



Project
MUSE[®]

Today's Research. Tomorrow's Inspiration.


Constraint Application with Higher-Order Programming for Modeling Music Theories

Torsten Anders
Eduardo R. Miranda

Computer Music Journal, Volume 34, Number 2, Summer 2010,
pp. 25-38 (Article)

Published by The MIT Press



 For additional information about this article
<http://muse.jhu.edu/journals/cmj/summary/v034/34.2.anders.html>

Torsten Anders and Eduardo R. Miranda

Interdisciplinary Centre for Computer Music Research (ICCMR)
University of Plymouth
Drake Circus
Plymouth PL4 8AA, United Kingdom
{torsten.anders, eduardo.miranda}@plymouth.ac.uk

Modeling music theories with computer programs has attracted composers and scholars for a long time. On the one hand, the resulting programs can serve as algorithmic composition tools. On the other hand, such an approach leads to a better understanding of existing as well as newly developed theories, which in turn can lead to a better understanding of music, as well as to better ways to retrieve music from databases.

Constraint programming (Apt 2003) has often been used to create computational models of music theories and composition. Constraint-based harmonization systems were surveyed by Pachet and Roy (2001); other examples include purely rhythmic tasks (Sandred 2003), Fuxian counterpoint (Schottstaedt 1989), Ligeti-like textures (Chemillier and Truchet 2001; Laurson and Kuuskankare 2001), and instrument-specific writing (Laurson and Kuuskankare 2001).

Many music-constraint systems have been proposed in which users implement their own music theory models. Two seminal systems are PWConstraints (Laurson 1996) and Situation (Rueda et al. 1998). Carla (Courtot 1990) is one of the earliest systems. Other examples include the aggregation of the music representation MusES with the constraint system BackTalk (Pachet and Roy 1995), Arno (Anders 2000), OMClouds (Truchet and Codognot 2004), and Örjan Sandred's PWMC defined on top of PWConstraints (Sandred 2010, in this issue). A survey of music-constraint programming in general and a detailed comparison of existing systems is provided by Anders and Miranda (in press).

Each system provides the following components, which are essential for solving musical *constraint-satisfaction problems* (CSP). It defines a *music representation* (score) where some aspects

Constraint Application with Higher-Order Programming for Modeling Music Theories

are represented as *variables* (unknowns). For example, the durations of notes in the score or the underlying harmonic structure may be unknown. Variables have a *domain*—a set of values they may take in a solution. The relations between these variables are restricted by *constraints* (rules), and a music-constraint system provides some mechanism for applying constraints to variables in the music representation. A CSP usually presents a combinatorial problem. A *constraint solver* finds one or more solutions for the problem. In a *solution*, the domain of each variable is reduced to a single value that is consistent with all of its constraints.

Nevertheless, existing systems differ widely; for example, they define diverse music representations. As a result, different systems are suitable for different types of CSPs. For instance, the system Situation is well suited for creating Messiaen-like chord progressions, and the PWConstraints subsystem Score-PMC supports complex polyphonic CSPs.

A constraint is often defined by a Boolean expression that involves a set of variables. This approach is fully generic; arbitrary constraints can be stated that way.

This article studies how constraints are applied to the score. How can we control which variable sets in the score are affected by a given constraint? Note that existing systems differ greatly in the mechanisms by which they apply constraints to the score. These mechanisms were often designed for convenience with specific uses in mind. For example, sets of score objects in a sequence (e.g., notes in a melody) are conveniently constrained in all systems. We will later survey existing constraint-application approaches in detail.

The problem is that complex sets of variables are hard to constrain with existing approaches. For example, a common contrapuntal constraint permits dissonant note pitches for passing tones on a weak

Figure 1. Solution to a music-constraint problem that models a conventional music theory.



An Example

When modeling music theories by constraint programming, we must deal with diverse sets of variables. This section illustrates this through an example.

Figure 1 shows one solution of a constraint-satisfaction problem that models a conventional music theory by constraining many musical aspects. Obviously, this example constrains the underlying harmony. It implements Schoenbergian rules for traditional chord progressions (Schoenberg 1922). Further, the example restricts the rhythmic structure, and melodic aspects are controlled by constraints. Some constraints enforce the repeated use of specific motifs. Finally, the example treats non-chord tones according to conventional contrapuntal rules.

Note that this CSP is only one example of a music theory that users of a music constraint system can define and implement. The actual power of these systems is that users implement their own theories, whether conventional or not. We have chosen to use a conventional, tonal musical idiom here to facilitate understanding of the principles of constraint programming, but we could have equally well shown examples of, say, serial music, microtonal music, or music that is rhythmically complex.

Each constraint in this example is applied to specific score object sets and in that way to sets of variables. Figure 2 displays a number of exemplary variable sets graphically. By no means do they cover all constraints defined by this example, but they give an impression of the range of the variable sets involved.

In case (a), a constraint is applied to every element of a sequence: every bass note is constrained to sound the root of the present chord. Note that this case already applies a single constraint multiple times, namely, to all bass notes.

Case (b) shows score object sets that are typical for melodic constraints. In general, it is common

beat when the note is below a certain duration. This constraint involves a more complex variable set. It is difficult and sometimes even impossible to apply such complex constraints with approaches designed for very different use cases. On the other hand, using generic control structures such as iteration for the constraint application results in highly complex programs.

We propose a formal approach based on functional programming that combines convenience with full generality. In this approach, a constraint is a first-class function. A *first-class function* is a function that can be treated as data; for example, it can be handed to another function as an argument (Abelson, Sussman, and Sussman 1985). A constraint applicator is a *higher-order function*: a constraint applicator expects a constraint as argument and traverses the music representation to apply this constraint to variable sets. The term *higher-order programming* refers to a programming style that makes use of higher-order functions.

This approach has been implemented in the music-constraint system Strasheela (Anders 2007), which is freely available at <http://strasheela.sourceforge.net>. However, the present article attempts to provide general principles instead of introducing Strasheela from a user's point of view. For this purpose, the text uses mathematical notation and will show actual Strasheela code only briefly at the end.

We present constraint applicators that are suitable for many musical CSPs and that reproduce important mechanisms of existing systems. Most importantly, users can define their own constraint applicators with this approach. The proposed constraint-application mechanism is independent of the actual music-representation format. In principle, it can be used for any representation format (e.g., a simple event list or a MIDI-like representation). Nevertheless, a hierarchic representation format is better suited for expressing more complex music CSPs. For example, one may use a variant of CHARM or Smoke (Wiggins et al. 1993) for polyphonic CSPs. In our implementation, we use the Strasheela music representation, which provides a rich interface for accessing score information (Anders 2007).

Figure 2. Examples of object sets constrained in this musical CSP: (a) every element in a sequence; (b) pairs of consecutive objects in a sequence; (c) simultaneous object

pairs; (d) objects that sound at the start time of a chord object; and (e) consecutive chord-object pair with notes in one voice at the boundaries of these

chords. For visual clarity, only a few representative object sets are marked in (b)–(e). (See text for an explanation of the specific constraints.)

(a)

(b)

(c)

(d)

(e)

that constraints hold for multiple object sets. The constraint “a melodic interval must not exceed an octave” is applied to every pair of consecutive notes in each voice. (For visual clarity, only selected object sets are marked in this and the following subfigures of Figure 2.)

The variable set “every pair of consecutive notes” can be generalized to “every n consecutive objects in a sequence,” and it is also not limited to melodic notes. For example, a constraint can be applied to triples of consecutive chords. Schoenberg differs between so-called ascending progressions (e.g., V–I

or III–I) and descending progressions (e.g., I–V or III–V, cf. Schoenberg 1922). Schoenberg recommends that in a sequence of three chords C_1, C_2, C_3 , the sequence C_1, C_2 can be descending only if C_1, C_3 is ascending (e.g., III–V–I, as occurring in the example at measure 3ff).

Case (c) applies a harmonic constraint to every simultaneous note pair. This example applies the following constraint: If two non-chord tones occur simultaneously, then they must be consonant with each other. Although conventional composition textbooks may not explicitly state this rule, it is commonly followed in classical music.

The next two cases constrain more complex score object sets. Case (d) constrains all notes that sound at the time a new chord starts. These notes are constrained to express all pitch classes of this chord to clarify the chord change. In this specific solution, the chord and the corresponding notes always start at the same time, but the constraint also covers notes tied over a bar line.

Finally, case (e) applies a contrapuntal constraint that depends on the harmony. This constraint is intended for situations where non-chord tones can occur at the time a new chord starts, that is, when either the constraint of case (d) is not applied or more voices than chord tones are composed. The constraint is applied to two consecutive chord objects as well as to the last note of the preceding and the first note of the succeeding chord from a single voice. (This can actually be the same note, which is tied over.) For a clearer harmonic structure, no more than one of these two notes can be non-chord tones. This constraint also enforces proper suspensions: If the first note of a chord is a non-chord tone, then the last note of the previous chord must have the same pitch.

All the cases shown so far affect note pitches and chord objects alone. Rhythmic constraints instead affect variables such as note start times, durations, meter objects, and so forth. Furthermore, constraints can affect variable sets from several musical aspects. For example, the notion of a passing tone involves the harmonic, melodic and rhythmic structure at the same time.

This section presented a diverse set of score objects and related variables. As mentioned previously,

Figure 3. Constraint applied to multiple variable sets by a loop with explicit variable access, as proposed by Roy and Pachet (1997).

$$\forall i \in \{1, \dots, \text{length}(\text{Notes}) - 1\} : \text{myConstraint}(\text{Notes}_i, \text{Notes}_{i+1})$$

we aim to produce a generic system where users can model a wide range of music theories. Such a system must be general enough so that users can potentially constrain any variable sets in the music representation. On the other hand, applying constraints must also be convenient so that the system is usable in practice. The next section surveys the constraint-application mechanisms of seminal existing systems. Subsequently, we propose a new approach to applying constraints.

Previous Work

Compositional constraints are often applied to multiple variable sets at the same time, as shown in the previous section. Most systems support this in principle, but some systems do so in a highly restricted way. For example, OMClouds supports only the first two cases of Figure 2 (every element, or sets of consecutive elements in a list are constrained), and all constraints are invariably applied to the same variable sets.

Other systems use common control structures such as loops for applying constraints. Roy and Pachet (1997) apply a melodic constraint (Figure 2b) by a loop whose running index i is used to access neighboring note pairs in a melody. Figure 3 shows such a loop in first-order logic notation, a widely used formalism for reasoning about arbitrary objects and their relations (Kelly 1997).

For the less technical reader, this example reads in English as follows. For every (\forall) value i that is a member of (\in) the set of note numbers (except the last one), apply *myConstraint* to the i th note and its successor note. The colon ($:$) separates the introduction of i and its use for clarity. Throughout this article, we will notate constrained variables (and data structures containing variables) starting with an upper-case letter, and functions and constraints are notated using lower case.

For complex rules, however, it becomes tedious to explicitly access all the variables involved. Even for this simple example, we needed to perform arithmetic operations on indices. Existing systems

Figure 4. Pattern-matching based approach of PWConstraints: The melodic interval between any two consecutive pitches must not exceed an octave.

| | |
|-------------------|-----------------------------|
| $[\ast, X, Y]$ | pattern matching expression |
| $ X - Y \leq 12$ | constraint body |

therefore often provide convenient mechanisms to apply a constraint to several score object sets at the same time. This section outlines the constraint-application mechanisms of the two seminal systems PWConstraints and Situation. Both systems were originally developed for PatchWork (Laurson 1996). They are currently available in the PatchWork successor systems OpenMusic (Assayag et al. 1999) and PWGL (Laurson, Kuuskankare, and Norilo 2009).

PWConstraints

PWConstraints (Laurson 1996; Rueda et al. 1998) proposes a pattern-matching mechanism to apply constraints to the score. This mechanism plays such an important role for the system that subsystems were named accordingly (e.g., the basic subsystem is PMC, which stands for “pattern matching constraints”). A constraint consists of a pattern-matching part and a body. The body—a Boolean expression—is applied to any set of score objects that matches the corresponding pattern. For example, the pattern $[\ast, X, Y]$ matches every two neighbors of a sequence. (The asterisk matches zero or more objects.)

Figure 4 shows this pattern in a melody constraint: The interval between any two consecutive pitches must not exceed an octave (cf. Figure 2b). PWConstraints rules are defined in Lisp; but for consistency, generality, and simplicity, this text uses a mathematical notation instead. The interval is measured in semitones, so an octave is 12 pitches. Pattern-matching variables are implicitly bound to the matching values (e.g., the first two pitches, then the second two pitches, and so on). The free variables X and Y in the body (second line) use this binding. Pattern matching is a convenient application mechanism for sequential score object sets (e.g., a sequence of melody notes).

However, polyphonic CSPs often include non-sequential score-object sets (e.g., simultaneous notes that occur in different voices). PWConstraints’s polyphonic subsystem Score-PMC addresses

non-sequential score-object sets with a polyphonic music representation. This music representation provides access to the following score contexts: the sequence of melodic notes, simultaneous notes, and the metric position of a note. An extended pattern-matching language simplifies access to these contexts (Laurson and Kuuskankare 2005). Instead of matching only individual notes of a single melody, new language keywords support the matching of simultaneous note sets and note sets at specific metric positions. Complex polyphonic CSPs can be expressed with Score-PMC.

Composition systems are often designed in a user-extendable way, because their authors acknowledge the fact that they cannot foresee all potential needs of users. Nevertheless, Score-PMC users cannot introduce new pattern-matching keywords to access further score contexts (nor can they extend the music representation). Moreover, pattern matching is axiomatically limited to what can be expressed by a pattern: PWConstraints's pattern-matching language cannot express every possible combination of elements in a sequence. For example, a single PWConstraints pattern cannot express "any pair of consecutive variables in a sequence such that pairs do not overlap": for the sequence $[a, b, c, d]$, there is no pattern that only matches $X = a \wedge Y = b$ and $X = c \wedge Y = d$, but not $X = b \wedge Y = c$. (Note that \wedge denotes logical "and.") Also, PWConstraints cannot apply a constraint to all pairwise variable combinations in the sequence, whether consecutive or not.

Situation

Situation (Rueda et al. 1998) was originally conceived in collaboration between the composer Antoine Bonnet and the computer scientist Camilo Rueda as a constraint system for solving a range of harmonic CSPs. Its music representation still reflects the history of the system: Music is represented as a sequence of composite score objects (originally chords; currently also rhythmic motifs).

The constraint-application mechanism of Situation is tailored to this sequential music representation: Constraints are applied to sets of objects that are identified by their numeric index in the

sequence. For example, Situation makes it easy to constrain the first and the fifth chord in a chord sequence to contain specific pitches.

Any possible score object set can be expressed by an index-based constraint-application mechanism. Nevertheless, such a mechanism is only convenient for a sequence of score objects, but it is less suitable for constraining complex score topologies (e.g., hierarchically structured scores). For example, when constraining simultaneous notes, users would need to know the sets of numeric indices of simultaneous note sets. If the CSP also searches for the rhythmic structure, and in the CSP definition it is therefore not known which score objects will be simultaneous, then an index-based approach cannot constrain simultaneous score object sets, because the index sets of the simultaneous objects are unknown.

Constraint Application with Higher-Order Programming

We aim at producing a generic music-constraint system in which users can define a wide range of musical CSPs, including rhythmic, harmonic, melodic, and contrapuntal problems. For such a generic system, the existing constraint-application mechanisms are limiting. We therefore propose an approach that makes constraint-application mechanisms freely programmable.

Our approach is based on a well-understood formalism: A constraint is a first-class function, and the application of a constraint to the score is simply a function application. In contrast to the approaches of PWConstraints and Situation, this formalism fully decouples the definition and the application of a constraint. Besides defining constraints, users can also define convenient constraint-application mechanisms. Such mechanisms are implemented as higher-order functions that expect a constraint (i.e., another function) as an argument.

Unlike previous systems that provide a single and limiting constraint-application mechanism, higher-order programming makes it possible for users to freely select from a range of predefined constraint-application mechanisms, including the index-based application mechanism of Situation

Figure 5. Constraining every element in a sequence: Every bass note sounds the root of the present chord (cf. Figure 2a).

$$\bigwedge \text{map}(\text{getNotes}(\text{BassVoice}), \text{isChordRoot})$$

and the pattern-matching-based mechanism of PWConstraints. In addition, Strasheela provides several mechanisms not supported by previous systems (e.g., the application of a constraint to any score object that meets some user-defined condition). Most importantly, users can also define new constraint-application mechanisms.

Applying a Constraint to Every Element in a List

As discussed before, we often want to apply a constraint to multiple score-object sets simultaneously. Any control structure can be used for that. For example, a constraint can be applied to all notes in a voice by iterating through all these notes in a loop (cf. the earlier MusES plus BackTalk example). However, more complex score-object selections, as shown for example in Figure 2, would result in complex nested looping constructs, whereas higher-order functions can hide these low-level details.

The higher-order function *map* is a simple example whose effect is very similar to iteration. It applies a function to every element in a list and returns another list with the collected results. In Figure 5, *map* applies the constraint *isChordRoot* to every note of the bass voice. This constraint expects one argument (a *unary* constraint), and it was discussed in the conventional music theory example herein (Figure 2a). The function *map* returns a list of Boolean variables, and the complete expression in Figure 5 returns the conjunction of all these values (\bigwedge).

(Recall that for better comprehensibility, this article uses well-known mathematical notation instead of actual Strasheela code of the examples. The Strasheela code that corresponds to Figure 5 is presented later in Figure 14.)

Applying a Constraint to Neighbors in a List

The next example constrains consecutive pairs of score objects (see Figure 6). It implements a simplified version of Schoenberg's rules on chord-root progressions—one that his harmony textbook describes prior to presenting his full root-progression rules. (His full rules were discussed in connection

Figure 6. Constraining pairs of consecutive objects in a sequence: Neighboring chords must share at least one common pitch class. A solution is shown in Figure 8.

$$\bigwedge \text{map2Neighbours}(\text{MyChords}, f : f(C_1, C_2) := (C_1 \cap C_2) \neq \emptyset)$$

Figure 6.

$$\text{map2Neighbours}(Xs, fn) := \text{zip}(\text{butLast}(Xs), \text{tail}(Xs), fn)$$

Figure 7.

with Figure 2.) In a sequence of chords *MyChords*, every pair of consecutive chords is constrained to share at least one common pitch (not necessarily a root). More specifically, the intersection of the pitch classes of these two chords must not be empty. The constraint applicator is the function *map2Neighbours*, which expects as arguments a list and a *binary* function *f* (i.e., *f* expects two arguments). It applies *f* to every pair of neighboring elements in the list. Note that the applicator *map2Neighbours* is also often used for melodic rules (e.g., to constrain the interval between neighboring notes, as shown in Figure 2b).

This article uses the *where-notation* (Landin 1966) as “syntactic sugar” for an anonymous function, i.e., a function that is defined and used at the same time. For example, the function *f* serves the purpose of an anonymous function in $f : f(x) = x^2$, the colon (:) reads “where.” The symbol := means “is defined as,” so it can be distinguished from the equality constraint (=).

User-Defined Constraint Applicators

These examples demonstrate how higher-order functions are used as constraint applicators. This section shows how users define such applicators. This ability marks an important distinction between Strasheela and existing systems.

The higher-order function *map* (applied in Figure 5) is widely known in functional programming, and its definition can be studied in many textbooks on functional programming languages, such as Abelson, Sussman, and Sussman (1985). Figure 7 defines the higher-order function *map2Neighbours*, which was used in the last example (Figure 6). The definition is very brief and consists of only a call to the function *zip*, a relative of the function *map*.

Figure 7. User-defined constraint applicator: The higher-order function *map2Neighbours*, which was used in Figure 6.

Figure 8. Solution to a harmonic CSP in which the dissonance degree of chord intervals gradually changes.



Figure 8.

```
permittedIntervalsSpec := [(1#3)#[3, 4, 5, 7, 8, 9, 12, 15, 16]
(4#6)#[2, 3, 4, 6, 8, 9, 10, 14, 15, 16]
(7#9)#[1, 2, 3, 6, 8, 10, 11, 13, 14, 15]
(10#11)#[2, 3, 4, 6, 8, 9, 10, 14, 15, 16]
12#[4, 5, 7, 8, 9, 12, 15, 16]]
```

(a)

```
restrictIntervals(C, Is) :=  $\bigwedge$  mapPairwise(map(C, getNotePitches),
f : f(P1, P2) := |P1 - P2| ∈ Is)
```

(b)

```
 $\bigwedge$  mapIndex(MyChords, permittedIntervalsSpec, restrictIntervals)
```

(c)

Figure 9.

The function *zip* expects two lists *Xs* and *Ys* and a binary function *fn*, and it collects the results of all calls *fn*(*X_i*, *Y_i*), where *X_i* and *Y_i* are each the *i*th element of *Xs* and *Ys*, respectively. The Common Lisp equivalent of *zip* is *mapcar*, which supports mapping over any number of lists. The function *map2Neighbours* calls *zip* with three arguments, namely, two sublists of its list argument *Xs* (one containing all but the last element and the other all but the first element of *Xs*) plus the binary function argument *fn*.

Whereas *map2Neighbours* applies a binary constraint to every pair of two neighboring list elements, a generalized function *mapNeighbours* applies an *n*-ary function to every sublist of *n* neighbors in a list. For example, melodic constraints that affect more than two neighboring notes (such as “a large skip should be followed by a step in the opposite direction”) can be applied with *mapNeighbours*. Also, the Schoenbergian rule on chord root progressions that “resolves” a descending progression (see the discussion of Figure 2) has been applied by this function.

Modeling Existing Approaches

This article argues that using higher-order constraint applicators is more generic than the constraint-

Figure 9. Situation’s index-based approach reproduced with higher-order programming: (a) the intervals are permitted in specific

chords, declared with a mini-language (see text for details); (b) constraint definition: chord *C* consists only of specific intervals *Is*; (c) constraint

application: chords at specific positions use only specific intervals, and a solution is shown in Figure 8.

application mechanisms of existing systems. To substantiate this claim, this section reproduces the constraint-application mechanisms of Situation and PWConstraints as higher-order functions. Note that these systems do not support each other’s application mechanisms. Also, recall that this article uses a mathematical notation instead of the original Lisp syntax of Situation and PWConstraints.

Index-Based Constraint Application

Situation offers multiple constraint-application mechanisms that are all index-based. The system introduces various constraint-specific mini-languages for controlling the constraint application to score objects. This section describes a typical Situation example and reproduces it using a higher-order constraint applicator.

Figure 8 shows a solution to a harmonic CSP in which all consecutive chords share common pitches (cf. Figure 6) and where the soprano voice forms an arch. In addition, the set of intervals allowed between chord tones gradually changes from consonant to dissonant and then back to more consonant. (It ends with an augmented triad.) The intervals between all pairwise note combinations within a chord are constrained. Figure 9a specifies chord-index ranges and specifies which intervals are permitted for these chords. The specification uses a mini-language. For example, the first line of the specification indicates that for the first three chords (index range 1#3), only consonances are permitted (intervals [3, 4, 5, 7, 8, 9, 12, 15, 16]). More generally, this mini-language consists of a list of pairs *indices#args*, where *indices* specifies one or more indices to which a constraint is applied, and *args* is a list of additional arguments given to the constraint.

The constraint on the chord intervals is applied in Figure 9c using the constraint applicator *mapIndex*. This function models a typical Situation constraint-application mechanism: A constraint is applied to score objects at specified indices together with index-specific constraint arguments. The function *mapIndex* expects the list of chords, the interval specification *permittedIntervalsSpec*, and the actual constraint *restrictIntervals* as arguments. For example, *mapIndex* applies *restrictIntervals* to the first

Figure 10. PWConstraints’s pattern-matching based approach reproduced with higher-order programming: The interval between two consecutive pitches does not exceed an octave (cf. the PWConstraints definition in Figure 4).

$$\bigwedge \text{mapPM}(\text{Pitches}, [*, x, x], \\ f : f([P_1, P_2]) := |P_1 - P_2| \leq 12)$$

mapPM. It expects three arguments: a list of score objects *Xs*, a pattern-matching expression *pattern* (using the syntax above), and a unary constraint *f* that expects a list. The function *mapPM* applies *f* to every sublist of *Xs* that matches *pattern*. Figure 10 uses *mapPM* to apply a melodic constraint to pairs of consecutive note pitches. The function *mapPM* is defined in the appendix.

Note that the approach proposed here is more general than PWConstraints’s constraint-application mechanism. The function *mapPM* can be used on any data sequence. For example, *mapPM* can apply constraints to any score-object sequence extracted from a nested music representation. This is similar to the effect of the new pattern-matching keywords introduced by Laurson and Kuuskankare (2005) discussed earlier, but here users are not limited to a set of predefined keywords. Also, *mapPM*—and *mapIndex*—can be used within another constraint definition, leading to nested constraint applicators (see the following).

Further Examples

Constraint Application to Selected Objects in a Score Hierarchy

The constraint applicators introduced so far apply a constraint to specific elements or element sets in a sequence. These mechanisms were implemented by higher-order functions that traverse a sequence for applying a constraint. However, higher-order functions can define control structures that traverse arbitrary data structures. The present section proposes a technique that applies a constraint to all score objects in a hierarchic music representation that meet a user-defined condition.

Strasheela supports the hierarchic nesting of score objects to express, for example, which objects form a voice, a motif, or other object sets. Score objects that hold other objects are called *containers*. Strasheela containers understand a method *map* that generalizes the *map*-function discussed previously.

element in *MyChords* with a list of only consonant intervals, as specified for the index 1. The constraint applicator *mapIndex* is defined in the appendix.

The constraint *restrictIntervals* enforces that the absolute interval between any pitch pair in a chord *C* is always an element of the given list of permitted intervals *Is* (Figure 9b). This constraint is concisely defined with another constraint applicator, *mapPairwise*. *mapPairwise* applies a binary constraint *f* to all pairwise combinations of a given list (here, the list of chord note pitches).

Our reproduction of Situation’s application mechanism is even more general than the original. In Situation, different mechanisms are hard-wired to a specific constraint. In contrast, higher-order functions can be used for applying any constraint. Also, the original mechanism is restricted to the sequential score topology of Situation. Instead, the mechanism of this section can be used on any score object set that can be represented as a sequence (e.g., the list of motifs in a specific voice).

Constraint Application with a Pattern-Matching Language

This section reproduces the constraint-application mechanism of PWConstraints. In PWConstraints, a constraint is applied to all object sets that match the pattern-matching expression of the constraint header (see Figure 4).

The present section defines a pattern-matching language similar to PWConstraints’s, and which defines three symbols. There are two placeholder symbols: *?* (question mark) matches exactly one sequence element, and *** (asterisk) matches zero or more elements. Instead of PWConstraints’s pattern-matching variables, this language provides the symbol *x*: Every occurrence of a pattern-matching variable in PWConstraints is substituted by the unvarying symbol *x* here. The following example shows an expression that matches any but the first pair of two successive elements: $[?, *, x, x]$. Here, the pair *x, x* must be preceded by exactly one element (*?*) and additionally zero or more elements (***).

PWConstraints’s constraint-application mechanism is reproduced by the higher-order function

Figure 11. Solution of Figure 1 repeated: In every instance of motif *a*, the maximum pitch occurs exactly once.



This method traverses a score hierarchy instead of a list. The method also supports optional arguments. For example, the argument *test* expects a Boolean function: Only objects for which the test returns true are processed; other objects are skipped.

We can use this method *map* for applying a constraint to multiple objects nested arbitrarily deep in the hierarchic representation, but which all hold a specific condition. Figure 11 repeats the CSP solution of Figure 1, but it now marks all occurrences of a specific motif *a*. With *map*, we can constrain all these motif instances to have a single melodic peak. Although this section proposes an approach in which motifs are “boxed” in a container, the actual implementation of this CSP is based on a more flexible model in which motifs are emerging in the solution.

Containers can be marked in the CSP definition, either manually or algorithmically, to represent specific motifs, and then they can apply additional constraints that shape these motifs. In Figure 12a, *map* applies the constraint *hasUniquePeak* to all objects that are a container marked with the tag *motif_a*. This condition is defined by the test function *f*. The function *hasThisInfo* returns a Boolean indicating whether a given object is tagged with a specific symbol. The constraint *hasUniquePeak* (Figure 12b) enforces that the melodic peak (i.e., the maximum pitch) occurs exactly one time in all the notes contained in its argument motif *M*. The keyword **let ... in ...** declares local variables. The constraint *once* enforces that *MaxP* occurs only once in *Ps*.

Nested Constraint Application

Earlier, we showed a number of constrained score-object sets in an example (Figure 2). Most of these score-object sets are more complex and have not been covered by the constraint-application examples discussed so far. These more complex cases are addressed by nesting applicators.

Figure 12. Constraining selected objects in the score hierarchy: (a) apply *hasUniquePeak* to every motif *a* nested somewhere in the score; (b) *hasUniquePeak* as defined such that the maximum pitch in motif *M* occurs exactly once.

$$\bigwedge \text{map}(\text{MyScore}, \text{hasUniquePeak}, \text{test} : f : f(X) := \text{isContainer}(X) \wedge \text{hasThisInfo}(X, \text{motif}_a))$$

(a)

$$\text{hasUniquePeak}(M) := \text{let } Ps := \text{map}(M, \text{getPitch}, \text{test} : \text{isNote})$$

$$\text{MaxP} := \text{max}(Ps)$$

$$\text{in } \text{once}(\text{MaxP}, Ps)$$

(b)

Figure 2c depicts the application of a contrapuntal constraint: All pairs of simultaneous non-chord tones are constrained to be consonant with each other. The rhythmical structure of the music can be arbitrarily complex. This constraint uses a constraint applicator for pairs of simultaneous objects, *mapSimNotePairs*. It is defined by combining mapping with the filtering of score objects from a score hierarchy. This section studies the definition of *mapSimNotePairs*. The other cases in Figure 2 are defined similarly.

Figure 13a shows the definition and application of this contrapuntal constraint. The constraint applicator *mapSimNotePairs* traverses all notes in the score and applies the constraint *isConsonant* to all pairs of simultaneous notes that are both non-chord tones. The method *collect* is a relative of the method *map* discussed previously: *collect* traverses the score hierarchy below a given container and selects all score objects for which a given Boolean function returns true. In Figure 13a, *collect* returns all non-chord notes in *MyScore*.

The constraint applicator *mapSimNotePairs* is realized with two nested mappings (Figure 13b). The outer mapping traverses *Notes*, a given list of note objects. For every note N_1 , the function *getHigherSimNotes* returns all notes simultaneous with N_1 but which are situated in a higher voice (discussed subsequently). The inner mapping applies the given function *myConstraint* to every note N_1 and any of its simultaneous notes.

The function *getHigherSimNotes* filters all notes from the whole score that fulfill certain conditions (Figure 13c): The notes must be simultaneous to the given note N_1 and occur in a higher voice. (In the container holding all voices, the voice of N_2 is situated before the voice of N_1 , i.e., N_2 's voice position is lower.) N_1 itself is excluded from the result. This function exploits the fact

Figure 13. Constraining simultaneous note pairs by nesting constraint applicators: (a) apply constraint that simultaneous non-chord tones are consonant with each other, using applicator `mapSimNotePairs`; (b) `mapSimNotePairs` is defined with nested mapping and `getHigherSimNotes`; and

(c) the accessor function `getHigherSimNotes` collects notes that meet certain conditions; they are simultaneous to note N_1 , but situated in a higher voice.

```

 $\bigwedge$  mapSimNotePairs( collect(MyScore, test: isNonharmonicNote),
                    isConsonant)
(a)

mapSimNotePairs(Notes, myConstraint) :=
  map(Notes,
      f : f(N1) :=  $\bigwedge$  map(getHigherSimNotes(N1, Notes),
                          g : g(N2) := myConstraint(N1, N2))
      )
(b)

getHigherSimNotes(N1, Notes) :=
  let test(N2) := N1 ≠ N2
                ∧ isNote(N2)
                ∧ getVoicePos(N1) > getVoicePos(N2)
                ∧ isSimultaneous(N1, N2)
  in filter(Notes, test)
(c)

```

that score objects in Strasheela’s hierarchic music representation are bidirectionally linked: Every container has access to its contained score objects (e.g., notes or other containers) and vice versa. That way, any note object N_1 can access the top-level container of the score, *Top*. The container *Top* then collects all score objects that meet the mentioned conditions, using the method *collect* just introduced. The constraints *isSimultaneous* and *isConsonant* are implemented in Anders (2007).

Note that users of *mapSimNotePairs* do not need to know the definition of *getHigherSimNotes* and *mapSimNotePairs*. The applicator *mapSimNotePairs* is convenient to use: Only a list of score objects and a constraint are required. The function hides all the details.

The constraint-application mechanisms of Situation and PWConstraints do not support nesting. For example, in PWConstraints, a pattern-matching expression can solely occur in the header of a constraint but not in the constraint body. Also, users cannot encapsulate a constraint in a function, nor can a constraint call other constraints. Pattern matching is only supported by the top-level constraint applicator provided by the solver.

Relation Between the Proposed Approach and Its Implementation in Strasheela

This article has so far discussed general principles; this section presents how the proposed approach is

realized in Strasheela. Some of the details presented in this section are intended for readers with a computer-science background; understanding these details is not required for understanding the main ideas presented by the rest of the article.

Strasheela is implemented in the multi-paradigm programming language Oz (van Roy and Haridi 2004), and Strasheela inherits many features from this language. At its core, Oz is a concurrent constraint-programming language, but a rich set of further programming paradigms is defined on top of this core. Like Common Lisp—the implementation language of Situation and PWConstraints—Oz supports the two programming paradigms of functional programming and object-oriented programming. This text utilizes the former, and Strasheela’s music representation is founded on the latter. As a music-constraint system, Strasheela obviously benefits from Oz’ excellent support of logic and constraint programming. The influence of concurrent programming on Strasheela is discussed further herein.

For many computer-aided composition systems, the implementation language also serves as their user interface, because such a design results in a particularly flexible system. Example systems include Common Music (Taube 2004), which is implemented in Common Lisp; the Haskell-based system Haskore (Hudak 1996); and JMSL, implemented in Java (Didkovsky and Burk 2001). Strasheela adopts this approach: The Oz programming language serves as its user interface, so Strasheela users must have at least basic Oz programming skills.

Figure 14 shows a constraint-application example in actual Strasheela code. This example corresponds to Figure 5: It constrains every bass note to sound the root of the present chord (cf. Figure 2a). More specifically, the method `getNotes` is applied to the score object `BassVoice`, and the constraint `IsChordRoot` is applied with the constraint applicator `ForAll` to each element of the resulting list of notes. For an introduction to Oz programming, see the comprehensive guide (van Roy and Haridi 2004) or the brief Oz tutorial at the Strasheela Web site (strasheela.sourceforge.net).

Besides the clear differences in syntax, Figure 14 and Figure 5 differ in the following. The higher-order

Figure 14. Translation of Figure 5 into Strasheela code.

```
{ForAll {BassVoice getNotes($)} IsChordRoot}
```

function *map* of Figure 5 has been replaced by the higher-order Oz procedure `ForAll`, and no surrounding conjunction constraint is required. The reason for this difference is that constraints in Oz are first-class concurrent procedures instead of first-class Boolean functions. Constraints in Oz run in parallel in an attempt to reduce the domains of their variables (Schulte 2002). They do not necessarily return anything, and their conjunction is implicit. Nevertheless, for more complex constraint problems, Oz also supports constraints for explicit logical connectives like conjunction (`and`), negation (`not`), or implication (`if...then`).

A constraint applicator definition in Strasheela code is shown in Figure 15. For simplicity, this definition corresponds exactly to an applicator discussed before and shown in Figure 7. The applicator `Map2Neighbours` is a function, but its definition shows that functions are only a special case of procedures in Oz whose last argument is the result. (Oz also supports a more convenient function notation in which the argument for the return value is implicit.) Strasheela's procedure `For2Neighbours` has the same arguments as `Map2Neighbours` except for the return value.

Unexpectedly at first, the declarative concurrent programming model of Oz turned out to be beneficial for the approach presented. Several examples in this text perform complex operations when accessing score objects for constraint application (e.g., a traversal of all score objects in the score). In systems like *Situation* and *PWConstraints*, costly access operations hamper search efficiency, because these operations are not performed until the validity of a variable value is checked, and they are therefore always performed again after backtracking. In Strasheela, on the other hand, such operations are performed only once. After constraints are applied to the variables, the constraints watch and affect the variables as concurrent agents.

In some cases, constraints cannot be applied immediately, because information is missing in the CSP. For example, in Figure 13 it might not be known in the CSP definition which notes are

Figure 15. Translation of Figure 7 into Strasheela code.

```
proc {Map2Neighbours Xs Fn Result}
  Result = {List.zip {List.take Xs {Length Xs}-1} Xs.2 Fn}
end
```

simultaneous or which notes are non-chord tones. In such a situation, the concurrent programming model of Oz simply blocks and the constraint application waits until the required information becomes available during the search process.

Oz integrates constraint programming into a concurrent language by encapsulating the search (Schulte 2002). By contrast, conventional constraint programming languages like Prolog feature global backtracking, and their constraint solver is situated at the top-level, which makes Prolog incompatible with concurrent programming. The combination of concurrent programming and encapsulated search has proven useful for real-time constraint programming (Anders and Miranda 2008).

Summary and Discussion

When one models music theories, constraints are applied to diverse variable sets. In this article, we have proposed the use of higher-order programming for this purpose. Experience with Strasheela shows that CSPs which make use of suitable higher-order procedures are often more concise than equivalent CSPs that apply constraints directly (e.g., by explicit nested loops). Higher-order procedures hide complex control structures, which is particularly important when modeling complex music theories.

The constraint-application mechanisms of existing systems result in concise programs as well, and they are thus convenient. However, they are not generic. By contrast, we argue that higher-order programming is both concise and generic. Arbitrary control structures can be defined in terms of higher-order procedures, including the constraint-application mechanisms of previous systems. As a demonstration, the application mechanisms of *Situation* and *PWConstraints* have been reproduced.

In addition, higher-order programming supports constraint applicators which cannot be adequately reproduced by *Situation* and *PWConstraints*. The mechanisms of the latter two systems are suited

to constraining elements in a sequence, because these systems rely on positional relations. First-class procedures, on the other hand, can process arbitrary data structures, such as trees or graphs, besides sequences. For example, a constraint can be applied to all score objects in a hierarchic score representation for which some test procedure returns “true.” Hence, a constraint-application mechanism based on the notion of higher-order procedures is more generic than the mechanisms of Situation and PWConstraints.

Various constraint applicators are presented in this article that are suitable for modeling a wide range of theories. Strasheela provides further applicators which implement mathematical concepts (e.g., the Cartesian product), or knowledge about the music representation (e.g., implicitly traversing a score hierarchy). Most importantly, however, users can define their own constraint applicators, which was impossible in existing systems.

Because higher-order constraint applicators allow for a concise and expressive CSP definition, other systems may be interested in adopting these mechanisms. For example, higher-order constraint applicators can be easily reproduced in the combination of MusES and BackTalk. MusES and BackTalk are implemented in Smalltalk, and Smalltalk code blocks are essentially first-class functions.

The idea of higher-order constraint applicators for musical CSPs has been adopted by the recent system PWMC, which is implemented on top of PWConstraints, and which is also presented in this issue of *Computer Music Journal* (Sandred 2010). In PWMC, a constraint definition consists of two parts. A *logic statement* is always wrapped in a PWGL sub-patch in lambda-state: It is a first-class function. A logic statement constrains its sub-patch arguments. Logic statements are applied to the music representation using an *access box*, which is a higher-order function that expects a logic statement. Sandred (2010) explains a number of rather flexible access boxes that can be seen as further constraint-applicator examples. However, PWMC adapts the approach presented in the present article only partly and is thus less general: Although users can freely define constraints (logic statements), users cannot define constraint applicators (access

boxes), because their definition depends on complex internal details. Nevertheless, the convenient visual-programming interface of PWMC greatly simplifies constraint programming for users without much programming experience. It also shows that higher-order programming is a suitable paradigm even for such users.

Higher-order programming is used solely for composition in this text, but it can serve for music analysis as well. The pattern-matching mechanism has also been applied to analysis (Kuuskankare and Laurson 2008). Higher-order programming can be used, for example, to search for occurrences of specific events in music (say, Italian sixth chords, or the pitch sequence B-A-C-H) with the common higher-order function *filter*. For complex analysis questions, it will be beneficial again that higher-order programming is both convenient and fully generic. For example, a higher-order function can encapsulate arbitrary score traversals for accessing analytical information.

Acknowledgments

This work was supported by the EPSRC-funded project Learning the Structure of Music (LeStruM), Grant EPD063612-1.

References

- Abelson, H., G. J. Sussman, and J. Sussman. 1985. *Structure and Interpretation of Computer Programs*. Cambridge, Massachusetts: MIT Press.
- Anders, T. 2000. “Arno: Constraints Programming in Common Music.” *Proceedings of the 2000 International Computer Music Conference*. San Francisco, California: International Computer Music Association, pp. 324–327.
- Anders, T. 2007. “Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System.” PhD thesis, Queen’s University Belfast.
- Anders, T., and E. Miranda. 2008. “Constraint-Based Composition in Realtime.” *Proceedings of the 2008 International Computer Music Conference*. San Francisco, California: International Computer Music Association. Available on-line at classes.berklee.edu/mbierylo/ICMC08/defevent/papers/1509.pdf. Accessed 6 December 2009.

- Anders, T., and E. Miranda. In press. "Constraint Programming Systems for Modelling Music Theories and Composition." *ACM Computing Surveys*.
- Apt, K. R. 2003. *Principles of Constraint Programming*. Cambridge: Cambridge University Press.
- Assayag, G., et al. 1999. "Computer Assisted Composition at IRCAM: From PatchWork to Open Music." *Computer Music Journal* 23(3):59–72.
- Chemillier, M., and C. Truchet. 2001. "Two Musical CSPs." *Seventh International Conference on Principles and Practice of Constraint Programming, Musical Constraints Workshop*. Available on-line at recherche.ircam.fr/equipes/repmus/cpws/chemillier.ps. Accessed 6 December 2009.
- Courtot, F. 1990. "A Constraint Based Logic Program for Generating Polyphonies." *Proceedings of the 1990 International Computer Music Conference*. San Francisco, California: International Computer Music Association, pp. 292–294.
- Didkovsky, N., and P. Burk. 2001. "Java Music Specification Language, an Introduction and Overview." *Proceedings of the 2001 International Computer Music Conference*. San Francisco, California: International Computer Music Association, pp. 123–126.
- Hudak, P. 1996. "Haskore Music Tutorial." In J. Launchbury, E. Meijer, and T. Sheard, eds. *Advanced Functional Programming*. Berlin/Heidelberg: Springer, pp. 38–67.
- Kelly, J. 1997. *The Essence of Logic*. Upper Saddle River, New Jersey: Prentice Hall.
- Kuuskankare, K., and M. Laurson. 2008. "Survey of Music Analysis and Visualization Tools in PWGL." *Proceedings of the 2008 International Computer Music Conference*, San Francisco, California: International Computer Music Association. Available on-line at classes.berkeley.edu/mbierylo/ICMC08/defevent/papers/cr1160.pdf. Accessed 6 December 2009.
- Landin, P. J. 1966. "The Next 700 Programming Languages." *Communications of the ACM* 9(3):157–166.
- Laurson, M. 1996. "PATCHWORK: A Visual Programming Language and some Musical Applications." PhD thesis, Sibelius Academy, Helsinki.
- Laurson, M., and M. Kuuskankare. 2001. "A Constraint Based Approach to Musical Textures and Instrumental Writing." *Seventh International Conference on Principles and Practice of Constraint Programming, Musical Constraints Workshop*. Available on-line at recherche.ircam.fr/equipes/repmus/cpws/laurson.ps. Accessed 6 December 2009.
- Laurson, M., and M. Kuuskankare. 2005. "Extensible Constraint Syntax Through Score Accessors." *Journées d'Informatique Musicale*, Talence: Association Française d'Informatique Musicale. Available on-line at jim2005.mshparisnord.org/download/5.Extensible.pdf. Accessed 6 December 2009.
- Laurson, M., M. Kuuskankare, and V. Norilo. 2009. "An Overview of PWGL, a Visual Programming Environment for Music." *Computer Music Journal* 33(1):19–31.
- Pachet, F., and P. Roy. 1995. "Mixing Constraints and Objects: a Case Study in Automatic Harmonization." In I. Graham, B. Magnusson, and J.-M. Nerson, eds. *Proceedings of TOOLS-Europe'95*. Upper Saddle River, New Jersey: Prentice Hall, pp. 119–126.
- Pachet, F., and P. Roy. 2001. "Musical Harmonization with Constraints: A Survey." *Constraints Journal* 6(1):7–19.
- Roy, P., and F. Pachet. 1997. "Reifying Constraint Satisfaction in Smalltalk." *Journal of Object-Oriented Programming* 10(4):43–51.
- Rueda, C., et al. 1998. "Integrating Constraint Programming in Visual Musical Composition Languages." *ECAI 98 Workshop on Constraints for Artistic Applications*. Available on-line at www.cs.vu.nl/~eliens/poosd/@online/@share/archive/ecai98/rueda.ps. Accessed 6 December 2009.
- Sandred, Ö. 2003. "Searching for a Rhythmical Language." *PRISMA 01*. Milano: Euresis Edizioni, pp. 117–127.
- Sandred, Ö. 2010. "PVMC, a Constraint Solving System for Generating Music Scores." *Computer Music Journal* 34(2):8–24.
- Schoenberg, A. 1922. *Harmonielehre*, rev. ed. Vienna: Universal Edition.
- Schottstaedt, W. 1989. "Automatic Counterpoint." In M. V. Mathews and J. R. Pierce, eds. *Current Directions in Computer Music Research*. Cambridge, Massachusetts: MIT Press, pp. 199–214.
- Schulte, C. 2002. *Programming Constraint Services*. LNAI 2302. Berlin/Heidelberg: Springer.
- Taube, H. 2004. *Notes from the Metalevel: Introduction to Algorithmic Music Composition*. Lisse: Swets and Zeitlinger.
- Truchet, C., and P. Codognet. 2004. "Solving Musical Constraints with Adaptive Search." *Soft Computing* 8(9):633–640.
- van Roy, P., and S. Haridi. 2004. *Concepts, Techniques, and Models of Computer Programming*. Cambridge, Massachusetts: MIT Press.
- Wiggins, G., E. Miranda, A. Smaill, and M. Harris. 1993. "A Framework for the Evaluation of Music Representation Systems." *Computer Music Journal* 17(3):31–42.

Figure 16. Definition of the constraint applicator `mapIndex`. See the “Modeling Existing Approaches” section for a discussion of this function.

```
mapIndex(xs, decls, fn) :=
  let /* matchingXsWithArgs is a list of pairs x#args. Each x is an element in xs which matches
      the index part of decls, and args is a list of arguments specified for this index in decls as well.
      More precisely, decls is a list consisting of pairs idx#args, where idx is either single index or a
      pair startIdx#endIdx: and args is a list of arguments. */
      matchingXsWithArgs :=
        /* mappend(xs, fn) is a variant of map(xs, fn). The mappend argument fn must return a
           list. mappend appends all collected sublists (hence the name: map-append or mappend). */
           mappend(decls,
             f : f(idx#args) :=
               if isPair(idx)
                 /* idx is a start#end pair. The expression sublist(xs, start, end) accesses the
                    sublist of xs that consists of the start-th to end-th elements (including). The
                    subsequent mapping then adds args to each element of this sublist. */
                 then let start#end := idx
                       in map(sublist(xs, start, end),
                             g : g(x) := x#args)
                 /* idx is a single index. */
                 else [nth(xs, idx)#args])
           in map(matchingXsWithArgs,
                 h : h(x#args) := fn(x, args))
```

Figure 16.

Figure 17. Definition of the constraint applicator `mapPM`. See the “Modeling Existing Approaches” section for a discussion of this function.

```
mapPM(xs, patternExpr, fn) :=
  let /* collectPM is a highly recursive auxiliary function which returns in a list any list of elements from xs
      (a list) which match the patternExpr (a list of pattern symbols). The argument matchingXs (initially
      nil) is used to pass matching elements accumulated so far to the next recursive call of collectPM. */
      collectPM(xs, patternExpr, matchingXs) :=
        /* Abort condition for recursion */
        if patternExpr = nil ∨ xs = nil
        then [reverse(matchingXs)]
        else /* Recursively call collectPM – depending on the first symbol in patternExpr. */
            let symbol := head(patternExpr)
                xsTail := tail(xs)
                patternTail := tail(patternExpr)
            in result where
                {
                  collectPM(xsTail, patternTail, matchingXs)          if symbol = ?
                  collectPM(xsTail,
                           patternTail,
                           cons(head(xs), matchingXs))              if symbol = x
                  let /* minimal length of remaining pattern is length of
                      patternTail without any occurrence of * (Kleene star) */
                      minLength := length(filter(patternTail,
                                                  f : f(x) := x ≠ *))
                  in /* mappendTail recursively applies f to every non-nil tail
                     of xs and appends the resulting lists. */
                     mappendTail(xs,
                                 g : g(sublist) :=
                                   if length(sublist) ≥ minLength
                                   then collectPM(sublist,
                                                  patternTail,
                                                  matchingXs)
                                   else nil)
                }
            in map(collectPM(xs, patternExpr, nil), fn)
```

Figure 17.

Appendix: Definition of `mapIndex` and `mapPM`

This appendix defines the two constraint applicators `mapIndex` and `mapPM`, discussed in the section “Modeling Existing Approaches.” Both definitions are documented by inline comments (see Figures 16 and 17).

For conciseness and simplicity, functions in these definitions support pattern matching

(as supported by Oz, but also languages like Haskell or ML): function arguments implicitly decompose composite data structures. For example, function arguments can be pairs such as `x#y`, where the variables `x` and `y` can be referenced directly. The notion of pattern matching in `PWConstraints` refers to a somewhat different concept, although the name is the same.