Demand-driven Search in Functional Logic Programs

Michael Hanus*
RWTH Aachen

Pierre Réty†
Université d'Orléans

Abstract

In this paper we discuss the advantage of lazy functional logic languages to solve search problems. We show that the lazy evaluation strategy of such languages can be easily exploited to implement a solver that explores only the dynamically demanded parts of the search space. In contrast to pure logic programming, the use of non-deterministic functions enables a modular and simple implementation without the risk of floundering. Furthermore, a local encapsulation of search is useful to avoid the combinatorial explosion of the demanded search space. The necessary features (laziness, non-deterministic functions, encapsulated search) are available in Curry, a new declarative language intended to combine functional and logic programming techniques.

We demonstrate the advantage of this approach with a musical application implemented in Curry: the generation of appropriate chords for the accompaniment of a given melody.

1 Introduction

Declarative programming is motivated by the fact that a higher programming level using powerful abstraction facilities leads to reliable and maintainable software. Thus, declarative programming languages are based on mathematical formalisms and abstract from many details of the concrete hardware and the implementation of the programs on this hardware. For instance, pointers are avoided and replaced by the use of algebraic data types, and complex procedures are split into easily comprehensible parts using pattern matching and local definitions.

Unfortunately, this general view of declarative programming is not supported by an underlying programming paradigm since declarative programming is currently split into the areas of functional programming and logic programming. This situation has negative consequences w.r.t. to teaching (usually, there are different courses on functional programming and logic programming, and students do not see many similarities between them), research (each field has its own community, conferences, and journals, and sometimes similar solutions are developed twice), and applications (each field has its own application areas and some effort

*Informatik II, RWTH Aachen, D-52056 Aachen, Germany, `hanus@informatik.rwth-aachen.de`.
Michael Hanus was partially supported by the German Research Council (DFG) under grant Ha 2457/1-1 and by a grant from the Université d'Orléans (invited professor position number 0542).

†LIFO - Université d'Orléans, B.P. 6759, F-45067 Orléans cedex 2, France, `rety@lifo.univ-orleans.fr`

has been done to show that one paradigm can cover applications of the other paradigm [27] instead of showing the advantages of declarative programming in various application fields, which might be also a reason for the quite limited influence of declarative programming to "real world" computing).

Each paradigm has its advantages (functional programming: nested expressions, efficient evaluation by deterministic (often lazy) evaluation, higher-order functions; logic programming: existentially quantified variables, constraints, partial data structures, built-in search). On the other hand, functional and logic languages have a common kernel and can be seen as different facets of a single idea. For instance, the use of algebraic data types instead of pointers and the definition of local comprehensible cases by pattern matching and local definitions instead of complex procedures are emphasized in functional as well as logic programming. Therefore, many researchers proposed integrated functional logic languages which cover features from functional as well as logic programming (see [10] for a survey). Most of the recent proposals advocate lazy evaluation strategies (e.g., Babel [20], Curry [15], Escher [17], K-LEAF [7]). More recently, a number of interesting techniques and extensions have been developed in this area, like optimal evaluation strategies [2, 3], non-deterministic functions [2, 8], or encapsulated search [14, 24]. This raises the question how these different features can be exploited in practice. In the following we want to answer it.

For this purpose we consider the multi-paradigm language Curry [15], a new declarative language intended to combine functional, logic and concurrent programming paradigms (we ignore the concurrent aspects of Curry here since they are not important for the application area considered in this paper). Since Curry contains the features mentioned above, it is an appropriate basis to discuss the advantages of combining them in a single language. We show that the lazy evaluation of non-deterministic functions can be easily exploited to implement a solver that explores only the dynamically demanded parts of the search space. In contrast to pure logic programming, the use of non-deterministic functions in an integrated language provides for a modular implementation without the risk of floundering. Furthermore, we can avoid the combinatorial explosion of the demanded search space by the encapsulation of local search problems. The latter is also necessary to combine the non-deterministic evaluation strategy from logic programming with the monadic I/O concept [28] from functional programming. We demonstrate the advantages of these programming techniques with a musical application implemented in Curry: a program for the generation of appropriate chords for the accompaniment of a given melody.

In the next section, we introduce the basic features and computation model of Curry. Section 3 discusses general methods to implement demand-driven search strategies. The description of our musical application is contained in Section 4.

2 Basic Features of Curry

This section provides an informal introduction to the computation model and basic features of Curry which are used in the subsequent sections. More details can be found in [11, 14] and in the language definition [15].

A Curry program is a set of functions operating on values described as algebraic data types. To be more precise, we consider *values* as data terms constructed from constants and data constructors. These are introduced through *data type declarations* like[1]

```
data Bool   = True | False
data Nat    = Z    | S Nat
data List a = []   | a : List a
```

`True` and `False` are the Boolean constants, `Z` and `S` are the zero value and the successor function to construct natural numbers,[2] and `[]` (empty list) and `:` (non-empty list) are the constructors for polymorphic lists (`a` is a type variable ranging over all types).

A *data term* is a well-formed expression containing variables, constants and data constructors, e.g., `(S Z)` or `[x,y]` (the latter stands for `x:y:[]`). The meaning of *functions* operating on data terms is specified by *rules* (or *equations*) of the form l `|` c `=` r where l is a *pattern*, i.e., l has the form $f\,t_1 \ldots t_n$ with f being a function, $t_1, \ldots, t_n$ data terms and each variable occurs only once, and r is a well-formed *expression* containing function calls, constants, data constructors and variables from l and c. The (optional) *condition* c is a *constraint* which consists of a conjunction of equations (or other expressions of type `Constraint`) and may contain a list of locally declared variables (which are considered as existentially quantified), i.e., a constraint can have the form "`let` $v_1, \ldots, v_k$ `free in` e_1`&...&`e_n" where the variables v_i are only visible in the equations $e_1, \ldots, e_n$. If a local variable v of a condition should be also visible in the right-hand side, the rule is written as l `|` c `=` r `where` v `free`. A rule can be applied if its condition is satisfiable. A *head normal form* is an expression without a defined function at the root. A *Curry program* is a set of data type declarations and equations.

Example 1 Assume that the above data type declarations are given. Then the following rules define the addition on natural numbers:

```
add Z     n = n
add (S m) n = S(add m n)
```

Using this addition function, we can define the subtraction of natural numbers as follows:

```
sub m n | add n d =:= m  = d  where d free
```

By solving the equation "`add n d =:= m`", the difference `d` between `m` and `n` is computed. □

To compute the *value* of an expression, i.e., a data term which is equivalent (w.r.t. the program rules) to the initial expression, we apply the program rules from left to right to (sub)expressions. For instance, we compute the value of `add (S Z) (S Z)` by applying the rules for addition to this expression:

```
add (S Z) (S Z)  →  S(add Z (S Z))  →  S(S Z)
```

To provide optimal evaluation strategies and to support demand-driven programming techniques (e.g., infinite data structures or demand-driven search), Curry is based on a lazy

[1] The syntax of Curry is a slight extension of Haskell [22].

[2] Curry has also built-in integer values and arithmetic functions. We use here the explicit definition of naturals only to provide some simple and self-contained examples.

evaluation strategy, i.e., if more than one function call occurs in an expression, outermost expressions are evaluated first.[3] For instance, in order to evaluate the expression "`add (add Z (S Z)) Z`", the first subterm `(add Z (S Z))` is the leftmost outermost reducible expression which must be evaluated to head normal form (in this case: `(S Z)`) since its value is required by all rules defining `add` (such an argument is also called *demanded*).

Note that the definition of a "program" as a set of equations allows the definitions of functions which could have more than one result value for a given input. Such functions are called *non-deterministic* and their declarative meaning and use is justified by a framework for non-deterministic rewriting [8].

Example 2 Consider the non-deterministic function `choose` which selects one of its arguments:

```
choose x y = x
choose x y = y
```

The evaluation of the expression "`choose 1 2`" has the result `1` or `2` which is written as

```
choose 1 2  →  1 | 2
```

("`|`" denotes a disjunction). Based on this function, we can define a non-deterministic function `insert` for inserting elements in a list and a non-deterministic function `permute` for computing all permutations of a list:

```
insert x []     = [x]
insert x (y:ys) = choose (x:y:ys) (y:insert x ys)

permute []     = []
permute (x:xs) = insert x (permute xs)
```

Thus, `permute [1,2,3]` evaluates to all permutations of the list `[1,2,3]`:

```
permute [1,2,3]  →*
  [1,2,3] | [2,1,3] | [2,3,1] | [1,3,2] | [3,1,2] | [3,2,1]
```

□

Non-deterministic evaluation is one aspect of logic programming. The other, equally important aspect is the computation with partially instantiated structures and existentially quantified variables. For this purpose, an expression may contain free variables (introduced by a surrounding declaration of the form `let...free` or `where...free`). In this case the expression may not be simply reducible but the free variables must be instantiated in order to apply a reduction step. Fortunately, it requires only a slight extension of the reduction strategy to deal with non-ground expressions and variable instantiation: if the value of a free variable is demanded by the left-hand sides of program rules in order to proceed the computation, the variable is non-deterministically bound to the different demanded values.

Example 3 Consider the function `f` defined by the rules

```
f 0 = 2
f 1 = 3
```

[3] Usually, the leftmost outermost reducible expression is evaluated first, but there are also situations where the evaluation of another outermost reducible expression is preferred, see [11] for details.

Then the expression "`f x`" with the free variable `x` is evaluated to `2` or `3` by binding `x` to `0` or `1`, respectively. □

A *computation step* is called *non-deterministic* if it reduces an expression to a disjunction with more than one alternative, otherwise it is called *deterministic*. Functional programming is the special case where all steps are deterministic and all computed substitutions are the identity. For inductively sequential programs [1] (these are, roughly speaking, function definitions without overlapping left-hand sides), the outlined evaluation method is identical to *needed narrowing* [3] which enjoys several optimality properties. Needed narrowing computes the shortest possible successful derivations (if common subterms are shared) and a minimal set of solutions, and it is fully deterministic if free variables do not occur.

To solve *equations between expressions* containing defined functions, as required by the application of conditional rules (see function `sub` in Example 1), both sides must be reduced to unifiable data terms. For instance, to evaluate the expression "`sub S(S Z) (S Z)`" w.r.t. Example 1, the equation "`add (S Z) d =:= S(S Z)`" is solved by evaluating the left-hand side to `S d` and by unifying it with `S(S Z)` which yields the binding {`d` ↦ `S Z`}. In general, an *equation* or *equational constraint* e_1`=:=`e_2 is satisfied if both sides e_1 and e_2 are reducible to a same data term. As a consequence, if both sides are undefined (non-terminating), then the equality does not hold.[4] Operationally, an equational constraint e_1`=:=`e_2 is solved by evaluating e_1 and e_2 to unifiable data terms where the lazy evaluation of the expressions is interleaved with the binding of variables to constructor terms [18]. Constraints can also be evaluated concurrently similarly to the framework of concurrent constraint programming [23] but we omit the concurrency features of Curry here since they are not important for the application area considered in this paper.

The final feature of Curry which is important in this paper is the *encapsulation of search* [14, 24]. As shown above, the evaluation of expressions might produce non-deterministic steps due to non-deterministic functions or non-deterministic variable bindings. Instead of fixing a particular strategy to explore all alternatives of non-deterministic computation steps, like backtracking in Prolog, Curry is based on a more flexible approach: the programmer can decide how the search space is traversed. For this purpose, Curry has a primitive function `try` with type

```
try :: (a->Constraint) -> [a->Constraint]
```

Thus, `try` takes a search goal as input and produces a list of search goals as output. A *search goal* is a constraint together with an abstracted search variable for which we want to compute values, i.e., if we are interested in solutions for the variable x such that the constraint c is satisfied, then the search goal has the form `\`x`->`c (this is the notation for the lambda abstraction $\lambda x.c$). `try` evaluates the constraint of the argument search goal until the computation finishes or does a non-deterministic step. In the latter case, the different alternatives of the disjunction are returned as a list of search goals so that the programmer can decide which search goal is evaluated next. Based on `try`, one can define various search strategies, like depth-first search, breadth-first search, iterative deepening, or best solution search with branch and bound (see [14, 24]), as ordinary Curry programs. For instance, `all` is

[4] This notion of equality is also known as *strict equality* [7, 20] and is the only reasonable notion of equality in the presence of non-terminating functions.

a search operator which returns a list of all solutions found by a depth-first search strategy. Thus, the expression "`all \x->add (S Z) x =:= S(S Z)`" returns the singleton solution list `[\x->x=:=(S Z)]`. The concrete value for the search variable can be accessed by applying the solved search goal to an unbound variable. This is done in the search operator `findall` [14] which behaves like `all` but yields a list of values (bindings for the search variable) instead of a list of solved search goals, i.e., "`findall \x->add (S Z) x =:= S(S Z)`" evaluates to `[(S Z)]`.

The important aspect of the search operator is not only its possibility to implement easily different search strategies but also its ability to *encapsulate* non-deterministic computations. On the one hand, this is necessary to combine non-deterministic evaluations with a main program doing monadic I/O [14]. On the other hand, this can also avoid the combinatorial explosion of the search space in case of independent subgoals. For instance, consider the predicates $p(x)$ with solutions $\{x \mapsto 0\}, \{x \mapsto 1\}, \{x \mapsto 2\}$ and $q(y)$ with solutions $\{y \mapsto 2\}, \{y \mapsto 3\}, \{y \mapsto 4\}$. The combined goal $p(x) \wedge q(y)$ has the 9 solutions

$$\begin{array}{lll}
\{x \mapsto 0, y \mapsto 2\} & \{x \mapsto 0, y \mapsto 3\} & \{x \mapsto 0, y \mapsto 4\} \\
\{x \mapsto 1, y \mapsto 2\} & \{x \mapsto 1, y \mapsto 3\} & \{x \mapsto 1, y \mapsto 4\} \\
\{x \mapsto 2, y \mapsto 2\} & \{x \mapsto 2, y \mapsto 3\} & \{x \mapsto 2, y \mapsto 4\}
\end{array}$$

This explosion can be avoided by representing the solutions as independent disjunctions, e.g., by evaluating the expression `[all \`x`->`$p(x)$`, all \`y`->`$q(y)$`]` to

```
[[\x->x=:=0,\x->x=:=1,\x->x=:=2], [\y->y=:=2,\y->y=:=3,\y->y=:=4]]
```

Such a "disjunctive" representation of solutions is useful in applications where several good solutions exist for independent subproblems. An example for such an application will be shown in Section 4.

Although many other programming languages also provide techniques to encapsulate search or non-deterministic computations, like `findall/bagof` in Prolog, list comprehensions in Haskell, or the search operators in Oz [24], the important aspect of Curry's search facilities is the embedding into a lazy context. This supports more modularity (as discussed in [14]) and a simple implementation of demand-drive search strategies, which will be discussed in the next section.

3 Programming Demand-driven Search

One of the distinguishing ideas of declarative programming is to describe problems by (executable) specifications instead of (low-level) imperative programs. Therefore, declarative programming, in particular (constraint) logic programming, is often applied in areas where non-deterministic search is used instead of applying exact algorithms to solve the problem (which are often unknown). The problem of programming with search is to formulate the program in such a way that the explored search space is not too large in order to compute a solution. In this section we discuss various known methods to implement search problems before we describe a demand-driven method which can be easily implemented in a functional logic language like Curry.

Since the theory of logic programming is based on the non-deterministic resolution principle (which is implemented in Prolog by backtracking), logic programming is often considered as an appropriate language to implement search problems. The simplest search method well known in algorithm design is generate-and-test. The idea is to provide a non-deterministic generator which generates candidate solutions to the problem and a tester which checks whether a candidate solution is an actual solution to the problem. In Prolog [25], one can express the combination of the generator and the tester to solve a search problem by the clause

```
solve(X) :- generate(X), test(X).
```

Due to the left-to-right selection strategy of Prolog, the generator `generate` binds `X` to a first candidate solution which is then checked by `test(X)`. If this test fails, the next candidate solution is computed by backtracking, tested and so on. For instance, a naive program to sort a list `Xs` by enumerating and testing all permutations of `Xs` is expressed as follows (see [25, Program 3.20] for the concrete definition of `permute` and `ordered`):

```
psort(Xs,Ys) :- permute(Xs,Ys), ordered(Ys).
```

Generate-and-test programs in this naive form are often highly inefficient since all candidate solutions are completely generated by backtracking before each of them is tested (the above program has a complexity of $O(n!)$ for an input list of length n).

The techniques to implement backtracking in a lazy functional language [27] are not very helpful to avoid the complete exploration of the search space. Backtracking can be programmed in a functional language by implementing the non-deterministic generator as a function which returns the list of all candidate solutions and the tester as a filter on this list. For instance, if the function `perms` returns the list of all permutations of its argument list and the Boolean function `sorted` is true if its argument is a sorted list, then we can compute the sorted lists by the following list comprehension in Haskell [22]:

```
psort xs = [ys | ys<-perms xs, sorted ys]
```

The lazy evaluation strategy of Haskell has the effect that some parts of the permutations are not considered if they already start in a wrong order (e.g., `[4,3,...]`). However, the entire list of all permutations is generated since `sorted` acts as a filter on all list elements, i.e., the complexity is still $O(n!)$.

The efficiency of generate-and-test can be improved by intertwining the tester with the generator by "pushing" the tester inside the generator. This is possible by changing the program code for each individual generate-and-test program, but it is not satisfying since it requires a lot of effort for each program and it decreases the modularity of the original generate-and-test formulation. In Prolog systems with coroutining, one can intertwine the tester and generator without loosing modularity by transforming generate-and-test into test-and-generate [21], i.e., the general scheme is changed to

```
solve(X) :- test(X), generate(X).
```

where the predicate `test` is delayed until its argument is sufficiently instantiated to check the solution in a deterministic way. This is possible by adding a "wait" declaration (or `when`, `freeze`, `block` etc. in other Prolog systems) to the predicate `test`. For instance, we can rewrite the clause for `psort` above as

```
psort(Xs,Ys) :- ordered(Ys), permute(Xs,Ys).
```

and add a wait declaration to `ordered`. Then the goal `?- psort([4,3,2,1],S)` is executed in the following way: After applying the above clause to this goal, the literal `ordered(S)` is delayed and the literal `permute([4,3,2,1],S)` will be evaluated. If `S` is bound to the first part of a permutation of `[4,3,2,1]` (i.e., a list with two elements and a variable at the tail), then `ordered(S)` is activated. If the first two elements of `S` are in the wrong order, then the computation fails and another permutation is tried, otherwise `ordered` is delayed again until the next part of the permutation is generated. Hence, not all permutations are completely computed and therefore the execution time is better than in the generate-and-test approach. Naish [21] presented an algorithm which generates the wait declarations from a given program and transforms the program by reordering the goals in a clause. Although this approach seems to be attractive, it has some problems. For instance, the generation of wait declarations is based on heuristics and therefore it is unclear whether these heuristics are generally successful. Moreover, it is possible that the annotated program flounders, i.e., all subgoals are delayed which is considered as a run-time error. Hence completeness of SLD-resolution can be lost when transforming a logic program into a program with wait declarations (see [9, 26] for some examples). Other interesting approaches to decrease the search space of logic programs by improving the control behavior are presented in [5, 26] but they are also based on heuristics and complex program analysis techniques.

We argue that the use of functional logic languages provides much simpler and modular methods to reduce the search space. This was discussed for the first time in [6] and confirmed by results from a practical implementation in [9]. These improvements were based on functional logic languages (ALF [9], SLOG [6]) that combine an overall eager evaluation strategy with a simplification of goals between non-deterministic evaluation steps. This allows to keep the general generate-and-test formulation of the programs, but the simplification has the effect that partially instantiated test expressions like `ordered([4,3|Xs])` are reduced to `false` which causes the failure of that computation branch. Due to the existing completeness results for this strategy, the incompleteness (floundering) problems of the "test-and-generate" technique above are avoided. The remaining problem of this technique is the generation of appropriate rules for simplification ([9] discusses methods for this).

If we use a functional logic language with a lazy evaluation strategy, we can easily implement a demand-driven search strategy without complicated simplification rules. The idea is to implement the generator as a non-deterministic function which computes all candidate solutions and the tester as a function taking a candidate solution and yielding an appropriate result if its argument is an actual solution to the problem. Thus, the problem is solved by evaluating the expression

```
test(generate)
```

Due to lazy evaluation, the argument `generate` is only evaluated as requested by the tester, i.e., only those parts of the search space are explored that are necessary to compute the actual solutions to the problem. Thus, we obtain a *demand-driven search strategy in a modular way* (i.e., without intertwining the actual code for the generator and tester).

Example 4 Consider the non-deterministic function `permute` defined in Example 2 which is a generator for permutations. The following function `sorted`, which is the identity on ascending sorted lists, acts as a tester:

```
sorted []  = []
sorted [x] = [x]
sorted (x:y:ys) | x<=y  = x : sorted (y:ys)
```

Thus, we obtain the entire solution to the sorting problem by:

```
psort xs = sorted (permute xs)
```

Operationally, this implementation behaves (simplified) as follows. Consider the evaluation of `psort` [n,$n-1$,...,2,1]. By definition of `sorted`, at least the first two elements of the list `permute` [n,...,2,1] must be computed in order to apply a rule to the call for `sorted`. Thus, one alternative evaluates `permute` [n,...,2,1] to n:$n-1$:`permute` [$n-2$,...,2,1]. This alternative is immediately discarded, since `sorted` is not defined on such a list, which completely avoids the enumeration of the permutations of [$n-2$,...,2,1] (i.e., $(n-2)!$ alternatives). The following table contains the number of alternatives in the search space for the generate-and-test implementation and this demand-driven implementation "test-of-generate" (the initial list has always the form [n,$n-1$,...,2,1]).

Length of the list:	4	5	6	7	8	9	10
generate-and-test	24	120	720	5040	40320	362880	3628800
test-of-generate	19	59	180	544	1637	4917	14758

Thus, the "test-of-generate" approach yields the same size of the search space as in the "test-and-generate" approach in Prolog with coroutining, as described above, but without the risk of floundering. □

Note that the demand-driven generation of the search space is independent of the actual strategy to traverse the search space like depth-first or breadth-first search. Therefore, we will combine the demand-driven search technique with the encapsulation of search to avoid a combinatorial explosion caused by combining the results of independent subproblems (as discussed in the previous section) in our application which will be presented in the next section.

4 A Musical Application

4.1 The Problem

The problem to be solved by our application is easily stated: For a given melody of a piece of music, which is provided by the user, an appropriate accompaniment composed of chords should be computed. For instance, these chords can be used to accompany the melody with a guitar. For each bar of the melody, the program should propose one or several choices for the chords (frequently, one major chord and one minor chord), among which the musician can choose according to his/her sensibility. Nevertheless, all proposed chords should be in harmony with given melody, i.e., big dissonances should be avoided. A change of a chord is allowed at the beginning of each half bar, if useful from a musical point of view. Otherwise, the same chord may cover a whole bar or even several consecutive bars.

The **input** to the program is the *melody* represented as a list of bars. A *bar* is a list of pairs of the form `(note,dur)` where `dur` is the duration of the note, i.e., an integer in the interval [1..8]. The duration 1 denotes a eighth note, so the duration 8 denotes a whole note, i.e., a note that fills an entire 4/4 bar. Thus, the sum of all durations in each bar must be 8 where rests can be included like notes.[5]

The **output** of the program is a list of accompaniment choices for each bar of the input. Each accompaniment choice contains one or several choices for the chords which are represented similarly as the input melody. The different choices are separated by the symbol "`||`" on the printed output and each accompaniment choice is printed in one line. For instance, the output line

```
F_maj/8 || A_min/4 D_min/4
```

means that the F major chord suits for the whole bar but also A minor for the first half bar and D minor for the second one.

4.2 Musical Rules Used in our Solution

The musical harmonization rules depend on the style of music. We consider here the style of traditional songs. Each bar is considered independently of the other since only the melody inside a bar is relevant for the chords in that bar (with a possible exception for the first or final bar of a song). The general idea is to compute the chords with a minimal dissonance between the notes in the chord and the notes in the melody while playing this chord. For this purpose, we use a local criterion for each chord and a global criterion for each bar.

The local criterion. A chord C fits to a melody bar or half-bar M, if every note included in C belongs to the scale of M[6] and the dissonance between the notes of C and those of M is not greater than some bound.

When two notes are sounding at the same time, there is a dissonance[7] if the distance between them is 1/2 or 1 tone. In other words, the unison as well as a bigger distance causes no dissonance. The dissonance between a given note n and a chord C is the dissonance of the smallest distance between n and the notes of C. The *dissonance value* between n and C is 1 in case of a dissonance and otherwise 0. The dissonance value is also 0 if n is a rest.

Since both M and C may contain several notes, there is necessarily some quantity of dissonance when C is sounding while M is being played. However, to be pleasant, this quantity must not be too large. Formally, if M is `[(`n_1`,`d_1`),...,(`n_p`,`d_p`)]` ($d_1, \ldots, d_p$ are the durations of the notes $n_1, \ldots, n_p$) and $diss_i$ denotes the dissonance value between n_i and C, we define the *dissonance value* $dv(C, M)$ between chord C and melody M by

$$dv(C, M) = 3(\sum_{i=1}^{p} diss_i * d_i) + diss_{1or2}$$

where $diss_{1or2}$ is $diss_1$, if n_1 is not a rest, and $diss_2$ otherwise. By adding $diss_{1or2}$, the first non-silence note has a slightly stronger weight.

[5]For the sake of simplicity, we consider only 4/4 bars and melodies written in the C major scale (or A minor) without accidentals, but our solution can be easily extended to other tonalities and kinds of bars.

[6]With our current restrictions, this means that C contains neither flat nor sharp notes.

[7]This causes an unpleasant effect for the listener.

The *bound* (the maximal allowed dissonance value for a chord) is $d'_1 + \cdots + d'_p$, where $d'_i = d_i$ if n_i is not a rest, otherwise $d'_i = 0$. Since we defined that there is no dissonance between a chord and a rest, we must not take into account rests when computing the bound. Note that if there is no rest, the bound is equal to the duration of M.

Example 5 Consider the melody half bar `[(C,2),(A,1),(G,1)]`. The C major chord contains the notes `C`, `E`, `G`. Thus, the only dissonance between it and the melody comes from the note `(A,1)` which yields the dissonance value 3. The bound for this half-bar is 4 since it does not include any rests. Therefore, the C major chord satisfies the local criterion. □

The global criterion. This criterion becomes relevant if there is more than one chord in a bar in order to obtain a nice accompaniment. Therefore, the global criterion is identical to the local if M is an entire melody bar with a single chord of the duration of the bar. On the other hand, if no single chord satisfies the local criterion, it is necessary to split the melody bar M into two halves M_1 and M_2 and to compute two chords C_1 and C_2 that fit to M_1 and M_2, respectively. From a musical point of view, we cannot deduce immediately that the composition $C_1 \cdot C_2$ fits to the entire bar M. We require that the use of two chords C_1 and C_2 for covering a bar should produce a global dissonance less than if using one chord, since the possibility of changing chords at the middle of the bar allows to get chords that fit better. If M does not contain any rest, the local criterion forces both dissonance values $dv(C_1, M_1)$ and $dv(M_2, C_2)$ to be less or equal to 4. Our global criterion defines that the sum $dv(C_1, M_1) + dv(M_2, C_2)$ must be less or equal to 6. As a consequence, C_1 and C_2 cannot be considered independently. For instance, if C'_1 also fits to M_1, then `[(`C_1`,4),(`C_2`,4)]` may be a solution whereas `[(`C'_1`,4),(`C_2`,4)]` may be not due to the global criterion.

However, for some melody bars it is even not possible to find an accompaniment with two chords because it would produce too big dissonances. Thus, we also allow the use of four chords in one bar if the previous global criterion cannot be satisfied, but in this case we require that the sum of all dissonance values in this bar must be less or equal to 4 in order to obtain an appropriate accompaniment.

Example 6 Consider the melody bar `[(C,2),(A,1),(G,1),(B,1),(F,1),(D,2)]` and the chord bar `[(C_maj,4), (D_min,4)]`. For the chord `C_maj`, the local dissonance is 3, as already seen in Example 5. The chord `D_min` contains the notes `D`, `F`, `A`. The dissonance between it and the second half bar comes from the note `(B,1)`, which yields the dissonance value 3+1 since `(B,1)` is the first note of this half bar. Both chords satisfy the local criterion, but the chord bar does not satisfy the global criterion since the global dissonance value is 7. □

4.3 The Implementation

In this section we sketch our implementation and explain how it profits from the described features of the functional logic language Curry: non-deterministic functions, laziness, and encapsulated search.

The *data types* `Note` and `Chord` define the possible notes (`R` denotes a rest) and chords occurring in the program:

```
data Note = C | D | E | F | G | A | B | R

data Chord = C_maj | D_maj | ... | B_min
```

Thus, the melody is a list of bars where each bar is a list of note/duration pairs, i.e., the main program takes an argument of type `[[(Note,Int)]]`.

As already mentioned, our program deals with each bar independently of the others. For each bar, it behaves as a generate-and-test solver. The generation function non-deterministically produces all chords that belong to the C major scale, and the test function filters the chords that fit to the melody bar, i.e., it is a partial identity function defined for convenient accompaniments, similarly to the function `sorted` in Example 4. Thus, the generators are defined as

```
aChord = C_maj
aChord = F_maj
...
aChord = A_min

one_chord_bar  = [(aChord, 8)]
two_chord_bar  = [(aChord, 4), (aChord, 4)]
four_chord_bar = [(aChord, 2), (aChord, 2), (aChord, 2), (aChord, 2)]
```

Using a non-deterministic function for generating candidate solutions becomes relevant for bars with more than one chord (i.e., `two_chord_bar` and `four_chord_bar` here and in possible extensions for other rhythms) since the second and further chords are only generated if the previous chords satisfy the local criterion.

¿From a musical point of view, it is useless to change the chord at the middle of a bar if there exists a chord that can fit the whole bar. In other words, we look for solutions of the type "one chord per half bar" only if there is no solution of the type "one chord per bar". This requirement can be easily implemented by the encapsulation of the local search. Thus, the main function to compute all appropriate chords for a melody bar is defined as follows:

```
compute_bar :: [(Note, Int)] -> [([(Chord, Int)],Int)]
compute_bar mbar =
  if one_chord_solutions == []
  then if two_chord_solutions == []
       then bestOf (findall \x -> checkBarDiss four_chord_bar mbar =:= x)
       else bestOf two_chord_solutions
  else bestOf one_chord_solutions
    where one_chord_solutions =
                findall \x -> checkBarDiss one_chord_bar mbar =:= x
          two_chord_solutions =
                findall \x -> checkBarDiss two_chord_bar mbar =:= x
```

The tester `checkBarDiss` takes a chord bar and a melody bar as input and yields the input chord bar as output together with the overall dissonance value if the chord bar satisfies the local and global criteria presented in Section 4.2. The dissonance value is used to filter the best solutions with the smallest dissonance values (up to a variance of 1), which is done by the function `bestOf`. Due to this implementation, the alternatives with more chords per bar are only computed if the list of solutions w.r.t. less chords are empty.

The encapsulation of search is also necessary to avoid the combinatorial explosion of solutions by combining all solutions for each bar. The function `compute_bar` computes a list of the best solutions for each bar (usually 2 or 3 different solutions). Combining these solutions for a complete melody with n bars, as done in a simple Prolog implementation, would result in approximately 2^n different solutions. Instead of this naive approach, we present the alternative solutions for each single bar, as explained in Section 4.1, so that the output size is linear in the input size. Thus, the main program applies `compute_bar` to each bar of the given melody and prints the result in a readable way.[8] The main function `run` of our implementation is defined as

```
run melody = foldr (>>) done (map (print_chord_alts . compute_bar) melody)
```

where `print_chord_alts` maps the alternative chords for a bar (computed by `compute_bar`) into an appropriate print action to produce the output shown in Section 4.1 (see [28] for a description of the monadic I/O technique).

This implementation can be extended in various ways. For instance, one could include other tonalities, rhythms or beats. Another interesting extension is the connection of our system with the Haskore libraries [16] in order to play the solutions computed by our system.

4.4 Benchmarks

To evaluate the practical effect of using a demand-driven search strategy in our application, we have compared the runtimes of the demand-driven search strategy with a strict search strategy where all alternatives are computed. The entire application is written in Curry and compiled into Java with the compiler described in [13]. The following table shows the runtimes in seconds on a Sun Ultra-2 system and the speedups due to the demand-driven search strategy for different melodies. Details about the melodies can be found in the appendix.

Melody (number of chords):	Sounds (14)	Nicolas (16)	Homeward (16)	French (8)
strict search:	3.2	1.1	65.9	48.5
demand-driven search:	1.3	0.7	1.6	2.0
speedup:	2.4	1.6	41.2	24.3

The accompaniments for the first two melodies do not contain more than two chords per bar, whereas these for the last two melodies need also four chords in some bars. Note that an implementation of this application in Prolog with `findall` or in Haskell with list comprehensions would have a similar effect as the strict search strategy, as discussed in Section 3. Thus, the numbers in this table shows that the use of a demand-driven search strategy is essential for this and similar applications.

[8] The use of the monadic I/O technique [28] to print the final results is another argument for the necessity of encapsulating the local search performed for each bar.

5 Conclusions

We have discussed the advantages of functional logic languages with a lazy evaluation strategy to solve search problems. By the use of generators implemented as non-deterministic functions, one can implement a demand-driven generation of the search space in a simple and modular way. Moreover, the local encapsulation of search avoids the combinatorial explosion caused by the combination of independent subproblems and is an appropriate programming technique to combine non-deterministic logic-based computations with the functional concept of doing input/output. Furthermore, we have shown how these programming techniques can be fruitfully used in a musical application, namely the generation of appropriate chords for the accompaniment of a given melody.

Since we are not aware of precise musical rules to compute good accompaniments for songs, we have developed the criteria presented in Section 4.2 by our own. The use of a high-level declarative language like Curry was very useful to find these criteria since it supported the straightforward implementation and test of different criteria and measurements. We have tested our criteria with existing songs and the results indicate that the criteria are appropriate (see [12] for details).

For future work we plan to integrate further constraint systems than the current equations between data terms into Curry, following the proposals [4, 19]. This opens the possibility to exploit lazy evaluation and demand-driven search strategies also in applications from constraint logic programming.

Acknowledgements. The authors are grateful to Frank Steiner for his comments on this paper and for providing the implementation of encapsulated search.

References

[1] S. Antoy. Definitional Trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 143–157. Springer LNCS 632, 1992.

[2] S. Antoy. Optimal Non-Deterministic Functional Logic Computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pp. 16–30. Springer LNCS 1298, 1997.

[3] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pp. 268–279, Portland, 1994.

[4] P. Arenas-Sánchez, T. Hortalá-González, F.J. López-Fraguas, and E. Ullán-Hernández. Functional Logic Programming with Real Numbers. In *Proc. JICSLP'96 Workshop on Multi-Paradigm Logic Programming*, pp. 47–57. TU Berlin, Technical Report No. 96-28, 1996.

[5] M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling Control. *Journal of Logic Programming (6)*, pp. 135–162, 1989.

[6] L. Fribourg. SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 172–184, Boston, 1985.

[7] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel LEAF: A Logic plus Functional Language. *Journal of Computer and System Sciences*, Vol. 42, No. 2, pp. 139–185, 1991.

[8] J.C. Gonzáles-Moreno, M.T. Hortalá-Gonzáles, F.J. López-Fraguas, and M. Rodríguez-Artalejo. A Rewriting Logic for Declarative Programming. In *Proc. ESOP'96*, pp. 156–172. Springer LNCS 1058, 1996.

[9] M. Hanus. Improving Control of Logic Programs by Using Functional Logic Languages. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pp. 1–23. Springer LNCS 631, 1992.

[10] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.

[11] M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.

[12] M. Hanus and P. Réty. Demand-driven Search in Functional Logic Programs. Research Report RR-LIFO-98-08, Univ. Orléans, 1998.

[13] M. Hanus and R. Sadre. An Abstract Machine for Curry and its Concurrent Implementation in Java. *Journal of Functional and Logic Programming (to appear)*, 1999. Previous version in: Proc. ILPS'97 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages.

[14] M. Hanus and F. Steiner. Controlling Search in Declarative Programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pp. 374–390. Springer LNCS 1490, 1998.

[15] M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.5). Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1999.

[16] P. Hudak. Haskore Music Tutorial. Technical Report, Yale University, 1997.

[17] J.W. Lloyd. Combining Functional and Logic Programming Languages. In *Proc. of the International Logic Programming Symposium*, pp. 43–57, 1994.

[18] R. Loogen, F. Lopez Fraguas, and M. Rodríguez Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pp. 184–200. Springer LNCS 714, 1993.

[19] F.J. López Fraguas. A General Scheme for Constraint Functional Logic Programming. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 213–227. Springer LNCS 632, 1992.

[20] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The Language BABEL. *Journal of Logic Programming*, Vol. 12, pp. 191–223, 1992.

[21] L. Naish. *Negation and Control in Prolog.* Springer LNCS 238, 1987.

[22] J. Peterson et al. Haskell: A Non-strict, Purely Functional Language (Version 1.4). Technical Report, Yale University, 1997.

[23] V.A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

[24] C. Schulte and G. Smolka. Encapsulated Search for Higher-Order Concurrent Constraint Programming. In *Proc. of the 1994 International Logic Programming Symposium*, pp. 505–520. MIT Press, 1994.

[25] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Massachusetts, 2nd edition, 1994.

[26] K. Verschaetse, D. De Schreye, and M. Bruynooghe. Generation And Compilation of Efficient Computation Rules. In *Proc. Seventh International Conference on Logic Programming*, pp. 700–714. MIT Press, 1990.

[27] P. Wadler. How to Replace Failure by a List of Successes. In *Functional Programming and Computer Architecture*, pp. 113–128. Springer LNCS 201, 1985.

[28] P. Wadler. How to Declare an Imperative. In *Proc. of the 1995 International Logic Programming Symposium*, pp. 18–32. MIT Press, 1995.

A Examples

The following examples are written in the tonality of C major (or A minor), which is not necessarily the one used in the actual records.

Example 7 "The Sounds of silence" (Simon and Garfunkel):

The melody written as a list of melody bars is as follows (R denotes a rest):

```
[[(R,2),(A,1),(A,1),(C,1),(C,1),(E,1),(E,1)], [(D,8)],
 [(R,2),(G,1),(G,1),(B,1),(B,1),(D,1),(D,1)], [(C,8)],
 [(R,2),(C,1),(C,1),(E,1),(E,1),(G,1),(G,1)], [(A,2),(A,1),(G,5)],
 [(R,2),(C,1),(C,1),(E,1),(E,1),(G,1),(G,1)], [(A,2),(A,1),(G,5)],
 [(R,2),(C,1),(C,1),(A,3),(A,1)], [(A,2),(A,1),(B,1),(C,3),(C,1)],
 [(C,1),(B,1),(A,2),(G,4)], [(R,2),(A,1),(G,1),(E,4)],
 [(R,1),(A,1),(A,1),(C,1),(G,4)], [(R,1),(B,3),(C,1),(A,3)]];
```

The tunings chosen by Paul Simon are (written as a list of chord bars):

```
[[(A_min,8)], [(G_maj,8)], [(G_maj,8)],
 [(A_min,8)], [(A_min,4),(C_maj,4)], [(F_maj,4),(C_maj,4)],
 [(C_maj,8)], [(F_maj,4),(C_maj,4)], [(C_maj,4),(F_maj,4)],
 [(F_maj,8)], [(F_maj,4),(C_maj,4)], [(C_maj,8)],
 [(A_min,4),(C_maj,4)], [(G_maj,4),(A_min,4)]]
```

Our program finds this solution, up to a few slight differences. The output of our program is:

```
Amin/8
Gmaj/8 || Dmin/8
Gmaj/8 || Gmaj7/8
Fmaj/8 || Amin/8
Cmaj/8
Fmaj/4 Cmaj/4 || Fmaj/4 Emin/4 || Amin/4 Cmaj/4 || Amin/4 Emin/4
Cmaj/8
Fmaj/4 Cmaj/4 || Fmaj/4 Emin/4 || Amin/4 Cmaj/4 || Amin/4 Emin/4
Fmaj/8 || Amin/8
Fmaj/8 || Amin/8
Fmaj/4 Cmaj/4 || Fmaj/4 Emin/4 || Amin/4 Cmaj/4 || Amin/4 Emin/4
Cmaj/8 || Emin/8 || Amin/8
Cmaj/8
Gmaj/4 Fmaj/4 || Gmaj/4 Amin/4 || Emin/4 Fmaj/4 || Emin/4 Amin/4
```

Note that without this "disjunctive" representation by the use of encapsulated search, 24576 different accompaniments would be computed for this melody. □

Example 8 "Nicolas and Bart" (by J. Baez):
The melody written as a list of melody bars is as follows:

```
[[(E,6),(C,2)], [(D,6),(B,2)], [(C,2),(B,2),(A,4)], [(B,6),(R,2)],
 [(E,6),(C,2)], [(D,6),(B,2)], [(C,2),(B,2),(A,4)], [(D,6),(R,2)],
 [(G,6),(E,2)], [(F,6),(D,2)], [(D,2),(E,2),(F,4)], [(E,6),(R,2)],
 [(E,6),(C,2)], [(D,6),(B,2)], [(C,4),(B,4)], [(A,8)]];
```

The tunings chosen by J. Baez are :

```
[[(C_maj,8)], [(G_maj,8)], [(A_min,8)], [(G_maj,8)],
 [(C_maj,8)], [(G_maj,8)], [(A_min,8)], [(G_maj,8)],
 [(C_maj,8)], [(D_min,8)], [(G_maj7,8)], [(C_maj,8)],
 [(C_maj,8)], [(D_min,8)], [(A_min,4),(E_maj,4)], [(A_min,8)]]
```

Our program finds this solution, up to one difference. The last but one tuning E˙maj includes the foreign note G sharp. However replacing `Emin` by `Emaj` is frequent, although not systematic. This scenario is not included in our program. It then yields `Emin` for this half-bar.

```
Cmaj/8
Gmaj/8 Dmin/8
Fmaj/8 || Amin/8
Gmaj/8 || Emin/8
Cmaj/8
Gmaj/8 || Dmin/8
Fmaj/8 || Amin/8
Gmaj/8 || Dmin/8
Cmaj/8 || Emin/8
Dmin/8
Gmaj7/8 || Dmin/8
Cmaj/8
Cmaj/8
Gmaj/8 || Dmin/8
Fmaj/4 Gmaj/4 || Fmaj/4 Emin/4 || Amin/4 Gmaj/4 || Amin/4 Emin/4
Fmaj/8 || Amin/8
```

□

Example 9 "Homeward bound" (by Simon and Garfunkel) :
We only consider the chorus, not the entire melody :

```
[[(C,4),(B,2),(C,2)], [(A,8)],
 [(R,2),(G,2),(A,2),(G,2)], [(E,1),(D,1),(C,4),(R,2)],
 [(C,4),(B,2),(C,2)], [(A,8)], [(A,8)], [(E,2),(R,3),(E,1),(E,1),(F,1)],
 [(F,2),(E,2),(D,2),(C,2)], [(E,2),(R,3),(E,1),(E,1),(F,1)],
 [(F,2),(E,2),(D,2),(C,2)], [(E,2),(R,3),(E,1),(E,1),(F,1)],
 [(F,2),(E,2),(D,2),(C,2)], [(D,2),(E,2),(D,4)],
 [(C,2),(G,6)], [(G,8)]]
```

The chords chosen by Paul Simon are :

```
[[(C_maj,8)], [(F_maj,8)],
 [(F_maj,8)], [(C_maj,8)],
 [(C_maj,8)], [(F_maj,8)], [(F_maj,8)], [(C_maj,8)],
 [(D_min,2),(C_maj,2),(B_flat_maj,2),(F_maj,2)], [(C_maj,8)],
 [(D_min,2),(C_maj,2),(B_flat_maj,2),(F_maj,2)], [(C_maj,8)],
 [(D_min,2),(C_maj,2),(B_flat_maj,2),(F_maj,2)], [(G_maj7,8)],
 [(C_maj,8)], [(C_maj,8)]]
```

Our program finds this solution, except for the third bar, for which Paul Simon has chosen the same chord as in the second bar, i.e. `Fmaj`, although this causes a big dissonance with the melody. Besides, `B_flat_maj` cannot be generated by our program.

```
Cmaj/8 || Fmaj/8 || Amin/8
Fmaj/8 || Amin/8
Cmaj/8 || Gmaj/8 || Gmaj7/8 || Emin/8
Cmaj/8 || Amin/8
Cmaj/8 || Fmaj/8 || Amin/8
Fmaj/8 || Amin/8
Fmaj/8 || Amin/8
Cmaj/8 || Emin/8 || Amin/8
Dmin/2 Cmaj/2 Gmaj/2 Fmaj/2 || Dmin/2 Cmaj/2 Gmaj/2 Amin/2
              || Dmin/2 Cmaj/2 Dmin/2 Fmaj/2 || Dmin/2 Cmaj/2 Dmin/2 Amin/2
Cmaj/8 || Emin/8 || Amin/8
Dmin/2 Cmaj/2 Gmaj/2 Fmaj/2 || Dmin/2 Cmaj/2 Gmaj/2 Amin/2
              || Dmin/2 Cmaj/2 Dmin/2 Fmaj/2 || Dmin/2 Cmaj/2 Dmin/2 Amin/2
Cmaj/8 || Emin/8 || Amin/8
Dmin/2 Cmaj/2 Gmaj/2 Fmaj/2 || Dmin/2 Cmaj/2 Gmaj/2 Amin/2
              || Dmin/2 Cmaj/2 Dmin/2 Fmaj/2 || Dmin/2 Cmaj/2 Dmin/2 Amin/2
Gmaj/8 || Gmaj7/8 || Dmin/8
Cmaj/8 || Emin/8
Cmaj/8 || Emin/8
```

□

Example 10 A french song:

The melody:

```
[[(E,2),(E,1),(C,1),(D,3),(E,1)], [(C,2),(B,2),(A,1),(B,1),(A,2)],
 [(E,2),(E,1),(C,1),(D,3),(E,1)], [(C,2),(D,2),(E,1),(D,1),(E,2)],
 [(A,2),(A,1),(A,1),(G,2),(F,1),(E,1)], [(F,2),(A,2),(E,4)],
 [(D,2),(E,1),(D,1),(C,2),(B,1),(A,1)], [(C,2),(D,2),(B,4)]]
```

The chords chosen by the author are:

```
[[(C_maj,4),(D_min,4)], [(A_min,2),(E_maj,2),(A_min,4)],
 [(C_maj,4),(D_min,4)], [(F_maj,2),(G_maj,2),(C_maj,4)],
 [(F_maj,4),(G_maj,4)], [(D_min,4),(C_maj,4)],
 [(D_min,4),(A_min,4)], [(D_min,4),(E_maj,4)]]
```

Our program finds this solution, except that :

- it finds `Emin` instead of `Emaj`, like in Example 8.
- for the last bar as well as the last but two bars, it generates one chord for the whole bar, with the dissonance 7, that is to say a dissonance value close to the bound 8.

```
Cmaj/4 Gmaj/4 || Cmaj/4 Dmin/4
Fmaj/2 Gmaj/2 Fmaj/2 Fmaj/2 || Fmaj/2 Gmaj/2 Fmaj/2 Amin/2
              || Fmaj/2 Gmaj/2 Amin/2 Fmaj/2 || Fmaj/2 Gmaj/2 Amin/2 Amin/2
              || Fmaj/2 Emin/2 Fmaj/2 Fmaj/2 || Fmaj/2 Emin/2 Fmaj/2 Amin/2
              || Fmaj/2 Emin/2 Amin/2 Fmaj/2 || Fmaj/2 Emin/2 Amin/2 Amin/2
              || Amin/2 Gmaj/2 Fmaj/2 Fmaj/2 || Amin/2 Gmaj/2 Fmaj/2 Amin/2
              || Amin/2 Gmaj/2 Amin/2 Fmaj/2 || Amin/2 Gmaj/2 Amin/2 Amin/2
              || Amin/2 Emin/2 Fmaj/2 Fmaj/2 || Amin/2 Emin/2 Fmaj/2 Amin/2
              || Amin/2 Emin/2 Amin/2 Fmaj/2 || Amin/2 Emin/2 Amin/2 Amin/2
Cmaj/4 Gmaj/4 || Cmaj/4 Dmin/4
Fmaj/2 Gmaj/2 Cmaj/2 Cmaj/2 || Fmaj/2 Dmin/2 Cmaj/2 Cmaj/2
              || Amin/2 Gmaj/2 Cmaj/2 Cmaj/2 || Amin/2 Dmin/2 Cmaj/2 Cmaj/2
Fmaj/4 Cmaj/4 || Fmaj/4 Gmaj/4 || Fmaj/4 Emin/4 || Amin/4 Cmaj/4
       || Amin/4 Gmaj7/4 || Amin/4 Emin/4
Amin/8
Gmaj/4 Fmaj/4 || Gmaj/4 Amin/4 || Gmaj7/4 Fmaj/4 || Gmaj7/4 Amin/4
       || Dmin/4 Fmaj/4 || Dmin/4 Amin/4
Gmaj/8 || Gmaj7/8
```

□