, der in der Basisklasse CgBaseTest konsolidiert ist. Dazu gehört die Konstruktion von Dateinamen und die Aufrufe von Vergleichs-, Programmerzeugungs-, Programmausführungs- und Programmentfernungscode. Die Codegen-Testfälle sind Teil des `testcase`-Pakets (Abbildung 5.1).

Die `CgBaseTest`-Testfall-Klasse ist sprachunabhängig designt. Die Standardausgabe von Code, welcher von unterschiedlichen Sprach-Backends generiert wird, unterscheidet sich. Natürlich ist die Kompilation des vom ADPC ausgegebenen Zielcode sprachabhängig, da unterschiedliche Compiler mit unterschiedlichen Optionen verwendet werden müssen. Ebenso ist die Ausführung sprachabhängig, weil z. B. kompilierte Java-Programme beim JVM-Aufruf als Parameter übergeben werden müssen. Zur Zeit ist sogar die Verwendung von einem bestimmten Metafrontend für den ADP-Compiler sprachabhängig. Für C-Code ist dies der XML-Interfacer, welcher stark mit make und anderen Unix-Tools interagiert, und für Java-Code wird das neue Metafrontend (siehe Abschnitt 3.4) verwendet. Deswegen ist die sprachabhängige Konstruktion und Durchführung der externen Programmaufrufe nicht der Teil der `CgBaseTest`-Klasse, sondern ist in das Paket `cg` ausgelagert. Ebenso ausgelagert ist das Parsen der sprachabhängigen Standardausgabe. Für diese Aufgaben

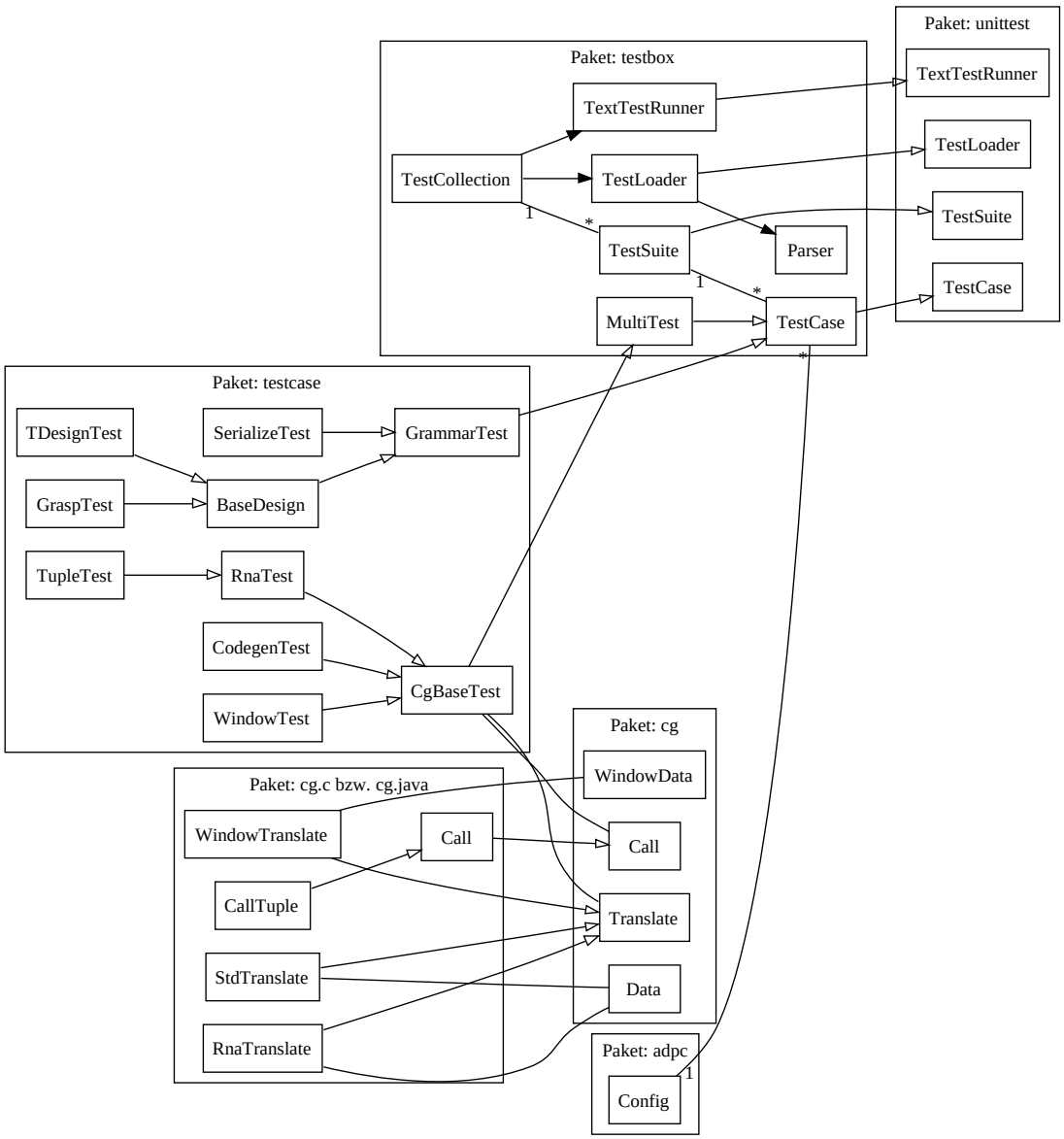

Abbildung 5.1: UML-Klassendiagramm [12] der ADPC-Testsuite bzw. des Testframeworks. Aus Gründen der Übersichtlichkeit sind einige Klassen in dem Diagramm nicht enthalten. Nicht enthalten sind z. B. verschiedene Exception-Klassen, spezielle Datenstrukturklassen und Hilfsklassen. Die Pakete `cg.c` und `cg.java` implementieren die Basisklassen aus dem Paket `cg` und stellen somit die Test-Abdeckung für die Zielsprachen C bzw Java bereit. Da das Paket `cg.java` bis auf die `CallTuple`-Klasse Klassen mit den gleichen Namen wie in `cg.c` enthält, ist es nicht als weiterer Paket-Cluster in dem Diagramm enthalten. Das Paket `unittest` ist Teil der Python Standard-Library.

enthält das Paket `cg` die abstrakte Basisklasse `Call` bzw. `Translate`. Für jedes neue zu testende Sprach-Backend des ADP-Compilers muss nun ein Subpaket unterhalb von `cg` erstellt werden, welches die beiden abstrakten Basisklassen implementiert und somit eine abstrakte Schnittstelle bereitstellt, welche von den sprachabhängigen Merkmalen abstrahiert, und von der `CgBaseTest`-Klasse erwartet wird. Das Paket `cg` enthält eine Symboltabelle, die die sprachspezifischen `Call`-Klassen den verwendeten Sprachen zuordnet. Die abgeleiteten Translate Klassen, die die verschiedenen Standardausgaben in eine allgemeingültige Datenstruktur übersetzen, sind in einer weiteren Symboltabelle enthalten, da sie nicht nur von der Sprache, sondern auch von Programmvarianten wie Window-Mode und RNA-spezifischen ADP-Programmen abhängen können. Die `CgBaseTest`-Klasse greift für den Test eines Backends über die Symboltabellen des `cg`-Moduls auf die korrekten Subklassen von `Call` bzw. `Translate` zu. Da die von `Translate` abgeleiteten Klassen die Programmausgabe in eine allgemeine Datenstruktur übersetzen, muss der Vergleich der Ausgaben nur einmal implementiert werden und nicht mehrmals für jede Kombination von Ausgabe-Sprache und Standardausgabe. Die `CgBaseTest`-Klasse serialisiert diese Datenstruktur als Normausgabe, so dass die Normprogrammausgabe des Zielcodes eines Sprach-Backends mit der Ausgabe des Zielcodes eines anderen Sprachbackends verglichen werden kann. Zur Zeit verwendet die Testsuite als Standard die Normausgabe der C-Backend-Tests auch als Normausgabe für weitere Sprach-Backend-Tests, da das C-Backend sehr stabil ist. Die allgemeinen Datenstrukturen sind Teil des Paketes `cg`. Die Klasse `Data` enthält den optimalen Score, die verwendete Algebra und eine Liste von (sub-)optimalen Kandidatenstrukturen und Scores. Die Klasse `WindowData` ist eine Aggregation von `Data` (Abbildung 5.1). Der Vergleichscode ist jeweils Teil der Datenstruktur-Klassen.

Der Testfall `SerializeTest` testet das ADPC-Modul, welches den annotierten Dependency-Graph zu einer ADP-Grammatik generiert. Instanzen dieses Testfalls unterscheiden sich durch den Dateinamen der ADP-Programme. Der Testfall ruft den ADP-Compiler mit bestimmten Optionen auf, so dass die Kompilierung frühzeitig abgebrochen und der Dependency-Graph in einem einfachen ASCII-Format in eine Datei serialisiert wird. Mit diesem Testfall ist ein großer und wichtiger Teil des Frontends abgedeckt. Denn Fehler in diesem Teil des Compilers, z. B. Änderungen im Parser oder im Ableitungscode, können Folgefehler in den Tabellierungsmodulen verursachen, da sie die Informationen des annotierten Dependency-Graphen als Eingabe für die Berechnung der optimalen Tabellenkonfiguration verwenden. Ein Fehler könnte z. B. die Ausgabe einer nicht-optimalen Tabellenkonfiguration als optimale Tabellenkonfiguration sein. Durch Verwendung einer nicht-optimalen Tabellenkonfiguration könnte im Folgenden Code erzeugt werden, der zwar eine korrekte Ausgabe liefert, aber dessen Laufzeit nicht asymptotisch optimal ist. Der `SerializeTest`-Testfall grenzt somit bei einem Fehler eines Tabellierungs-Tests zu einer Grammatik den Bereich ein, der den Fehler verursacht hat: Bei zusätzlichem Fehlschlagen unter derselben Grammatik wurde der Fehler offensichtlich durch eine Änderung im Frontend (mit-)verursacht; schlägt er nicht fehl, dann ist der Fehler durch eine Änderung in den Tabellierungsmodulen entstanden. Zusätzlich werden

durch diesen einfachen Testfall Fehler gefunden, die nicht durch Codegenerierungstests gefunden werden können, da diese nicht die asymptotisch optimale Laufzeit vergleichen.

Die Testfälle `TDesignTest` bzw. `GraspTest` testen das Tabellierungsmodul zum exakten bzw. zum approximativen Tabellierungsdesign. Die Test-Instanzen unterscheiden sich durch den Dateinamen, welcher die zu testende Grammatik enthält. Durch ein Hilfsprogramm, das durch zahlreiche Parameter die interne Schnittstelle nach außen bereitstellt, welche auch der ADP-Compiler intern verwendet, können die Module unabhängig von dem restlichen ADP-Compiler in verschiedenen Konfigurationen getestet werden. Als Eingabe wird die serialisierte Ausgabe des Dependency-Graphen der ADP-Grammatik des ADP-Compilers verwendet.

Die Testsuite enthält eine Verzeichnis-Hierachie, welche die Tests strukturiert. Im Unterverzeichnis `input` sind alle Eingabedaten für die Codegenerierungstests abgelegt. Das Unterverzeichnis `state` enthält die Normausgaben der einzelnen Tests und das Verzeichnis `output` die Ausgaben des letzten Testsuite-Laufs. Im Unterverzeichnis `grammar` sind die verwendeten ADP-Programme enthalten und das Verzeichnis `workdir` wird als current-working-directory für die temporären Dateien und die Programmerzeugung während der Codeerzeugung verwendet.

5.2.2 Testframework

Das allgemeine Testframework, das die Testsuite verwendet, ist im Paket `testbox` erstellt worden. Das Paket ist von einem in [9] beschriebenen Testframework abgeleitet (siehe Abbildung 5.1). Es hat keine Abhängigkeiten zu den anderen Paketen der ADP-Compiler-Testsuite und ist somit auch als Basis für weitere Testsuites geeignet, welche nichts mit ADP zu tun haben. Die Verwendung dieses Testframeworks erspart die wiederholte Entwicklung eines spezifischen Konfigurationsdateiformats, der Implementierung eines Konfigurationsdateiparsers, eines CLI und der restlichen Infrastruktur. Für neue Testsuites müssen nur neue Testfälle entwickelt werden.

Im Gegensatz zu dieser klassischen Architektur bedeutet der Error-Status von Tests einen Fehler im Testfall, der korrigiert werden soll, und nicht nur das Auftreten eines unvorhergesehenen Fehlers. Der Fail-Status steht für das Fehlschlagen eines Tests.

Wenn ein Testfall vor dem eigentlichen Test benötigte Abhängigkeiten erstellen und diese nach dem Test aufräumen muss, kann dies geschehen, indem beim Ableiten von `TestCase` die Methoden `setUp` bzw. `tearDown` überschrieben werden. Diese Funktionalität wird bei den Codegenerierungstestfällen benötigt, da vor dem eigentlichen Test der Zielcode erzeugt und kompiliert werden muss. Um Seiteneffekte auf die Erzeugung von nachfolgendem Zielcode zu vermeiden, muss nach dem Test das `workdir` aufgeräumt werden. Das Problem hierbei ist, dass aufeinanderfolge Codegenerierungstests, welche dasselbe ADP-Quellprogramm verwenden und sich nur in der Programmeingabe unterscheiden, unnötig redundant und zeitaufwändig das benötigte Programm erzeugen. Deswegen enthält das Testframework zusätzlich die von `TestCase` abgeleitete Klasse `MultiTest`, durch deren Ableitung und

-g	Zeichnet den aktuellen Status, d. h. die Normausgaben auf.
-v	Schaltet die Ausgabe von detaillierten Informationen der ausgeführten Tests an. Doppelte Angabe schaltet zusätzlich Debug-Information an.
-l	Listet die verfügbaren Testsuites auf
-i file	Die Konfigurationsdatei der Testsuite (default: ./test.conf).
-c	Entfernt die Ausgabe aus dem output-Verzeichnis.
-s	Setzt das Verzeichnis der Normausgaben zurück.
`[testsuites]`	Optionale Angabe auszuführenden Testsuite-Namen (default: all).
-h	Hilfe

Abbildung 5.2: Optionen des Testframeworks von der `TestCollection`-Klasse des `testbox`-Pakets bereitgestellten Testframeworks.

Implementierung der beiden Methoden `prepareTestSuite` und `cleanTestSuite` Testfälle konstruiert werden, die den Vor- und Nachbedingungscode nur unter konfigurierbaren Bedingungen ausführen. Im Codegenerierungs-Testfall geschieht dies, wenn die Dateinamen-Attribute aufeinanderfolgender Tests sich unterscheiden.

Die Klasse `TestCase` ist um eine Methode erweitert, welche eindeutige Dateinamen für die Ausgaben einer Test-Instanz erzeugt. Spezifische Testfälle können diese Methode überschreiben, um ein eigenes Namensschema zu verwenden.

Die klassische Architektur enthält als Strukturierung von Tests die Klasse `TestCase` und die Klasse `TestSuite`, welche eine Aggregation von `TestCase`-Instanzen ist. In dieser Modellierung können entweder einzelne Tests oder komplette `TestSuites` ausgeführt werden. Das `testbox`-Testframework erweitert diese Modellierung um eine Klasse `TestCollection`, welche eine Aggregation von TestSuites ist. Somit können Gruppen von Tests zu Testsuites zusammengefasst werden, welche in einer `TestCollection` zusammengefasst sind. Entweder kann die gesamte Testcollection ausgeführt werden oder nur einzelne Testsuites. Dies ist eine natürliche Modellierung, da nach einer Änderung in einem abgeschlossenen Bereich, wie z. B. einem Tabellierungsmodul, aus Zeit- bzw Übersichtlichkeitsgründen nur die zugehörige Testsuite ausgeführt werden kann. Beispielsweise besteht die konkrete ADPC-Testsuite aus den `TestSuite`-Instanzen `codegen`, `serial`, `tdesign`, `grasp`, `tupel` und `window`, welche die entsprechenden Bereiche abdecken. Die `TestCollection`-Klasse ist nicht nur eine zusätzliche Aggregation von Testsuites, sondern sie stellt auch ein CLI bereit, das sich dynamisch an die konfigurierten Tests bzw. Testsuites anpasst. So kann z.B. angezeigt werden, welche Testsuites zur Verfügung stehen, einzelne Testsuites oder die vollständige Testcollection ausgeführt und Normausgaben in das zentrale Verzeichnis aufgenommen werden. Eine komplette Übersicht der Optionen gibt Abbildung 5.2.

Deswegen wird auch der `TextTestRunner` erweitert, der die Tests aufruft und ihre Ergebnisse übersichtlich formatiert auf die Standardausgabe des Terminals aus-

gibt. In der erweiterten Form kann er zusätzlichen mit `TestCollection`-Instanzen umgehen, und er erzeugt ein übersichtliches Gesamtergebnis über alle beteiligten `TestSuite`-Instanzen.

Die Klasse `TestLoader` stellt mehrere Convenience-Methoden bereit, mit deren Hilfe `TestsSuite`-Instanzen erstellt werden können. Beispielsweisekann ein Klassenname übergeben werden, und es wird für jede Klassenmethode, welche das Präfix `test` besitzt, eine eigene Testfall-Instanz angelegt, die beim Testaufruf diese bestimmte Klassenmethode aufruft und der `TestSuite`-Instanz hinzugefügt. Es kann auch ein Modul-Name angegeben werden, so dass alle von `TestCase` abgeleiteten Klassen des Moduls nach dem vorherigen Schema behandelt werden. Der abgeleitete `TestLoader` des `testbox`-Pakets erweitert den klassischen `TestLoader` um die Möglichkeit, eine `TestCollection`-Instanz aus dem Inhalt einer externen Konfigurationsdatei zu erstellen. Damit werden in Abhängigkeit von der Konfigurationsdatei eine `TestCollection`-Instanz und Instanzen der verschiedenen spezifizierten Testfälle angelegt. Die Attribute der Testfall-Instanzen werden entsprechend ihrer Deklaration und Definition aus der Konfigurationsdatei initialisiert. Außerdem berücksichtigt der abgeleitete `TestLoader` das Präfix gen von Testfall-Methodennamen und erzeugt getrennt für diese Methoden eine `TestCollection`-Instanz, welche für die Generierung von Normausgaben verwendet wird.

Das Konfigurationsdateiformat verfolgt die Abseitsregel [23], d. h. das Einrückungen, also Whitespace von Zeilenbeginn zum ersten Symbol, nicht beliebig ist sondern eine Bedeutung hat. Die Erhöhung der Einrücktiefe im Vergleich zur vorherigen Zeile impliziert den Beginn eines neuen Konfigurationsblocks, und die Verringerung der Einrücktiefe beendet so viele Blöcke, wie im vorhinein durch größere Einrürcktiefen begonnen wurden. Ein Beispiel für das Konfigurationsdateiformat ist in Abbildung 5.3 angegeben. Der Vorteil dieses Formats liegt darin, dass Hierachien von Direktiven kompakt aufgeschrieben und mit einem Texteditor von einem Benutzer einfach geändert werden können. Im Gegensatz hierzu ist ein XML-Format weniger kompakt und übersichtlich, da Blöcke von öffnenden und schließenden XML-Tags eingeschlossen werden müssen. Außerdem ist XML eher für ein Datenaustauschformat, welches automatisch verarbeitet wird, geeignet, als für ein Konfigurationsdateiformat, welches von Menschen manuell editiert werden soll, da nicht-triviale XML-Dateien schnell unübersichtlich werden und mit einem normalen Texteditor schwer zu pflegen sind. Dem Autor ist kein existierendes Konfigurationsdateiformat bekannt, welches die Abseitsregel implementiert. Konfigurationsdirektiven, die nicht reserviert sind, deklarieren und definieren ein Attribut einer Testfall-Instanz. Die Hierachie-Ebenen von Konfigurationsblöcken sind als Ebenen eines Baumes zu betrachten. Jede Direktive, der kein tiefer eingerückter Block folgt, ist ein Blatt, und für jedes Blatt wird vom `TestLoader` eine Testfall-Instanz erzeugt, auf die die Direktiven seiner Vorgängerknoten angewendet werden. Die reservierten Direktiven sind `testsuite`, welche den Namen der Testklasse definiert, auf die sich alle tiefer eingerückten Direktiven beziehen, und `class`, welche die Testfall-Klasse angibt, von der eine Instanz erzeugt wird. Des Weiteren definiert `def` einen Block, der mit der Direktive `$name` mehrmals in Konfigurationshierachien referenziert werden

```
def el
  grammar ElMamun.lhs
    filename el.small.inp
      algebra buyer
      algebra seller
...

testsuite codegen 'Codegen part'
  class CodegenTest
   parameter -d 0
     language C
       $el
     language java
       $el
  class RnaTest
    grammar RNAfold.lhs
      filename rnafold.small.inp
        language java
        language C
...

default codegen serial tdesign grasp tuple window
```

Abbildung 5.3: Exemplarischer Auszug aus der Konfigurationsdatei der ADP-Compiler-Testsuite. Es gilt die Abseitsregel. Die Direktiven def, testsuite und default sind reserviert. Die restlichen Direktiven deklarieren und definieren Attribute der Testfall-Instanzen. Das Fragment erzeugt sechs Testfall-Instanzen.

kann. Der Parser erkennt beim Parsen unerlaubte Zyklen im Direktiven-DAG[3], um Endlosschleifen in der weiteren Verarbeitung zu vermeiden. Die Direktive default gibt eine Liste von Testsuite-Namen an, welche als Standard bei der Ausführung der `TestCollection`-Instanz ausgeführt werden. Durch Angabe von Optionen kann diese Liste überschrieben werden. Der Parser dieses Konfigurationsdateiformats ist in der Klasse `Parser` des `testbox`-Pakets enthalten.

5.3 Implementation

Das vorgestellte Design der Testsuite bzw. des Testframeworks ist in Python implementiert worden. Python [34] ist eine dynamische Skriptsprache, die auch Objek-

[3]Directed Acyclic Graph

torientierung und Exceptions unterstützt. Dynamische Sprachmerkmale sind das dynamische Typsystem und die Veränderbarkeit von Klassen und Modulen während der Laufzeit. So werden z. B. die Klassenattribute eines Objekt erst während der Laufzeit bei der ersten Zuweisung dynamisch hinzugefügt, und die Typprüfung der Variablen geschieht erst während der Laufzeit. Der Python-Interpreter kompiliert den Quellcode in einen Bytecode, der von einer VM interpretiert wird. Die Vorteile von Python liegen in seiner Plattformunabhängigkeit, einer umfangreichen Standardbibliothek und dem Vorhandensein von Highlevel-Konstruktionen, wie z. B. Listenbeschreibungen, Listen und Hash-Tabellen, die Teil der Sprache sind. Die dynamischen Sprachmerkmale sind zum einen ein Vorteil, da während der Laufzeit nach dem Parsen der Konfigurationsdatei die Attribute der Test-Instanzen den Objekten einfach hinzugefügt werden können, und diese Attribute in den Testfall-Klassen verwendet werden können, da sie während der Compile-Zeit nicht überprüft werden. Auf der anderen Seite sind die dynamischen Merkmale nachteilig, da Compilezeitfehler durch Laufzeitfehler eingetauscht werden und die Typinformation fehlt, welche die Wartbarkeit von Programmen erhöht. Weitere Nachteile von Python sind einige inkonsequente Sprachmerkmale[4], die Existenz von zwei verschiedenen Klassensystemen[5] und unnötige Implementationsnachteile[6].

Die Python-Standardbibliothek enthält das Paket `unittest`[25], welches das in [9] beschriebene Smalltalk-Testframework auf die Gegebenheiten in Python abbildet. Das Paket ist ausführlich dokumentiert und wird als Basis für das neuentwickelte `testbox`-Testframework verwendet.

Die Implementation der Testsuite und des Testframeworks sind komplett in Python-Pakete unterteil. Sie liegen unterhalb des Unterverzeichnises `lib`. Das Paket `testbox` enthält das allgemeine Testframework, das Paket sysext stellt eine übersichtliche Schnittstelle für den Aufruf von externen Programmen bereit, das Paket `adpc` enthält die ADP-Testsuite-spezifische Konfigurationsklasse, das Paket `testcase` enthält die konkreten Testfälle, und das Paket `cg` enthält die Klassen, welche für die Codgenerierungstestfälle von den Sprachbackends abstrahieren (siehe Abbildung 5.1).

Die einzelnen Pakete machen umfangreichen Gebrauch von spezialisierten Exceptions, um Fehlerbedingungen oder fehlgeschlagene Tests zu signalisieren. Dabei wird durchgehend auf die Sammlung von für den Endanwender hilfreichen Informationen in den Exception-Objekten geachtet, damit der Exception-behandelnde Code diese Informationen ausgeben kann. So enthalten z. B. die Exceptions des Konfigurationsdateiparsers die Zeilennummer und die fehlerhafte Zeile der Konfigurationsdatei neben dem Fehlergrund, damit der Endbenutzer einen Hinweise auf

[4] Z.B. wird die Länge eines Listenobjekts mit der globalen Funktion len ermittelt, andere Funktionen auf Listen sind aber Methoden der Listenklasse und Lambda-Ausdrücke dürfen nur einen Parameter besitzen.

[5] New-style-Classes die nicht in der Standard-Dokumentation dokumentiert sind und Old-Style-Classes

[6] Z.B. wird Constant-Folding erst ab Version 2.5 durchgeführt und das Bytecodeformat ändert sich häufig zwischen den Python-Versionen.

die konkrete Fehlerursache erhält. Ebenso melden z. B. die Codegenerierungstestfälle im Falle eines Unterschiedes in der Programmausgabe detailliert, um welche Art von Unterschied es sich handelt, welche Kandidaten betroffen sind, zusätzliche Kontextinformationen, wie die Strukturen und die Scores der Kandidaten, und die Namen der Normausgabe- bzw. Programmausgabedateien.

Für die bedingte Ausgabe von Logging-Informationen wird das `logging`-Framework der Python-Standardbibliothek verwendet. Dieses Framework ist sehr flexibel verwendbar. Es unterstützt unter anderem die Filterung von Meldungen, viele verschiedene Ausgabe-Streams, das Multiplexen von Ausgabe-Streams und hierachische Meldungen. Die verwendeten Log-Level sind `Error`, `Debug` und `Info`. Als Standard werden alle Nachrichten der Stufe `Error` und `Info` in die Log-Datei `test.log` im akutellen Arbeitsverzeichnis geschrieben. Die Stufe Error wird zusätzlich auf die Standardausgabe des Terminals ausgegeben. Optional kann der Level der Ausgabe auf der Standardausgabe erhöht werden. Eine ausführliche Protokollierung der Testläufe wird mit der Log-Stufe `Info` erzeugt. Teile dieser Protokollierung sind z. B. die verschiedenen externen Programmaufrufe inklusive ihrer Parameter und die Ergebnisse der Tests. Die Protokollierung in eine Log-Datei wird durchgeführt, da somit bei von der Testsuite gemeldeten Fehlern auf fremden Rechnersystemen die Anwender einfach die Log-Datei und das `output`-Verzeichnis, welches die Sammlung aller Test-Ausgaben enthält, in ein Archiv packen und diese Daten an die ADPC-Autoren schicken können, welche dann hinreichend Informationen besitzen, um den Fehler nachvollziehen zu können.

Die Testfälle implementieren das im vorherigen Abschnitt beschriebene Design. Neben dem detaillierten Vergleich der Programmausgaben werden auch einige Konsistenz-Checks auf jede einzelne Ausgabe angewendet. Beispielsweise testet der Window-Mode-Codegenerierungstestfall, ob die Länge der RNA-Sekundärstrukturen mit der Länge der Eingabesequenz übereinstimmt und ob überhaupt Kandidaten in den Fenstern gefunden werden. Abgeschnittene Kandidatenstrukturen in der Ausgabe des Window-Mode war ein mehrmals berichteter Fehler von ADPC-Benutzern. Für die Serialisierung der allgemeinen Daten-Objekte der Codegenierungstestfälle wird das Paket `pickle` der Python-Standardbibliothek verwendet. Dieses Paket serialisiert beliebige Python-Objekte, so dass die Erzeugung von Boilerplate-Code zur Serialisierung von Objekten vermieden wird. Das für die Serialisierung verwendete Format ist rückwärtskompatibel zwischen verschiedenen Versionen von Python.

6 Anwendungsfall: ADPDemo

ADPDemo ist ein GUI-Beispielprogramm, welches die einfache Integration eines ADP-Programms, das mit dem ADP-Compiler nach Java übersetzt wird, in ein Java-Programm demonstriert. Als ADP-Programm wird RNAfold (implementiert den RNAfold-Algorithmus [20]) ausgewählt. RNAfold enthält eine minimum-free-energy (mfe) Algebra, welche die freie Energie eines Kandidaten in Bezug auf seine Sekundärstruktur minimiert. Die RNAfold-Grammatik besteht aus 11 Nichtterminalen. Der Suchraum der Sekundärstrukturen wird durch Modellierung von verschiedenen Schleifen[1] und geschlossenen Strukturen beschrieben, welche in eingeschränkter Art und Weise kombiniert werden können. Zur Darstellung der optimalen bzw. suboptimalen Kandidaten wird eine Baumdarstellungskomponente und RNAMovies2 [21] verwendet. Die Baumdarstellungskomponente ist im Rahmen der Diplomarbeit entwickelt worden und implementiert einen einfachen Layout-Algorithmus, der Top-Down Platz für die Kinder-Knoten reserviert. RNAMovies2 ist ein eigenständiges Java-Programm, welches einen Algorithmus implementiert, der aus den Basenpaarungen und der RNA-Sequenz die Koordinaten der Sekundärstruktur in der Ebene berechnet und zweidimensional darstellt. Die Hauptkomponente von RNAMovies2 lässt sich aber auch einfach als Bibliothek in andere Java-Programme integrieren.

Der Anwender kann die Sequenz über ein Texteingabefeld eingeben und als Parameter den Bereich der während des Backtracing anzuzeigenden suboptimalen Kandidaten wählen. Die optimalen bzw. suboptimalen Kandidaten werden in einem Listenauswahlfeld mit ihrem Wert absteigend sortiert angezeigt. Bei Auswahl eines dort dargestellten Kandidaten werden die beiden graphischen Darstellungskomponenten aktualisiert, so dass sie den ausgewählten Kandidaten in einer Baumdarstellung bzw. graphischen Sekundärstrukturdarstellung anzeigen. Abbildung 6.1 zeigt einen Screenshot von ADPDemo.

Der Aufbau der internen Datenstruktur, die der vom ADP-Compiler generierte Backtracing-Code verwendet, wird für jedes ADP-Programm spezifisch generiert. Das hat zur Folge, dass eine allgemeine Baumdarstellungskomponente diese Datenstruktur nicht verwenden kann, da von den Implementationsdetails nicht durch eine Schnittstelle abstrahiert wird. Deswegen ist eine einfache allgemeine Baumdatenstruktur entwickelt worden, welche aus den Klassen `Tree` und `Node` besteht. Sie sind Teil des `Tree`-Pakets[2]. Der ADP-Compiler erzeugt zusätzlichen Code, welcher die implementationsspezifische Backtracing-Datenstruktur in die allgemeine Baumdatenstruktur konvertiert. Die abstrakte Basisklasse `Algebra` enthält die Methode

[1] z.B. hairpin-loop

[2] Verfügbar in `java/tree` des `adp2java`-Entwicklungszweiges des ADP-Compilers.

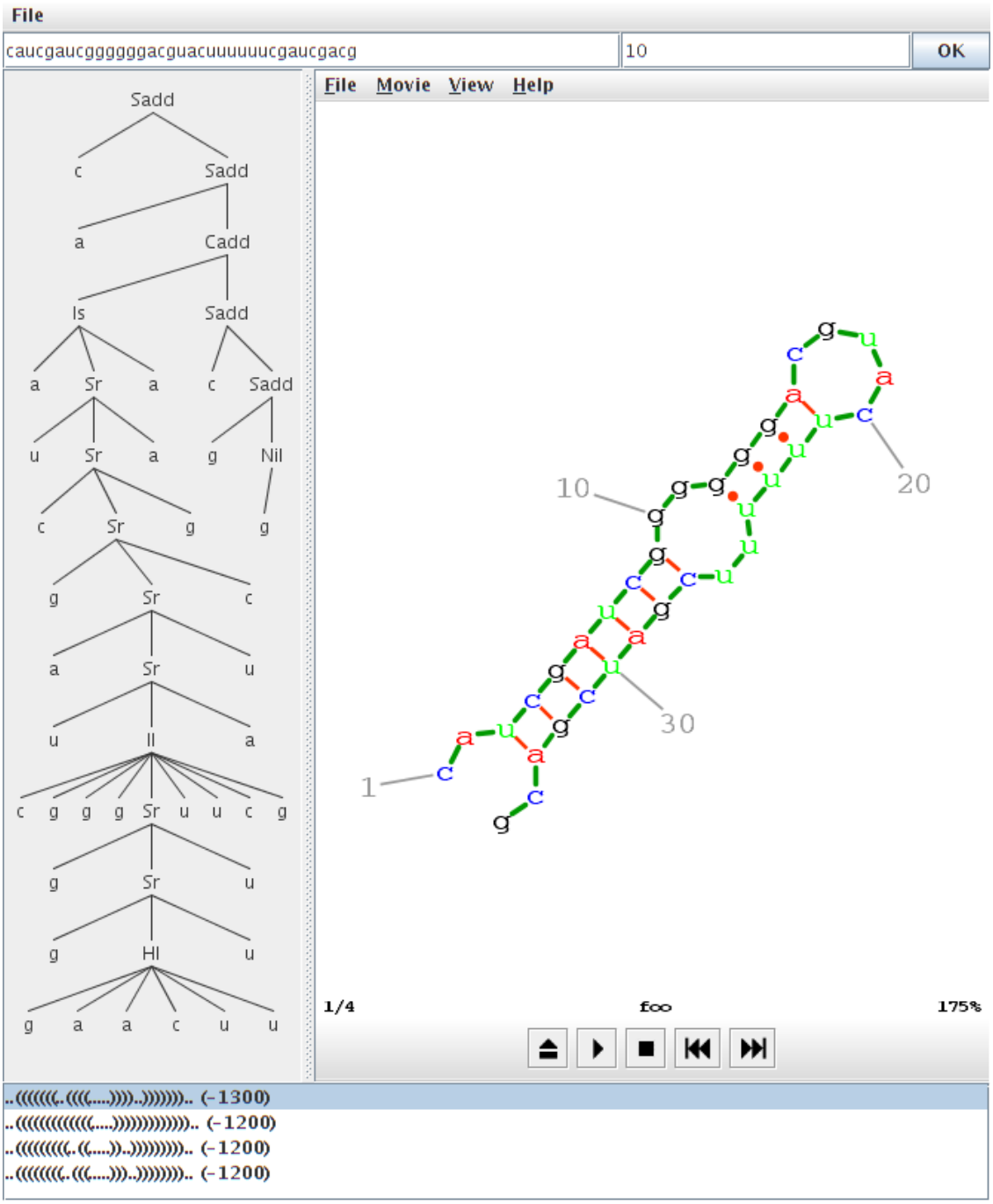

Abbildung 6.1: Screenshot von ADPDemo.

`getTree()`, welche die Konvertierung durchführt. Wenn die Baumdarstellung nicht benötigt wird, entsteht kein Overhead während der Laufzeit des ADP-Programms, da die Konvertierung erst mit dem Aufruf von `getTree()` durchgeführt wird.

Das Design von ADPDemo verwendet durchgehend das Listener/Event-Konzept. Ein Listener ist ein Interface, welches eine Klasse implementiert, die von einer anderen Klasse über bestimmte in dem Interface spezifizierte Events benachrichtigt werden soll. Dazu muss das Objekt einer Listener-implementierenden Klasse bei dem Objekte einer Event-erzeugenden Klasse registriert werden. Der Vorteil dieser Architektur ist, dass Event-Erzeuger und Listener einfach austauschbar sind und die konkrete Zusammenschaltung von Objekten außerhalb der beteiligten Objekte geschieht, also bei Änderungen dieser Zusammschaltungen nicht die beteiligten Klassen verändert werden müssen.

Konkret implementiert die Klasse `Compute` das Interface `InputListener` und wird mit diesem Interface bei der Klasse InputPanel `registriert`. `Compute` startet das RNAfold-ADP-Programm, welches über die abstrakte Klasse `StdAlgebra` angesprochen wird. Über `StdAlgebra` wird die `Output`-implementierende Klasse `SortOutput` registriert, welche in dem `ResultListener` spezifizierte `Result`-Events verschickt, nachdem die Ausgabe des ADP-Programms abgeschlossen und sortiert ist. Den `ResultListener` implementieren die Klassen `CandidatePanel` und `MainPanel`, welche beide bei neuen Ergebnissen benachrichtigt werden müssen, damit sie ihre Anzeigen aktualisieren können. `MainPanel` implementiert außerdem das Interface `SelectionListener`, mit dem es bei `CandidatePanel` registriert wird, damit die zwei Darstellungskomponenten des `MainPanel` über die Auswahl von einem dargestellten Kandidaten benachrichtigt und aktualisiert werden. Die Zusammenschaltung aller Komponenten geschieht in der `main()`-Methode der `Main`-Klasse, und die Berechnung und erste Anzeige der Kandidaten wird durch die Erzeugung von einem `Input`-Event im `InputPanel` gestartet.

Der Code von ADPDemo ist in dem Verzeichnis `java/ADPDemo` des `adp2java`-Entwicklungszweiges enthalten. Die GUI ist mit der Swing-API realisiert worden, welche Bestandteil der standardisierten Java-Bibliotheks-API ist. Die Klassen, welche ein Panel als Suffix im Namen enthalten, sind zentrale Komponenten der GUI.

7 Fazit

Die Zielsetzung, ein System zur automatischen Übersetzung von ADP-Programmen nach Java zu erstellen, ist erreicht worden. Das System besteht aus der Erweiterung des ADP-Compilers um das Java-Backend (Kapitel 3), notwendiger Modifikationen in weiteren Modulen des ADP-Compilers und der Erstellung des Metafrontends (Abschnitt 3.4). Die Laufzeit der untersuchten Java-Zielprogramme ist im Vergleich zu den C-Zielprogrammen um den Faktor 1.5 bis 4 höher (siehe Kapitel 4), was für die Verwendung einer sicheren Sprache und einer VM sehr gut ist. Benchmark Ergebnisse anderer Systeme, welche automatisch C-Code nach Java-Byte-Code übersetzen, bescheinigen in der Regel eine Verschlechterung der Laufzeit um den Faktor zwei bis zehn [8].

Wie im Abschnitt 1.2 vermutet, hat der von dem Java-Backend generierte Zielcode aufgrund der sicheren Sprachmerkmale von Java tatsächlich Fehler aufgedeckt, von denen auch der C-Zielcode betroffen ist und die dort aber nicht bemerkt wurden, weil diese Fehler im C-Code zu keinen Programmabbrüchen führen. Bei den Fehlern handelt es sich um Array-Zugriffe in einer RNA-Hilfsbibliothek außerhalb der Array-Grenzen. In einigen Test-ADP-Programmen führt die Korrektur zu keinen unterschiedlichen Ergebnissen in der Programmausgabe. Ein von Locomotif generiertes Beispielprogramm zeigt aber Abweichungen im Score und der Struktur einzelner Kandidaten unter einer minimum-free-energy-Algebra.

Als positive Seiteneffekte der Entwicklung des ADP-nach-Java-Übersetzungssystems ist ein annotationsbasierter Optionsparser für Java (Abschnitt 3.5), eine allgemeine Baumdarstellungskomponente für Java (Kapitel 6) und ein allgemeines Testframework (Kapitel 5) entwickelt worden. Diese Komponenten sind jeweils unabhängig von den restlichen Teilen des ADP-Compilers wiederverwendbar. Ein existierender annotationsbasierter Optionsparser für Java bzw. die vorherige Verwendung eines Konfigurationsdateiformats, welches die Abseitsregel implementiert, ist dem Autor nicht bekannt. Das Konfigurationsformat wird in dem Testframework zur Definition der Tests verwendet.

Die erstellte Testsuite (Kapitel 5) enthält umfangreiche Testfälle für verschiedene Bereiche des ADP-Compilers. Abgedeckt werden das Frontend, die in C geschriebenen Tabellierungsmodule und die Codeerzeugung des Compilers bzw. die Zielcodeausgabe der Sprach-Backends in verschiedenen Varianten. Dadurch wird die weitere Entwicklung des ADP-Compilers unterstützt, da Optimierungen, Refaktorisierungen und die Implementation neuer Features einfach und automatisiert auf Regressionen getestet werden und die Neueinführung von Fehlern in eine umfangreiche Testmenge verhindert werden kann. Die Automatisierung der Tests entlastet die Entwickler von mühseligen und nervigen manuellen Tests und ermöglicht eine

regelmäßige Ausführung der Tests.

Der ADPDemo-Anwendungsfall (Kapitel 6) demonstriert, wie einfach und allgemeingültig verwendbar die in Abschnitt 3.3 beschriebenen Schnittstellen des erzeugten Java-Zielcodes sind. Durch diese Schnittstellen wird die Integration von nach Java kompilierten ADP-Programmen in umfangreichere Java-Programm-Systeme ermöglicht. Der Austausch von ADP-Programmen bzw. die Abänderung von Algebren und Grammatiken ist durch die Schnittstellen ohne bzw. nur durch minimale Modifikationen im abhängigen Java-Modul möglich.

8 Ausblick

Das entwickelte Java-Backend für den ADP-Compiler ist insofern fertiggestellt, dass für ADP-Programme, welche die von dem ADP-Compiler unterstützten ADP-Merkmale verwenden, Java-Zielcode erzeugt wird. ADP-Merkmale, wie z. B. die Verwendung von Tupeln in pretty-Print-Algebren, die das ADP-Frontend zur Zeit nicht unterstützt, können natürlich auch nicht nach Java übersetzt werden. Ein weiteres Beispiel ist die Übersetzung von allgmeinen Multitrack-ADP-Grammatiken, die der ADP-Compiler noch nicht unterstützt. Mit der Integration von weiteren ADP-Merkmalen in den ADP-Compiler muss in Zukunft unter Umständen auch die TL erweitert bzw. verändert werden, so dass auch die Sprachbackends angepasst werden müssen.

Die Überarbeitung der C-zentrierten TL ist für sich allein genommen ein sinnvolles Projekt, das allerdings größere und damit zeitaufwändige Refaktorisierungen und Neuimplementierungen mehrerer Modulen des ADP-Compilers zur Folge hätte (siehe Abschnitt 3.2). Der Austausch der TL durch eine nicht-C-zentrierte Variante, würde das Java-Backend vereinfachen und viele Sonderfallbehandlungen überflüssig machen. Die Sonderfallbehandlungen sind prinzipiell problematisch, da auf Grund dieser Sonderbehandlungen die TL nicht in allen Kombinationen in der Codegenerierung frei verwendet werden kann. Außerdem würde die Erweiterung des ADP-Compilers um weitere Sprach-Backends vereinfacht.

Für die Abbildung von Tupel-Datentypen nach Java, welche in ADP-Algebren als Antwort-Datentypen verwendet werden, könnte neben dem in Abschnitt 3.2.5 beschriebenen Schema ein Alternativ-Schema evaluiert werden. Alternativ können die Tupel im Tabellierungsfall komponentenweise in Arrays gespeichert werden. Im Gegensatz zu der Erzeugung von einzelnen Objekten pro Tupel und der Tabellierung ihrer Referenzen wird in diesem Schema Speicher gespart, da pro Komponente nur ein Array-Objekt eines elementaren Datentyp erzeugt wird. Außerdem liegen die Komponenten jeweils zeilenweise im Speicher, so dass möglicherweise von der CPU-Cache profitiert werden kann. Ob der Zugriff auf Tupel in diesem Schema in der Java-VM effizienter ist, muss durch Profiling beider Schemata untersucht werden.

Das Metafrontend (Abschnitt 3.4) ist zwar in Hinsicht auf einfache Erweiterbarkeit für weitere Zielsprachen neben Java entwickelt worden, aber die Unterstützung der Zielsprache C ist noch nicht implementiert worden. Für den Umstieg von dem XML-Interfacer auf das Metafrontend bei der C-Codeerzeugung spricht die Eliminierung der in Abschnitt 3.4 beschriebenen Nachteile, wie z.B. eingeschränkte Portabilität und Probleme bei der Mitteilung von Fehlern in ADP-Programmen.

Die Testsuite enthält umfangreiche Testfälle, besonders die Codegenerierung wird umfangreich abgedeckt. Noch nicht implementiert ist allerdings die Integration von

ADP-Programmen, welche in der in Haskell eingebetteten ADP-Variante geschrieben sind. Die Ausgaben dieser Referenzimplementation könnten dann mit den Programmausgaben der von dem ADP-Compiler erzeugten ADP-Programme automatisch verglichen werden, so dass Fehler in der Codeerzeugung gefunden werden, welche die verschiedenen Sprach-Backends in der gleichen Art und Weise beeinflussen, aber nicht in der Referenzimplementierung enthalten sind. Auf Grund der sprachunabhängigen Architektur der Codegenerierungstests ist die Erweiterung der Testsuite wenig aufwändig (siehe Abschnitt 5.2.1. Allerdings müssen Abweichungen in den ADP-Dialekten beachtet werden. Beispielsweise enthalten die ADP-Programme, die der ADP-Compiler übersetzt, zwingend Compiler-Direktiven im nicht-literate-Teil des Programms, die kein gültiges Haskell sind. Ebenso können einige häufig in Pretty-Print-Algebren verwendeten Haskell-Konstruktionen nicht unmodifiziert kompiliert werden. Neben den in Abschnitt 5.2.1 beschriebenen Testfällen sind weitere Unittests sinnvoll, die eine kleinere Einteilung vornehmen,und einzelne zentrale Funktionen der einzelnen Haskell- und C-Module des ADP-Compilers testen, um bestimmte Fehler im Fehlerfall noch einfacher eingrenzen zu können.

Literaturverzeichnis

[1] Gnu compiler collection. Verfügbar in: http://gcc.gnu.org/.

[2] Java annotations. Verfügbar in: http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.

[3] Junit. Verfügbar in: http://junit.sourceforge.net/.

[4] Openjdk. Verfügbar in: http://openjdk.java.net/.

[5] Ruby language home page. Verfügbar in: http://www.ruby-lang.org/en/.

[6] Gnuplot, 2007. Verfügbar in: http://www.gnuplot.info/.

[7] Jakarta commons cli library 1.1, 2007. Verfügbar in: http://jakarta.apache.org/commons/cli/.

[8] Brian Alliet und Adam Megacz. Complete translation of unsafe native code to safe bytecode. In *Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators*, 2004. Verfügbar in: http://www.megacz.com/research/papers/nestedvm.ivme04.pdf.

[9] Kent Beck. Simple smalltalk testing: With patterns. *Smalltalk Report*, 10 1994. Verfügbar in: http://www.xprogramming.com/testfram.htm.

[10] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[11] Johan Bengtsson. Memtime 1.3, 2002. Verfügbar in: http://freshmeat.net/projects/memtime.

[12] Grady Booch, James Rumbaugh und Ivar Jacobson. *Unified Modeling Language User Guide, The*. Addison-Wesley Object Technology Series. Addison-Wesley Professional, 2nd edition, 2005.

[13] Manuel M. T. Chakravarty. *The Haskell 98 Foreign Function Interface 1.0: An Addendum to the Haskell 98 Report*, 2003. Verfügbar in: http://www.cse.unsw.edu.au/~chak/haskell/ffi/ffi.pdf.

[14] Stuart Dreyfus. Richard bellman on the birth of dynamic programming. *Operations Research*, 50(1):48–51, 2002. Verfügbar in: http://www.eng.tau.ac.il/~ami/cd/or50/1526-5463-2002-50-01-0048.pdf.

[15] Sharon Zakhour et. al. *The Java Tutorial*, chapter The Reflection API. Prentice Hall, 2006. Verfügbar in: `http://java.sun.com/docs/books/tutorial/reflect/`.

[16] R. Giegerich, C. Meyer und P. Steffen. *Towards a dicipline of dynamic programming*, volume P-19 of *GI Edition - Lecture Notes in Informatics*, pages 3–44. Bonner Köllen Verlag, 2002. Verfügbar in: `http://bibiserv.techfak.uni-bielefeld.de/adp/ps/adp_discipline.pdf`.

[17] Robert Giegerich und Peter Steffen. Challenges in the compilation of a domain specific language for dynamic programming. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1603–1609, New York, NY, USA, 2006. ACM Press. Verfügbar in: `http://www.techfak.uni-bielefeld.de/~psteffen/pub/exbedding.pdf`.

[18] Robert Giegerich, Björn Voß und Marc Rehmsmeier. Abstract shapes of rna. *Nucleic Acids Research*, 32(16):4843–4851, 2004. Verfügbar in: `http://nar.oxfordjournals.org/cgi/reprint/32/16/4843.pdf`.

[19] James Gosling, Bill Joy, Guy Steele und Gilad Bracha. *The Java(TM) Language Specification*. Prentice Hall PTR, 3rd edition, 2005. Verfügbar in: `http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf`.

[20] I. Hofacker, W. Fontana, P. Stadler, L. Bonhoeffer L, M. Tacker und P. Schuster. Fast folding and comparison of rna secondary structures. *Monatshefte Chemie*, (125):167–188, 1994.

[21] Alexander Kaiser, Jan Krüger und Dirk J. Evers. Rna movies 2: sequential animation of rna secondary structures. *Nucleic Acids Research*, 2007. Verfügbar in: `http://nar.oxfordjournals.org/cgi/reprint/gkm309v1.pdf`.

[22] Razya Ladelsky. Matrix flattening and transposing in gcc. In *2006 GCC Summit Proceedings*, 2006. Verfügbar in: `http://www.gccsummit.org/2006/2006-GCC-Summit-Proceedings.pdf`.

[23] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966. Verfügbar in: `http://www.cs.utah.edu/~eeide/compilers/old/papers/p157-landin.pdf`.

[24] Christian Lang und Georg Sauthoff. *Table Design in Dynamic Programming*. Projektarbeit, Universität Bielefeld, 2005.

[25] Python Software Foundation. *unittest - Unit testing framework*, 2006. Verfügbar in: `http://www.python.org/doc/2.5/lib/module-unittest.html`.

[26] Janina Reeder. *Locomotif - a Graphical Programming System for RNA Motif Search*. Dissertation, Universität Bielefeld, 2006. Verfügbar in: `http://bieson.ub.uni-bielefeld.de/volltexte/2007/1064/pdf/thesis.pdf`.

[27] Janina Reeder und Robert Giegerich. A graphical programming system for molecular motif search. In *Proceedings of the 5th international Conference on Generative Programming and Component Engineering*, pages 131–140, Portland, Oregon, USA, October 22 - 26 2006. ACM Press, New York, NY. GPCE'06.

[28] Jens Reeder und Robert Giegerich. Design, implementation and evaluation of a practical pseudoknot folding algorithm based on thermodynamics. *BMC Bioinformatics*, 5(104), 2004. Verfügbar in: `http://www.biomedcentral.com/content/pdf/1471-2105-5-104.pdf`.

[29] Aaron M. Renn. Gnu getopt - java port 1.0.13, 2006. Verfügbar in: `http://www.urbanophile.com/arenn/hacking/download.html`.

[30] Stefanie Schirmer. *A Frontend for the ADP compiler*. Diplomarbeit, Universität Bielefeld, 2006.

[31] Peter Steffen und Robert Giegerich. Versatile and declarative dynamic programming using pair algebras. *BMC Bioinformatics*, 6:224, 2005. Verfügbar in: `http://www.biomedcentral.com/content/pdf/1471-2105-6-224.pdf`.

[32] Peter Steffern. *Compiling a Domain Specific Language for Dynamic Programming.* Dissertation, Universität Bielefeld, 2006. Verfügbar in: `http://www.techfak.uni-bielefeld.de/~psteffen/pub/diss.pdf`.

[33] Sun Microsystems. *Java Native Interface Specification*, 2003. Verfügbar in: `http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html`.

[34] Guido van Rossum. *Python Language Reference Manual*. Python Software Foundation, 2006. Verfügbar in: `http://www.network-theory.co.uk/docs/pylang/`.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig bearbeitet und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Ort, Datum	Unterschrift