Live-Musikprogrammierung in Haskell

Henning Thielemann

Institut für Informatik
Martin-Luther-Universität Halle-Wittenberg
Von-Seckendorff-Platz 1
06122 Halle
henning.thielemann@informatik.uni-halle.de

Abstract: Ziel unserer Arbeit ist es, algorithmische Musik interaktiv und mit mehreren Teilnehmern zu komponieren. Dazu entwickeln wir einen Interpreter für eine Teilsprache der nicht-strikten funktionalen Programmiersprache Haskell 98, der es erlaubt, das Programm noch während seiner Ausführung zu ändern. Unser System eignet sich sowohl für den Live-Einsatz zur Musikprogrammierung als auch als Demonstrations- und Lernumgebung für funktionale Programmierung.

1 Einführung

Unser Ziel ist es, Musik durch Algorithmen zu beschreiben. Wir wollen Musik nicht wie auf dem Notenblatt als mehr oder weniger zusammenhanglose Folge von Noten darstellen, sondern wir wollen Strukturen ausdrücken. Beispielsweise wollen wir nicht die einzelnen Noten einer Begleitung aufschreiben, sondern die Begleitung durch ein allgemeines Muster und die Folge von Harmonien ausdrücken. Als weiteres Beispiel mag ein Komponist dienen, der eine Folge von zufälligen Noten verwenden möchte. Er möchte die Noten aber nicht einzeln aufschreiben, sondern die Idee ausdrücken, dass es eine zufällige Folge von Noten ist. Dem Interpreten wäre es damit freigestellt, eine andere aber ebenso zufällige Folge von Noten zu spielen.

Der Programmierer soll den Grad der Strukturierung frei wählen können. Beispielsweise soll es möglich sein, „von Hand“ eine Melodie zu komponieren, diese mit einem Tonmuster zu begleiten, für das lediglich eine Folge von Harmonien vorgegeben wird, und das ganze mit einem vollständig automatisch berechneten Rhythmus zu unterlegen.

Mit dem bewussten Abstrahieren von der tatsächlichen Musik wird es aber schwierig, beim Programmieren das Ergebnis der Komposition abzuschätzen. Bei Musik, die nicht streng nach Takten und Stimmen organisiert ist, fällt es schwerer, einen bestimmten Zeitabschnitt oder eine bestimmte Auswahl an Stimmen zur Probe anzuhören. Auch der klassische Zyklus von „Programm editieren, Programm prüfen und übersetzen, Programm neu starten“ steht dem kreativen Ausprobieren entgegen. Selbst wenn Übersetzung und Neustart sehr schnell abgeschlossen sind, so muss doch das Musik erzeugende Programm und damit die

Musik abgebrochen und neu begonnen werden. Insbesondere beim gemeinsamen Spiel mit anderen Musikern ist das nicht akzeptabel.

In unserem Ansatz verwenden wir zur Musikprogrammierung eine rein funktionale Programmiersprache mit Bedarfsauswertung [Hug89], die nahezu eine Teilsprache von Haskell 98 [PJ+98] ist. Unsere Beiträge zur interaktiven Musikprogrammierung sind Konzepte und ein lauffähiges System, welches folgendes bieten:

- Algorithmische Musikkomposition, bei der das Programm verändert werden kann, während die Musik läuft (Abschnitt 2.1),
- gleichzeitige Arbeit mehrerer Programmierer an einem Musikstück unter Anleitung eines „Dirigenten“ (Abschnitt 2.2).

2 Funktionale Live-Programmierung

2.1 Live-Coding

Wir wollen Musik ausgeben als eine Liste von MIDI-Ereignissen [MMA96], also Ereignissen der Art „Klaviaturtaste gedrückt“, „Taste losgelassen“, „Instrument gewechselt“, „Klangregler verändert“ und Warte-Anweisungen. Ein Ton mit Tonhöhe C-5, einer Dauer von 100 Millisekunden und einer normalen Intensität soll geschrieben werden als:

```
main =
  [ Event (On c5 normalVelocity)
  , Wait 100
  , Event (Off c5 normalVelocity)
  ] ;

c5 = 60 ;
normalVelocity = 64 ;
.
```

Mit der Listenverkettung „++“ lässt sich damit bereits eine einfache Melodie beschreiben.

```
main =
   note qn c ++ note qn d ++ note qn e ++ note qn f ++
   note hn g ++ note hn g ;

note duration pitch =
  [ Event (On pitch normalVelocity)
  , Wait duration
  , Event (Off pitch normalVelocity)
  ] ;
```

```
qn = 200 ;  -- quarter note - Viertelnote
hn = 2*qn ; -- half note    - halbe Note

c = 60 ;
d = 62 ;
e = 64 ;
f = 65 ;
g = 67 ;
normalVelocity = 64 ;
```

Diese Melodie lässt sich endlos wiederholen, indem wir am Ende der Melodie wieder mit dem Anfang fortsetzen.

```
main =
   note qn c ++ note qn d ++ note qn e ++ note qn f ++
   note hn g ++ note hn g ++ main ;
```

Die so definierte Liste `main` ist unendlich lang, lässt sich aber mit der Bedarfsauswertung schrittweise berechnen und an einen MIDI-Synthesizer senden. Dank der Bedarfsauswertung kann man die Musik als reine Liste von Ereignissen beschreiben. Das Programm muss und kann selbst keine Ausgabebefehle ausführen. Der Versand der MIDI-Kommandos wird vom Interpreter übernommen.

In einem herkömmlichen interaktiven Interpreter[1] wie dem `GHCi` würde man die Musik etwa so wiedergeben:

```
Prelude> playMidi main        .
```

Will man die Melodie ändern, müsste man die Musik beenden und die neue Melodie von vorne beginnen. Wir wollen aber die Melodie ändern, während die alte Melodie weiterläuft, und dann die alte Melodie nahtlos in die neue übergehen lassen. Mit anderen Worten: Der aktuelle Zustand des Interpreters setzt sich zusammen aus dem Programm und dem Zustand der Ausführung. Wir wollen das Programm austauschen, aber den Zustand der Ausführung beibehalten. Das bedeutet, dass der Zustand in einer Form gespeichert sein muss, der auch nach Austausch des Programms einen Sinn ergibt.

Wir lösen dieses Problem wie folgt: Der Interpreter betrachtet das Programm als Menge von Termersetzungsregeln und die Ausführung des Programms besteht darin, die Ersetzungsregeln wiederholt anzuwenden, solange bis der Startterm `main` so weit reduziert ist, dass die Wurzel des Operatorbaums ein Terminalsymbol (hier: ein Datenkonstruktor) ist. Für die musikalische Verarbeitung testet der Interpreter weiterhin, ob die Wurzel ein Listenkonstruktor ist und falls es eine nichtleere Liste ist, reduziert er das führende Listenelement vollständig und prüft, ob es ein MIDI-Ereignis darstellt. Ausführungszustand des Interpreters ist der reduzierte Ausdruck. Während der Interpreter die vorletzte Note in der Schleife des obigen Programms wiedergibt, wäre dies beispielsweise:

[1] Der Interpreter wäre hier im wahrsten Sinne des Wortes der musikalische Interpret

```
Wait 200 :
    (Event (Off g normalVelocity) : (note hn g ++ main))
.
```

Der Ausdruck wird immer so wenig wie möglich reduziert, gerade so weit, dass das nächste MIDI-Ereignis bestimmt werden kann. Das erlaubt es zum einen, eine unendliche Liste wie `main` zu verarbeiten und zum anderen führt es dazu, dass in dem aktuellen Term wie oben angegeben, noch die Struktur des restlichen Musikstücks zu erkennen ist. Der abschließende Aufruf von `main` ist beispielsweise noch vorhanden. Wenn wir jetzt die Definition von `main` ändern, wird diese veränderte Definition verwendet, sobald `main` reduziert wird. Wir können auf diese Weise die Melodie innerhalb der Wiederholung ändern, beispielsweise so:

```
main =
    note qn c ++ note qn d ++ note qn e ++ note qn f ++
    note qn g ++ note qn e ++ note hn g ++ main ;
.
```

Wir können aber auch folgende Änderung vornehmen

```
main =
    note qn c ++ note qn d ++ note qn e ++ note qn f ++
    note hn g ++ note hn g ++ loopA ;
```

und damit erreichen, dass nach einer weiteren Wiederholung der Melodie die Musik mit einem Abschnitt namens `loopA` fortgesetzt wird.

Wir halten an dieser Stelle fest, dass sich die Bedeutung eines Ausdrucks während des Programmablaufs ändern kann. Damit geben wir eine wichtige Eigenschaft der rein funktionalen Programmierung auf. Wenn wir von Live-Änderungen Gebrauch machen, ist unser System also nicht mehr „referential transparent“. Beispielsweise hätten wir die ursprüngliche Schleife auch mit der Funktion `cycle` implementieren können

```
main =
  cycle
    ( note qn c ++ note qn d ++ note qn e ++ note qn f ++
      note hn g ++ note hn g ) ;
```

und wenn `cycle` definiert ist als

```
cycle xs = xs ++ cycle xs ;
```

dann würde dies reduziert werden zu

```
( note qn c ++ note qn d ++ note qn e ++ note qn f ++
  note hn g ++ note hn g )
```

```
++
cycle
   ( note qn c ++ note qn d ++ note qn e ++ note qn f ++
     note hn g ++ note hn g ) ;
.
```

Die Schleife könnte dann nur noch verlassen werden, wenn man die Definition von `cycle` ändert. Diese Änderung würde aber alle Aufrufe von `cycle` im aktuellen Term gleichermaßen betreffen. Zudem wäre es bei einem strengen Modulsystem ohne Importzyklen unmöglich, im Basis-Modul `List`, in dem `cycle` definiert ist, auf Funktionen im Hauptprogramm zuzugreifen. Dies wäre aber nötig, um die `cycle`-Schleife nicht nur verlassen, sondern auch im Hauptprogramm fortsetzen zu können.

Wir erkennen an diesem Beispiel, dass es vorausschauend besser sein kann, mit einer „von Hand“ programmierten Schleife der Form `main = ... ++ main` eine Sollbruchstelle zu schaffen, an der man später neuen Code einfügen kann.

Neben der seriellen Verkettung von musikalischen Ereignissen benötigen wir noch die parallele Komposition, also die simultane Wiedergabe von Melodien, Rhythmen usw. Auf der Ebene der MIDI-Kommandos bedeutet dies, dass die Kommandos zweier Listen geeignet miteinander verzahnt werden müssen. Wir wollen die Definition der entsprechenden Funktion „`=:=`“ der Vollständigkeit halber hier wiedergeben.

```
(Wait a : xs) =:= (Wait b : ys) =
  mergeWait (a<b) (a-b) a xs b ys ;
(Wait a : xs) =:= (y : ys) =
  y : ((Wait a : xs) =:= ys) ;
(x : xs) =:= ys  =  x : (xs =:= ys) ;
[] =:= ys  =  ys ;

mergeWait _eq 0 a xs _b ys =
  Wait a : (xs =:= ys) ;
mergeWait True d a xs _b ys =
  Wait a : (xs =:= (Wait (negate d) : ys)) ;
mergeWait False d _a xs b ys =
  Wait b : ((Wait d : xs) =:= ys) ;
```

Die grafische Bedienschnittstelle unseres Systems ist in Abbildung 1 zu sehen. Im linken oberen Teil kann der Benutzer den Programmtext eingeben. Mit einer bestimmten Tastenkombination kann er den Programmtext auf Syntaxfehler prüfen und in den Programmspeicher des Interpreters übernehmen. Das vom Interpreter ausgeführte Programm ist im rechten oberen Teil zu sehen. In diesem Teil hebt der Interpreter außerdem hervor, welche Teilausdrücke reduziert werden mussten, um den vorhergehenden in den aktuellen Ausdruck zu überführen. Auf diese Weise kann man den Verlauf der Melodie visuell verfolgen. Der aktuelle Term des Interpreters ist im unteren Teil der Bedienoberfläche dargestellt. Die in der Abbildung wiedergegebenen Texte entsprechen im Wesentlichen unserem

einführenden Beispiel, weisen aber unsere Melodie zusätzlich noch einem MIDI-Kanal zu und verwenden Definitionen von ++ und `map`, welche auf `foldr` aufbauen.

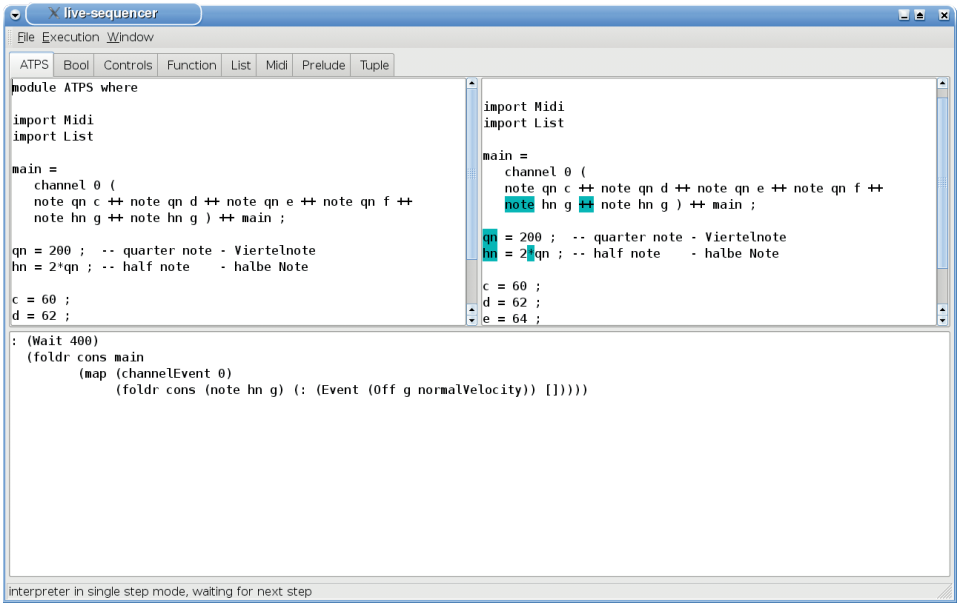

Abbildung 1: *Der Interpreter im Betrieb*

Das System verfügt über die Ausführungsmodi „Echtzeit“, „Zeitlupe“ und „Einzelschritt“. Der Echtzeit-Modus gibt die Musik wieder, so wie es die Notenlängen erfordern, während die anderen beiden Modi die Warte-Ereignisse ignorieren und stattdessen nach jedem Element der MIDI-Kommandoliste eine Pause machen. Die beiden letzten Modi sind zur Beobachtung der Ausführung und zur Fehlersuche gedacht. Sie eignen sich auch für den Einsatz in der Lehre, zur Erläuterung, wie ein Interpreter einer funktionalen Sprache mit Bedarfsauswertung im Prinzip funktioniert.

Der Interpreter ist in Haskell mit dem Glasgow Haskell Compiler GHC [PJ+12] implementiert und verwendet WxWidgets [SRZ+11] für die grafische Bedienoberfläche. Die verarbeitete Sprache unterstützt „Pattern matching“, vordefinierte Infix-Operatoren, Funktionen höherer Ordnung, unvollständige Funktionsanwendung. Aus Gründen der einfachen Implementierung gibt es bislang folgende Einschränkungen: Die interpretierte Sprache ist dynamisch typisiert, und kennt als Objekte ganze Zahlen, Texte und Konstruktoren. Sie ist formatfrei, weswegen nach Deklarationen stets Semikola gesetzt werden müssen. Viele syntaktische Besonderheiten werden nicht unterstützt, beispielsweise „List Comprehension“, „Operator Section“, Do-Notation, Let- und Case-Notation, frei definierbare Infix-Operatoren. Ein- und Ausgabeoperationen sind ebenfalls nicht verfügbar.

2.2 Verteilte Programmierung

Unser System soll es auch erlauben, das Publikum in eine Aufführung oder Studenten in die Vorlesung durch Programmieren einzubeziehen. Die typische Situation dafür ist, dass der Vortragende die Bedienoberfläche des Programms an die Wand projiziert, die Zuhörer die erzeugte Musik über eine Musikanlage hören und dass die Zuhörer über ein Funknetz und einen Browser Kontakt mit dem Vortragsrechner aufnehmen können.

Die implementierte funktionale Sprache verfügt über ein einfaches Modulsystem. Der Vortragende kann auf diese Weise ein Musikstück in mehrere Abschnitte oder Tonspuren zerlegen, und jeden dieser Teile in einem Modul ablegen. Die Module wiederum kann er Zuhörern zuweisen. Außerdem kann der Vortragende durch Einfügen eines bestimmten Kommentars festlegen, ab welcher Zeile der Zuhörer den Modulinhalt verändern darf. In den Zeilen davor stehen üblicherweise der Modulname, die Liste der exportierten Bezeichner, die Liste der importierten Module und grundlegende Definitionen. Auf diese Weise kann der Vortragende eine Schnittstelle für jedes Modul vorgeben.

Ein Zuhörer kann nun über einen WWW-Browser ein Modul abrufen und den veränderbaren Teil editieren. (Siehe Abbildung 2) Nach dem Editieren kann er den veränderten Inhalt an den Server schicken. Dieser ersetzt im Editor den Modultext unterhalb des Markierungskommentars mit dem neuen Inhalt. Dann wird der Text syntaktisch geprüft und im Erfolgsfall an den Interpreter weitergeleitet. Ist der Text syntaktisch nicht korrekt, so bleibt er im Editor, damit er notfalls vom Vortragenden überprüft und korrigiert werden kann.

Im Allgemeinen wird es nicht gelingen, ohne Vorbereitung auf diese Weise ein völlig neues Musikstück zu erschaffen. Der Vortragende kann aber seine Aufführung vorbereiten, indem er sich eine Aufteilung in Module überlegt und diese mit grundlegenden Definitionen füllt. Dies können Definitionen von Funktionen sein, die eine Liste von Nullen und Einsen in einen Rhythmus verwandeln, oder eine Liste von Zahlen in ein Akkordmuster oder eine Bassbegleitung. Der Vortragende kann dann durch Vorgabe von Takt und von Harmonien sicher stellen, dass die einzelnen Stücke zusammenpassen. Der Vortragende übernimmt in diesem Szenario nicht mehr die Rolle des Komponisten, sondern eher die Rolle des Dirigenten.

3 Verwandte Arbeiten

Algorithmische Komposition hat inzwischen eine lange Tradition. Als Beispiele seien hier nur Mozarts musikalische Würfelspiele und die Illiac-Suite [HI59] genannt. Auch gibt es mit Haskore [HMGW96] seit einiger Zeit die Möglichkeit, Musik in Haskell zu programmieren und damit verschiedene Klangerzeuger über MIDI zu steuern oder mit CSound, SuperCollider oder mit in Haskell geschriebenen Audiosynthesefunktionen Audiodateien zu erzeugen. Haskore baut ebenfalls auf der Bedarfsauswertung auf und erlaubt die elegante Definition von formal großen oder unendlichen Musikstücken bei geringem tatsächlichem Speicherverbrauch bei der Interpretation. Das kreative Komponieren wird allerdings da-

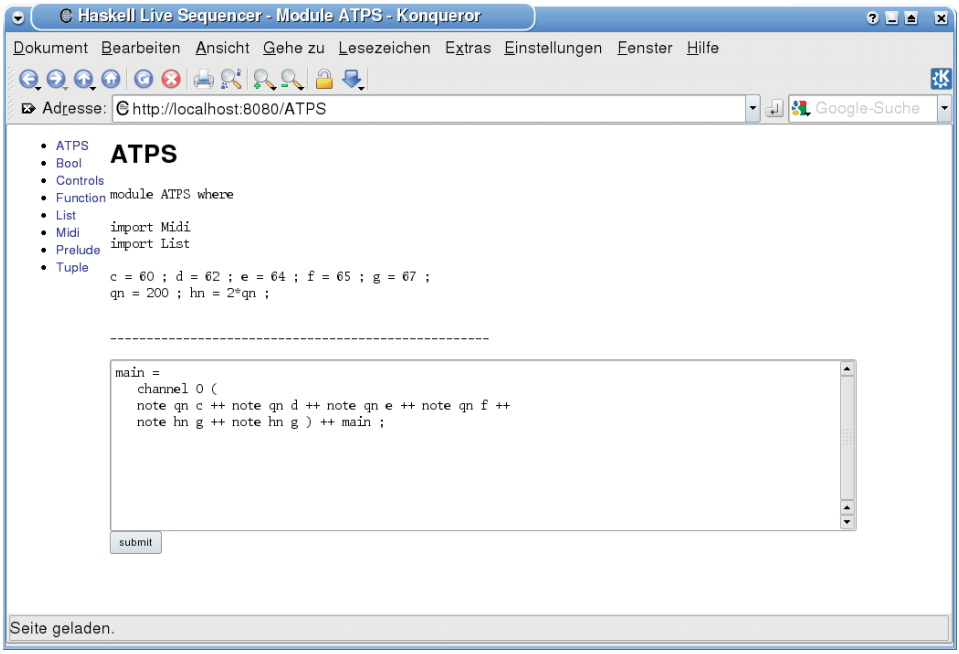

Abbildung 2: *Zugriff auf ein Modul über das Netz*

durch erschwert, dass man Änderungen erst nach Abbruch und Neustart des Programms hören kann.

Ein sehr populärer Ansatz zur Programmierung von Animationen, Robotersteuerungen, grafischen Bedienschnittstellen und Audiosignalverarbeitung in Haskell ist die funktionale reaktive Programmierung [EH97]. Wie für unsere Musikstücke wird auch hier der zeitliche Verlauf einer Animation, einer graphischen Bedienoberfläche oder ähnlichem durch eine Art unendliche Liste beschrieben. Darüber hinaus soll ein FRP-Programm auch auf äußere Ereignisse, wie zum Beispiel Bewegungen der Computermaus reagieren können. Im Gegensatz zu unserer Arbeit ist bisher allerdings nicht möglich, ein FRP-Programm während seiner Ausführung zu ändern.

Eine funktionale, aber nicht rein funktionale, Programmiersprache, die Änderungen des Programmtextes während der Programmausführung erlaubt, ist Erlang [Arm97]. Erlang folgt der strengen Semantik. Man könnte in Erlang eine Folge von MIDI-Kommandos nicht durch eine (lazy) Liste von Konstruktoren beschreiben, sondern bräuchte Iteratoren oder ähnliches. In ein laufendes Erlang-Programm kann neuer Programmcode auf zwei verschiedene Weisen eingebracht werden, entweder in dem das laufende Programm Funktionen (zum Beispiel Lambda-Ausdrücke) aufruft, die ihm als Nachrichten zugesandt werden oder indem ein Erlang-Modul durch ein neues Modul ersetzt wird. Wird ein Erlang-Modul nachgeladen, so behält das Laufzeitsystem die alte Version des Moduls im Speicher, um laufende Programmteile ausführen zu können. Lediglich modul-externe Aufrufe

springen in das neue Modul, wobei man einen externen Aufruf auch innerhalb desselben Moduls starten kann. Auch in Erlang müssen also Sollbruchstellen (externe Aufrufe oder Funktionsübernahme aus Nachrichten) zum späteren Einfügen von neuem Code geschaffen werden.

Unser Ansatz zur Änderung von Programmtext während der Programmausführung ist damit sehr ähnlich zum „Hot Code loading“ in Erlang. Die Bedarfsauswertung in unserem Interpreter führt jedoch dazu, dass beträchtliche Teile des Programmtextes im aktuellen Term enthalten sind. Diese werden von Änderungen an einem Modul nicht unmittelbar betroffen. Auf diese Weise ist es in unserem Ansatz nicht unbedingt nötig, zwei Versionen eines Moduls im Speicher zu halten, um einen glatten Übergang von altem zu neuem Programmtext zu erreichen.

Sogenanntes Musik-Live-Coding, also das Programmieren eines musikerzeugenden Programms, während die Musik läuft, war bislang Spezialsprachen wie ChucK [WC04] und SuperCollider/SCLang [McC96] und ihren Implementierungen vorbehalten. Diese Sprachen sind beim Kontrollfluss an das imperative Programmierparadigma angelehnt und beim Typsystem an die objektorientierte Programmierung. Im wesentlichen funktionieren beide wie eine Client-Server-Lösung, wobei der Server die Klänge erzeugt und parallel zu einem Kommandozeileninterpreter läuft, von dem aus man Befehle an den Server schicken kann.

Auch in unserer Architektur läuft die Klangerzeugung parallel zur eigentlichen Programmierung und wird mit (MIDI-)Kommandos gesteuert. Jedoch wird in unserem Ansatz nicht programmiert, wie sich die Klangerzeugung ändern soll, sondern das erzeugende Programm wird direkt geändert.

4 Folgerungen und zukünftige Arbeiten

Unsere vorgestellte Technik zeigt einen neuen Weg der Live-Programmierung von Musik auf, der sich möglicherweise auch auf die Wartung anderer lange laufender funktionaler Programme übertragen lässt. Dennoch zeigt sich, dass man schon beim Programmieren einer ersten Version gewisse Sollbruchstellen vorsehen muss, an denen man später im laufenden Programm Änderungen einfügen kann. Auch mit automatischen Optimierungen des Programms müssen wir jetzt vorsichtig sein, denn eine Optimierung könnte eine solche Sollbruchstelle entfernen. Wenn ein Programm zur Laufzeit geändert wird, so sind Funktionen eben nicht mehr „referential transparent“, womit wir eine wichtige Eigenschaft der funktionalen Programmierung aufgeben.

Typsystem Um die Gefahr zu verringern, dass ein Musikprogramm nach einer Änderung wegen eines Programmfehlers abbricht, ist der nahe liegende nächste Schritt der Einsatz eines statischen Typprüfers. Dieser müsste nicht nur testen, ob das vollständige Programm nach Austausch eines Moduls noch typkorrekt ist, sondern er müsste zudem testen, dass der aktuelle Term im Interpreter bezüglich des neuen Programms typkorrekt ist.

Noch wichtiger wird ein Typprüfer im Mehrbenutzerbetrieb. Der Leiter einer Programmierveranstaltung mit mehreren Programmierern könnte jedem Teilnehmer Typsignaturen im nicht editierbaren Bereich seines Moduls vorgeben, die der Teilnehmer implementieren muss. Der Typprüfer würde dafür sorgen, dass Teilnehmer nur Änderungen einschicken können, die zum Rest des Musikstücks passen.

Auswertungsstrategie Derzeit ist unser Interpreter sehr einfach gehalten. Der aktuelle Term ist ein reiner Baum. In dieser Darstellung können wir nicht ausdrücken, dass der Wert eines Terms mehrmals verwendet wird. (Es gibt also kein „sharing“.) Wenn beispielsweise `f` definiert ist als `f x = x:x:[]`, dann wird der Aufruf `f (2+3)` reduziert zu `(2+3) : (2+3) : []`. Wenn weiterhin das erste Listenelement zu `5` reduziert wird, wird das zweite Listenelement nicht reduziert. Wir erhalten also `5 : (2+3) : []` und nicht `5 : 5 : []`. Da der Term nur ein Baum ist und kein Graph, brauchen wir keine eigene Speicherverwaltung, sondern können uns auf die automatische Speicherverwaltung des GHC-Laufzeitsystems verlassen, in dem der Interpreter läuft. Wenn ein Teilterm nicht mehr benötigt wird, so wird er aus dem Operatorbaum entfernt und früher oder später vom Laufzeitsystem des GHC freigegeben.

Selbst einfache korekursive Definitionen wie die der Fibonacci-Zahlen durch

```
fix (\fibs -> 0 : 1 : zipWith (+) fibs (tail fibs))
```

führen bei diesem Auswertungsverfahren zu einem unbegrenzten Anstieg der Termgröße. In Zukunft sollen daher weitere Auswertungsstrategien wie zum Beispiel die Graphreduktion mit der STG-Maschine [PJ92] hinzukommen, die dieses und weitere Probleme lösen. Anstelle eines Operatorbaums würde der aktuelle Term dann aus einem Operatorgraph bestehen, die Anwendung der Funktionsdefinitionen und damit die Möglichkeit der Live-Änderung einer Definition bliebe prinzipiell erhalten. Die Gefahr bei Live-Musikprogrammierung liegt natürlich darin, dass Programmänderungen abhängig von der Auswertungsstrategie verschiedene Auswirkungen auf den Programmablauf haben können. Das Verwenden des gleichen Objektes im Speicher an verschiedenen Stellen im aktuellen Term („sharing“) würde zwar im obigen Beispiel der Fibonacci-Zahlen den Speicherverbrauch begrenzen, könnte aber auch verhindern, dass eine geänderte Definition der aufgerufenen Funktionen noch berücksichtigt wird. Der Einzelschrittmodus würde es ermöglichen, in der Lehre verschiedene Auswertungsverfahren zu demonstrieren und Vor- und Nachteile miteinander zu vergleichen.

Offen ist, ob und wie wir unser System, das Änderungen des Programms während des Programmablaufs zulässt, direkt in eine existierende Sprache wie Haskell einbetten können. Dies würde es uns vereinfachen, die Wechselwirkung zwischen Programmänderungen, Optimierungen und Auswertungsstrategien zu untersuchen.

Hervorhebungen Es gibt noch ein weiteres interessantes offenes Problem: Wie kann man Textstellen im Programm passend zur erzeugten Musik hervorheben? Es liegt nahe, die jeweils gespielten Noten hervorzuheben. Dies wird zur Zeit dadurch erreicht, dass in

einer Wartephase alle die Symbole hervorgehoben werden, welche seit der letzten Wartephase durch den Interpreter reduziert wurden. Wenn aber eine langsame Melodie parallel zu einer schnellen Folge von Regleränderungen abgespielt wird, so führt das dazu, dass die Noten der Melodie nur kurz hervorgehoben werden, nämlich immer nur für die kurze Zeit, in der der Reglerwert konstant bleibt. Wir würden aber erwarten, dass die Hervorhebung eines Musikteils nicht von parallel laufenden Teilen beeinflusst wird. Formal könnten wir es so ausdrücken: Gegeben seien die serielle Komposition `++` und die parallele Komposition `=:=`, die sowohl für Terme als auch für Hervorhebungen definiert sein sollen. Gegeben sei weiterhin die Abbildung `highlight`, welche einen Term seiner Visualisierung zuordnet. Dann soll für zwei beliebige Musikobjekte `a` und `b` gelten:

```
highlight (a ++  b) = highlight a ++  highlight b
highlight (a =:= b) = highlight a =:= highlight b
```

Wenn man alle Symbole hervorhebt, die mittelbar an der Erzeugung eines `NoteOn`- oder `NoteOff`-MIDI-Kommandos beteiligt waren, dann erhält man eine Funktion `highlight` mit diesen Eigenschaften. Allerdings führt sie dazu, dass die Notenaufrufe kumulativ hervorgehoben werden. In

```
note qn c ++ note qn d ++ note qn e ++ note qn f
```

werden bei Wiedergabe von `note qn e` auch `note qn c` und `note qn d` hervorgehoben, denn diese erzeugen Listen und dass diese Listen endlich sind, ist ein Grund dafür, dass aktuell `note qn e` wiedergegeben werden kann. Die Aufrufe `note qn c` und `note qn d` sind also notwendigerweise daran beteiligt, dass `note qn e` reduziert werden kann.

Zeitsteuerung Eine weitere Schwierigkeit besteht im zeitlich präzisen Versand der MIDI-Kommandos. Bislang wartet der Interpreter bis zu dem Zeitpunkt, an dem eine MIDI-Nachricht verschickt werden soll, und beginnt dann erst mit der Berechnung des entsprechenden Listenelements. Wir vertrauen also darauf, dass die Berechnung schnell genug beendet wird und sich der Versand nicht allzu stark verzögert. Bei komplizierteren Berechnungen trifft diese Annahme natürlich nicht zu. Eine höhere Präzision könnten wir erreichen, indem wir die MIDI-Nachrichten mit einem Zeitstempel versehen und einige Zeit im Voraus verschicken. Das wirft neue Probleme der Synchronisation von Musik und grafischer Darstellung des Interpreterzustandes auf und es würde auch heißen, dass die Musik erst verzögert angehalten werden kann und man nur verzögert in einen anderen Ausführungsmodus wechseln kann.

5 Danksagungen

Dieses Projekt basiert auf einer Idee von Johannes Waldmann, die ich gemeinsam mit ihm entwickelt und in einen Prototypen umgesetzt habe. Ich möchte mich bei ihm, Heinrich

Apfelmus und den anonymen Gutachtern für das gründliche Lesen und für zahlreiche Verbesserungsvorschläge zu diesem Artikel bedanken.

Informationen über dieses Projekt zu Programmentwicklung, Demonstrationsvideos und Artikeln können Sie unter

```
http://www.haskell.org/haskellwiki/Live-Sequencer
```

abrufen.

Literatur

[Arm97] Joe Armstrong. The development of Erlang. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP 1997, Seiten 196–203, New York, NY, USA, 1997. ACM.

[EH97] Conal Elliott und Paul Hudak. Functional reactive animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, Jgg. 32, Seiten 263–273, August 1997.

[HI59] Lejaren A. Hiller und Leonard M. Isaacson. *Experimental Music: Composition With an Electronic Computer*. McGraw-Hill, New York, 1959.

[HMGW96] Paul Hudak, T. Makucevich, S. Gadde und B. Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3), June 1996.

[Hug89] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[McC96] James McCartney. Super Collider. `http://www.audiosynth.com/`, March 1996.

[MMA96] MMA. Midi 1.0 detailed specification: Document version 4.1.1. `http://www.midi.org/about-midi/specinfo.shtml`, February 1996.

[PJ92] Simon Peyton Jones. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[PJ+98] Simon Peyton Jones et al. Haskell 98 Language and Libraries, The Revised Report. `http://www.haskell.org/definition/`, 1998.

[PJ+12] Simon Peyton Jones et al. GHC: The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`, 2012.

[SRZ+11] Julian Smart, Robert Roebling, Vadim Zeitlin, Robin Dunn et al. wxWidgets 2.8.12. `http://docs.wxwidgets.org/stable/`, March 2011.

[WC04] Ge Wang und Perry Cook. ChucK: a programming language for on-the-fly, real-time audio synthesis and multimedia. In *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, Seiten 812–815, New York, NY, USA, 2004. ACM.