Tinte: developing a prototype for typesetting music in Clean – a case study

Sven Eric Panitz
Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt
Germany
e-mail: panitz@informatik.uni-frankfurt.de

Abstract

This paper describes the development of a typesetting program for music in the lazy functional programming language Clean. The system transforms a description of the music to be typeset in a dvi-file just like TEX does with mathematical formulae.

The implementation makes heavy use of higher order functions. It has been implemented in just a few weeks and is able to typeset quite impressive examples. The system is easy to maintain and can be extended to typeset arbitrary complicated musical constructs.

The paper can be considered as a status report of the implementation as well as a reference manual for the resulting system.

1 Introduction

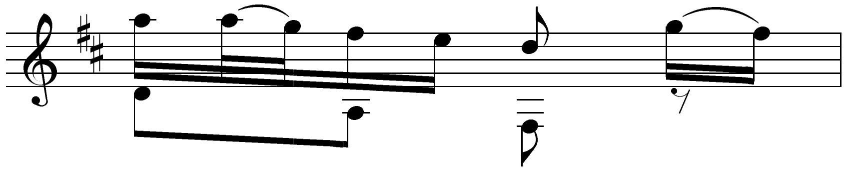
1.1 Musical Typesetting

As chemical or mathematical notation musical notation is a two-dimensional representation of information. The horizontal position represents a point of time relatively to other points, i.e. notes which are closer to the left are to be played before notes further to the right. The vertical position represents the pitch at which a note is to be played. Thus typesetting music seems to be a quite easy task: place certain graphical elements, such as semibreves, minums, crotchets, quavers, semiquavers, demisemiquavers, hemidemisemiquavers and corresponding rests on a two-dimensional area. Elements which are to be played at the same point of time are on the same vertical line. They form a chord. It seems that all that is needed is a description which can express which musical elements are to be played (set) at the same time (place) and which are to be played (set) in a sequence, where each musical element has a pitch i.e. a vertical position.

Unfortunately there is more to the story. First of all the number of musical elements is numerous (consider e.g. things like grace notes etc.) and one cannot expect to have a complete specification of all elements may be required beforehand (consider e.g. weird things like Gregorian note shapes or percussion note shapes etc.).

Furthermore there exist not only simple musical elements but also grouping of elements with beams and slurs as well as ornaments and marks for fingering and dynamical expression signs.

Just to get an impression of some arbitrary musical notation, consider the following bar taken from Divertimento II op. 33 by Antonio Nava:



Last but not least, we demand of a typesetting system not only to place musical elements regarding their sequence and pitch, but also to split the music to be set in lines and stretch the elements of a line in such a way, that a line is completely filled.

As a matter of fact to the authors knowledge all efforts to define a standardized description format for the typesetting of music seemed to have failed.[1] There are of course standardized formats for exchanging music, such as *midi*, but these are unsuitable for expressing how this music is to be typeset. They do not regard things like beams, i.e. they can express the information that there are 4 semiquavers at certain pitches to be played in sequence, but cannot express whether these semiquavers are to be set in a group with one beam or in two groups with a beam each or as something else.

To typeset music on a computer there are generally two different approaches:

- **Drawing**: One possible solution is to use an ordinary drawing program (with some nice macros for typesetting music) and to place musical elements manually at certain points of the two-dimensional area. This solution has the disadvantage, that not any use is made of the musical information. There are only geometrically objects with positions. These objects are in no way connected with some musical semantics. Things like transposing or playing the music are almost impossible, even reformatting is generally not an easy task. As great advantage, one is completely free to typeset any musical construct or even to invent some new notation.

- **One-dimensional description**: The other solution is, to have a one-dimensional description (i.e. a string of characters) of the music to be typeset. This description is then to be transformed into a printable file format, which represents the actual music. This approach has the great

[1] There seem to have been some efforts to define a uniform musical notation standard file format (NIFF) in the 1995, but to our knowledge this project did not come to a successful end.

advantage that the description of the music cannot only be used for printing the music, but also for transforming it into a midi-file or for further transformations like transposing. The great disadvantage of this approach is that a description language has to be defined completely beforehand. Things that cannot be expressed with this language cannot be typeset without extending the typesetting system.

Between this two approaches there are a number of possible mixtures. e.g. programs which appear like a drawing program but have a internal representation of the music as a one dimensional description, which can be used for further transformations; or a description language that not only allows to define musical elements but also to place arbitrary graphical elements at certain points.

Most commercial musical typesetting programs supply a graphical user interface and hide the one-dimensional description from the user. A wide-spread commercial typesetting program for music is *Finale* distributed by *Coda Music Technology*: `www.codamusic.com`.

Systems that are completely in the second category above are the TEXmacro package for typesetting music called MusicTEX[Tau93] and the system *MUP* [Ark] written in C. Unfortunately the author was ignorant of *MUP* when he started this project.[2].

The system Tinte[3] we are developing here, will completely follow the second approach, i.e. it will transform a given one-dimensional description of the music to be typeset into a printable file. To be most flexible with this approach, the system can easily be extended with new elements.

1.2 Functional Programming

This work is considered as a case study for developing a prototype in a lazy functional language. Lazy functional languages offer a great amount of abstraction and are known to be perfect tools for rapid prototyping [HJ94].

Especially higher order functions offer a great amount of declarativity. Objects can be declared in a module even if not all information which is needed for creating these objects is given yet, by way of functions. This enables a greater flexibility in modularization of the complete system.

To the same effect lazy evaluation can make certain control structures obsolete. Expressions are only evaluated, when their value is really necessary for the result of the main function. This offers an automatic mechanism for backtracking. A function in one module can offer a list of several solutions (e.g. parse-trees), which are tested sequentially by another function in another module for further calculations. Lazy evaluation will ensure that only the solutions are created by the first function which are needed by the consuming function.

Today there are two main lazy functional languages: Haskell [HAB+96] and Clean [Pv97][4].

[2] Thanks to Philip Smith who pointed me to *MUP* after he had read the first version of this report

[3] *Tinte* is an acronym for *tinte is not TEX* as well as the German word for *ink*.

[4] For historical reason one could also add the programming language called after the Shakespear character *Miranda*[Tur85].

We will prefer Clean as implementation language for several mundane reasons:

- there is no Haskell compiler for my Power Mac at home nor for my Apple Power-Book on the road. And furthermore Clean runs on the workstation at work.
- the system independent I/O-library of Clean may be required for extending the resulting system with a graphical interface.
- potential users of the system are likely to have some hardware and operating system where Clean is available for.
- the fast compilation times of clean!

Nevertheless this shall not be considered as an evaluation Haskell vs. Clean. Some smaller features of Haskell as special monad syntax and some predefined classes were definitely missing.

1.3 History of this work

This work results from my private interest in 19th century guitar music. In the first half of the 19th century guitar music reached a peak. Publications for guitar music were numerous: this did not only include solo works but lieder with guitar accompaniment and chamber music. The vast majority of these publications today can only be found in libraries and private collections and are not available in modern editions (just to give a number: some private collections contain about 4000 pieces). The idea is to revive these works and typeset them on a computer in order to offer them to broad public on the Internet. Thus I started to typeset some Sonata with mtex [SS87], a TEX [Knu91] macro package which provides fonts for musical notation and allows to typeset music. The mtex package is the basis for MusicTEX, a quite impressive typesetting system which uses TEX [Tau93]. MusicTEX is able to typeset multi-staff music and enhanced the original mtex in many respects.

Unfortunately, it turned out that guitar music is very demanding for a typesetting program, because it comprises polyphonic music in one single staff. Rather then trying to enhance the TEX macro package, which seems to be a tedious task for someone used to modern functional languages, the idea of implementing an own typesetting system as a case study occured.

2 Specification

2.1 Output format

One demand of the typesetting program is that it is able to print sheet music in a high quality standard on any modern printing device. A suitable format for this demand is the *dvi*-format, which had been defined in 1979 [Fuc80]. As a matter of fact TEX produces dvi-files and there are a lot of tools for printing dvi-files, displaying them and transforming them into some other format like postscript. Furthermore musical fonts are available for dvi-files.

A dvi-file is a sequence of commands which allows to place characters of chosen fonts on the printable area. Some further commands are available to draw rectangles, i.e. horizontal or vertical lines. There are no commands which allow to draw lines with an arbitrary angle of gradient. This has as consequence that beams for groups of notes have to be designed by characters of a font.

The dvi-commands form a little stack machine, where actual positions on the printable area can be pushed onto a stack.

To have some sort of back-end some modules for handling dvi-files were implemented first. The dvi-commands were modeled as an algebraic data-type `DVI` where a dvi-file is a list of dvi-commands packed in a constructor such that instances of type-classes can be made when required:

```
:: DVIl = D [DVI]
```

Useful functions for handling dvi-files were defined, such as reading a dvi-file and displaying its command list on screen, writing its command list in a clean file `*.icl` (for importing the commands again) and of course creating a valid dvi-file out of an object of type `DVIl`.

No special technique was applied to develop these functions. Getting some data sheet about the dvi-format together with implementing the functions took about a day. However, the functions in these modules are rather ad-hoc and very inefficient but they do their job. Some sophisticated library with functions like to write integers of different length (2 byte, 3 byte...) in twos-complement notation would have been rather welcome.

2.2 Input format

In the beginning the specification of the input for the system was quite unclear. The only precisely made demand was that a one-dimensional description of the music to be typeset was to be used as input. On the one hand a rather high-level musical description was desirable, on the other hand the ability to typeset the music exactly like its 19th century original was demanded. The decision had to be made, how much the system is to decide on its own, where to put a certain musical element and how much influence the user has, to determine how certain things are to be set, which in generally is not information about the musical structure. As an example consider the question of where to put the stem of a note. There is a rule how to decide this: for notes up to a certain pitch the stem has to show upwards, for notes above this pitch downwards. However this is just a default rule and in certain situations it is desired not to obey this rule, e.g. if there is some other element above the note which belongs to another part in polyphonic music, then the stem will better go downwards. Maybe the decision of where to put the stem depends on some optical question which may depend on things like slurs and beams, fingering indications and who knows what else.

As can be seen with this example, there does not seem to be a deterministic algorithm of how a certain musical structure is to be typeset. This means that the system cannot be create its output from a description of the musical structure alone and has to be very flexible for a user.

3 The overall structure

Let us shortly consider which subtasks have to be performed and how the system can be split into different modules. Roughly speaking our system divides into the following subtasks:

- handling dvi files, i.e. reading and writing dvi files and providing an algebraic data type for dvi code.
- creating dvi code out of a description.
- formatting, i.e. modules for breaking lines and pages.
- parsing, i.e. as soon as we have settled on a input language read some file of this language and transform it into its internal representation.

There are further functionalities which may someday be wished to be incorporated into the system and which we at least might consider as future modules:

- interpreting dvi code and displaying the result on the screen. This requires to read and decompress pk files.
- transforming the one-dimensional description into a midi file, for proof hearing the typeset music.
- integrating an editor which has functionalities that help typesetting music.
- automatically generating the one dimensional description of music from other formats.

A graphical overview is given in figure 1. The precise interface description cannot yet be made, because we still have not settled on an input language. We will try to come to a decision in the next section.

4 Solutions

As we have seen in the previous section, the specification was quite unclear. In order to get a less opaque view of how the system is to work, we made the decision to implement a tiny system, which is only able to typeset notes and rests and neglects chords, grouping, slurs etc. The development of the tiny prototype system took a few days. It is introduced in the next subsection.

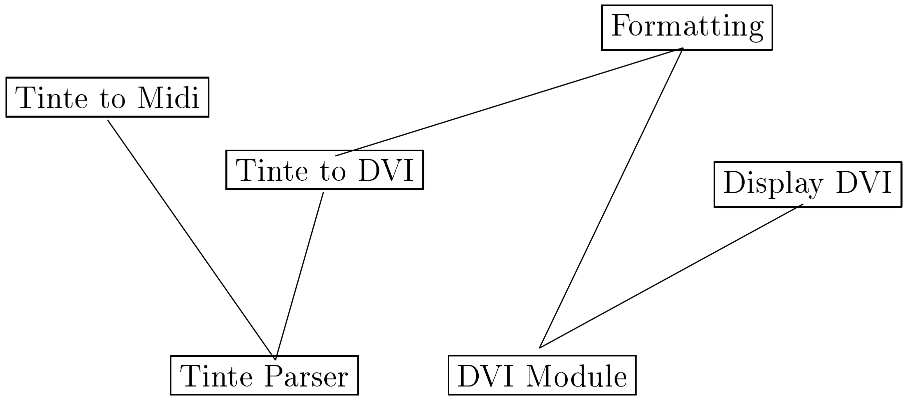


Figure 1: A rough overview of the to be expected structure of the system

4.1 Algebraic data type based solution

Since we decided to develop a tiny system, we are now able to completely specify, what the system is to be able to typeset.

We want to be able to typeset notes. The most natural idea is to define an algebraic data type which contains constructors for all musical elements, which we want to be able to define. So let us start with constructors for notes:

```
::Element
       =  N1  Char Int
         |N2  Char Int
         |N4  Char Int
         |N8  Char Int
         |N16 Char Int
         |N32 Char Int
         |N64 Char Int
```

`(N4 'c' 1)` shall denote a crotchets of note c', `(N8 'a' 2)` a quaver of note a" etc. Further elements are the corresponding rests:

```
         |P1|P2|P4|P8|P16|P32|P64
```

Furthermore we can express sharps, flats and naturals around an element.

```
         |Is Element
         |Es Element
         |Na Element
```

We define a bar as a list of elements[5].

[5]Throughout the system we sometimes use German identifiers. *Takt* is the German word for *bar*.

```
::Takt :== [Element]
```

The main function for the tiny system has to transform a list of bars into a dvi-file:

```
music2dvi :: [Takt] -> DVIl
```

For implementing this main function the problem is not to create the dvi code, which sets a certain element, but the spacing next to an element.

4.1.1 Spacing

After each element we expect a space. The width of this space is dependent on the value of the element. There has to be a wider space next to a quaver than next to a crotchet.[6] Unfortunately this space is not of a fixed length for each element, but may be widened or narrowed a bit, in order to fit several bars exactly on one line (this is what is called glue in TEX). This means that we cannot determine the complete dvi-code for an element until we know how far this element has to be stretched in the context of a complete line. Higher order programming offers a nice and simple solution to this problem: we can define a default length of space for each element. Then we can define a function which produces dvi-code for an element dependent on a numerator and a denumerator, which denote the stretching factor of the space next to the element:

```
::DVIcode :== Int Int -> DVIl

element2dvi :: Element -> (DVIcode,Int)
```

The second tuple argument of the result gives the default length of the element. Now we can produce dvi code for a bar:

```
bar2dvi :: Takt -> (DVIcode,Int)
bar2dvi els = (\z n -> concat (map ((\x-> x z n) o fst) dviels)
              ,sum (map snd dviels))
  where
    dviels = map element2dvi els
```

Now we know how long each bar is by default and can split the bars into lines. This is done in an separate module. Although this effects the dvi code of each element this does not have to worry us, very much. To draw everything together we can now calculate the numerator and denumerator which denote the stretching factor for each line.

```
line2dvil :: [(DVIcode,Int)] Int -> DVIl
line2dvil bars linelength
 = concat (insertbarline
```

[6] Unfortunately the space next to a quaver is not double the length of the space next to a crotchet. This is what makes typesetting polyphonic music so difficult.

```
    (map ((\x-> x linelength defaultlength) o fst) bars))
 where
  defaultlength = sum (map snd bars)
```

As can be seen higher order functions give a straightforward solution to the spacing problem. It provides some good means for splitting the system in modules. Although the line break function effects the dvi code which has to be produced for each element (and this effect can be tremendously as soon as we introduce grouping and slurs in the system, as can be seen in the next section), we could separate these subtasks from each other.

We now need to specify the different alternatives for setting different elements. We give the example of a rest:

```
element2dvi P4
 = (\z n -> odvi (D ([Push,Set_char p4char,Pop
                      ,Right ((p4restlength*z)/n)]))
   ,p4restlength)
```

`p4char` and `p4restlength` are constants, which are set in some global environment. `p4char` is the character of the music font, which denotes the rest symbol. `Push` etc. are constructors of the type `DVI`.

4.1.2 Modifying elements

We have not yet treated the recursive constructors of the type `Element`. It has to be specified, how `element2dvi` is going to behave on an element like `Is(N4 'c' 2)`. This will of course involve some call of `element2dvi` to `N4 'c' 2` and then add dvi-code which puts a sharp in front of c". This would mean that we need rule alternatives like:

```
element2dvi (Is (N1 c i))
 = (\z n ->setsharpfor c i+++l z n,l+sharplength)
 where
   (n1dvi,l) = element2dvi (N1 c i)
```

As can be seen `element2dvi` gets pretty blown up this way. Nevertheless, we get unspecified patterns this way. How is `element2dvi` to behave on expressions like `Is(Na(N4 'c' 2))`, which may denote meaningful musical expression. As can be seen this way we are forced to specify a huge number of different cases and `element2dvi` degenerates into a very large function which only dispatches different alternatives. This will in the large scale get a very hard to maintain system.

The solution we applied to this problem is, to change `element2dvi` such that it not only results a tuple consisting of the dvi-code and the length of the element but as much information as we can gather about the element. In order to have some easy means to extend the implementation all this information is best collected into a record structure:

```
::Elementrecord = {pitch  :: Int
                  ,length :: Int
                  ,dvicode:: DVIcode
                  ,definition :: Element
                  }
```

Now the rules for sharps look like the following:

```
element2dvi el=:(Is e) = {e & dvicode   = newcode
                            ,length     = newlength
                            ,definition = el}
 where
   l = e.length
   n1dvi = e.dvicode
   pi = e.pitch
   newcode z n = setsharpatpitch pi+++l z n
   newlength   = l+sharplength
```

Now we have automatically defined all recursive elements as e.g.
`Is(Is(Na(Es(N32 'd' 0))))`

4.1.3 The first nasty things

Even in our tiny setting of this experimental system we forgot some special cases. Usually an element has only space behind it; but in some special situations it is required to have also some space in front of it. This is for example the case, when there is only a single element in a bar. Then this element is centered in the bar. We did not provide any means neither to express this situation explicitly, or to typeset elements implicitly according to this rule. In an explicit solution, we would introduce a further recursive constructor `Center` to the type `Element`, which denotes that half of the space from behind the element is to be set in front of the element. An implicit solution, i.e. a solution where the system decides alone that an element is to be displayed centered, would require a special rule alternative for `bar2dvi [el]`.

4.1.4 Evaluation of the tiny system

From the tiny experimental system we have drawn some important design decisions: elements get transformed into a record which contains as much information as possible of the element and the context it stands in as we can gather. This is necessary to be flexible and to have all required information for further extensions. The dvi code which is created is dependent on the stretching factor of an element which will be known only after line breaking has been done.

Among the things we have to look for a better solution is the representation of the input language. We were using an algebraic data type and a function which makes a very large pattern matching on this data type. The data type will grow rapidly with further extensions. This makes the implementation rather clumsy.

During the overall development of the system we extended the tiny test system to chords, which involved stemless notes. This is not reported any further in this paper.

4.2 Function type based solution

Instead of defining an input language for the musical notation and representing this language as an algebraic data type, we will represent musical elements as functions. This means that we do not need any more a function which matches patterns on musical elements and calls different functions for the different elements. A musical element is simply this function. This means e.g. instead of the constructor `N4` we supply a function `n4` of type:

```
n4 :: Int Char -> Elementrecord
```

Such that `(element2dvi (N4 i c))`= `n4 i c`.

This makes the system more flexible during its development. New elements can be constructed easily from scratch. The elements which we can adopt from the tiny test system of the last section are:

```
n1  :: Char Int -> Elementrecord
n2  :: Char Int -> Elementrecord
n4  :: Char Int -> Elementrecord
n8  :: Char Int -> Elementrecord
n16 :: Char Int -> Elementrecord
n32 :: Char Int -> Elementrecord
n64 :: Char Int -> Elementrecord
p1  :: Elementrecord
p2  :: Elementrecord
p4  :: Elementrecord
p8  :: Elementrecord
p16 :: Elementrecord
is  :: Elementrecord  -> Elementrecord
es  :: Elementrecord  -> Elementrecord
na  :: Elementrecord  -> Elementrecord
```

As can be seen there are basic element functions which denote an element and functions which modify elements. We still have no functions which can combine two or more elements into one element. This is quite similar to technique used for the pretty printing library be Hughes [Hug95]. As a matter of fact the basic question for typesetting music is the same as for pretty printing. The difference is that in typesetting music we have a lot more parameters to take into consideration.

Functions which allow horizontal and vertical combination of elements will be introduced in the next subsections.

4.2.1 Sequences

The most natural combination of two elements is the sequence. These are elements which are to be typeset next to each other. We introduce a function which can combine two elements into a new element which denotes an element that represents the sequence of two elements:

```
(--) infixl 1 :: Elementrecord  Elementrecord->  Elementrecord
(--) e1 e2 = {e1 &length     = l1+l2
                ,dvicode     = \z n -> (dvi1 z n+++dvi2 z n)}
  where
    l1    = e1.length
    l2    = l2.length
    dvi1  = e1.dvicode
    dvi2  = e2.dvicode
```

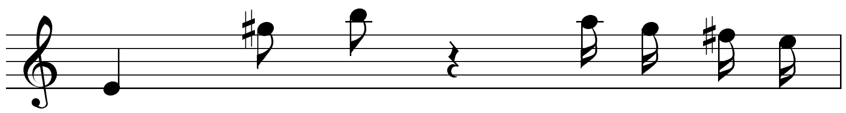
The pitch of the combined element is the pitch of its first sub-element. The pitch is used for modifying functions such as `is`. The default length of the sequence is the sum of the lengths of the parts. The dvi-code is the concatenation of the parts.

We only gave a very stripped down version of the type `Elementrecord` in this presentation. The different modifier functions need much more information about elements. A full description of the type `Elementrecord` can be found in appendix C.

By now we can express a simple sequence of different notes, as e.g.:

```
n4 'e' 1--is(n8 'g' 2)--n8 'h' 2--p4--n16 'a' 2--n16 'g' 2--
is(n16 'f' 2)--n16 'e' 2
```

which will produce the following output:



4.2.2 Chords

Up to now we can only typeset monophonic music. The first step to polyphonic music are chords. A chord consists of different note-heads which are on the same horizontal position together and which may share one common stem. As can be seen, chords are a combination of note heads, which get a modifier that places a stem to this group. In the same way a note can be seen as a chord with only one element, which is then modified by a stem adding function. Therefore we introduce new basic element records, which represent only the note heads[7]:

```
k4  :: Char Int -> Elementrecord
k8  :: Char Int -> Elementrecord
k16 :: Char Int -> Elementrecord
k32 :: Char Int -> Elementrecord
k64 :: Char Int -> Elementrecord
```

Now we introduce the new modifier functions, which add stems to elements. One for downwards and one for upwards directed stems[8]:

[7] *head* means *Kopf* in German, therefore the `k`

[8] The German meaning of *stem* is *Hals*. The `o` indicates an upward directed stem (*oben* in German), the `u` a downwards directed stem (*unten* in German).

```
halso :: Elementrecord -> Elementrecord
halsu :: Elementrecord -> Elementrecord
```

This makes notes no longer primitive elements, but constructed elements, which can be defined as a macro function:

```
n4 :: Char Int -> Elementrecord
n4 c i |pitch c i > pitch ’h’ 1 = n4u c i
       |otherwise               = n4O c i

n4u c i = halsu (n4 c i)
n4o c i = halso (n4 c i)
```

Now, where we have the more basic primitives of note-heads, we can combine elements on one horizontal point, in order to form chords. We define the basic function to build chords:

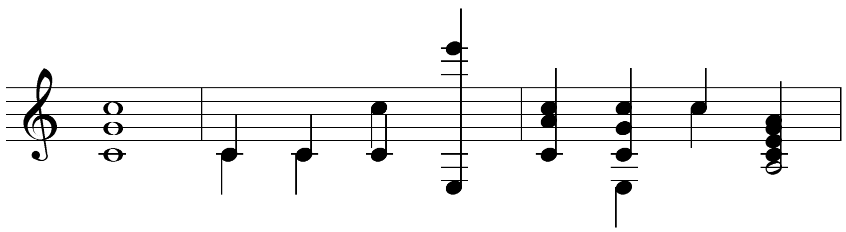
```
akk :: [Elementrecord] -> Elementrecord
```

Instead of taking one combinator which combines exactly two elements into a new one, the function `akk` combines a list of elements.

We can specify some algebraic laws for the function `akk`. Naturally we demand that `akk [e]` prints the same result as `e`. Furthermore we want the result of `akk` to be independent of the order of the list elements. Some further property of `akk`, we expect is:
`akk [e,akk c]=akk [e:c]`

Now let us see an example how we can combine chords. In the following we show the code and display the result for a sequence of chords[9]:

```
[akk [n1 ’c’ 1,n1 ’c’ 2,n1 ’g’ 1]
,akk [n4 ’c’ 1,halsu (k4 ’c’ 1)]--
 akk [akk [halsu (k4 ’c’ 1),n4 ’c’ 1]]--
 akk [n4 ’c’ 1,n4 ’c’ 2]--
 halso (akk [k4 ’e’ 0,k4 ’e’ 3])
,halso (akk [k4 ’c’ 1,k4 ’c’ 2,k4 ’4’ 1])--
 akk [halso(akk [k4 ’c’ 1,k4 ’c’ 2,k4 ’g’ 1]),halsu(k4 ’e’ 0)]--
 halso (akk [n4 ’c’ 2])--
 halso (akk [n2 ’a’ 0,n4 ’c’ 1,n4 ’e’ 1,n4 ’g’ 1,n4 ’a’ 1])
]
```



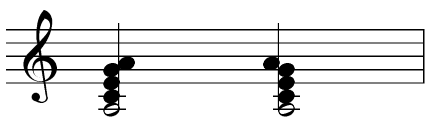
[9]In *tinte* bars are simply represented as a list of elements

In order to have all these functionality of `akk` we have to collect some more information in an element record. In order to place the right stem, we need to know which is the highest note and which is the lowest note of an element. This adds two more entries in an element record.

As we have seen in the last chord of our example, the function `akk` does not behave quite satisfactory in all situations. Notes which are directly next to each other in a chord, are usually not placed exactly at the same horizontal point, but one of them is shifted a note-head width. Therefore we introduce two further functions which can build chords and regard the rule of shifting certain notes of a chord:

```
akko :: [Elementrecord] -> Elementrecord
akku :: [Elementrecord] -> Elementrecord
```

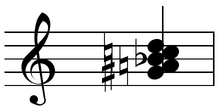
Now we can set the last chord of the previous example, such that note heads are shifted:



```
[halso (akko [k2 'a' 0,k4 'c' 1,k4 'e' 1,k4 'g' 1,k4 'a' 1])--
 halso (akku [k2 'a' 0,k4 'c' 1,k4 'e' 1,k4 'g' 1,k4 'a' 1])]
```

One aspect we neglected in this section are accidentals. Also these have to be respected when forming a chord. In order to be able to implement `akk`, `akko` and `akku` with the desired behavior for accidentals we have to add some more information to the element record. We need to know what accidentals an element has. Eventually we created chord building functions which respect all the relevant cases such that e.g. the following chord can be set:

```
halso
 (akko [is(k4 'g' 1),na(k4 'a' 1)
       ,es(k4 'h' 1),na (k4 'c' 2)
       ,k4 'd' 2])
```



With the functions introduced in this section, we are not only able to typeset chords, but have also the key for typesetting polyphonic music. Elements of a chord are not demanded to have one common stem such that they can belong to different voices in polyphonic music.

4.2.3 Grouping

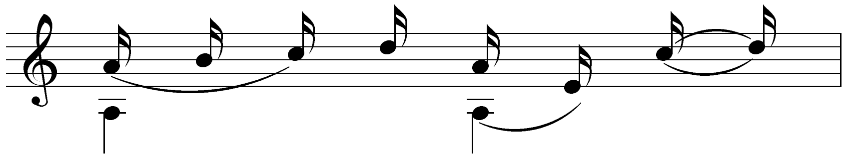
In the last subsection we introduced functions for vertical grouping, i.e. for typesetting chords. Now we want to take a closer look at the sequential combination of elements. We already can combine sequences of elements.

slurs Now we would like to add beams and slurs to a sequence of elements. What we would like to have, is a function which adds slurs to a sequence of elements. For such a function we have to specify, which are start and end note for the slur and if the slur is above or underneath the elements. Since the building of chords can make elements rather complex, we decided to have as further arguments the height, where to put the slur explicitly. This can be given as an integer, where 0 denotes a slur for the pitch e', n a slur for the pitch which is ($n *$ note-head-height/2) above e' and n a slur for the pitch which is ($n *$ note-head-height/2) below e'. We define two functions for adding slurs to elements[10]:

```
bogenu :: (Int,Int) (Int,Int) Elementrecord -> Elementrecord
bogenu :: (Int,Int) (Int,Int) Elementrecord -> Elementrecord
```

The first position of the two tuple arguments denote the number of the element and the second position pitch height of the slur for start and end element of the slur. With this functions we can typeset the following sequence:

```
[bogenu (1,~3) (3,~5)
  (akk [n16o 'a' 1,n4u 'a' 0]--n16o 'h' 1--
   n16o 'c' 2--n16o 'd' 2)--
 bogeno (3,~5) (4,~6)
  (bogenu (3,~5) (4,~6)
   (bogenu (1,4) (2,0)
    (akk [n16o 'a' 1,n4u 'a' 0]--n16o 'e' 1--
     n16o 'c' 2--n16o 'd' 2)))]
```



In order to implement these slur functions we need again more information about an element in a record. An element which has been produced by the sequencing operator `--` has to have the information of which components it consists in its record. Therefore we have enlarged the type `Elementrecord` with a further field, where the parts of the element are stored as a list. The full type `Elementrecord` is given in appendix C.

In order to typeset a slur, we need to know its width. This width can only be known after the line-break function has calculated the actual space between the elements. Again higher order programming can give an easy solution to this problem. Since we modeled the dvi-code as a function depending on the stretching factor for the line, we can make the selection of the correct slur character dependent on this factor.

beams Adding beams to a sequence of elements is quite similar to adding slurs. As for slurs there are different characters in a font which denote beams with different angles of gradient. In order to get a beam of an arbitrary length a character is repeated several times and set overlapped with itself. A function for setting beams has to perform several task:

[10] Guess what: *Bogen* is the German word for slur

- calculate an angle of gradient for the beam. This will be dependent on the stretching factor.
- calculate the length of the beam.
- select the correct character for the beam.
- calculate, how many beams element of the group requires.
- calculate the position and length of stems.

We have to provide some further information for the stems of a group: we have to say which notes of a vertical combined element are to be connected with a stem to the beam. This leads to the following functions for beams above and below a group of elements[11]:

```
balko :: [(Int,Int,Int,Int)] Elementrecord -> Elementrecord
balku :: [(Int,Int,Int,Int)] Elementrecord -> Elementrecord
```

The list argument of these functions denotes the elements which are to be connected with a beam. The first tuple number is the number of the element that is to participate, the second number denotes the value of the element, i.e. 8 denotes quavers, 16 semiquavers and so on. This determines the number of beams on each element. The third and fourth number of the tuple denote highest and lowest pitch which will be connected with a beam. Now we can see the beaming functions in action. The following sequence describes the very first example of this paper as has been printed out in section 1.1:

```
[bogeno (2,~10) (3,~9)
 (balku
  [(1,16,~10,~10),(2,32,~10,~10),(3,32,~9,~9),
   (4,16,~8,~8),(5,16,~7,~7)]
 (balku [(1,8,1,1),(4,8,4,4)]
  (akko [k8 'd' 1,k16 'a' 2]--k32 'a' 2--k32 'g' 2
   --akko [k8 'a' 0,k16 'f' 2]--k16 'e' 2)))--
 akko [n8u 'f' 0,n8o 'd' 2]--
 bogeno (1,~9) (2,~8) (balku [(1,16,~9,~9),(2,16,~8,~8)]
  (akko [k16 'g' 2,p8u]--k16 'f' 2))
]
```

In some publications beams aree not only found only above or underneath a group of notes but both. For this situation a third function for adding a beam to a group has been defined:

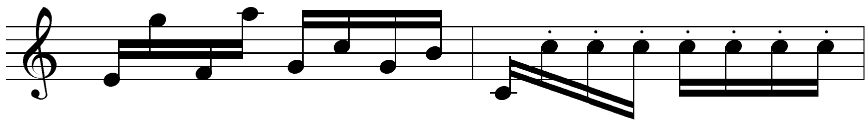
```
balku :: [(Int,Int,Int,Int)] [(Int,Int,Int,Int)]
          Elementrecord -> Elementrecord
```

As an example for this function we give 2 bars taken from a Sonata by Pierre Porro. The *Tinte* code is:

[11] Guess again: *Balken* is the German world for *beam*

```
balkou  [(1,16,0,0),(3,16,~1,~1)] [(2,16,~9,~9),(4,16,~10,~10)]
 (k16 'e' 1--k16 'g' 2--k16 'f' 1--k16 'a' 2)
 --balkeno [k16 'g' 1,k16 'c' 2,k16 'g' 1,k16 'h' 1]
,balkou [(1,16,2,2)][(2,16,~5,~5),(3,16,~5,~5),(4,16,~5,~5)]
 (k16 'c' 1--stao (k16 'c' 2)--stao (k16 'c' 2)--stao (k16 'c' 2))
 --balkenu (map stao [k16 'c' 2,k16 'c' 2,k16 'c' 2,k16 'c' 2])
```

Which generates the following output:



4.3 Introducing monads into the system

In order to generate dvi-code *tinte* needs quite a lot global parameters, as e.g. the length of a line, the key of the piece to be typeset, the width of characters and so on. Up to now these parameters were assumed to be given globally in some module as macro definitions. These parameters do not appear as function arguments and therefore have to stay constant for a piece that is typeset. But we might like to change these parameters in the middle of a bar, e.g. when typesetting grace notes, because grace notes are smaller and stand closer to each other. The standard way to incorporate a global state which can change during evaluation into a functional program is by way of a monad [Wad90, Wad92, Wad95]. Unlike Haskell [HAB+96] Clean does not provide a special syntax for monads.

For our needs we have to use a state monad, which allows to combine some data type with a global state. This means that instead of the type `Elementrecord` the musical elements are objects of the following type[12]:

```
:: GenElem :==  Global -> (Elementrecord,Global)
```

where `Global` is a record type which contains all global information that is needed.

We can define the usual function which transforms objects of type `Elementrecord` into objects of type `GenElem`:

```
result :: Elementrecord -> GenElem
result dvielem = \s -> (dvielem,s)
```

This way we can easily transform all functions which define an element record into the state monad. Now we have to redefine the combinator functions anew such that they combine no longer records, but monadic objects, i.e. functions. For the sequential combination we use the original definition of `--` and call it `bi` such that we can use it in a new definition of `--`:

[12]With polymorphic typing the state monad can of course be defined more generally. For simplicity we just define the specialized version here. As a matter of fact: the state monad can be found in a Clean library.

```
(--) infixl 1 :: GenElem GenElem -> GenElem
(--) e1 e2 = binde e1 e2
              where
               binde e1 e2 s = (e11 bi e21,s2)
                where
                  (e11,s1) = e1 s
                  (e21,s2) = e2 s1
```

This way we are able to change the types of our musical elements but do not need to change the source code of the examples we have already programmed.

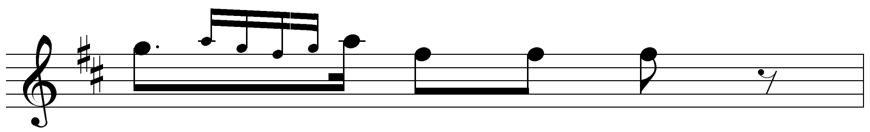
Now we can define musical elements which change the global state. Such changes are to switch into a grace mode where all spaces are smaller and a smaller font is used or to switch back into normal mode:

```
small :: GenElem
small = \s ->  ({initdvielem & parts = [pa]}
                 ,{s&grace = Grace})
 where
  pa= { initpart &
        pvorsatzdvi = odvi (D [Fnt_num fntgracemusic])}

normal :: GenElem
normal  = \s -> ({initdvielem & parts = [pa]}
                  ,{s&grace = Normal})
 where
  pa= { initpart &
       pvorsatzdvi = odvi (D [Fnt_num fntmusic])}
```

Now we can express a sequence which contains grace notes like in the following bar taken from the flute part of Divertimento II op. 33 of Antonio Nava.

```
[balku [(1,8,~9,~9),(7,16,~10,~10)]
 (pu (k32 ’g’ 2)--
  small--balkeno [k16 ’a’ 2,k16 ’g’ 2,k16 ’f’ 2,k16 ’g’ 2]--
  normal--k16 ’a’ 2--
  balkenu [k8 ’f’ 2,k8 ’f’ 2]--n8 ’f’ 2--p8 )]
```



The astute reader will have noticed that a grouping with a beam can overlap with changes from and to the grace mode.

5 Comparisions to other systems

Tinte as a prototype follows the same guide lines as MusicTeX and *Mup*. It has been developed independently from *Mup* but used mtex and MusicTeX as example. The musical description language of *Tinte* goes more in the line of mtex than *Mup*. *Mup* provides a higher level of abstraction and can describe different voices in a single staff independently, a feature which is currently not supported in *Tinte*. The syntax of *Mup* is somewhat slimer, because many default parameters are used and only changes from these need to be recorded. E.g. the introductory example of this paper can be expressed in *Mup* as:

```
staff 1
vscheme=2f
music
1 1: 16a+bm;32a+<>;32g+;16f+;e+ebm;8d+;16g+<>bm;f+ebm;
1 2: 8dbm;a-ebm;e-;r;
bar;
```

As can be seen in *Mup* there are more things decided by the system. However, it is not always possible to overwrite the default settings of *Mup*. The code above will not place the beams of the first voice downwards as we could determine in *Tinte* and as our predecessor in the 19th century has done.

Although *Mup* has a wide range of features and has been developed over several years some smaller features seem to be missing: e.g. beams which are laying between note-heads cannot be expressed.

6 Conclusion

We implemented a basic typesetting program for typesetting classical guitar music. The system offers a wide flexibility in usage. Extended examples[13] can be down-loaded at:
`http://www.ki.informatik.uni-frankfurt.de/persons/panitz/guitar.html`

New musical elements that have not been thought of in the original design can be added easily as long as they stay in the restriction to single staff music. The prototype directs the way to how a complex system with a more general specification can be designed. Nevertheless we must admit that the system does not offer any comfort in usage and lacks any user interface. The music is simply coded in a Clean module and then linked with the Clean libraries of *Tinte*.

Thus lazy functional language have again proved to be an adequate tool for prototyping. The implementation of the overall system was done in just a few weeks time, which included the implementation of some complex examples. The runtime behavior is quite satisfying, drawing into account that no effort for an efficient solution was made. 14 pages of rather complex guitar music are transformed into a dvi file in about 30s–120s depending on the systems architecture (we were using several Power Mac, 68k Mac and Sun workstations).

One feature which is missing in strongly typed lazy functional languages is an evaluator of the language within itself, i.e. there is no `eval`-function as known from lisp. This would have been rather convenient for our implementation,

[13]The created Clean libraries which form *Tinte* are only given by personal request, because they do not have the quality which one would expect of a freely available product.

because this would provide a parser and a type-checker for the language. Music would no longer be required to be typeset as a Clean module and compiled, but could be interpreted from the main system *Tinte*.

Now the only solution is to write an own interpreter of the functions which are needed for typesetting music. This interpreter could be extended with some standard Clean functions, which are useful for typesetting music. Consider e.g. a list of notes which have all the same accidental. Here we can use the function `map`:

```
akk (map (\x->is(x 1) [k4 'c',k4 'd',k4 'f',k4 'g'])
```

This way, how the concept of the implementation language effects the concept of the resulting system, is similar to the experiences made in [DH93, HRE95].

A Summary of musical constructs

We give the interface (`*.dcl`)-files for the functions that can be used to describe music. Unfortunately there are not yet comments to the functions, but most of them should have been explained in this paper.

A.1 Note heads

```
definition module ppnotenkoepfe

import pptintdata

k1 :: Char Int -> GenElem;
k2 :: Char Int -> GenElem;
k4 :: Char Int -> GenElem;
k8 :: Char Int -> GenElem;
k16 :: Char Int -> GenElem;
k32 :: Char Int -> GenElem;
k64 :: Char Int -> GenElem;
gr8o :: Char Int -> GenElem;
```

A.2 Stems

```
definition module pphals
import pptintdata

halsol :: Int !GenElem -> GenElem
halsul :: Int !GenElem -> GenElem
halso ::  !GenElem -> GenElem
halsu ::  !GenElem -> GenElem
hals :: !GenElem -> GenElem
```

A.3 Flags

```
definition module ppfahnen
import pptintdata

f8o :: GenElem -> GenElem
f8u :: GenElem -> GenElem
```

```
f8  :: GenElem -> GenElem

f16o :: GenElem -> GenElem
f16u :: GenElem -> GenElem
f16  :: GenElem -> GenElem

f32o :: GenElem -> GenElem
f32u :: GenElem -> GenElem
f32  :: GenElem -> GenElem

f64o :: GenElem -> GenElem
f64u :: GenElem -> GenElem
f64  :: GenElem -> GenElem

f  :: GenElem -> GenElem
fo :: GenElem -> GenElem
fu :: GenElem -> GenElem

grfo :: GenElem -> GenElem
```

A.4 Rests

```
definition module pppausen
import pptintdata

p1::GenElem
p2::GenElem
p4::GenElem
p8::GenElem
p16::GenElem
p32::GenElem
p64::GenElem
p1o::GenElem
p1u::GenElem
p2o::GenElem
p2u::GenElem
p4o::GenElem
p4u::GenElem
p8o::GenElem
p8u::GenElem
p16o::GenElem
p16u::GenElem
p32o::GenElem
p32u::GenElem
p64o::GenElem
p64u::GenElem
```

A.5 Chords

```
definition module ppakkord

import pptintdata

akk :: [GenElem] -> GenElem
akko :: [GenElem] -> GenElem
akku :: [GenElem] -> GenElem
```

A.6 Slurs

```
definition module ppbogen

import pptintdata

bogenu :: (Int,Int) (Int,Int) GenElem -> GenElem
bogenlu :: [((Int,Int), (Int,Int))] GenElem -> GenElem
bogeno :: (Int,Int) (Int,Int) GenElem -> GenElem
bogenlo :: [((Int,Int), (Int,Int))] GenElem -> GenElem
```

A.7 Beams

```
definition module ppbalken
import pptintdata

balkeno :: [GenElem] -> GenElem
balkenu :: [GenElem] -> GenElem

balko :: [(Int,Int,Int,Int)] GenElem -> GenElem
balku :: [(Int,Int,Int,Int)] GenElem -> GenElem
balkou :: [(Int,Int,Int,Int)] [(Int,Int,Int,Int)] GenElem -> GenElem
```

A.8 Miscellaneous

```
definition module pp
import pptintdata, pppausen, ppfahnen, ppnotenkoepfe,
import ppakkord, pphals,ppbalken,ppbogen,ppcres

n1::Char Int ->GenElem
n2::Char Int ->GenElem
n4::Char Int ->GenElem
n8::Char Int ->GenElem
n16::Char Int ->GenElem
n32::Char Int ->GenElem
n64::Char Int ->GenElem
n1o::Char Int ->GenElem
n2o::Char Int ->GenElem
n4o::Char Int ->GenElem
n8o::Char Int ->GenElem
n16o::Char Int ->GenElem
n32o::Char Int ->GenElem
n64o::Char Int ->GenElem
n1u::Char Int ->GenElem
n2u::Char Int ->GenElem
n4u::Char Int ->GenElem
n8u::Char Int ->GenElem
n16u::Char Int ->GenElem
n32u::Char Int ->GenElem
n64u::Char Int ->GenElem

metrum :: Int Int ->GenElem
klammer :: Int GenElem ->GenElem

textmfnt :: Int Int String GenElem -> GenElem
textmfnto :: Int  String GenElem -> GenElem
textmfntu :: Int  String GenElem -> GenElem
texto :: String GenElem -> GenElem
textu :: String GenElem -> GenElem
```

```
text:: Int String GenElem -> GenElem
pu :: GenElem ->GenElem
stau :: GenElem ->GenElem
stao :: GenElem ->GenElem
akzu :: GenElem ->GenElem
akzo :: GenElem ->GenElem
endwiederholung :: GenElem
endwiederholungmitte :: GenElem
anfwiederholung :: GenElem
anfwiederholungmitte :: GenElem
anfendwiederholung :: GenElem
endstrich :: GenElem
breittaktstrich :: GenElem
fermateo :: GenElem ->GenElem
fermateu :: GenElem ->GenElem
tro :: GenElem ->GenElem
tru :: GenElem ->GenElem

takt :: GenElem
dicktaktstrich :: GenElem
left :: Int -> GenElem
ohnzw :: GenElem -> GenElem


liftd :: Int DVIelem -> DVIelem
lift :: Int GenElem -> GenElem


is :: GenElem -> GenElem
es :: GenElem -> GenElem
na :: GenElem -> GenElem

small  :: GenElem
normal :: GenElem

center ::GenElem -> GenElem
```

B Example

We give an extended example. The following is the complete guitar part of the adagio of Divertimento I by Antonio Nava. The output produced by Tinte can be seen after the code.

```
[textmfnto 14 "Adagio Cantabile" (metrum 6 8)--p8
,textu "p"
 (balku  [(1,8,4,4),(3,8,4,4),(5,8,4,4)]
 (balko  [(1,16,~5,~3),(2,16,~7,~7),(3,16,~5,~3),(4,16,~7,~7)
         ,(5,16,~5,~3),(6,16,~7,~7)]
  (akko [k8 'a' 0,k16 'a' 1,k16 'c' 2]--k16 'e' 2--
   akko [k8 'a' 0,k16 'a' 1,k16 'c' 2]--k16 'e' 2--
   akko [k8 'a' 0,k16 'a' 1,k16 'c' 2]--k16 'e' 2))--
 balku  [(1,8,7,7),(3,8,7,7),(5,8,7,7)]
 (balko  [(1,16,~6,~4),(2,16,~7,~7),(3,16,~6,~4),(4,16,~7,~7)
         ,(5,16,~6,~4),(6,16,~7,~7)]
  (akko [k8 'e' 0,k16 'h' 1,k16 'd' 2]--k16 'e' 2--
   akko [k8 'e' 0,k16 'h' 1,k16 'd' 2]--k16 'e' 2--
   akko [k8 'e' 0,k16 'h' 1,k16 'd' 2]--k16 'e' 2))
,akko [halsu (k4 'a' 0),halso (akko [k4 'a' 1,k4 'c' 2])]--
 akko [fu (halsu (k8 'e' 1)),fo (halso (k8 'h' 1))]--
 akko [(halsu (k4 'c' 1)),(halso (k4 'a' 1))]--
```

```
 akko [fu (halsu (k8 'f' 1)),is(fo (halso (k8 'a' 1)))]
,gr8o 'c' 2--
 textu "f"
 (balku  [(1,8,1,1),(3,8,1,1),(5,8,1,1)]
 (balko  [(1,16,~4,~4),(2,16,~3,~3),(3,16,~4,~4),(4,16,~5,~5)
         ,(5,16,~6,~6),(6,16,~7,~7)]
  (akko [k8 'd' 1,k16 'h' 1]--is(k16 'a' 1)--
   akko [k8 'd' 1,k16 'h' 1]--k16 'c' 2--
   akko [k8 'd' 1,k16 'd' 2]--k16 'e' 2))--
 bogenlu [((2,~9),(3,~8)),((4,~8),(5,~7)),((6,~7),(7,~6))
         ,((8,~6),(9,~5)),((10,~5),(11,~4))]
 (balku  [(1,8,1,1),(4,8,1,1),(8,8,1,1)]
 (balko [(1,16,~9,~9),(2,32,~9,~9),(3,32,~8,~8),(4,32,~8,~8),(5,32,~7,~7)
        ,(6,32,~7,~7),(7,32,~6,~6),(8,32,~6,~6),(9,32,~5,~5),(10,32,~5,~5)
        ,(11,32,~4,~4)]
  (akko [k8 'd' 1,k16 'g' 2]--k32 'g' 2--k32 'f' 2--
   akko [k8 'd' 1,k32 'f' 2]--k32 'e' 2--k32 'e' 2--k32 'd' 2--
   akko [k8 'd' 1,k32 'd' 2]--k32 'c' 2--k32 'c' 2--k32 'h' 1)))
,bogenlu [((4,~10),(5,~9)),((6,~8),(7,~7)),((8,~6),(9,~5))]
 (balko  [(1,16,~3,~3),(2,16,~5,~5),(3,16,~7,~7)
         ,(4,32,~10,~10),(5,32,~9,~9),(6,32,~8,~8),(7,32,~7,~7)
         ,(8,32,~6,~6),(9,32,~5,~5)]
  (akko [halsu(k4 'e' 1),k16 'a' 1]--k16 'c' 2--k16 'e' 2--k32 'a'2
   --k32 'g' 2--
   akko [p8u,k32 'f' 2]--k32 'e' 2--k32 'd' 2--k32 'c' 2))--
 akko [halsu (k4 'e' 0),halso (akko [k4 'g' 1,k4 'h' 1])]--akko[p8,p8u]
,decrt 4 1 5
 (balko [(2,16,~5,~5),(3,16,~7,~7),(4,16,~10,~10),(5,16,~7,~7),(6,16,~5,~5)]
 (balku [(1,8,~3,~3),(3,8,~3,~3),(5,8,~3,~3)]
  (akko [p16o,k8 'a' 1]--k16 'c' 2--
   akko [k16 'e' 2,k8 'a' 1]--k16 'a' 2--
   akko [k16 'e' 2,k8 'a' 1]--k16 'c' 2)))--
 decrt 4 1 5
 (balko [(2,16,~2,~2),(3,16,~5,~5),(4,16,~7,~7),(5,16,~5,~5),(6,16,~2,~2)]
 (balku [(1,8,0,0),(3,8,0,0),(5,8,0,0)]
  (akko [p16o,is(k8 'e' 1)]--k16 'g' 1--
   akko [k16 'c' 2,k8 'e' 1]--is(k16 'e' 2)--
   akko [k16 'c' 2,k8 'e' 1]--k16 'g' 1)))
,decrt 4 1 5
 (balko [(2,16,~3,~3),(3,16,~5,~5),(4,16,~8,~8),(5,16,~5,~5),(6,16,~3,~3)]
 (balku [(1,8,~1,~1),(3,8,~1,~1),(5,8,~1,~1)]
  (akko [p16o,k8 'f' 1]--k16 'a' 1--
   akko [k16 'c' 2,k8 'f' 1]--k16 'f' 2--
   akko [k16 'c' 2,k8 'f' 1]--k16 'a' 1)))--
 decrt 4 1 5
 (balko [(2,16,~4,~4),(3,16,~6,~6),(4,16,~8,~8),(5,16,~6,~6),(6,16,~4,~4)]
 (balku [(1,8,1,1),(3,8,1,1),(5,8,1,1)]
  (akko [p16o,(k8 'd' 1)]--k16 'h' 1--
   akko [k16 'd' 2,k8 'd' 1]--(k16 'f' 2)--
   akko [k16 'd' 2,k8 'd' 1]--k16 'h' 1)))
,decrt 4 1 5
 (balko [(2,16,~3,~3),(3,16,~5,~5),(4,16,~7,~7),(5,16,~5,~5),(6,16,~3,~3)]
 (balku [(1,8,0,0),(3,8,0,0),(5,8,0,0)]
  (akko [p16o,k8 'e' 1]--k16 'a' 1--
   akko [k16 'c' 2,k8 'e' 1]--k16 'e' 2--
   akko [k16 'c' 2,k8 'e' 1]--k16 'a' 1)))--
 decrt 4 1 5
 (balko [(2,16,~2,~2),(3,16,~4,~4),(4,16,~7,~7),(5,16,~6,~6),(6,16,~4,~4)]
 (balku [(1,8,7,7),(3,8,7,7),(5,8,7,7)]
  (akko [p16o,(k8 'e' 0)]--k16 'g' 1--
   akko [k16 'h' 1,k8 'e' 0]--(k16 'e' 2)--
   akko [k16 'd' 2,k8 'e' 0]--k16 'h' 1)))
```

```
,akko [halsu (k4 'a' 0),halso (akko [k4 'a' 1,k4 'c' 2])]--
 akko [fu (halsu (k8 'e' 1)),fo (halso (k8 'h' 1))]--
 akko [(halsu (k4 'c' 1)),(halso (k4 'a' 1))]--anfendwiederholung--
 n8 'e' 2
,decr 1 3
 (balko [(1,8,~7,~3),(2,8,~5,~5),(3,8,~3,~3)]
  (akko [k8 'e'2,k8 'c' 2,is(k8 'a' 1),halsu (k8 'f' 1)]
   --k8 'c' 2--akko [p8u,k8 'a' 1]))--
 decrt 1 1 6
 (balko [(1,16,~9,~3),(2,16,~7,~7),(3,16,~5,~5),(4,16,~3,~3)
        ,(5,16,~4,~4),(6,16,~5,~5)]
 (akko [na(k16 'g' 2),k16 'c' 2,is(k16 'a'1),halsu (k16 'e' 1)]
   --k16 'e' 2--k16 'c' 2--
  k16 'a' 1--akko [p8u,k16 'h' 1]--k16 'c' 2))
,gr8o 'c' 2--
 balko [(1,16,~4,~4),(2,16,~3,~3),(3,16,~4,~4),(4,16,~5,~5)
       ,(5,16,~6,~6),(6,16,~5,~5)]
   (akko [k16 'h' 1,halsu (k16 'd' 1)]--is (k16 'a' 1)--k16 'h' 1--
   k16 'c' 2--akko [p8u,k16 'd' 2]--k16 'c' 2)--
 decrt 4 1 3
 (balko [(1,8,~4,~4),(2,8,~4,~4),(3,8,~3,~3)]
  (balku [(1,8,1,1),(2,8,1,1),(3,8,1,1)]
  (akko [k8 'h'1,k8 'd'1]--akko [k8 'h'1,k8 'd'1]
   --akko [na(k8 'a' 1),is(k8 'd'1)])))
,decrt 1 1 3
 (balku [(1,8,0,0),(2,8,7,7),(3,8,~1,~1)]
  (akko [k8 'e' 1,n4 'g' 1]--k8 'e'0 --akku [na(k8 'f' 1),n8 'g' 1]))--
 decrt 4 1 3
 (balku [(1,8,0,0),(2,8,1,1),(3,8,3,3)]
  (akko [k8 'e' 1,n4 'g' 1]--k8 'd'1 --akku [(n8 'g'1),k8 'h' 0]))
,textu "f" (balku [(1,8,2,2),(2,8,0,0),(3,8,2,2)]
  (akko [na(k8 'c' 1),n4 'a' 1]--k8 'e'1 --akku [(n8 'a'1),k8 'c' 1]))--
 akko [(halsu (k4 'a' 0)),(halso (k4 'a' 1))]--akko[p8,p8u]
,textu "p" (akko [halso (akko [na(k4 'c'2),k4 'a'1]),halsu (is (k4 'd'1))])--
 akko [f8o(halso (akko [(k8 'c'2),k8 'a'1])),f8u (halsu ( (k8 'd'1)))]--
 akko [halso (akko [(k4 'c'2),k4 'a'1]),halsu ( (k4 'd'1))]--
 akko [f8o(halso (akko [(k8 'c'2),k8 'a'1])),f8u (halsu ( (k8 'd'1)))]
,textu "f" (akko [halso (akko [(k4 'h'1),k4 'g'1]),halsu (k4 '1'1)])--
 akko[p8,p8u]--
 textu "p" (akko [halso (akko [na(k4 'c'2),k4 'a'1]),halsu (is (k4 'd'1))])--
 akko [f8o(halso (akko [(k8 'c'2),k8 'a'1])),f8u (halsu ( (k8 'd'1)))]
,balko [(1,8,~4,~2),(2,8,~7,~7),(3,8,~6,~6)]
 (balku [(1,8,0,0),(2,8,0,0),(3,8,1,1)]
  (akko [k8 'e'1,k8 'g'1,k8 'h' 1]--akko [k8 'e'1,k8 'e'2]
   --akko [na(k8 'd' 1),(k8 'd'2)]))--
 balko [(1,8,~5,~3),(2,8,~6,~6),(3,8,~6,~1)]
  (akko [halsu(na(k8 'c'1)),k8 'a'1,na(k8 'c' 2)]--(k8 'd' 2)
   --textu "sf" (akko [f8u(halsu(k8 'a' 0)),na(k8 'f'1),k8 'a'1,is(k8 'd'2)]))
,akko[halso (akko (map pu [k4 'h'0,k4 'd'1,k4 'g'1,k4 'h'1,k4 'e'2]))
     ,halsu(pu (k4 'e'0))]
 --small--
 bogenu (1,~5)(2,3)
 (balko [(1,32,~7,~7),(2,32,~4,~4),(3,32,~2,~2),(4,32,0,0),(5,32,3,3)]
  (k32 'e'2--k32 'h' 1--k32 'g' 1--k32 'e' 1--k32 'h' 0))
 --normal--n4 'e'0--fermateo p8--endstrich]
```

C Data definitions and code fragments

The most fundamental type used in the system is the type `Elementrecord`, which describes a musical element.


Adagio Cantabile

```
:: Odvi   = O (DVI1 -> DVI1)

instance +++ Odvi where
  (+++) (O odvi1) (O odvi2) = O (odvi1 o odvi2)

::Elementrecord
   = {maxbreite :: Int //maximal and minimal length
     ,minbreite :: Int //of the whole element
     ,minzw     :: Int //minimal and maximal length
     ,maxzw     :: Int //of the space after the element
     ,zwischenr :: Int Int -> Int
                      //function how to determine space after element
     ,hoehe     :: Int //highest and lowest point of the element
     ,tiefe     :: Int //
     ,offset    :: Int //length of space before actual element starts
     ,dvi       :: Int Int -> Odvi
                      //function which produces the actual dvi code
     ,vorsatzdvi:: Odvi//dvi code for things like sharps etc
     ,vorsatzart:: [(Element,Int)]
                      //definitions of the elements sharps etc.
     ,schlaege  :: [(Int,Int)]
                      //fraction of the beats. Necessary for polyphonic
                      //combination, which is not yet made, therefore
                      //yet unused
     ,def       :: Element
                      //definition of the element (not used anymore)
     ,ref_hoehe :: Int //main pitch of the element
     ,parts     :: [Part]
                      //parts of a combined element

::Part
 = {poffset :: Int
   ,pwoliegt :: Int Int -> Int
   ,ptiefe :: Int
   ,phoehe :: Int
   ,pvorsatzdvi :: Odvi
   ,pdvi :: Int Int -> Odvi
   ,pschlag :: (Int,Int)
   ,pvorsatzart :: [(Element,Int)]}


initdvielem :: Elementrecord
initdvielem
 = {maxbreite = 0
   ,minbreite = 0
   ,maxzw = 0
   ,minzw = 0
   ,zwischenr = \z n ->0
   ,hoehe = 0
   ,tiefe = 0
   ,offset = 0
   ,dvi = \z n -> odvi (D [])
   ,vorsatzdvi = odvi (D [])
   ,vorsatzart = []
   ,schlaege = []
   ,def = abort "nie Definition gegeben"
   ,ref_hoehe = 0
   ,parts = []}

initpart :: Part
initpart
 = {poffset = 0
```

```
   ,pwoliegt= \z n -> 0
   ,ptiefe  = 0
   ,phoehe  = 0
   ,pvorsatzdvi = odvi (D [])
   ,pdvi    = \z n ->odvi (D [])
   ,pschlag = abort "undefinierter Schlag in part"
   ,pvorsatzart = []}


:: Global
  = {tonartzeichen                  :: Odvi
    ,anzahlzeilenersteseite         :: Int
    ,anzahlzeilenweitereseiten      :: Int
    ,teiler                         :: Int
    ,artab                          :: Int
    ,grace                          :: Grace
    ,zeilenlaenge                   :: Int
    ,seitennr                       :: Int}


initialglob :: Global
initialglob
 = {anzahlzeilenweitereseiten = 9
   ,anzahlzeilenersteseite    = 8
   ,tonartzeichen             = cdur
   ,artab                     = 0
   ,teiler                    = 30
   ,grace                     = Normal
   ,zeilenlaenge              = langl
   ,seitennr                  = 1}
```

D How this paper has been typeset

This paper has been typeset using the LaTeX macro package for TeX. The musical examples have been created using *tinte*. The resulting dvi-files have been transformed to encapsulated postscripts files by *dvips* with the option flag `-E`. The resulting epsf-file was then included in the TeX-file with the LaTeX command: `\includegraphics`.

References

[Ark] Arkkra Enterprises. *Mup Music Publishers Users's Guide.* `http://www.arkkra.com/`.

[DH93] W. De Hoon. Designing a spreadsheet in a pure functional graph rewriting language. Master's thesis, Catholic University Nijmegen, 93.

[Fuc80] David Fuchs. The format of TeX's DVI files. *TUGboat*, 1(1):17–19, October 1980.

[HAB+96] K. Hammond, L. Augustsson, B. Boutel, W. Burton, J. Fairbairn, J. Fasel, A. Gordon, M. Guzmán, J. Hughes, P. Hudak, T. Johnsson,

M. Jones, D. Kieburtz, R. Nikhil, W. Partain, J. Peterson, S. Peyton Jones, and P Wadler. Report on the programming language Haskell 1.3. Technical report, Department of Computer Science, University of Glasgow, 1996.

[HJ94] Paul Hudak and Mark P. Jones. Haskell vs Ada vs. C++ vs. Awk vs. ... an experiment in software prototyping productivity. to appear in the Journal of Functional Programming, 7 1994.

[HRE95] W. de Hoon, L. Rutten, and M.C.J.D. van Eekelen. Implementing a functional spreadsheet in clean. *Journal of Functional Programming*, 3, 1995.

[Hug95] John Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995.

[Knu91] Donald E. Knuth. *Computers and Typesetting Vol. A-E*. Addison-Wesley Co., Reading, MA, 1987–1991.

[Pv97] M. J. Plasmeijer and M van Eekelen. *Concurrent Clean Language Report 1.2*. University of Nijmegen, 1997.

[SS87] Angelika Schofer and Andrea Steinbach. Automatisierter Notensatz mit TEX. Technical report, Rheinische Friedrich-Wilhelms-Universität Bonn, 8 1987. in German.

[Tau93] Daniel Taupin. MusicTEX: Using TEX to write polyphonic or instrumental music. *TUGboat*, 14(3):203–211, October 1993.

[Tur85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 1–16. Springer, 1985.

[Wad90] P. Wadler. Comprehending monads. In *Proceedings of Symposium on Lisp and Functional Programming*, pages 61–78, Nice, France, June 1990. ACM.

[Wad92] P. Wadler. The essence of functional programming. In *Proceedings 19th Symposium on Principles of Programming Languages*, pages 1–14, January 1992.

[Wad95] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 24–52. Springer, 1995.