Abstract Representation of Music: A Type-Based Knowledge Representation Framework

Nicholas Harley

Submitted in partial fulfilment of the requirements of the Degree of Doctor of Philosophy

Queen Mary
University of London

School of Electronic Engineering and Computer Science
Queen Mary University of London
United Kingdom

December 16, 2019

Statement of Originality

I, Nicholas Harley, confirm that the research included within this thesis is my own work or that where it has been carried out in collaboration with, or supported by others, that this is duly acknowledged below and my contribution indicated. Previously published material is also acknowledged below.

I attest that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge break any UK law, infringe any third party's copyright or other Intellectual Property Right, or contain any confidential material.

I accept that the College has the right to use plagiarism detection software to check the electronic version of the thesis.

I confirm that this thesis has not been previously submitted for the award of a degree by this or any other university.

The copyright of this thesis rests with the author and no quotation from it or information derived from it may be published without the prior written consent of the author.

Signature: [illegible]

Date: December 16, 2019

Abstract

The wholesale efficacy of computer-based music research is contingent on the sharing and reuse of information and analysis methods amongst researchers across the constituent disciplines. However, computer systems for the analysis and manipulation of musical data are generally not interoperable. Knowledge representation has been extensively used in the domain of music to harness the benefits of formal conceptual modelling combined with logic based automated inference. However, the available knowledge representation languages lack sufficient logical expressivity to support sophisticated musicological concepts.

In this thesis we present a type-based framework for abstract representation of musical knowledge. The core of the framework is a multiple-hierarchical information model called a constituent structure, which accommodates diverse kinds of musical information. The framework includes a specification logic for expressing formal descriptions of the components of the representation. We give a formal specification for the framework in the Calculus of Inductive Constructions, an expressive logical language which lends itself to the abstract specification of data types and information structures. We give an implementation of our framework using Semantic Web ontologies and JavaScript. The ontologies capture the core structural aspects of the representation, while the JavaScript tools implement the functionality of the abstract specification. We describe how our framework supports three music analysis tasks: pattern search and discovery, paradigmatic analysis and hierarchical set-class analysis, detailing how constituent structures are used to represent both the input and output of these analyses including sophisticated structural annotations. We present a simple demonstrator application, built with the JavaScript tools, which performs simple analysis and visualisation of linked data documents structured by the ontologies. We conclude with a summary of the

contributions of the thesis and a discussion of the type-based approach to knowledge representation, as well as a number of avenues for future work in this area.

Acknowledgements

This research would not have been possible, but for the contributions of a great many people. First and foremost, I wish to thank my supervisor, professor Geraint Wiggins, for his inimitable supervision and general mentorship. I have thoroughly enjoyed my time working with him and learned so much. Without his constant encouragement and advise this PhD would not have been completed. I would also like to give a big thanks to my advisory panel, professors Mark Sandler and Elaine Chew, for their incisive and constructive input throughout the course of my PhD. I gratefully acknowledge the Transforming Musicology project for the funding which made my PhD possible and for the opportunity to be a part of such an exciting collaboration. I wish to thank all the members of the Transforming Musicology team whose inspiring work, excellent advice and kind encouragement has helped me no end. I also wish to thank my colleagues in the Computation Creativity Lab whose spirited pub discussions never failed to provoke flashes of inspiration. Finally, I would like thank my friends and family for their endless support, understanding and patience. Thanks to my Alexandra Road friends for tolerating my constant stays in their house over the past year. Thanks to my Boundary Road friends for their understanding of my apparent withdrawal for society. Very special thanks to my parents for their constant love and support, both emotional and financial; nothing would be possible without them. Last but not least, massive thanks to Georgia, for being there every step of the way, regardless of how insufferable I was.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Research Problem

1.1.1 Research Topic

The topic of this thesis is computer representation of music. This encompasses many aspects of computer science, such as information modelling, ontology and knowledge representation. We are primarily interested in representations which support information systems and software tools for computational musicology. In particular, we are interested in representations which admit sophisticated reasoning with information. As such knowledge representation plays a central role in the topic. The intersection of knowledge representation and music is heavily dominated by work related to the Semantic Web. Although the focus of this thesis is not the Semantic Web, it will nonetheless play a role; we examine representations from a broader point of view, and regard the Semantic Web as a set of technologies which may or may not be applicable to the wider task.

1.1.2 Relevance of the Topic

Scientific music research is highly interdisciplinary, encompassing acoustics, cognitive science, information retrieval and traditional musicology, to name but a few. The ubiquitous application of computers leads computer science to the forefront of many endeavours, and representations of music and related information underpin all computer-based music research. The different research activities across these disciplines each consume and produce data. As a result, computational musicology is increasingly data-driven. The plethora of available data prompts new research questions which explore the connection between perspectives from different disciplines. Addressing these interdisciplinary questions requires the integration of data and methods across research activities. **The way that data is stored, accessed and processed is therefore of central importance.** A considerable amount of research is dedicated to these aspects of information management in music, with knowledge representation playing a crucial role. The complexity of music as a subject of scientific study poses a great many challenges from the point of view of knowledge engineering, and there remain open problems.

1.1.3 Research Problem

Despite the overlapping aims of different research tasks, **computer systems for the communication, analysis and manipulation of music are in general not interoperable**. This problem stems from the underlying design of the representations involved. There exist numerous low-level encoding formats for musical information which generally cater towards a particular musical repertoire, notational convention or processing application. Meaningful comparison of these encodings is not possible without explicit tools for conversion. These tools are often developed ad hoc and embedded in larger software tools whose primary purpose is not the integration of data. In addition, these basic encodings do not support more sophisticated musicological information.

This problem can be addressed by applying the principles of knowledge representation. However, music is a highly complex domain and presents a number of specific challenges. **Many knowledge representation languages lack sufficient expressive power to capture sophisticated musicological concepts.** Music representations based on these languages do fully accommodate the complexity of music, resulting in application-specific models which are often not conceptually interoperable. In addition, music representations must support a multitude of different processing applications such as querying, data integrity checking, logical inference and application-specific (user-defined) analysis algorithms. Existing information management solutions often treat these kinds of applications separately. For example, knowledge representation systems are generally designed to support a specific kind of automated reasoning and do not easily integrate with programming languages in which users can design bespoke analysis algorithms. The multitude of different applications of music representations, combined with conceptual complexity of the domain, requires that representations be modular and layered. Abstraction and modularity are two aspects of software development whose purpose is to manage this kind of complexity. However, these design aspects are absent from many knowledge representation languages.

1.2 Research Question and Objectives

1.2.1 Question

Our motivation for the current work has highlighted the importance of knowledge representation, as well as aspects of software engineering, to the task of representing music in computers. We therefore declare the research question of this thesis as follows:

Question: How can the principles of knowledge representation and techniques of software engineering be unified to assist in the design of general purpose music representations which support information systems and software tools for computational musicology?

1.2.2 Objectives

To address the research question, we define the objectives of the current work as follows:

Objective I: Ascertain the major requirements of music representations. This requires establishing the diversity and commonality of different music research tasks in terms of their representation requirements and examining these requirements from the point of view of knowledge representation. Furthermore, we aim to examine and evaluate existing methods of music representation in terms of these requirements.

Objective II: Develop a general purpose music representation system. The representation system should address the issues and general requirements ascertained through Objective I. The representation system should be formally specified using software engineering methods to allow it to be implemented in computer systems for digital musicology. The specification for the system should be implementation-independent; it should not constrain the user of the system to any particular software platform of programming environment.

Objective III: Demonstrate the utility of the proposed representation. This requires an implementation of the system. The implementation must use mainstream technologies and tools so that it may be easily accommodated into existing systems and methodologies by researchers and developers. Finally, we must demonstrate the utility of the representation system by showing how it supports a variety of typical applications and musicological research tasks.

1.3 Research Methods and Scope

In this section we outline the methods that are used to tackle the research question of this thesis. In addition, we clarify the scope of the work by stating the bounds of our focus.

1.3.1 Research Methods

The method is divided into three parts, each of which addresses one of the objectives outlined in the previous section. These three methods are as follows:

Method I: Literature survey The first part of our method is a literature survey which addresses Objective I. We review work on the principles of knowledge representation and music representation and enumerate the specific requirements for a general purpose music representation system. We survey existing knowledge representation and music representation systems and consider them in terms of these requirements.

Method II: Specification The second part of the method is a formal specification for a music representation framework which addresses Objective II. The specification is given using dependent type theory and addresses the requirements identified. We give a functional specification of a multiple-hierarchical information model called a constituent structure. We define a specification logic for constituent structures in which the structural properties of represented entities can be formally expressed. We describe a number of extensions to the specification which realise a range of different representational features and requirements.

Method III: Demonstration The third part of our method is a implementation of the specification which addresses Objective III. We present a number of Semantic Web ontologies which capture the constituent structure model and specification logic of the framework. We implement a number of JavaScript modules which provide the core functionality of the representation. We illustrate how the representation can be used in support of three music analysis tasks. Finally, we use the implementation to build a demonstrator application which performs simple analysis and visualisation of musical material.

1.3.2 Scope of Research

It is important at this stage to clarify precisely the scope of the current work. Firstly, we emphasis that the main focus of the work is on the general methodology of computer-based music research. It is not concerned with any particular analysis algorithm, research task or method of computational musicology. We regard representation as being the cornerstone of computer-based methodology and a subject of study independent of any particular application. Secondly, the emphasis of this work is on formal specification at an abstract level. We are, in general, not concerned with concrete implementations of systems. We are not proposing a new data format or standard and are especially not interested in implementation details such as space and time complexity. We regard implementations as being concrete instantiations of the representation specification.

1.4 Thesis Outline and Contributions

This thesis is divided into four parts. Parts 1-3 include the major work of the thesis. Part 4 includes conclusions and future work.

1.4.1 Part I

Part I introduces the relevant background and related work. It includes the literature survey of Method I. The key contribution of this part is a review of the issues and existing approaches to knowledge representation and music representation. The part is divided into three chapters. Chapter 2 describes the relevant background and related work in the areas of type theory and category theory and describes their relevance to knowledge representation. Chapter 3 includes the relevant background in the topic of knowledge representation and reasoning. We discuss the methodological principles of knowledge representation with a particular focus on interoperability of representations. The key contribution of this chapter is a survey of a number of existing technologies with particular assessment of aspects which will be relevant to music. Chapter 4 is the principle literature review chapter, containing a discussion of the core issues of music representation. It draws on the background presented in the previous chapter to assess the requirements of general purpose music knowledge representations. The key contribution is a survey of existing approaches to music representation, with particular attention paid to Semantic Web representations and the CHARM system.

1.4.2 Part II

Part II contains the core theoretical work of the thesis. It describes a type-based framework for music knowledge representation. It is divided into three chapters. Chapter 5 describes a general purpose representation of music based on abstract data types, called constituent structures. The key contribution of this chapter is a functional specification of the constituent structure representation given in dependent type theory. Chapter 6 describes a specification logic for constituent structures. It provides an expressive language for capturing logical descriptions of the structure of represented entities. Chapter 7 describes several extensions to the both the constituent structure representation and the specification logic which capture a range of desirable representational features. This chapter aims to demonstrate how the type-based method supports the formalisation of many sophisticated representational approaches.

1.4.3 Part III

Part III describes an implementation of the representation using Semantic Web ontologies and JavaScript modules. The core contribution of this part is a demonstration of the type-based knowledge representation framework. Chapter 8 describes three Semantic Web ontologies which capture the different parts of the framework. It includes an ontology for representing constituent structures, an ontology for describing functional specifications of abstract data types based on type theory, and an ontology which captures expressions of the specification logic. Chapter 9 describes a number of JavaScript tools which implement the core functional behaviours of the representation framework, and provide a basis for developing music analysis applications. Chapter 10 gives concrete examples of the use of the framework, describing how the representation supports three different music analysis tasks: pattern search and discovery, paradigmatic analysis, and hierarchical set-class analysis. In addition, it presents a web-based demonstrator application, built using the JavaScript tools, to perform paradigmatic analysis and visualisation of linked data documents, structured by the ontologies.

1.4.4 Part IV

Finally, Part IV contains the conclusions of the thesis and proposals for future work. Chapter 11 summaries the contributions of the work and gives a broad discussion of the type-based approach to knowledge representation with reflection and comparison on existing approaches. Chapter 12 identifies aspects of the current work which require further development, as well as a number of potential avenues of future research within the topic.

Part I

Background and Literature Review

Summary

In this part, we introduce the requisite background material and give a review of related work. The current work draws on a wide variety of disciplines, including theoretical computer science, artificial intelligence, and computational musicology. These fields are not substantially cross pollinated, and as such a principle aim of this part is organise the requisite material in such a way as to draw out the connections between them. Therefore, we take care in clearly describing the connections between the topics and their application to computer representation of music. The part is divided into three chapters, each broadly dealing with a different discipline. Chapter 2 deals with topics from mathematics and computer science. It introduces two areas of mathematics, namely type theory and category theory, which have wide applications in computer science. It introduces the requisite background and reviews a number of applications, each of which serve to illustrate the relevance of these disciplines to the field of knowledge representation and reasoning. Chapter 3 presents the background material on the topic of knowledge representation and reasoning. It provides a high-level overview of the theoretical aspects of the field, and reviews a number of existing knowledge representations systems. Various theoretical aspects and existing technologies are highlighted due to their importance in the subsequent discussion of music representation. Chapter 4 deals with the topic of computer representation of music from the perspective of knowledge representation, with reference to the insight of the previous chapter. It highlights the specific knowledge representation requirements and challenges encountered in the domain of music. It reviews a number of existing music representation systems, regarding each in terms of its suitability as a general purpose music knowledge representation system.

Chapter 2

Type Theory and Category Theory

2.1 Introduction

This chapter introduces the requisite mathematical topics, namely type theory and category theory. Of the two, type theory plays a greater role in the current work. It is used as a formal basis for the music knowledge representation framework described in Part II. In this chapter we introduce the requisite type-theoretic material for what follows. Category theory plays a smaller role. However, its application in areas related to knowledge representation, as well as its deep connection to type theory, is what prompts its inclusion.

Chapter Outline

The outline of the chapter is as follows: §2.2 introduces the requisite background on type theory. In particular, it includes a description of the specific metalanguage and notation used in Chapter II, and a review of three applications of type theory which possess certain conceptual and methodological aspects in common with the current work. §2.3 introduces the requisite category-theoretic material and reviews two applications of category theory with close connections to the current work. §2.4 provides summary of the conclusions of the chapter, with a specific look ahead to the involvement of these mathematical topics in the discussion of knowledge representation and music representation.

2.2 Type Theory

2.2.1 Overview

What is Type Theory?

Type theory is concerned with formal systems for rewriting certain strings of symbols. It formalises the notion of a mathematical term, in particular the notions of variable, function and substitution. In type theory each term has a type. Terms are written as (explicitly typed) lambda terms and can be seen as a logical extension to the lambda calculus (Church, 1936).

Why is it Relevant?

In this thesis we propose the use of type theory as a basis for knowledge representation. The motivation for doing so is two-fold: First, **type theories are systems of constructive logic** via the Curry-Howard correspondence (Curry and Feys, 1958; Howard, 1980). Logic is central to the field of knowledge representation as a basis for mechanical reasoning. In this thesis we argue that constructive logic is an attractive alternative to classical first-order logic as basis for knowledge representation. Second, **type theories are mathematical formalisations of programming languages**. In this thesis we argue that a knowledge representation language *is* a programming language, and as such, the theory of programming languages should be involved in the task of knowledge representation. The Curry-Howard correspondence originated through the identification of a precise correspondence between intuitionist propositional logic and simply-typed lambda calculus. It has since been developed into the more general paradigm of "propositions as types; proofs as programs" in which every theorem statement of a logic can be identified with a type, and every proof of a theorem can be identified with a program of that type. This paradigm has been highly influential in theoretical computer science and logic and is the principle motivation behind the use of type theory in the current work.

2.2.2 Prerequisites

Judgements and Inference Rules

A type theory is a system of natural deduction for deriving typing judgements of the form $\Gamma \vdash M : A$, read as "M is a term of type A in context Γ", where M is a lambda term, A is called the type of M, and the context Γ is a sequence of variable decalrations $(x_1 : T_1, ...x_n : T_n)$ which includes the free variables of M and A. The symbol $\vdash$ denotes an entailment relation between contexts and terms. Other forms of judgement capture the notions of definitional equality between types, terms and contexts. For example, $\Gamma \vdash M \equiv N : A$ captures when M and N are equal terms of type A. The inference rules of the system govern the formation of types, the introduction and elimination of terms of those types, the computational properties of terms with respect to definitional equality, and the structural rules for contexts with respect to the entailment relation. For a full discussion of judgements and rules we refer the reader to Jacobs (1999). There are many such systems, each characterised by their rules for forming types and terms.

The Lambda Cube

Systems can be characterised by the variable dependencies which exist between the objects of the theory. The lambda cube (Barendregy, 1991) is a three dimensional cube-shaped graph which summaries the relationships between a number of different typed lambda calculi. Each vertex of the cube represents a typed lambda calculus and each edge is labelled with an arrow denoting the inclusion of one calculus within another. The three dimensions of the cube represent three forms of abstraction: terms depending on types (polymorphism), types depending on types (higher-order types or type operators), and types depending on terms (dependent types). The simplest calculus of the structure is the simply-typed lambda calculus (STT: Church, 1940), in which the only dependency is between terms and other terms. In STT, types are generated from a set of base types using the constructors $\rightarrow$, $\times$ 1, $+$ and 0. Moving along the edges of the cube introduces other kinds of dependency. For example, System F (second-order or polymorphic lambda calculus: Reynolds, 1974) is an extension to STT which allows terms to depend on type variables. The most expressive calculus of the cube is The Calculus of Constructions (CC: Coquand and Huet, 1988), a higher-order dependently-typed polymorphic lambda calculus, which admits all three kinds of dependency resulting in a highly expressive programming language and logical calculus.

Dependent Types

In this thesis we are interested in use of dependent types as they correspond logically to predicates. A typical example of a dependent type is the type

$$(n : \mathsf{Nat}) \vdash \mathsf{NatList}(n) : \mathsf{Type}$$

of lists of natural numbers of length n (where Nat is the type of natural numbers). The term $\mathsf{NatList}(n)$ is referred to as a family of types indexed by elements of the type Nat. Given such a family $B(x)$ indexed by terms of a type A, there are two ways of constructing dependent types called products and sums. The dependent product (or function) type, written $\Pi x : A.B(x)$, is the type of functions whose return type varies according to the value of the argument. For example, given $f : \Pi x : A.B(x)$ and $a : A$, $f(a)$ has type $B(a)$. The dependent sum type, written $\Sigma x : A.B(x)$, is the type of pairs (a,b) where a has type A and b has type $B(a)$. Dependent products and sums generalise the notions of function type and product type respectively. Π and Σ are both variable binding operations; they can be seen as the constructive equivalents of the logical operations $\forall$ and $\exists$, respectively.

Pure Type Systems

Pure type systems (Barendregy, 1991) are a unified framework for characterising dependently-typed λ-calculi. Terms of a pure type system are explicitly typed lambda terms with a single type constructor for dependent products, which are the type of lambda abstractions. Pure type system are characterised by a set of sorts (type universes) which capture the types of types, a set of axioms which define the typing rules for sorts and other constants of the systems, and a set of rules which specify the types of dependent products. All of the calculi in the lambda cube, as well as many others, can be characterised in this way.

2.2.3 The Calculus of Inductive Construction

When augmented with inductive types, CC becomes The Calculus of Inductive Constructions (CIC: Bertot and Castéran, 2004). CIC is a pure type system with an infinite hierarchy of type universes given by the set $\{\mathsf{Prop}, \mathsf{Set}, \mathsf{Type}_i | i \in \mathbb{N}\}$, where Prop is the type of logical propositions, Set is the type of data types (such as Booleans and natural numbers as well as function types and product types), and Type_i is the type of higher-order types. All sorts have a type which is also a sort. The typing of sorts is given by the axioms $\mathsf{Prop} : \mathsf{Type}_1$, $\mathsf{Set} : \mathsf{Type}_1$ and $\mathsf{Type}_i : \mathsf{Type}_{i+1}$ for all $i \in \mathbb{N}$. The hierarchy of type universes is cumulative meaning that any term of Type_i is also a term of type Type_{i+1}.

Inductive Definitions

CIC includes rules for introducing arbitrary inductive types, as well as a restricted form of general recursion. Inductive types are defined by a finite number constructor operations. Terms of an inductive type are the least set of objects generated by a finite number of applications of the constructions. Typical examples of inductive types include natural numbers (via Peano axioms) and lists (as in Haskell or ML). For the precise rules governing inductive definitions in CIC we refer the reader to Bertot and Castéran

(2004), however precise understanding of these rules is not required for understanding the present work, or the use of inductive types in general. Functional programming languages like ML and Haskell admit a simple kind of inductive type via their data type definitions. All the standard data types and type structure form mainstream functional programming languages, such as ML and Haskell, can be defined in CIC using inductive definitions. As such, CIC is a highly expressive system for both functional programming and logic, and it is used as the underlying logical language of the Coq proof assistant (Bertot and Castéran, 2004).

2.2.4 Notation Used in This Thesis

In Part II we give a formal specification for a music representation system using CIC. Our notation for the basic calculus is fairly standard. Variables are written x, A, while constants and sorts are written x, A. We write lambda abstraction as $\lambda x : A.t$, dependent products as $\Pi x : A.B$, and functional application using brackets as $f(t)$, with $f(t, u)$ short for $f(t)(u)$. (As usual, application associates to the left.) We write the dependent product $\Pi x : A.P$ as $\forall x : A.P$ when its type is Prop. We write dependent products $\Pi x : A.B$ as $A \to B$ when x is not free in the type B. We sometimes omit the type of bound variables in variable-binding operations. For example, $\lambda x.t$ is shorthand for $\lambda x : A.t$ when the type of x is clear from the context (or could be inferred). We also adopt a recursive style of notation for nested binding operations. For example $\lambda x : A.\lambda y : B.t$ is written $\lambda(x : A)(y : B).t$ or $\lambda x\ y.t$.

In addition to the core calculus, we assume (inductive) definitions for the following type forming operations in the syntactic universe Prop: $\exists x : A.P$, $\top$, $P \wedge Q$, $\bot$ and $P \vee Q$ stand for existential quantifier, true, conjunction, false and disjunction, respectively. We also assume the type $\mathsf{option}(A)$ to be the polymorphic type for capturing the partiality of operations. We use the type $\mathsf{fset}(A)$ for the type of polymorphic finite sets with the usual set theoretic operations and predicates.

For simplicity, we do not include type contexts in the specification of Part II; the type of free variables in expressions is given separately when it is not obvious from the context. Constants are introduced as either assumptions or definitions. Assumptions are written simply as typing judgements $\mathsf{C} : T$ where C is the defined constant and T is its assumed type. Definitions are written $\mathsf{C} := t$ or $\mathsf{C} : T := t$ where C is the defined constant, T is its type and t is the term it is definitionally equivalent to. All typing judgements are assumed to be derivable in a global context which includes previously given definitions.

2.2.5 Relevant Applications of Type Theory

Programming Language Type Systems

The most commonly found application of type theory is in the theory and design programming languages. A programming language can be viewed as collection of types; the type structure of a language determines the types of computation it performs. Type systems found in languages such as ML and Haskell are derived from particular type-theoretic constructs and allow for the correctness (type safety) of a program to verified at compile time. Recently, the functional paradigm of programming has increased in popularity, as the strong mathematical basis provides powerful formal methods of reasoning about programs. In §4.3 we discuss a number of domain-specific programming languages for music which have been developed in Haskell.

Theorem Proving and Verified Software

The connection between type theory and logic has led type theory to be used in the field of theorem proving and program verification (Chlipala, 2007). Traditionally, programming and theorem proving are treated as two distinct things, with separate mechanisms and tools being developed for each. The type-theoretical approach uses the Curry-Howard correspondence to unify these two activities. The idea is that typed-lambda calculi can be used as languages for writing formal program specifications and as concrete formats for writing proofs of these specifications. This is merely an extension of the convention of writing type annotations in the source code of statically typed functional programs. Proofs of these trivial specifications (the programs) are proofs of type safety. Generalising this idea leads to considering statically-typed programming languages which include an embedded representation of proofs, thus unifying the practises of programming and proving.

This unified view of programming and proving is the basis behind a number of type-theoretic proof assistants. Coq (Bertot and Castéran, 2004) is arguably the most advanced of the type-theoretic proof assistants and contains a lot of support for activities in the area of formal verification including a sophisticated tactic language for automatically constructing proof terms, program extraction, and an extensive standard library. Coq has been used in a wide variety of verified software projects, including verification of low-level programming languages (Chlipala, 2013), deductive synthesis of abstract data types (Delaware et al., 2015) and verified database systems (Malecha et al., 2010; Dumbrava, 2016). The current work shares some conceptual overlap with these verified software projects, and throughout the work will draw attention to these connections.

Knowledge Representation

Despite the strong connections between logic and programming provided by type theory, very little existing work fully explores the practical application of type theory to the task of knowledge representation and AI. One notable exception is K-DTT (Barlatier and Dapoigny, 2012), an ontology language based on dependent type theory. The core aspect of this work is the adoption of a constructive logic with proof theoretic semantics as a basis for representing ontological structures. Again, there is some conceptual overlap with this approach and the current work, and the K-DTT system is discussed further in §3.4.3.

2.3 Category Theory

2.3.1 Overview

What is Category Theory?

Category theory is a branch of mathematics concerned with the algebraic study of structure. It can be seen as the abstract algebra of functions. It originated in algebraic topology but has since become a fundamental component of mathematical foundations, logic and computer science.

It plays a smaller role than type theory in the current work. However, it has deep connections with type theory and as such has many applications within computer science. Some of these applications have considerable conceptual overlap with the current work. In this section, we present the minimal background required for understanding these connections so that our work may be compared with these other areas.

Why is it relevant?

Category theory provides provides a framework for formalising many other aspects of mathematics and has close connections with both type theory and systems of logic. As such, it has many applications in computer science (Goguen, 1991) as a formal language for the abstract description of systems. Category theory provides a way of understanding patterns in systems. The use of category theory in formal specifications provides a high degree of conceptual hygiene and allows many seemingly unrelated systems to be meaningfully compared and combined.

Many concepts from category theory have correspondences in functional programming. It is used to understand type structures in functional programming languages. Most famously, the notion of Cartesian closed category has been found to correspond with STT (Lambek, 1980). Indeed, type theory can be seen as a formal calculus for category theory. Both lambda calculus and categories provide languages for specifying, manipulating and calculating with mathematical functions.

2.3.2 Prerequisites

Category theory uses the abstract (primitive) notions of object and morphism to describe collections of entities and their structural relationships. These descriptions, called categories, are often regarded as directed graphs, with objects as vertices and morphisms as edges. The defining characteristic of a category however, is that the relationships expressed by the morphisms must be reflexive (every object is related to itself), transitive (relationships can be composed) and associative (the order of composition is unimportant). In this section, we give informal definitions of two concepts from category theory, namely *category* and *functor*, which will be required in order to understand what follows. For more formal definitions of the concepts we refer the reader to Awodey (2010).

Category

The fundamental concept from category theory is that of the category itself. A category consists of a collection of objects and a collection of 'arrows' connecting those objects, called morphisms. Each morphism has a domain and codomain which are objects of the category. For each object there is a specific morphism called the identity morphism, whose domain and codomain is that object. In addition, there is a partial binary composition operation on morphisms; given two morphism where the codomain of one is the same object as the domain of the other, we can compose them to create a morphism from the domain of the first to the codomain of the second. This composition operation must satisfy two laws: it must be associative and the identity morphisms are units.

Functor

A functor is a structure preserving map between categories. It maps objects of the domain category to objects of the codomain category, and morphisms of the domain category to morphism of the codomain category in such a way as to preserve the identity and composition of morphism. Functors in category theory are a very powerful abstraction. They can be thought of as particular ‘diagrams’ in the codomain category whose ‘shape’ is determined by the domain category. In other words the shape of the first category is rendered in the second. Functors can also be though of as morphisms in a category of categories, in which the objects are categories. A final level of abstraction enables functors themselves to be viewed as objects in categories whose morphisms are called natural transformations. The precise ways in which these constructions are defined are not required for the current work, however we allude to them here to give the reader a sense of the richness and power of the abstractions of category theory.

Example

The canonical example of a category is the category **Set** in which the objects are sets and the morphisms are functions between sets. Composition of morphisms is given by the standard composition of set functions, and so by definition satisfies the laws of identity and associativity. The category **Set** has additional interesting structure. For example, given any two sets, both the Cartesian product and the set of functions between them is also a set. These two structural aspects of sets are merely specific instances of the more general notions of product and exponential constructions in category theory. Categories which allow for both these constructions possess what is referred to as Cartesian-closed structure. It is these kinds of constructions in category theory which bare a close correspondence with certain structures in type theory and logic. Again, the formal details of these constructions and their type-theoretic and logical counterparts are not required for an understanding of this work. However, we want to be emphatic regarding the importance of the connection; one desired outcome of the type-based approach to knowledge representation presented in this thesis is to provide a basis from which insight from category theory can be utilised in future work. In the next section we describe two fields of research in which category theory has been applied which bare a strong resemblance to the current work.

2.3.3 Relevant Applications of Category Theory

Information Systems

Category theory has been applied in the areas of information science and the design of databases. In particular, The Functorial Data Model (Spivak, 2012) uses a variety of categorical constructions to formalise various aspects of relational database languages. The basic idea is that database schemas can be specified as categories in which the objects of the category are database tables or data types. Each column of a table corresponds with a morphism from that object to either another table (foreign keys) or to a data type object. Instances of these databases schemas are set-valued functors, i.e. functors which map every object in the schema to a set, and every morphism to a set function. This categorical framework provides many advantages over other ways of specifying database languages. Firstly, database schemas in this language integrate better with programming languages, due to close connection between categories and lambda calculi. Secondly, categorical database schemas have available to them the full formal power of category theory for defining schema mappings as functors. This in turn leads to the notion of functorial data migration (Spivak, 2012), whereby functors mapping between schemas give rise to canonical transformations between instances of those schemas. Ontology Logs (Spivak and Kent, 2012) use the Functorial Data Model as basis for a knowledge representation framework, in which the database schema categories are viewed as simple ontologies. Ontology Logs have many aspects in common with the type-based approach to knowledge representation presented in this work, and so is discussed in more detail in §3.4.4.

Music Representation and Analysis

Category theory has been extensively applied to the study of music by the mathematician and musicologist Guerino Mazzola. The Topos of Music (Mazzola, 2012) outlines in considerable detail, a method for applying category to the representation, analysis and manipulation musicological knowledge. It uses a specific category as the formal basis for defining a representation language for music consisting of two kinds of entity: *Forms* and *Denotators*. Forms represent musical concept spaces, and correspond with functors from the category of modules to the category of sets. Denotators represent (generalised) points in those spaces, and correspond with morphisms in the category of modules. The precise formal set up is highly sophisticated and not required for understanding of the current work. Instead, we restrict ourselves to a high-level discussion of The Topos of Music regarding two remarkable aspects. Firstly, as the work constitutes a comprehensive and highly sophisticated examination of many aspects of music representation and formal analysis, it is remarkable how little it is referred to in the broader music research community. This is in part due to the complexity of the mathematics involved. However, there few aspects of music representation and analysis which are not addressed in some way throughout the work, and so there is a lot of potential for comparison with other work which has not been carried out. Secondly, it is remarkable how, throughout Mazzola's work, virtually no connection with type theory is drawn. Mazzola remarks on the appropriateness of the object-oriented programming paradigm as medium for implementing Forms and Detonators, and indeed the partial implementation Rubato Composer (Milmeister, 2014) is implemented in Java. However, the connection between functional programming languages and categorical structures of the kind used by Mazzola are reasonably well understood. It seems reasonable to believe that careful examination of the theory of Forms and Denotators from the perspective of type theory may reveal close connections to other areas of computer science and programming language theory. Whilst this kind of examination is not included within the scope of the current work, we do endeavour to address where possible the overlap and potential scope for further exploration. In addition, we provide a more detailed discussion of the Rubato Composer system in §4.3.

2.4 Conclusions

In this chapter we have introduced the topics of type theory and category theory and highlighted their connections with the topics of this thesis. We have given an overview of two applications of type theory, namely programming languages and verified software, and two applications of category theory, namely information systems and mathematical music theory, with strong relevance to the current research. The advantages of type theory as a formal specification language have been used extensively in computer science from the simple type-safety of programming languages to large verified software projects. The advantages of category theory as a high-level conceptual framework for understanding make it a useful organisational tool in software development. In the current work, category theory will play a subservient role to type theory although many aspects are expressible in both formalism.

Chapter 3

Knowledge Representation and Reasoning

3.1 Introduction

What is Knowledge Representation?

Knowledge Representation is a field of Artificial Intelligence concerned with formal systems for representing knowledge in computers. It is a predominantly theoretical practice, drawing on disciplines such as information science, cognitive psychology, mathematical logic and metaphysics. The aim is to represent information in a way such that computer systems can use it to solve problems. The use of represented knowledge to solve tasks is often referred to as automated reasoning (Brachman and Levesque, 2004).

Why is it relevant?

The ability of computer systems to share and automatically reason with information is the main motivation behind the practice, as well as the assertion that "better knowledge can be more important for solving a task than better algorithms" (Meghanathan et al., 2013, pp. 439). The relevance of knowledge representation in music has been widely acknowledged (Balaban, 1996; Wiggins and Smail, 2000; Fields et al., 2011) and many systems and technologies have been used as music representation methods (Wiggins et al., 1989; Marsden, 2000; Abdallah et al., 2006).

Chapter Outline

In this chapter we give a high-level overview of the issues and existing systems in the field of knowledge representation, identifying those aspects which will be of particular importance when considering music. §3.2 takes a closer look at the principles of knowledge representation from the perspective of AI systems. §3.3 discusses interoperability of representations, describing two distinct approaches. §3.4 contains a survey of existing knowledge representation technologies. §3.5 gives a more detailed examination of knowledge representation on the Semantic Web. §3.6 summarises the chapter and states the main conclusions.

3.2 Definition and Requirements of Knowledge Representation

In this section we clarify the definition of knowledge representation, and in doing so discuss some of the fundamental aspects of the field. The difficulty in defining knowledge representation stems from the difficulty of defining *Knowledge* itself. Knowledge is often defined by its opposition to other terms such as *data* and *information* (Ackoff, 1989; Rowley, 2007), however these definitions are informal, and introduce philosophically thorny terminology. Brachman and Levesque (2004) regard knowledge as propositional; a relationship between a knower and a logical proposition. Wiggins (2000) regards knowledge representation in a technical sense by its opposition to data encoding in the sense of Fisher et al. (2006).

Davis et al. (1993) give perhaps the most useful definition of knowledge representation (KR) in terms of five roles. The roles define what KR *is* from a number of different perspectives. From each one of these perspectives, it is possible to, at a high level, distinguish a number of particular requirements which should be considered in the design of knowledge representation systems. In this section we examine each one of Davis' roles (with a sub-section for each), highlighting what we believe to be their consequences from the perspective of system requirements. This section is intended to lay the groundwork for the more detailed survey of existing knowledge representation systems presented in §3.4, and the discussion of the music knowledge representation presented in Chapter 4.

3.2.1 Role I: A KR is a Surrogate

Represented knowledge exists internally to some agent in approximation of some portion of reality, allowing the agent to reason by acting on the representation rather than reality itself. The intended identity (meaning or denotation) of a representation, that is to say, the thing that it is a surrogate for, is often formalised through the use of a formal semantics, which defines the connection between the representation and a mathematical model of a conceptualisation. A formal denotational semantics constitutes a translation from one language (the knowledge representation language) to another (typically mathematics) in which it can be better understood in some sense. It is therefore an important aspect of any knowledge representation formalism that it be explicit about its denotation according to some mathematical model.

3.2.2 Role II: A KR is a Set of Ontological Commitments

A representation system must select certain aspects of reality to capture and omit others. These choices constitute what Davis refers to as *ontological commitments*. They can be understood as an answer to the question "How should we view the world?" The ontological commitment of a representation system is accumulated in layers, starting at the level of the modelling primitives of the language and accrued through the definition of vocabulary terms for a specific domain. Davis' use of the term "ontological commitment" differs considerably from how it sometimes used in the knowledge engineering community (Guarino and Oberle, 2009). The word 'ontology' is widely used in computer science to refer to formal models which specify how we should view the world. It is most commonly defined as an "explicit specification of a conceptualisation" (Gruber, 1993, p. 1). There has been some work dedicated to the clarification of this definition and surrounding terminology (Guarino and Oberle, 2009). Gruber (1993) used the notion of conceptualisation from Genesereth and Nilsson (1987) who defined it as a set-theoretic, extensional relational structure. Guarino and Giaretta (1995) provide an alternative, intentional definition in terms of possible worlds. In this set up, ontological commitment is defined as "a partial semantic account of the intended conceptualisation of a logical theory" (Guarino and Giaretta, 1995, p. 31). It is therefore an important aspect of knowledge representation formalisms that the layers ontological commitment of a system can be adequately controlled through the use of dedicated ontology languages.

3.2.3 Role III: A KR is a Fragmentary Theory of Intelligent Reasoning

A KR system explicitly or implicitly utilises or embeds a theory of how reasoning is performed by intelligent agents via the specific inferences which it supports. Davis et al. (1993) considers this role in terms of three parts: the fundamental conception of intelligent reasoning, the set of sanctioned inferences, and the set of recommended inferences. Theories of intelligent reasoning usually originate in different academic disciplines, including mathematics (logical deduction), psychology (frames and prototypes), neuroscience (neural networks) and statistics (causal networks). A knowledge representation should, therefore, be explicit in defining the modes of reasoning which it supports, so that the behaviour of any inference engine be transparently understandable. In this thesis we will primarily focus on methods of knowledge representation based on mathematical logic. These methods constitute formal languages whose meaning can be understood in terms of logical systems and so reasoning with these representations therefore amounts to some form of deduction.

3.2.4 Role IV: A KR is a Medium of Efficient Computation

Reasoning in machines is, by definition, a matter of computation. As such, a knowledge representation language *is* a programming language; a knowledge base is a program which must be parsed, interpreted or compiled by the computer system that is to reason with it. A central part of programming language specification is their formal semantics. There are many ways of defining the semantics of programming languages, including denotational, operational and axiomatic semantics. This role is connected to Role III by substantial technical consequences. Theories of intelligent reasoning, such as rule-based deduction or frames, each come tethered to certain paradigms and methods of computation, for example forward and backward chaining or classification. An important aspect of the design of knowledge representations is therefore that the connection between representation and computation (reasoning) be explicitly defined, via formal methods such as those used in the theory of programming languages.

3.2.5 Role V: A KR is a Medium of Human Expression

As with any programming language, a knowledge representation language must act as a form of communication between human programmers and machines. Two important aspects of programming languages are firstly, what ideas can be expressed, and secondly how intuitively and concisely these ideas can be expressed. This role is particularly divergent from Role IV; a medium of efficient computation is not necessarily a good medium of human expression. The design of programming languages inevitably involves striking a balance between human and machine readability. Typical programming systems involve layers of languages including low-level languages such as assembly, mid-level languages such as C, and high-level or domain-specific languages. The automation of translating between these different levels (compilation) is a key aspect of the design of software development tools. It is therefore important for the design of knowledge representation systems, to carefully control the relationships between high-level, user-friendly languages, and low-level, machine-usable encodings of data.

3.2.6 Summary

The roles help to clarify the definition and goals of knowledge representation, and provide a useful framework within within which to compare different technologies. Understanding knowledge representations in terms of these roles helps to design systems which better meet the requirements, as well as affording greater interoperability between systems, via the combination of different languages and modes of reasoning (Davis et al., 1993).

3.3 Conceptual Interoperability

Interoperability is the ability of computer systems to share information and functionality. It is important in information systems used for scientific research in order to facilitate the sharing of knowledge between researchers. It is particularly important in computer-based music research due to the significant benefits afforded by the sharing and integration of knowledge from the different constituent disciplines. In this section we attempt to clarify the definition and important aspects of interoperability in information systems, and review a number of methods employed.

3.3.1 Levels of Interoperability

The Levels of Conceptual Interoperability Model (LCIM: Tolk and Muguira, 2003; Wang et al., 2009) identifies six levels which characterise the different ways in which information systems can share understanding. These levels are defined as follows:

Level 0: No Interoperability Systems are centralised and isolated. No communication is possible.

Level 1: Technical Interoperability A communication protocol exists allowing systems to share bits and bytes.

Level 2: Syntactic Interoperability A common data format exists allowing systems to share structured data.

Level 3: Semantic Interoperability A common reference information model (ontology) is used allowing systems to share the meaning of structured data.

Level 4: Pragmatic Interoperability A common set of data processing methods and procedures is used allowing systems to share functionality.

Level 5: Dynamic Interoperability A common protocol for state and effects is used allowing systems to communicate synchronously.

Level 6: Conceptual Interoperability A common conceptual model is used in which a full and formal specification of the problem domain is given.

The LCIM was intended to give a systems engineering perspective to the task of modelling and simulation (Wang et al., 2009) and has been successfully applied to the development information systems such as digital libraries (Kostelic, 2017). The top level, conceptual interoperability, is highly desirable from the point of view of knowledge representation as it allows information system components, such as data schemas and ontologies, to be composed (Wang et al., 2009). Next, we examine two contrasting approaches to conceptual interoperability in ontology-based information systems.

3.3.2 Ontology Alignment

Ontology alignment refers to methods of specifying the formal relationships between ontologies. It constitutes a bottom-up approach to conceptual interoperability, introducing ad hoc methods of integrating existing information. Here, we examine approaches to ontology alignment based on category theory.

Cafezeiro and Haeusler (2007) use category theory to formalise various ontology operations such as alignment, merging, integration and matching. The method involves defining a category of *ontology structures* in which certain universal constructions can be used to combine and decompose ontologies. In this approach, the categorical definition of ontology structure is very basic, and morphisms of ontology structures do not preserve the logical content of ontology axioms. Zimmermann et al. (2006) use a more expressive category-theoretic notion of ontology to define a category of ontologies. They formalize various kinds of alignments as category-theoretic structures which form the basis of an algebraic language of ontology operations. These two approaches treat ontologies as (set-theoretic) objects of a category. An alternative approach to ontology alignment is that of the Functorial Data Model (Spivak, 2012) and Ontology Logs (Spivak and Kent, 2012) (see §2.3 for an infromal overview), in which schemas or ontologies are defined as categories and their alignments are defined via functors. As conceptual links between ontologies, functors are much richer than the morphisms used by Cafezeiro and Haeusler (2007) as they preserve logical structure. (Spivak and Kent, 2012) demonstrates how functors can be used to define systems of interconnected ontologies and, in doing so, shows how these systems can be used to capture Sowa's Lattice of Theories (Sowa, 2000) and Information Flow (Barwise and Seligman, 1997). The afforded conceptual interoperability is a considerable advantage of the categorial approach, and a more detailed examination of Ontology Logs is given in §3.4.4.

3.3.3 Upper Ontology

Upper ontologies are high-level, domain-neutral ontologies, designed to guide the development of interoperable, non-overlapping domain ontologies. It constitutes a top-down approach to conceptual interoperability, prescribing a common ontology architecture. Here, we examine three existing upper ontologies, and discuss their general suitability to the domain of music.

The Basic Formal Ontology (BFO: Arp et al., 2015) was developed to support the design and integration of data models for bio-informatics. It consists of a relatively small taxonomy of classes, developed from extensive and systematic ontology research. It has been highly successful in achieving interoperability of information, overseeing a suite of sub-ontologies each capturing a different corner of scientific research (Smith et al., 2007). However, it is highly focussed on representing objective reality from a scientific point of view; it contains no top-level classes for capturing entities which have subjective existence, such as perceived musical sounds. This is a disadvantage from the point of view of music representation, as much of the musical domain is distributed across perceptual, cognitive and physical realities (see Chapter 4).

The Descriptive Ontology for Linguistic and Cognitive Engineering (DOLCE: Gangemi et al., 2002) aims at capturing high-level concepts and relations from a more human cognitive point of view. It has much in common with BFO. Despite catering more towards descriptive representations of cognitive artefacts, it has clear bias towards the representation of linguistic concepts. From the perspective of music, linguistic concepts are insufficient as a basis for knowledge representation, as many musical concepts are linguistically ambiguous and tied to the perception of sounds.

The Suggested Upper Merged Ontology (SUMO: Niles and Pease, 2001) is another successful and widely used ontology project. However, it is less suitable than either BFO or DOLCE as an architecture for interoperable domain ontologies as it does not cleanly separate domain knowledge from the high-level concepts. For example, it contains descriptions of concepts from mathematics and biology. This means it cannot cleanly support the top-down method of conceptual interoperability (Arp et al., 2015).

3.4 Knowledge Representation Systems

In this section we review a number existing knowledge representation technologies. We examine each of them with reference to the five roles of Davis et al. (1993) and associated requirements given in §3.2. In particular, we examine their representational semantics, modelling primitives, their computational semantics and their expressive power.

3.4.1 KIF and FOL-Based Representations

The Knowledge Interchange Format (KIF: Genesereth, 1991) is a knowledge representation language based on classical first-order logic (FOL). KIF uses a lisp-like syntax (McCarthy, 1960) to encode information about objects, functions and relations. The meaning of KIF (Role I) is given by a model-theoretic semantics. The ontological commitment (Role II) is controlled by definitions of specific objects, functions and relations. The fundamental conception of intelligent reasoning (Role III) is given as a logical entailment relation, defined via the model-theoretic semantic. This implicit theory of reasoning is realised computationally by inference engines which typically perform forward and backward chaining and resolution. However, the way this is performed is not strongly guided by the representation. In addition, FOL-based representations have limited automated reasoning due to their undecidability. This can be addressed by restricting the language. Description Logics (DLs: Baader et al., 2007) are particular subsets of FOL with syntax aimed at capturing clear and concise descriptions of concepts. In this way, DLs sacrifice the logical expressiveness of FOL in favour of tractability and decidability of reasoning methods. DLs form the basis of knowledge representation languages on the semantic web (see §3.5). As for roles IV and V, KIF is neither a programming language nor is it intended to be a human language. Its primary purpose is as a concrete format for information exchange (Genesereth, 1991). As a medium of human expression, KIF is highly expressive, being able to represent arbitrary FOL formula, however it is not well suited to higher-level conceptual modelling (Corcho and Gomez-Perez, 2000), requiring augmentation with additional ontology languages such as Ontolingua (Gruber, 1992).

KIF has a number of related descendants. Common Logic (CL)[1] is an abstract syntax for FOL formula with a number of specific 'dialects'. Defining a language in this way has the advantage that it separates the logical meaning of represented knowl-

[1]The ISO standard that defines Common Logic is found at https://www.iso.org/standard/39175.html

edge from the concrete syntax used to express it. SUO-KIF is a variant of KIF that is used as the language underlying SUMO (Niles and Pease, 2001), and includes explicit language constructs to improve human-usability. Ontolingua (Gruber, 1992) is an ontology language built upon KIF and the Frame Ontology (Gruber, 1993), which introduces object-oriented descriptions of ontology components to create a more high-level, user-friendly modelling language.

Many other knowledge representation systems are based on FOL. Prolog (Bratko, 2012) is a logic programming language which uses rules in the form of Horn clauses. Datalog (Huang et al., 2011) is a subset of Prolog for defining and querying deductive databases. The Rule Markup Language (RuleML: Boley et al., 2001) is a subset of Datalog for for modelling production and business rules. The Rule Interchange Format (RIF: Kifer, 2008) and The Semantic Web Rule Language (SWRL: Horrocks et al., 2004) are both further restrictions of RuleML for integrating rules with semantic web ontology languages based on description logic. These languages are intended to provide common formats for the exchange of knowledge between information systems and as such do not provide the high-level modelling constructs of ontology languages (Corcho and Gomez-Perez, 2000). All these FOL-based representations are untyped, or more precisely, singly-typed (Jacobs, 1999). This inhibits the integration of represented knowledge with programming language data types (Cook and Ibrahim, 2006). In KIF and related languages, functions are understood as special types of relations. As such, they do not naturally support the representation of mathematical knowledge or algebraic theories[2]. This will turn out to be a significant and important aspect of musical knowledge, in which the algebraic structure of concepts is a central part of reasoning with and manipulating the representation.

[2]https://www.w3.org/TR/owl2-dr-linear/ is a proposed extension to OWL (Motik et al., 2009) for reasoning with the algebraic properties of numbers.

3.4.2 KL-ONE and Object-Oriented Representations

Frames (Minsky, 1974) are a general purpose knowledge representation method designed as data structures for representing stereotyped situations. A frame consists of a number of labelled fields called *slots* into which information can be put about the situation. Slots may contain data values, procedures or other frames. The frames paradigm was proposed primarily as a theory of intelligent reasoning which focuses on the organisation of concepts (Davis et al., 1993). However, they suffered initially from a lack of formal semantics and vague definitions of their components (Woods and Schmolze, 1992).

KL-ONE (Brachman, 1978) is an ontology language based on the frame paradigm. The aim was to marry the formality of FOL for automated reasoning with the organisational power of frames. The basic modelling primitive is the *Concept*. The local, internal structure of concepts is captured by roles (labelled slots) and *structural descriptions* which capture complex relationships and constraints amongst roles. Collections of concept descriptions can be understood as structured inheritance networks (with multiple inheritance). The expressive power of the concept description formalism can be understood in terms of FOL (Schmolze and Isreal, 1983), and as such supports classifiers which perform automatic deduction and validation (Schmolze and Lipkis, 1983). However, Brachman and Schmolze (1985, pp.192) acknowledges that "The need to handle the various possible relations among Roles makes the technical details of Structural Descriptions a bit messy."

KL-ONE influenced many subsequent knowledge representation systems (Woods and Schmolze, 1992). The most comprehensive and expressive of these is the LOOM system (Bates and MacGregor, 1987). LOOM incorporates a large number of representational features including production rules and procedural attachments, making it extremely powerful. However, its complexity makes it daunting as a solution for many applications. In addition, many of its representational features, such as effects, are not easily specified formally for the purposes of pragmatic interoperability (see §3.3.1). Frame Logic (Kifer and Lausen, 1989) is a syntactic extension of FOL with object oriented modelling constructs. It is less expressive than LOOM, but enjoys a formal logical semantics (Corcho and Gomez-Perez, 2000).

3.4.3 K-DTT and Type-Theoretic Representation

K-DTT (Barlatier and Dapoigny, 2012) is an ontology language based on dependent type theory. Ontological components are represented by dependent types, while individuals are represented by proof objects. The system consists of two layers. The lower layer is the is a type theory called the Extended Calculus of Constructions (ECC: Luo, 1989) and provides the logical basis. The upper layer defines various ontological primitives, such as classes, relations, properties and roles, as syntactic classes of terms from the lower layer.

The resulting ontology language is capable of formally capturing many ontological constructs and meta-properties of ontology components such as roles (Barlatier and Dapoigny, 2012), contexts (Dapoigny and Barlatier, 2010a) and part-whole relations (Dapoigny and Barlatier, 2010b). A notable feature of the approach is that the truth of logical statements is associated with formal proofs. This constitutes a departure from classical FOL approaches in which the theory of reasoning is based upon a system of constructive logic with a proof-theoretic semantics, rather than a model-theoretic one. This leads to a system which adopts an intermediary position between open and closed-world assumptions called the regular world assumption (Barlatier and Dapoigny, 2012).

The type-theoretic basis of K-DTT provides solutions to many formal problems in the areas of ontology. However, it has some disadvantages. For example, K-DTT uses type universes (sorts)and the cumulativity relation (see §2.2) to model upper-level classes and class subsumption, respectively. However, the universe hierarchy in ECC is linear. It is therefore not clear how, for example, K-DTT could be used to axiomatise domain-specific disjoint subclasses of upper-level classes. Secondly, representing individuals as variable declarations in type contexts prevents reasoning about the identity of individuals; the structural rules of ECC do not distinguish between two distinct variables of the same type.

3.4.4 Ologs and Category-Theoretic Representations

Ontology Logs (Ologs: Spivak and Kent, 2012) are a method for representing ontologies based on the Functorial Data Model Spivak (FDM: 2012). The modelling primitives are *types* (objects), *aspects* (functions) and *facts* (equations), which are understood as generators for categories. Ologs constitute a simple knowledge representation method which inherits all of the simplicity and functionality of the FDM, including instance definitions (as set-valued functors), functorial data migration and programming language integration (Spivak, 2012). Ologs are intended to be subjective world views, however the formalism does not include any higher-level structuring of ontologies. However, the category-theoretic basis affords conceptual interoperability via functors (see §3.3.2).

The categorical approach constitutes a significant paradigm shift compared to many alternative knowledge representation systems. The basis of the shift is the use of types and functions as modelling primitives rather than sets and functions. The Olog method is limited in two respects. Firstly, the Olog formalism is logically equivalent to multi-sorted equational logic (Jacobs, 1999), which is insufficient for some domains. However, Ologs are, in principle, able to internalise more expressive logical structure through the incorporation of sub-object classifiers into categorical models (Spivak and Kent, 2012), which essentially embeds a logical system, such as a boolean logic, as an algebraic theory in a category. Secondly, software implementations of the FDM for working with categorical data[3] are relatively primitive standalone applications, and have not been incorporated into more general software development ecosystems. Finally, the Olog formalism does not internalise a notion of proof and functional programming like K-DTT does.

[3]http://categoricaldata.net/fql.html

3.4.5 Conceptual Spaces

The theory of conceptual spaces (Gärdenfors, 2000) is an alternative approach to knowledge representation, developed as a meta-theory of cognitive representation on the same level as the symbolic or connectionist approaches. The theory of reasoning it embodies is aimed at capturing semantic similarity, and concept creation and combination. The modelling primitives are geometric spaces, regions and points. A conceptual space is comprised of a number of *quality dimensions*, intended to capture perceived aspects of objects. Particular individuals are identified with points in conceptual spaces; a collection of values, one for each quality dimension. The theory of conceptual spaces nicely incorporates many cognitive theories such as similarity, concept formation, natural concepts and prototype theory, via geometric methods.

Adams and Raubal (2009a) give a specification for a metric conceptual space algebra, including a structural description of the conceptual space representation and core operations for inclusion of points in regions, similarity operations between instances and concept regions and concept combination operations. This framework for working with conceptual spaces forms the basis for the Conceptual Space Markup Language (CSML) (Adams and Raubal, 2009b), a concrete format for representing geometric knowledge bases.

The perceptual and cognitive aspect of conceptual spaces means that they have limited applicability in some knowledge representation applications. However, they are attractive for application to music because music is fundamentally a perceptual and cognitive subject. Many aspects of music fit the paradigm of spaces and points. More generally, it is the formalisation of perceived aspects of objects as algebraic structures which makes the theory highly relevant to music knowledge representation.

3.5 Knowledge Representation on the Semantic Web

The Semantic Web is a collection of W3C standards and technologies aimed at supporting the publishing and reuse of information on the web. A large amount of the current research in knowledge representation and information systems is focussed on the Semantic Web due to the benefits afforded by linked open data. Many music researchers have also sought to harness the power of Semantic Web as platform for sharing knowledge (Raimond, 2008; Fields et al., 2011; Wissmann, 2012). These music-specific endeavours are discussed in detail in §4.4. In this section we review the core Semantic Web technologies, RDF (Lassila and Swick, 1999) and OWL (Motik et al., 2009), and consider their general limitations as knowledge representation languages.

3.5.1 Technologies and Standards

The Resource Description Framework (RDF: Lassila and Swick, 1999) is a data model and a declarative knowledge representation language. The RDF data model structures information about web resources as collections of triples called graphs. RDF graphs support querying via the SQL-like query language SPARQL[4]. The RDF language regards triples as assertions of atomic propositions and supports logical entailment via a model theoretic semantics. The RDF[5] and RDF Schema (RDFS)[6] vocabularies provide terms which can be used as simple ontology languages. Each vocabulary supports a different entailment regime via a corresponding extension to the model theoretic semantics[7]. RDF is a uniform solution for representing linked data, however it lacks expressive power as a knowledge representation language.

The Web Ontology Language (OWL: Motik et al., 2009) is a family of knowledge

[4]http://www.w3.org/TR/sparql11-overview/
[5]https://www.w3.org/TR/rdf-concepts/#section-URIspaces
[6]https://www.w3.org/TR/rdf-schema/
[7]https://www.w3.org/TR/rdf11-mt/

representation and ontology languages based on description logics, which add greater expressive power to that provided by the RDF modelling languages. OWL 2 DL[8] is based on the SROIQ(D) description logic profile (Baader et al., 2007) and is the most expressive of the languages. It has two different semantics: the Direct Semantics[9] directly corresponds with the standard description logic semantics, while the RDF Semantics[10] is given as extension to the RDFS semantics. OWL 2 DL supports automated reasoning, and a number of dedicated reasoners have been developed including FaCT++ (Tsarkov and Horrocks, 2006), Hermit (Motik et al., 2009) and Pellet (Sirin et al., 2007).

3.5.2 Relevant Existing Ontologies

Meta-Data and Provenance

There exist a wide variety of RDFS and OWL ontologies which are commonly used in research communities. Vocabularies such as The Simple Knowledge Organisation System (SKOS)[11], The Dublin Core Metadata Initiative (DCIM)[12], The Friend of a Friend Vocabulary (FOAF)[13] and the PROV Ontology[14] are used extensively for structuring metadata and provenance information.

OWL Time

One ontology that is relevant to the discussion of music is the OWL Time ontology[15]. It provides a vocabulary for representing the temporal structure of information including temporal entities, such as instants and intervals, and the temporal relations of Allen (1984). However, the semantics of the temporal relations, in terms of the algebraic

[8]https://www.w3.org/TR/owl2-syntax/
[9]https://www.w3.org/TR/owl2-direct-semantics/
[10]https://www.w3.org/TR/owl2-rdf-based-semantics/
[11]https://www.w3.org/2004/02/skos/
[12]http://dublincore.org/schemas/rdfs/
[13]http://xmlns.com/foaf/spec/20140114.html
[14]https://www.w3.org/TR/prov-o/
[15]https://www.w3.org/TR/owl-time/

properties of the temporal values of entities, is not constrained by the axioms of the ontology. The ontology includes a class of Temporal Reference Systems but does not provide any formal definition of the concept. The fundamental unconstrainedness of the OWL Time ontology is a symptom of a more general limitation of OWL in capturing the algebraic properties of data types. Ma et al. (2016) propose an ontology which captures semantic specifications of data types. However, the focus is on the high-level human interpretation of data values rather than their computational or algebraic properties for the purposes of reasoning.

Conceptual Spaces on the Semantic Web

Raubal et al. (2010), acknowledging this lack of expressive power, uses CSML to demonstrate the utility of conceptual spaces as knowledge representation on the semantic web. The introduction of conceptual spaces, with explicit geometric semantics, solves, albeit in a application-specific way, the problem of representing algebraic structures in conjunction with OWL ontologies.

3.5.3 Limitations of The Semantic Web

Logical Expressivity

OWL only works well for binary relations. A common solution to this problem is to represent associations as concepts (Jacobson et al., 2009); an n-ary relationship between entities is itself represented as an entity which is then associated with its relata by binary relations. The problem with this solution is that there exists no standard way of going about it. As a result different models capture n-ary relations in different non-compatible ways[16].

OWL 2 DL requires that the sets of classes, properties and individuals be disjoint. This means, for example, that classes cannot be made instances of meta-classes and relations cannot associate individuals with classes. OWL 2 DL under the Direct Se-

[16]see https://www.w3.org/TR/swbp-n-aryRelations/ for a discussion of different approaches.

mantics[17] allows “punning”[18], whereby the same name can be used for a class and an individual or property. However, these different uses are treated completely separately in the Direct Semantics, meaning that individuals representing the same entity are not necessarily interpreted as equivalent classes. OWL 2 Full under the RDF Semantics[19] removes these restrictions but has no reasoning support.

The Open World Assumption

The Semantic Web represents knowledge using the Open World Assumption (OWA), meaning information is assumed to be incomplete by default. Whilst this allows models and knowledge bases to be extended without invalidating any previous conclusions, some problems are inherently closed world. For example, consider a knowledge base which represents the notes of a piece of music. In an open world, asking questions which inherently quantify over all the notes of the piece will not necessarily give the desired result as the absence of any involved information is not enough to deduce that the answer is ‘no’. This can be mitigated by incorporating concept closures axioms into logical models (Rector et al., 2004). However, this is generally non-trivial and can require a large number of additional axioms. In the case of the musical piece example we would require an axiom to close the concept of the piece which explicitly enumerated every single one of the notes. Another solution is provided by SPARQL Inferencing Notation (SPIN)[20], a W3C member recommendation for representing rules and constraints by encoding them as SPARQL queries. This has the disadvantage that the logical meaning of the constraints is not contained within the ontology.

Representing Structured Objects

OWL has the tree model property (Vardi, 1996), which is exploited by automated reasoners for decidable model-checking. However, it means that certain things cannot

[17]https://www.w3.org/TR/owl2-direct-semantics/
[18]https://www.w3.org/2007/OWL/wiki/Punning
[19]https://www.w3.org/TR/owl2-rdf-based-semantics/
[20]http://spinrdf.org/

be modelled easily. In particular structured objects, that is objects consisting of a number of interconnected parts, cannot be expressed in OWL (Motik et al., 2008; Wissmann, 2012). Ontologies of domains which include such objects are often left under-specified and so do not support the desired inferences. The SWRL rule language can be used to axiomatise such structured objects but the combination of SWRL and OWL is undecidable (Parsia et al., 2005). Motik et al. (2008) proposed a solution to the problem by an extension to OWL called Description Graphs. However, these have not yet been standardised and no tool support exists.

Restriction Semantics

The semantic web represents knowledge using restrictions. Restrictions are axioms of a logical theory which restrict the number of permissible models and so increase the number of valid inferences that can be made. Constraints on the other hand are not axioms of the logical theory but instead capture properties which prevent invalid values being inserted into the model. A basic example of the difference is the domain and range properties of the RDF schema vocabulary. A model specifying that the domain of a property P is the class C allows a reasoner to infer that any individual with that property is of class C; it does not prevent P-property assertions being made on individuals who are not explicitly of class C. The distinction is subtle but is an important feature of Semantic Web modelling languages. The lack of constraints is most noticeable when trying to reconcile linked data models and other database schemas or data structures in programming language (Cook and Ibrahim, 2006). Database systems typically include mechanisms for data integrity constraints whilst programming languages constrain data via type systems.

Programming Language Integration

The lack of constraints in the fundamental barrier between the integration of semantic web technologies and programming languages. Semantic Web knowledge bases cannot be fitted with data integrity constraints. For example, aggregate data structures such as lists cannot be enforced using Semantic Web technologies. The RDF vocabulary includes terms for their description but their use is not compatible with OWL. The OWL list ontology axiomatises list structures but their is no standard tool support and the lack of constraints means their use in processing applications is limited. They are mainly intended for reasoning about sequences, rather than providing data structures for data processing applications.

No Conceptual Interoperability

Finally, the semantic web is limited in the level of interoperability it provides. The variety of different entailment regimes and OWL profiles means that these technologies do not entirely guarantee semantic interoperability. In addition, the lack of formal meta-modelling makes it difficult to achieve the level of conceptual interoperability. Ontologies are designed across the semantic web in ad hoc ways to model overlapping domains. Integrating information annotated with these ontologies, as any researcher involved in semantic web will attest, takes considerable manual effort. Many tools exist to assist integration of ontologies (McGuinness et al., 2000; Fridman and Musen, 2000), however these tools are not fully automated, requiring input from the user about the best way to proceed. The resulting alignments are as ad hoc as the original ontologies. Upper ontology is a less ad hoc method as it guides the design of non-overlapping domain ontologies (see §3.3.3).

3.6 Summary and Conclusions

In this chapter we have introduced the key aspects of knowledge representation and reasoning including its roles in AI systems and the notion of conceptual interoperability. We reviewed a number of existing technologies and examined them in terms of these key aspects.

The majority of knowledge representation systems are based on singly typed FOL, and support inference via a model-theoretic semantics. Amongst the reviewed systems there are three which clearly depart from this FOL approach:

The multi-sorted equational logic of the Olog language recognises the functional nature of databases and, as such, integrates much better with programming languages. However, the formalism as it is proposed lacks sufficient development and expressive power.

Conceptual spaces model algebraic structures which serve as frameworks reasoning about the formation and combination of concepts. Although they have limited applicability, they are tempting as an approach to music knowledge representation as we will discuss in Chapter 4.

The proof theoretic approach of K-DTT is attractive as an approach to expressive knowledge representation as reasoning is no longer limited by the ability to construct a model. However, the K-DTT formalism is limited by its focus on modelling ontological structures. In addition, certain aspects of the formalism seem a bit vague an impractical.

The semantic web has many attractive features. However, it lack adequate expressive power and does not easily integrate with programming languages.

Chapter 4

Computer Representation of Music

4.1 Introduction

This chapter reviews the literature relating to the main topic of the thesis, namely computer representation of music. There are many existing representation systems and much work has been done examining the issues and challenges of music representation, e.g. Dannenberg (1993); Balaban (1996); Wiggins (2000).

The Challenge of Music Knowledge Representation

In this thesis we examine music representation from the perspective of knowledge representation. The domain of music presents a number of challenges from the point of view of knowledge representation. In this chapter we examine these challenges in order to ascertain the requirements of a general purpose knowledge representation system for music. In doing so we consider two fundamentally important questions:

1. What are we trying to represent?
2. What is the purpose of the representation?

Chapter Outline

§4.2 examines the important issues in the topic of music knowledge representation relating to the two questions stated above, and proposes a number of requirements for general purpose music representation systems. §4.3 surveys a number of existing music representation methods and examines them in terms of the requirements. §4.4 focusses on existing approaches to representing music on the Semantic Web. §4.5 focusses on one particular representation system, CHARM (Wiggins et al., 1989), which comes closest to meeting the requirements we have identified.

4.2 Issues and Requirements of Music Representations

4.2.1 What Are We Trying to Represent?

It is commonly agreed that music as an entity is hard to define. The question of what it is we are trying to represent requires careful consideration. Babbitt (1965) identified three domains of music representations based on the portion of reality to which they refer:

Acoustic Music exists as a pressure wave in the air. The acoustic domain encompasses representations of physical sounds, such as digital audio signals and their features.

Graphemic Music exists in the form of physical information artefacts. The graphemic domain encompasses representations of notated music such as scores and tablature.

Auditory Music exists in the mind. The auditory domain encompasses representations of perceived music. This is the primary domain of music and the most difficult to access.

Wiggins (2000) considers whether these domains are complete or the most useful for some representations, in particular pondering where in the domains the intention of the composer is found. A more detailed examination of musical ontology was given by Mazzola (2012) who considers facts of music as being identified with coordinates in a three dimensional space. The dimensions are their coordinate values are as follows:

Reality Physical-Mental-Psychological

Communication Creator-Work-Listener

Semiosis Significant-Signification-Significate

The Reality dimension is comparable to Babbitt's view, however Mazzola's Physical coordinate subsumes both Babbitt's Acoustic and Graphemic domains, whilst the Mental and Psychological coordinates distinguish between objective mathematical or logical facts and subjective emotional states. The Communication dimension distinguishes between the creator's intention, such as a prescriptive score or compositional germ, and the listeners experience, such as a perceived sound or performers interpretation of a score, mediated by the neutral level of Work, which subsumes the objective existence of music. The Semiosis dimension acknowledges the semiotic structure of music in the sense of Barthes (1967).

4.2.2 What Is the Purpose of the Representation?

An alternative way to approach the design of music representations is to consider their purposes. Wiggins et al. (1993) identifies three broad purposes of representations from the point of view of the user:

Record The user wants to accurately document the musical object.

Analysis The user wants to extract information from and/or about the musical object.

Generation The user wants to build new musical objects from scratch or by transformation of existing objects.

In addition, Wiggins et al. (1993) propose two orthogonal dimensions along which music representations can be evaluated in terms of the levels of detail which they accommodate:

Expressive Completeness The degree to which the original object can be recreated from the representation.

Structural Generality The degree to which the structure of the object can be explicitly represented.

Wiggins et al. (1993) give a survey of nine different representations, evaluating each in terms of expressive completeness and structural generality. They conclude that systems which maximise both are more generally useful.

4.2.3 General Requirements

On the basis of our examination of these two questions, we now set out what we believe to be five requirements for general purpose music knowledge representations. These requirements are as follows:

1. Multiple Domains of Representation
2. Multiple Levels of Abstraction
3. Multiple Hierarchies of Musical Objects
4. Abstract Algebraic Specification of Data Types
5. Formal Description Languages

Multiple Domains of Representation

Having ascertained that music as a domain of representation is highly complex, high-level examination of musical ontology, in the manner of Babbitt (1965); Mazzola (2012), is essential in the design of more precise representations. Therefore, **a general purpose music representation system will have an explicit notion of ontological domain and accommodate multiple domains of representation.** In particular it must separate music-generic entities from domain or application-specific descriptions (Fields et al., 2011). Considering music representations in these terms clarifies their meaning and ontological commitment in the sense of Davis et al. (1993) (§3.2). However, a representation system should make only a minimum of ontological commitments and not prescribe any detailed classification of the musical universe, instead allowing the users to formally express their specific viewpoint alongside other viewpoints.

Multiple Levels of Abstraction

In each domain it is possible to view music at many different levels of detail (Dannenberg, 1993). For example, representations of the acoustic domain may be regarded at the level of digital sample, at the level of extracted features, or at the level of entire audio recordings. Representations of the auditory domain may be regarded at the level of perception or at higher-level cognitive structures. These different levels of abstraction are all involved in various music research tasks. Therefore **a general purpose music representation system will accommodate multiple levels of abstraction.** An important feature of representations is the precise specification of the specific level of detail it provides. The notion of *musical surface* refers to the lowest level of detail considered for a certain task (Nattiez, 1975; Huovinen and Tenkanen, 2007; Wiggins, 2000). Therefore, **a general purpose music representation system will have an explicit notion of musical surface and allow users to give precise definitions of new musical surfaces to suit a given application.**

Multiple Hierarchies of Music Objects

Musical objects at different levels of abstraction may be hierarchically related; collections of entities on one particular musical surface may be regarded as a single structural unit at a higher level of abstraction. This relationship is characterised by compositional containment. Many authors have acknowledged the importance of hierarchy in music (Lerdahl and Jackendoff, 1985; Marsden, 2005; Alvaro et al., 2005), and particularly that musical knowledge is multiple-hierarchical (Balaban, 1996; Smaill et al., 1993; Wiggins, 2000). That is, the same low-level entities can be viewed at higher levels of abstraction in multiple different ways. This multiplicity of viewpoints is essential for the meaningful analysis of manipulation of music in computer systems. Therefore, **a general purpose music representation system will accommodate multiple-hierarchies of musical objects.** In addition, Wiggins (2000) argues that there exist other relationships between the entities of musical hierarchies which are not characterised by compositional containment. Such annotations might include structural relationships, such as in Ockelford (2005), or co-reference annotations between entities of different musical domains (Wiggins, 2000).

Abstract Algebraic Specification of Data Types

Many music analysis tasks require comparison or transformation of musical objects from the perspective of particular characteristics, such as pitch, timbre or loudness (Lewin, 1987; Tymoczko, 2011; Martorell, 2015). Representations of these characteristics must come with an explicit notion of equality and the algebraic operations which capture their behaviour under transformations. In software development such representations are specified via algebraic specifications of abstract data types (Dale and Walker, 1996). Abstract data types allow for the precise behaviour of data to be defined independently of the concrete format in which it is encoded. This is particularly important for music in which there are numerous different but equivalent ways of encoding the same information (Wiggins, 2000). Therefore, **a general purpose music knowledge rep-**

resentation system will explicitly represent the abstract algebraic structure of musical data types. In addition, the representation system must allow users to provide their own specifications; the representation must acknowledge the multiplicity of musical viewpoints and not prescribe any fixed set of data type specifications.

Formal Description Language(s)

In order for a representation to meet the level required for knowledge representation, it must be possible to give formal descriptions of the structure and characteristics of entities (Balaban, 1996). Therefore, **a general purpose music knowledge representation system will include a formal description language for expressing specifications for musical entities.** Expressions of the language must encompass structural properties of musical entities and relationships which exist between them. In addition, these descriptions must be in a form such that a computer can automatically verify whether an entity satisfies the description, and perform more sophisticated automated reasoning and manipulations tasks. The language should be able to make reference to all the other representational features of the system. In particular, this will involve the algebraic properties of the data types which are used to characterise musical objects. A crucial feature is that the language be extensible in order to accommodate the new data types, musical surfaces and ontological domains. Extensions to the language will involve the introduction of new terminology via formal definitions in the language.

4.2.4 Summary

Our description of these requirements repeatedly highlights extensibility as an important feature of representations. Many authors have acknowledged the need for highly extensible representation systems due to complexity of music and the multiplicity of potential conceptualisations (Dannenberg, 1993; Balaban, 1996; Wiggins, 2000).

Representation systems which meet these requirements will, we argue, be highly expressively complete and structurally general. Expressive completeness is born out of the flexibility of abstract data types; it allows the user to represent anything by way of precisely choosing the mathematical properties which are required of it (Wiggins et al., 1993). A high level of structural generality is afforded through the ability to represent arbitrarily complex hierarchies with explicit association annotations, each fitted with a formal description of the structures or relationships which they represent. The ability to explicitly represent the formal properties of the hierarchical entities and associations is contingent on the expressive power of the description language.

Many other aspects of music representation are subsumed by the requirements. For example, the kinds of musical knowledge identified by Raimond (2008), namely *editorial*, *musicological* and *work-flow*. Editorial information, such as musical works, composers and dates, is catered for by the expressive completeness of abstract data types. Such bibliographic data is not normally required to posses mathematical structure, however the ability to explicitly name the type of such information and specify how it is accessed is important for the transparency of the representation. Musicological knowledge, such as rhythmic or harmonic analyses, is catered for by the structural generality. For example, the structure of the piece could be represented by a hierarchical configuration of entities capturing individual notes, rhythmic or metrical units, motifs, chords, harmonic progressions, movements, entire works and corpora. Annotations on this hierarchy could be used to represent relationships such as repetitions, transpositions, or other musical similarities. Work-flow information, that is "know-how for deriving new information by combing music processing tools and with existing information" (Raimond, 2008, p.13), can be represented by associations between entities which are fitted with formal descriptions of the process by which one was derived from the other.

Finally, we argue that the diversity of music research tasks can be unified under examination of the representational requirements. In particular, we argue that many research tasks can be formalised as a mapping between representations in different domains and different levels of abstraction. For example, "the problem of music description" (Sturm et al., 2014) can be though of as mapping from low-level representations in the acoustic domain to high-level representations in the auditory domain. Models of music perception and cognition (Wiggins, 2007) involve mappings in the auditory domain from low-level (perceptual) representations to high-level (cognitive) representations of the experienced structure. Finally, many musicological research tasks involve mapping from low-level graphemic representations to mid- or high-level graphemic or auditory representations so as to allow for the comparison of historical scores which use disparate notational conventions (Lewis et al., 2011). All tasks can benefit from formal, abstract definition of the process embodied in the system and rigorous treatment of data. Having a unified view of music representation can facilitate this on a wider scale.

4.3 Music Representation Systems

In this section we give a survey of existing representation systems for music. Rather than endeavour to give a comprehensive survey of all of the different approaches and systems which exist, we organise our survey according to three general paradigms or applications: encoding formats, programming languages and knowledge representation systems. For each of these categories we focus on a few prevalent examples which exemplify the paradigm. We discuss each system in terms of its advantages and disadvantages according to the requirements outlined in §4.2. In the case of music programming languages, we consider a further subcategory, namely music calculi. Here, we give special attention to three music-specific languages implemented as libraries for the functional programming language Haskell. These systems warrent special examination due to the close connections they bare to the type-based framework presented in this these. Our survey constitutes an update on the survey of Wiggins et al. (1993), and inherits much of the organisational structure, but instead focusses on systems which have emerged subsequently.

4.3.1 Encodings Formats

There exist many different encoding formats for music, most centred around simple, machine readable score representations e.g. DARMS (Erickson, 1975), Humdrum/Kern and MusicXML (Good, 2001). These formats possess little of the formal requirements of knowledge representation as they primarily focus on the syntactic and structural aspects, rather than the meaning and computational properties of the representation. However, in this section we discuss three prevalent encoding formats, each of which is worthy of examination due their widespread application in music research.

Midi

MIDI[1] is a binary protocol originally designed as a control format between digital keyboards and synthesisers. The MIDI file format has since become a popular method of exchanging basic note-level information in the form of piano-roll representations. It permits a very limited level of structure representation through the use of tracks and channels. Despite its popularity as a format for digital musicology, it fails to meet the formal requirements of knowledge representation. MIDI files consist of a sequence of messages which include note-on and note-off commands indexed by the keyboard key number they correspond to. Extracting a piano-roll representation from this sequence of messages involves an algorithm with marries sequential pairs of these messages for each key. However, this is not always possible; valid MIDI files may contain sequences of note-on and note-off messages which do not make musical sense. In general, the 'meaning', of MIDI representations is not explicitly defined; it could be used to encode any sequence of parameter information (for example, lighting controls), and often, the ability to use a MIDI file properly depends on implicit knowledge about how the file was encoded. Our inclusion of MIDI in this survey, is intended to highlight the kinds of representation aspects which need to be made precise when considering music from the perspective of knowledge representation.

[1]https://www.midi.org/specifications

MEI

The music Encoding Initiative (MEI: Roland, 2002) is an XML-based score encoding format, and is widely used in the support of complex musicology projects (Crawford and Lewis, 2016). It consists of a large and loosely structured XML schema, which aims to provide a single comprehensive format for a wide variety of different applications. The loose structuring of the schema means that a lot of information is optional, allowing the user flexibility in choosing particular aspects of the music to encode. The representation can also be specialised by defining smaller and more tightly structured schema subsets and extended via user defined parameters. This flexibility and extensibility is an important advantage of MEI and exemplifies design principles required for a general purpose music knowledge representation. In addition, the schema level of the representation makes the distinction between four ontological domains: logical (the musicological structure of the score), visual (the graphical structure of the engraving), analytic (annotations and commentary) and gestural (the expressive structure of a performance). This explicit representation of ontological domain distinguishes MEI from almost every other representation system. However, the analytical information it can express is limited to simple relationships between XML elements, and the meaning of these annotations is not formally defined. MEI is also limited by its commitment to the XML format. Firstly, it is only possible to explicitly represent a single hierarchy of musical objects. The representation of secondary hierarchical information, such as beams which cross bar lines, is done using XML attributes which point between elements. This fundamental deficit makes MEI, and XML in general, inappropriate as a general purpose music representation system (Wiggins et al., 1993). Secondly, the types of attribute values are taken from XML Schema and are not defined by their algebraic properties, instead being defined by their syntactic encoding. This precludes the explicit representation of the mathematical properties of musical attributes. In summary, MEI posses a large number of desirable qualities from the point of view of music knowledge representation, such as flexibility and extensibility, but has limited structural generality and expressive completeness by the singly-hierarchical nature of XML and the lack of algebraic specifications of data types.

JAMS

JSON Annotation Music Specification (JAMS: Humphrey et al., 2014) is a JSON format for representing music and audio feature data, and is widely used in the Music Information Retrieval (MIR) community (Nieto and Bello, 2015; Mcfee et al., 2016). A JAMS Object contains the audio file metadata (about the track; identifiers, duration in seconds, etc.) and a list of *Annotation objects*, each containing some information about the content of the audio file. The motivation behind the development of JAMS was to provide a common encoding format for the sharing of feature data and music analysis. However, it is worthy of examination in this survey because aspects of its general design and representation features are generalisable to other kinds of musical information. Firstly, the internal structure of an Annotation object is comparable to other event-based representations (Alvaro et al. (2005), for example); they contain time and duration values which describe the segment of audio being annotated, a *namespace* which describes the type of the annotation, and a list of data observations, each of which consists of a time, a duration and a value. The notable feature of this representation is that the type of the value of a data observation is determined by the namespace of the Annotation object. The JAMS documentation provides a number of namespace definitions each capturing a particular type of musical attribute such as pitch, onset, chord label and key signature. The explicit representation of type information within a more general purpose structure is an extremely flexible and extensible approach, and one which is particularly required for music, where the variety of possible musical annotations is limitless. It is this general approach which we adopt in Chapter 5 for the representation of musical attributes.

4.3.2 Programming languages

Over the years, many programming languages have been developed to address many different music analysis and composition applications. There are a wide variety of approaches to the design of such languages, each influenced by the specific purpose of the language. Loy and Abbott (1985) give a survey of a large number of programming languages, distinguishing those whose design is focussed around the bottom-up structuring of musical material from the acoustic level, and those whose focus is on the high-level control of abstract musical structures. Wiggins et al. (1993) identify one particularly important aspect of music programming languages from the perspective of knowledge representation, namely, the distinction between declarative and procedural programming paradigms. They judge that declarative languages are more suitable for knowledge representation, as knowledge represented as procedures can be obscured by the operational semantics of the language. In this section we examine two recent music programming languages, namely Music21 (Cuthbert and Ariza, 2010) and Rubato Composer (Milmeister, 2014), and discuss the advantages and disadvantages from the perspective of knowledge representation. In addition, we examine a particular kind of music programming language, namely music calculi, and discuss three particular examples which are implemented in the functional programming language Haskell.

Music21

Music21 (Cuthbert and Ariza, 2010) is a python tool kit for searching, analysing and manipulating musical score representations. Its underlying representation is defined using the Python class system. Atomic objects, such as notes and chords, are represented as Python objects, with their internal properties accessed by methods of their class. It allows for the representation of higher-level musical structures via the Stream class, which captures ordered collections of atomic entities and other streams. As such, it is capable of representing sophisticated hierarchical configurations of musical objects. In addition, it supports multiple hierarchies, as individual objects can be made elements of more than one stream object simultaneously. However, control of these multiple-hierarchical structures is limited by the fact that it depends on object pointers, which are not directly accessible in the Python language.

Music21 has become a popular tool in the musicological community due to the extensibility and flexibility of the Python language, allowing users to define and share new classes and operations to suit varied applications. However, it has a distinct disadvantage from the perspective of knowledge representation, which is best explained via the parallel between object-oriented programming languages and knowledge representations based on frames (Minsky, 1974). The fundamental difference lies in the intended purpose of the representation. In frame languages, objects are used to explicitly structure and connect the concepts of a domain so as to make explicit the ontology of the problem. In programming languages, objects are used to bundle together data and methods to enable efficient management of complex programs. As such, Music21 does not explicitly represent its ontology of musical objects and parameters. Rather, this information is implicit in the class definitions, which hard-code representations and bundle them with the methods which provide programming support. As a result, much of the knowledge represented by Music21 programs is obscured by the operational semantics of the underlying language; it is only accessible via the running of the program, and so it not generally available for reasoning with. This is not to say that object-oriented programming is incompatible with knowledge representation. We take the view that object-oriented languages such as Music21 can be seen as specialised implementations of more abstract knowledge representations. Indeed, it is highly desirable that a general purpose music knowledge representation should support such languages for music programming. The system proposed in this thesis (Part II) is intended to do just that.

Rubato Composer

Rubato Composer (Mazzola, 2006; Milmeister, 2014) is a Java tool-kit for analysis and composition of music. The system has two purposes. Firstly it is a application development environment for mathematical musicologists, with proficiency in the Java language, to build modular program units. Secondly, it is a high-level graphical programming environment aimed at composers and analysts, without mathematical knowledge, for exploring the use and combination of the modular elements build by developers. The underlying representation of the system is a partial implementation of the Forms and Denotators system (hereafter Denotator theory) of Mazzola (2012) based on category theory (see §2.3.3). As such it is highly expressive, allowing for the representation of complex musical structures and transformations between them. The entities of the representation are Java objects representing Denotators, which are stored, at runtime, in a repository which is organised into several different namespaces. All the entities of the representation can be serialised as XML for efficient storage exchange between users of the system. The substantial expressive power of the system is born from the underlying sophistication of the category-theoretic basis. It demonstrates how a layered approach to music programming languages and representation in general can be used to provide users with powerful high-level tools and components, whilst hiding the underlying complexity of the low-level formalism. This is a design principle which we adopt through the development of the type-based framework presented in Part II.

There are however, two aspects of both Rubato Composer and Denotator theory, which we believe make it less suitable as a general purpose music knowledge representation. The first aspect is that Denotator theory does not support the algebraic specification of abstract data types (requirement 4, §4.2). Atomic musical parameter spaces must be defined in terms of the algebraic structure of a mathematical module. Mazzola's motivation for this is the acknowledgement of the importance of, not only parameter values, but also the algebraic operations which relate them. Mazzola (2012, p.70), on the justification for selection of modules, says: "One could require that a determinate structural instance (such as additive closure of modules) has to play a musical or musicological role in all situations where the structure is present. However, such a requirement would be too restrictive since mathematics should make available to music what may possibly happen in music, and not what happens in any case." This argument is perhaps more valid in the context of mathematical musicology, however, we argue that for the purposes of music *knowledge* representation, the designer of the representation should be able to explicitly describe the specific 'theory' to which they are committing, rather than have it prescribed and choosing how to selectively apply it. This is a subtle distinction, but one that is an important aspect of the approach taken in this thesis; we use a sophisticated mathematical basis to provide users with a representation which is entirely comprehensive from a *descriptive* perspective. This is in contrast to a representation which embeds one specific theory of music that is designed to be comprehensive from a *definitive* perspective. The second aspect is that the system does not directly support logic-based descriptions and automated reasoning. Whilst the underlying category-theoretic machinery is capable of formalising logical systems (Mazzola, 2012, ch.18), Denotator theory does not come equipped with a human usable language for formally describing the logical properties of musical structures (requirement 5, §4.2). Instead, these properties must be implicitly encoded in the definitions of functors and module morphism (Mazzola, 2012, ch. 7). We argue that such a means of representing structural descriptions is, in general, less intuitive and user-friendly that a logic-based language, such as that presented in this thesis (Chapter 6).

Music Calculi

We now discuss a specific category of music programming languages called *music calculi*. Orlarey et al. (1994) propose an approach to the design of music programming languages based on lambda calculus. The basis of the idea is as follows: "Instead of building suitable music data structures and functions on an actual programming language, we suggest to build suitable programming languages on music data structures" (Orlarey et al., 1994, p.243). They describe a music calculi in which event and score objects are defined syntactically as terms of lambda calculus. These definitions are recursive, allowing score objects to be composed of both events and other scores. The different ways in which the objects can be composed is captured by the syntactic rules of the term language. This definition of terms is similar to that of grammar based representations, such as Bel and Kippen (1992). The distinguishing characteristic of the approach, however, is that the lambda calculus embeds a notion of function abstraction and computation, allowing the user to define operations which construct new objects from old.

A more expressive example of a music calculi is the Musical Structures system (Balaban, 1988, 1996). Atomic Music Structures are defined syntactically as pairs consisting of pitch and duration values, annotated with a temporal onset. More complex music structures can be constructed via the composition of existing structures. This approach simultaneously captures the hierarchical and temporal characterisics of music, and, in principle, allows for a great deal of structural generality and expressive completeness due to the open-endedness of the symbols used (Wiggins et al., 1993). A significant disadvantage to Balaban's approach, however, is that it fails to explicitly represent abstract algebraic properties of the musical parameters used. Temporal onsets are represented according to the real line, however, Wiggins et al. (1993, p.15) note that "...it is rare that musicians think in those terms. An improvement would be to use an algebra with the relevant properties of the real number, but with abstract syntax." This view is motivated by the desire to make precise the algebraic aspects of music which are being represented.

More recent work in the area of music calculi consists of domain-specific languages (DSLs) in the functional programming language Haskell. Haskore (Hudak, 1996) defines a representation of musical scores as recursive Haskell data types. This approach is similar to that of Orlarey et al. (1994) and Balaban (1988), however it is distinguished by the use of a typed host language; the data types and their operations are defined *in* a typed lambda calculus rather than *defining* an untyped one, such as Orlarey et al. (1994). A subsequent update to the Haskore system was Euterpea (Hudak, 2011) which added a great deal of expressive power and functionality to the original Haskell library. This library demonstrates how sophisticated analysis and manipulation of music can be achieved via (recursive) functional programs. A disadvantage of the approach, however, is that, again, the algebraic properties of musical attributes are not abstractly defined, but rather hard-coded in to the implementation. A separate attempt at developing a Hakell music library was the Music Suite (Hoglund, 2014). This approach is distinguished from Haskore and Euterpea in that it allows the user to define their own kinds of atomic musical entities. In addition, it provides abstract notions of musical parameter spaces which are parametric in the concrete type which encodes the value. In this sense, the Music Suite comprises a family of related music calculi, with a high level of extensibility and clean abstraction discipline. One disadvantage is shared by all these representations. From the perspective of knowledge representation, it is often desirable to be able to uniquely identify particular entities within a more complex structure. However, this is not possible in these functional DSLs without explicitly using a function which recursively traverses the data structure to the desired object. Furthermore, these functions are not invariant under transformation of the representation. This effectively means that the identity of the individual components of a representation is lost. The type-based framework presented in this thesis provides a solution to this problem in a functional setting, by defining a meta-level data structure which explicitly includes a type of identifiers.

4.3.3 Music Knowledge Representations

A large number of music representation systems have been developed which make use of dedicated knowledge representation technologies. In this section we review four representative examples, highlighting the advantages and disadvantages of each approach.

HARP

Hybrid Action Representation and Planning (HARP: Camurri and Frixione, 1992; Camurri et al., 1994) is a model of musical knowledge based on a twofold formalism. This formalism consists of an object-oriented programming environment for managing concrete representations of musical material and processing algorithms, and a high-level symbolic system based on KL-ONE (Brachman, 1978; Brachman and Schmolze, 1985, see §3.4) extended with concepts which capture temporal entities. The HARP model defines two basic musical entities as top-level concepts: the *music action* and the *compositional action*. Music actions represent abstract musical material at arbitrary levels of detail. They can can be composed using dedicated temporal relations, as well as related to concrete instances of sounds or scores. Compositional actions represent processes which manipulate music actions. They are defined via the types of music action which they take as input and produce as output, and can be associated with concrete instances of procedural algorithms in the object-oriented environment.

The system is intended to aid in the high-level manipulation of diverse kinds of musical material. It is highly general, relying on the user to define specific subclasses of music action and compositional action to achieve the desired functionality. As an approach to knowledge representation, it is notable due to the advantages afforded by the division of knowledge into different subsystems: the architecture maintains strict separation between the high-level relationships between abstract musical objects and the concrete instantiations of these objects along with the algorithms which process them. However, a distinct disadvantage of the system is that the precise meaning of the representation at the abstract level is obscured by the specific technologies used to implement the concrete level. The system presented in this thesis provides an alternative approach to this separation, whereby abstract data types are used to separate implementation detail from the abstract computational properties of musical entities.

EV-Meta Model

The EV meta model (Alvaro et al., 2005) is a frame-like system for music representation in computer-aided composition tools. The model is centred around the representation of time-based events at many different levels of abstraction. Events can be fitted with parameters, which capture the properties of the musical object, and a collection of sub-events, allowing for the representation of hierarchical musical structures. The approach is notable for its high-level conceptual foundation. The event representation is entirely general, allowing for the representation of arbitrary kinds of musical object via specific parameters. The values of parameters can in turn be represented by dynamic objects whose static value is a function of time. The ability to combine dynamic objects in recursive structures makes the representation highly expressively complete. However, there are two distinct disadvantages of the system. Firstly, the representation only supports a single hierarchy of musical events. In addition, the sub-events of an event must share the same object structure i.e. they must be the same kind of entity. This precludes the representation of events consisting of, for example, a collection of chords and notes. Secondly, the system does not support logic-based description of event structures in the style of KL-ONE structural descriptions (Brachman and Schmolze, 1985, see §3.4). Despite these disadvantages, the model epitomizes an important requirement of music representations, namely the provision of a general purpose scheme which can be specialised according to the requirements of the user. This is a design principle which is central to the framework presented in Part II.

Temporal Logic

Marsden (2000) gives a detailed analysis of a large number of approaches to music representation involving temporal logic. He gives a comprehensive discussion of the relevant aspects including a number of different approaches to ontology centred around either points, periods or events. The work provides some considerable insights into the theory and technical aspects of temporal logic approaches to the representation of music. However, there are two aspects of the representations discussed which limit their suitability as bases for general purpose music knowledge representations. Firstly, as noted by Wiggins (2000), the need for special temporal logic-based representations is not strictly necessary when working within a typed logic; the same knowledge can be represented by defining, at an abstract level, the types of temporal entities and their algebraic structure. One aspect of the temporal logic approach which is, however, not so easily captured by abstract data types, namely indeterminate sequences of entities and under-constrained representations. For example, using temporal logic, it is not possible to represent a set of temporal constraints which capture a class of possible concrete structures, without declaring their their temporal values. A second disadvantage is related to a technical issue regarding the representation of musical entities. When dealing with point and period ontologies, Marsden, represents musical objects without explicitly naming them; their existence is only implied via the truth of certain logical propositions. For example, the proposition 'sounding(C,**w,y**)' Marsden (2000, p. 38)) indicates that a 'C' is sounding between time points **w** and **y**; the identity of the extant musical object is not explicitly represented. This presents problems from the point of view of knowledge representation, as we cannot make statements about particular entities, such as individual notes or pieces of music. When dealing with an event-based ontology, Marsden (2000, p. 61) does "distinguish between identity and the temporal relation of equality: in the case of periods, $x = y$ means that x and y refer to the same period; it is possible for two events to be temporally 'equal' but to be different events." In this way, a musical object is represented as an event, and we can make statements about it directly. For example, the proposition 'C(x)' represents the fact that the event x is an instance of a tone whose pitch is C. This approach, whereby extant musical objects are given names to which descriptions of the objects are are associated, is the basis of the constituent structure representation described in Chapter 5.

Conceptual Spaces for Music

The theory of conceptual spaces (Gärdenfors, 2000, see §3.4) is attractive as a basis for representations of music based in the auditory domain. Forth et al. (2010) illustrate how a representation of musical melody might be constructed using conceptual spaces consisting of quality dimensions for pitch, interval, onset and inter-onset interval, and higher-level abstractions such as melodic contour and harmonic function. In addition, they give a more detailed conceptual space formalism of musical meter, highlighting the complexities involved in the selection and specification of the relevant quality dimensions. This work demonstrates how perceptually valid representations of music can be achieved via the definition of conceptual spaces with appropriate quality dimensions, normalised according to empirical evidence.

Chella (2015) also applies conceptual spaces to music, suggesting a cognitive architecture for music perception. The architecture consists of a *subconceptual* area concerned with the processing of sensory data, a *conceptual* area in which sensory data is organised into conceptual spaces, and a *linguistic* area in which symbols are assigned to perceived entities, allowing them to be described in terms of a logical language. The linguistic area is a hybrid knowledge representation, similar to that of HARP (Camurri and Frixione, 1992, see above), consisting of a *terminological* component for the description of concepts, and an *assertional* component that stores information about particular instances of concepts. This architecture is notable as it gives an explicit account of the connection between conceptual spaces and logical languages.

From the perspective of general music knowledge representation, accurate representations of the auditory domain open up exiting possibilities for computational exploration of music perception and cognition and models of musical creativity. The added ability to connect these representations with higher-level knowledge representation languages for more general purpose information management is a principle advantage of the approach presented in this thesis. The representation of musical parameters as abstract data types can be seen as a generalisation of the notion of quality dimension in the theory of conceptual spaces. As such, our framework is a suitable basis for users who wish to represent their own conceptual space formalisms.

Semantic Web Ontologies

A large number of representations of music have been developed using Semantic Web modelling languages. Due to popularity and relevance of Semantic Web in current research endeavours, a detailed examination of the various music-based ontologies will be given in the following section.

4.4 Music Representation on the Semantic Web

The Semantic Web has been used extensively for the representation of music and related information for a wide variety of tasks. In this section we survey seven existing models and methods of music representation from the literature, examining the advantages and disadvantages of each. Six of these methods are OWL ontologies which model musical information from the point of view of a certain task. The seventh is an approach to the representation of work-flow information, (Raimond, 2008) which incorporates aspects of both ontology and other Semantic Web technologies, the details of which are described in the relevant paragraphs. We summarise with a broader discussion of the various approaches and particularly highlight diversity and commonality of the models at the meta-conceptual level. Throughout this section, and the rest of the thesis, we use the notation "`namespace:fragment_identifier`" to refer to the classes and relations defined in the different OWL ontologies, where `namespace` is the conventional prefix abbreviation for the namespace URI of the ontology, and `fragment_identifier` is the name of a specific vocabulary term, usually an OWL class or property. For each ontology we specify the prefix used, where necessary.

The Timeline Ontology

The Timeline Ontology (Abdallah et al., 2006, prefix: `tl`) was developed to support the structuring and organisation of temporal relationships between musical entities. The ontology defines the `tl:Timeline` class, for modelling linear and coherent sections of time which can either be abstract or concrete. Temporal entities, such as instances of the `tl:Instant` and `tl:Interval` classes, are associated with a time line via the `tl:timeline` property. Time lines themselves are associated with coordinate systems (defined by XML-Schema datatypes) via specific sub-classes of `tl:Timeline`. Temporal entities defined on a time line can be linked with a literal value defining its coordinates whose type is restricted by the coordinate system of the time line. The temporal entity classes are taken from The OWL Time Ontology[2] and therefore inherit the structure of the model including temporal relations.

There are two crucial advantages to the model. Firstly, the `tl:Timeline` class serves as an explicit representation of a particular musical parameter space. Information represented using the ontology is explicitly labelled with the coordinate system or space in which it is structured. Although the model was developed specifically to deal with time, it is potentially generalisable; music is a domain with many dimensions comparable to time such as pitch and loudness. Secondly, the time line concept acts as a structuring mechanism, grouping together entities whose temporal coordinates are directly comparable. This, we argue, is part of the function of a musical surface (see §4.2), on which coexisting entities constitute the basis for musical analysis. Although the underlying logical basis of OWL is not sufficiently expressive to capture the algebraic properties of time lines, these two aspects of The Timeline Ontology are directly comparable to the approach of the current work.

[2]https://www.w3.org/TR/owl-time/

The Event Ontology

The Event Ontology (Raimond et al., 2007, prefix: `event`) was developed as a general purpose model for capturing a variety of different aspects of music. It is primarily used as a part of The Music Ontology (Raimond et al., 2007, discussed next) to structure complex interconnected information about musical works. The ontology defines the `event:Event` class, for modelling arbitrary space/time regions which can be linked to addition information including *factors* (things involved in an event), *agents* (people involved in the event) and *products* (things produced by events). This event-reification approach allows for a great deal of flexibility and extensibility, as new information can be attached to an existing events as and when it is available. The Timeline Ontology is used to add temporal structure to events by linking them with instances of `tl:Instant` or `tl:Interval`.

The main advantage of the model is that it is entirely general, providing a meta-level concept which subsumes many types of musical entity. It supports the development of more domain-specific models of musical objects which benefit from a high-degree of top-down conceptual interoperability (see §3.3). In addition, the model supports multiple-hierarchical event decomposition; events can be associated with sub-events via the `event:sub_event` property. These aspects of the model are highlighted due to their close correspondence with the approach to modelling musical objects presented in Chapter 5.

The Music Ontology

The Music Ontology (Raimond et al., 2007, prefix: `mo`) is an ontology for annotating online collections or recorded music in the form of audio files. It captures basic editorial information by extending the FRBR model with four top-level classes: `mo:MusicalWork`, `mo:MusicalManifestation`, `mo:MusicalExpression` and `mo:MusicalItem`. The ontology also models the music creation work-flow through a number of specific `event:Event` subclasses such as `mo:Composition`, `mo:Performance` and `mo:Recording`. The ontology constitutes a high-level information management model, which despite being primarily aimed at capturing knowledge about recorded pop music and MIR tasks, is able to capture the general conceptual connections between musical works and their concrete expressions, providing a strong basis for more domain-specific knowledge representation.

The notable feature about the ontology is that the high-level classes can be viewed as offering a comparable kind of classification to that of Babbitt's domains (Babbitt, 1965), albeit with a specific focus on recorded music, and omitting the auditory domain. Also of note is that all of these top-level classes are subclasses of the meta-concept `event:Event`. The approach to music representation presented in this thesis can be viewed as a generalisation these aspects of the model, which can incorporate information which is not necessarily centred around record audio files.

The Audio Features Ontology

The Audio Features Ontology (Allik and Sandler, 2016, prefix: `afo`,) provides a framework for representing information about audio features and extraction methods. It provides a common format for sharing audio feature data, as well as a model for representing computational extraction work-flows. The core of the model captures four levels of abstraction: The `afo:AudioFeature` class captures abstract conceptualisations of an audio feature type; the `afo:Model` class captures computational models of audio feature extraction methods; the `afo:FeatureExtractor` class captures concrete implementations of extraction methods from a specific software tool; and the `afo:Instance` class captures concrete extraction instances on a particular operating systems or hardware platform. The `afo:AudioFeature` class is a subclass of `event:Event`. Instances of the class are associated with computational models of the extraction work-flow which are in turn associated with sequences of operations. Operations are further classified as transformations, filters or aggregations. The Audio Feature vocabulary provides a large number of terms representing features, models and operations which are present in various feature extraction tools.

The ontology is notable because it models an explicit distinction between the abstract type of an audio feature and the concrete implementation details about how it was extracted. This distinction underpins the notion of an abstract data type, and is a fundamental motivation behind the abstract representation system presented in Part II. The use of it in the Audio Features Ontology is an acknowledgement of multiplicity of different implementations of software tools which perform feature analysis. However, the core principle is widely applicable to music representation as there exist multiple different ways of encoding musical parameters, such as pitch and time, which are often equivalent.

The Segment Ontology

The Segment Ontology (Fields et al., 2011, prefix: `seg`) was developed to provide a music-generic model of structural segmentations of a music piece. It defines the `seg:Segment` class as a subclass of `tl:Interval`. The model is an acknowledgement of segments as a general purpose abstraction used in musical analysis. Fields et al. (2011) use the Similarity Ontology (Jacobson et al., 2009) to link music-generic segmentations of a musical signal, with application-specific annotations such as labels and audio feature data. The Similarity Ontology (Jacobson et al., 2009) models similarities between entities as an OWL class, rather than a relation. This approach provides an anchor point for attaching additional information about the provenance of the association along with information about the computational process by which it was computed.

This approach was motivated by the desire to model a strict separation between music-generic information, such as the segments of a piece, and application specific information about those segments. The notable feature of this approach is its fundamental difference to that of The Event Ontology. The Event Ontology uses arbitrary resources to represent music-generic entities and describes their temporal characteristics by linking them with temporal entities. Conversely, the Segment Ontology uses temporal entities themselves to represent music-generic entities, and links them with additional information via the Similarity ontology.

Chord Sequences

The ontologies discussed up to this point have each aimed at capturing fairly general and high-level musical information. We now examine one attempt which has been made to use the OWL language to axiomatise more sophisticated musicological structures. The Music Entity Ontology (MEO: Wissmann, 2012, ch.7) defines an ontology of 'musical entities', including `Pitch`, `Duration`, `Note`, `Interval` and `Chord`. Notes are described by pitch and duration characteristics, whilst Chords are modelled using containment relations `contains` and `containedIn`, which capture the constituent notes of a chord. Sequences of chords are captured using a model of sequential patters called SEQ (Wissmann, 2012, ch.8), based on the OWL List Ontology[3].

It is highly successful in allowing automated tools to reason with chord sequence information, demonstrating how this supports both analysis and querying (Wissmann, 2012, ch.13). However, there are two major disadvantages to the model from the perspective of our requirements of music representation. Firstly, it is highly domain-specific and is not applicable to different music analysis contexts. For example, it focusses on a particular concrete representation of pitch, which is only applicable to particular forms of music, i.e. equal tempered, diatonic, western tonal music. This is done via specific sub-classes of the class `Note`, such as `MIDINote` and `ScoreNote`. This means that, in principle, the model could be extended to include other representations of pitch. However there are no axioms that model the structural or algebraic aspects that are common to these various representations, nor any way of separating music-generic information from domain-specific representations, in the manner of Fields et al. (2011). Secondly, the model is not structurally general. This was, of course, not the intended purpose of the model; it was intended to address a very specific problem i.e. chord sequence patters. However, the details of why it is not generalisable are highly relevant to the reasons why OWL is, in general, not a suitable basis for music knowledge representations which

[3]https://w3id.org/list

meet our intended requirements. Firstly, the representation of simultaneities using a containment relations is not capable of expressing multiple hierarchies. This is because, in order to enable efficient reasoning, the model defines the *sibling* relation which captures when two notes are contained within the same chord. This relations is then used in an axiom which says that the relation composition `sibling` ∘ `containedIn` is a subset of the relation `containedIn`. This explicitly prohibits the modelling of chords which, for example, share one underlying note; any siblings of the shared note will also automatically be `containedIn` both chords. This work highlights the challenges involved in the axiomatisation of musicological structures using description logics. It shows how sophisticated reasoning can be achieved for a particular specialised domain, using large ontologies, but these approaches are not generalisable. This, we argue, means that music requires a more expressive logical basis, and serves to motivate the type-based approach presented in Part II.

Work-flows: N3-Tr

Raimond (2008) proposed a method for representing information about music processing work-flows on the Semantic Web. The ability to explicitly document computational methods of analysis is an important part of reproducible research, and is a crucial step towards conceptual interoperability in computational musicology. The system proposed by Raimond (2008) consists of a logic called "N3-Tr", which is based on concurrent transaction logic (Bonner and Kifer, 1994). Expressions of the logic can be used to describe, at an abstract level, algorithms for music processing, with the stages executed concurrently or sequentially on some database. The precise details of this logic are unimportant for the purposes of this discussion. However, the notable aspect of the approach is how it is represented for the Semantic Web. Raimond (2008, ch. 5) defines a small Semantic Web vocabulary of terms which capture the abstract syntactic structure of formulae of the N3-Tr logic. This vocabulary is used in conjunction with the Notation3[4] (N3) serialisation format for RDF to encode logical formulae as linked data

[4]https://www.w3.org/TeamSubmission/n3/

graphs. The N3 format allows for the representation of RDF graph-literals, which Raimond (2008) uses to encode atomic formulae. Jacobson et al. (2009) demonstrates how N3-Tr representations of music processing work-flows can be used to augment the Similarity Ontology, by attaching to similarity annotations, a description of the algorithm which was used to compute the similarity. This method of embedding in RDF, formulae of a more expressive logical system, is the basis of our approach to the Semantic Web implementation of the type-based framework presented in Chapter 8.

Summary

The large body of work in music representation on the Semantic Web reflects the desire of music researchers to harness the considerable power of linked open data. It is for this reason that the implementation of the representation framework presented in this thesis incorporates Semantic Web technologies. Our survey or existing approaches to music representation on the Semantic Web has highlighted several broad patterns in the various models at the meta-conceptual level. In addition is has highlighted a particular conceptual incompatibility between the Segment and Event ontologies. One common pattern in existing models is the use of reification to circumvent the expressive limitations of the Semantic Web modelling languages and create stable, extensible information models. Event reification (Raimond et al., 2007) and association reification (Jacobson et al., 2009) exemplify this approach, and are comparable to earlier approaches from semantic networks, such as Conceptual Graphs (Sowa, 1976). In addition, the Time Line OntologyAbdallah et al. (2006) reifies the concept of a space or coordinate system used to structure musical entities, while the N3-Tr system Raimond (2008) reifies logical formula. We draw attention to these aspects here as these notions are generalised and subsumed by the approach to reification of type-theoretic expressions taken in §8.3 and §8.4.

4.5 The CHARM System

Common Hierarchical Abstract Representation of Music (CHARM: Wiggins et al., 1989; Harris et al., 1991; Smaill et al., 1993) is a proposal for a general purpose music representation system built from abstract data types. The representation framework presented in this thesis is heavily influenced by the insight provided by CHARM, as it comes the closest to fulfilling the requirements for a general purpose music knowledge representation systems described in §4.2. We begin this section by giving an overview of the system and describing previous work that has been done in demonstrating the power of the abstract approach. We then describe one attempt which has been made at an implementation of CHARM, namely AMusE (Lewis et al., 2011), and discuss why it falls short of completely realising the core principles. We conclude this section with discussion of how the original CHARM work could be usefully developed and, in doing so, provide motivation for the type-based representation framework presented in Part II.

4.5.1 Overview

Wiggins et al. (1989) propose a representation of note-based musical material based on abstract data types. They give an algebraic specification of the pitch and time dimensions of musical structures, which captures the structure implicit in many other representations. The specification of time is given in terms of sets, functions and relations, observing that the set of durations forms a linearly ordered commutative group under addition. Similar treatment is given to the pitch dimension, while other musical dimensions are left for future work. The basic representation consists of a set of *event* tuples of the form

$$\langle \texttt{identifier}, \texttt{pitch}, \texttt{time}, \texttt{duration}, \texttt{timbre} \rangle,$$

where the `identifier` is the unique name of the event and `pitch`, `time`, `duration` and `timbre` are values taken from an appropriate abstract data type. The values of events are accessed by dedicated destructor operations. For example, the operation getPitch takes and identifier and returns the pitch component of the corresponding event. This representation has much in common with conceptual spaces (Gärdenfors, 2000), where each abstract data type can be seen as representing a quality dimension of a musical conceptual space, and each destructor operation as capturing a projection operation.

In addition to the basic event representation, Wiggins et al. (1989) describe the *constituent* representation, a mechanism for structuring basic musical material hierarchically. Constituents are used to delimit groupings of basic musical events and other constituents in order to represent any higher-level musical structures of interest to the user. Harris et al. (1991) describe this constituent mechanism in more detail. They specify constituents, at the abstract level, as pairs of the form

$$\langle \texttt{Properties/Definition},\ \texttt{Particles} \rangle,$$

where `Properties/Definition` is a formal description of the structure denoted by the constituent and `Particles` is the set of basic events or other constituents of which the constituent is composed. Harris et al. (1991) distinguish between a constituent's structural `Properties`, which are *derivably* true and can be checked, and properties assigned by the user which are true by `Definition`. The structural `Properties` component of a constituent is further broken down as a pair

$$\langle \texttt{spec},\ \texttt{environment} \rangle,$$

where `spec` is a logical specification of the structure of the constituent's particles and `environment` is a (possibly empty) set of key-value pairs which allow the user to attach values of abstract data types to constituents. These values can be retrieved by applying the destructor operations, such as GetPitch, to the constituent.

Harris et al. (1991) propose an implementation of the `spec` component using first-order logic. They give a number of examples of constituent specifications including *stream* (Harris et al., 1991, p. 9), which describes a contiguous sequence of events in time. Stream is defined as follows:

$$\begin{aligned} \mathit{stream} \leftrightarrow \forall p_1.\neg\exists p_2.\ & p_1 \neq p_2 \land \\ & \mathsf{getTime}(p_1) \leqslant \mathsf{getTime}(p_2) \land \\ & \mathsf{getTime}(p_2) < (\mathsf{getTime}(p_1) + \mathsf{getDuration}(p_1)) \end{aligned}$$

This definition uses the destructor operation GetTime for accessing the onset times of basic objects, the operation $+$ and the relations $\leqslant$ and $<$ from the abstract data type for time, and the standard logical connectives. The range of significance of the quantifiers is the set of particles of the constituent.

4.5.2 Applications

Wiggins et al. (1989) illustrate how the representation can be used in practise with two examples. First, an implementation of Steedman's rhythmic analysis procedure (Steedman, 1977) applied to the first two bars of Mozart's Variations on "Unser dummer Pobel meint". Second, an implementation of Ruwet's paradigmatic analysis algorithm (Ruwet, 1972) applied to Debussy's Syrinx in the manner of Nattiez (1975). Harris et al. (1991) give further details of the former of these examples including the logical specification for a particular kind of rhythmic unit called a *dactyl* (Harris et al., 1991, p.15), which captures event sequences of the form "long-short-short". Smaill et al. (1993) elaborate on the second of the examples, describing the algorithm in more detail and showing how the output of the analysis can be represented as a collection of constituents. In addition, Smaill et al. (1993) give a comprehensive illustration of the power and flexibility of the abstract data type approach. They show that the analysis program produces the same results when applied to two different concrete encodings of the same musical piece. This is an extremely powerful advantage of CHARM. The ability to process representations using programs specified at an abstract level, in terms which capture the musical meaning of the analysis process independent of irrelevant implementation and encoding details, is a large step towards conceptually interoperable tools for computational musicology. In addition, Smaill et al. (1993) apply their analysis program to a piece which uses an entirely different pitch system, demonstrating how the level of the abstraction captured by the specification for pitch is not specific to any notational convention or tuning system.

4.5.3 The AMusE Implementation

Only one serious attempt has been made at an implementation of CHARM: Lewis et al. (2011) introduce the AMusE system, a collection of abstract classes that form a framework into which concrete implementations of musical concepts can be plugged. Implemented in LISP (SBCL), and based on the abstract data types of Wiggins et al. (1989), AMusE has demonstrated the advantages of abstract representation through allowing independent tools for analysis and visualisation to operate on varying kinds of underlying representation (Lewis et al., 2011).

However, AMusE is not a full implementation of CHARM. The constituent mechanism is underdeveloped, only allowing constituents that are defined by two time points, with limited scope for formally specifying their intrinsic properties. Another aspect that further removes it from CHARM is the precise nature of the abstraction used. For example, AMusE does not fully and cleanly decouple the concepts of *pitch*, *pitch implementation*, and *pitched event*. Rather, these concepts are subsumed in a large and rather complex hierarchy of classes. While the practical reasons for doing this were driven by the nature of the platform and the intended use of the system, this tower of abstraction presents new users with a substantial challenge, in comparison to say, the relative conceptual simplicity of the ML signatures and structures used by Smaill et al. (1993).

4.5.4 Scope for Development

The work on CHARM makes vital progress towards more general purpose and interoperable music representations. However, there has been little uptake of the core principles and ideas within the music research community. One possible reason for this is that work to date has not included a full specification of the technical and logical details of the representation. In this section, we examine these details, highlighting their importance from the perspective of knowledge representation, and propose two ways in which CHARM could be developed to add formal clarification.

The CHARM Interface

The first aspect of CHARM which could benefit from formal clarification is regarding the precise behaviour of events and constituents with respect to the interface operations. The work to date does not include a full specification of the interface. The most detailed description is given by Harris et al. (1991), who informally give three requirements of implementations. Firstly, they require that "[e]ach member of a *concrete* event structure is associated with a unique `Identifier`, for efficient reference by software routines" (Harris et al., 1991, p. 6). Secondly, they require implementations to define operations for accessing the values of events, and a PutEvent operation which takes an event tuple and returns the identifier associated with it (Harris et al., 1991, p. 6). They also require that "constituents have appropriate typing and destructor functions, as for events" (Harris et al., 1991, p. 10), and suggest informally, that the interface operations be used in conjunction with a database containing the constituent structure (Harris et al., 1991, p. 6). Thirdly, they propose that default values be built into the implementation of the destructor operations to ensure that, when applied to constituents which do not provide a value in their environment, they return a value which is meaningful to the implementer (Harris et al., 1991, p. 11).

This description leaves two things under-specified at the abstract level. Firstly, it does not explicitly define identity and equality of entities; they are implicit from the description but not formally defined at the specification level. We take the view that *identifiers* are, rather than an implementation detail, a core part of the represented knowledge. A development of CHARM could include an *abstract type of identifiers*, which explicitly captures identity of entities as equality of identifiers. This would ensure that all implementations of CHARM share a common understanding of the identity and equality of the represented entitles at the abstract level.

Secondly, the behaviour of the interface operations is not fully defined. In particular, what should happen when a destructor operation is passed an identifier which does not correspond with any entry in the database? This aspect may seem like an implementation detail, however it has significant consequences when considering the precise meaning of the constituent specification language, in which the interface operations are among the non-logical symbols. Consider, for example, the logical expression $\mathsf{GetTime}(e1) < \mathsf{GetTime}(e2)$, where $e1$ and $e2$ are event identifiers and $<$ is the ordering relation from the abstract data type for time. What would the meaning of this expression be if, for example, the event $e2$ did not exist? Should it be *false*? Should it cause an exception to be thrown? Or is it not even a well-formed proposition of the language? In order for CHARM to be a precise knowledge representation formalism, we must provide *an* answer to this question, so that checking procedures and automated reasoners may have the same behaviour independently of the implementation used. One approach is to give algebraic specifications for events, constituents and whole structures, in the manner used for the other abstract data types. These would provide a mathematical reference for implementations of CHARM, and strengthen the meaning of the constituent specification logic.

Finally, there is a disadvantage to the default value approach: it potentially allows two different implementations to exhibit different behaviour when applied to identical data. In our quest for conceptual interoperability, this is something which we would like to avoid. We propose instead, that the destructor operations be partial functions, and that their partiality be captured abstractly at the specification level.

Constituent Specifications

The second area of CHARM which could benefit from formal clarification is the logical language of constituent specifications. We would like this language to be defined at the abstract level, with a formal meaning which is independent of any implementation of the interface. The ultimate goal is to allow users to define and share sophisticated logical descriptions of musical structures, which can be used in software applications for data integrity checking and automated reasoning.

There are two aspects of the language in particular which require special attention. The first is regarding the nature of the quantifiers. Harris et al. (1991, p. 10) give an example of a specification, the *slice*, which, rather than quantifying over the constituent's particles, quantifies over the elements of an abstract type. This mixing of quantification over arbitrary abstract types and identifiers makes the language complex from both a theoretical and technical perspective. For example, it is not, in general, possible, at the mathematical level, to prove or disprove (decide) the existence of a particular element of an arbitrary abstract type using its interface alone. Any formal definition of a logical language must explicitly define the range of significance of its quantifiers, and any means by which this range may be bounded. The type-based framework presented in this thesis provides a uniform solution to this problem, in which the range of significance of any quantifier is a specific type.

A second aspect of the language is that the example specifications given by Harris et al. (1991) are not defined as logical predicates over constituents. Instead, they are defined as constants whose truth value is dependent on the object to which they are attached. This is more in the manner of the frame paradigm of knowledge representation (Minsky, 1974, see §3.4), where a constituent is a frame and the specification is a procedural attachment which will evaluate to either true or false for any particular frame to which it is attached. The disadvantage of this approach is that it prohibits the reuse and composition of specifications. For example, how could we give a specification for 'a stream of chords' by reusing existing specifications for *stream* and *chord*. Representing the definitions of specifications as predicates mitigates this problem. A representation of *stream* as a logical predicate would look as follows:

$$
\begin{aligned}
stream(x) \leftrightarrow & \forall p_1 \in \mathsf{GetParticles}(x). \\
& \neg\exists p_2 \in \mathsf{GetParticles}(x). \\
& p_1 \neq p_2 \wedge \\
& \mathsf{GetTime}(p_1) \leqslant \mathsf{GetTime}(p_2) \wedge \\
& \mathsf{GetTime}(p_2) < (\mathsf{GetTime}(p_1) + \mathsf{GetDuration}(p_1))
\end{aligned}
$$

Here *stream* is a unary predicate over constituents, in which the quantifiers are explicitly bounded to the set of particles of the argument. We can now write a specification for ‘a stream of chords’ as follows:

$$
\begin{aligned}
stream\_of\_chords(x) \leftrightarrow & stream(x) \wedge \\
& \forall p \in \mathsf{GetParticles}(x).\ chord(p)
\end{aligned}
$$

Notice how we are able to apply the predicates *stream* and *chord* to the particles or sub-constituents of a constituent inside the definition of its specification. This example serves to highlight the kind of composability we would like to achieve. This is an important feature of knowledge representation systems, particularly in complex domains such as music, where the descriptions of sophisticated structures could become extremely large. The specification logic defined in Chapter 6 as a part of the type-based framework defines specifications as predicates for exactly this reason, allowing complex structural descriptions to be built from simple equational properties of constituent structures.

Summary

The two areas of CHARM discussed above are both important from the perspective of knowledge representation, and require rigorous formalisation if we to reach the “…eventual situation where any researcher in computer music could use any program with his of her chosen representation system, limited only by the suitability of the program for computation over the data represented" (Smaill et al., 1993, p. 1). Moving CHARM towards a fully specified, implementation independent knowledge representation system is a crucial first step. The type-based framework presented in Part II inherits much of the initial groundwork laid by CHARM, and builds upon it to provides formal solutions to the aspects discussed in this section.

4.6 Summary and Conclusions

The challenge of designing a general purpose music knowledge representation system stems from both the ontological complexity of music as an domain, and from the diversity of different application requirements. Existing representations tend to be tailored to a specific application and are in general not directly interoperable, either syntactically or conceptually. Approaching music representation from the meta-conceptual level is essential in unifying different activities. In this section, we have identified five meta-conceptual requirements for general purpose music representations, and have surveyed a number of existing systems, highlighting how the variously address these requirements. In particular, many representations focus of some kind of musical event whose properties commonly include pitch and onset values. However, most representations are limited regarding the higher level structures which they can explicitly capture, and often espouse syntactic data encoding over semantic knowledge representation.

The Semantic Web is an attractive solution for knowledge representation in the field of music, and many ontologies and methods have been developed for specific purposes. However, these ontologies tend to be designed with a specific research application in mind, even when their intended domain is broad. In addition, the different fundamental perspectives of musical ontology, especially with regard to time, have given rise to different high-level models which are not conceptually interoperable. Finally, the expressive limitations of semantic web modelling languages prevent the axiomatisation of many kinds of musicological knowledge. Specifically, the open world assumption and restriction semantics of OWL force knowledge engineers to adopt complex solutions to the modelling of music which often stray outside the realms which are naturally accommodated by description logics.

The CHARM system is a strong attempt at unifying representation approaches. The abstract data type approach to modelling musical attributes is a key step in promoting interoperability between representations. However, the unfinished or underspecified aspects of the original CHARM work make it hard to directly implement.

The importance of types in the design of interoperable information systems is highlighted when considering a complex domain such as music. The widespread use of singly-typed FOL-based knowledge representations results in the limited applicability of existing technologies to the design of general purpose music representation systems.

In the next part, we present an approach to music knowledge representation using type-based methods of specification which attempts to remove some of the limitations of existing approaches.

Part II

A Framework for Music Representation

Summary

In this part we describe a type-based framework for music knowledge representation. The purpose of the framework is to provide music researchers with a formal basis on which to store, access and manipulate musical information in a uniform way. The framework consists of two parts. The first part (Chapter 5) is a multiple-hierarchical abstract information model called the constituent structure. The second part of the framework (Chapter 6) is a logical language which can be used to express the structural properties of the represented information. The framework is entirely general, incorporating diverse kinds of information through a fixed type system. The different information requirements of research tasks can be incorporated by extending the framework (Chapter 7) with new types and specification definitions predicated upon these types. The framework allows users to integrate and reason with diverse kinds of musical information in a controlled way and thus affords greater interoperability between computer systems for music research.

Chapter 5

The Constituent Structure Representation

5.1 Introduction

In this chapter we describe the constituent structure representation of music. A constituent structure is an abstract, multiple-hierarchical information model. Nodes in the hierarchy represent musical objects and can be fitted with attribute values that capture their inherent characteristics. The type of attribute information is defined through specifications of abstract data types. The hierarchical relation between nodes represents the compositional containment of musical objects. This structure provides a high-level framework which generalises many existing music representation systems and allows simple information to be integrated with with sophisticated structural and analytic annotations.

5.2 Conceptual Basis of the Representation

In this section we introduce the constituent structure representation by outlining the conceptual and ontological foundations on which it is based. The conceptual basis of the representation consists of four ontological assumptions. These are intended to be as general as possible, favouring pragmatism and generality over any claim as to a definitive musical ontology.

Musical Objects

First, we presuppose that the musical universe can be usefully decomposed into a domain of discrete *musical objects*. We use the term musical object here in the broadest possible sense: the perceived sound objects of performed music; the physical sound objects of recorded audio; the graphemic objects (from Babbitt's graphemic domain, discussed in §4.2) of a musical score; these are all examples of the kinds of musical object which may participate in human conceptualisations of music. We emphasise 'usefully' as the manner of decomposition to highlight that this conceptualisation is a pragmatic one, serving the conceptual needs of the user. That is, we do not assume that musical objects *define* music. Rather they are atomic entities, used in the conception of structure, to which which further description can be assigned. *Truthful* decomposition of the musical universe, of the kind involved in the philosophical research field of formal ontology (Guarino, 1998), is a job for music ontology proper, and, we argue, not required for a general purpose music knowledge representation system.

Hierarchy

Second, we presuppose that the domain of musical objects is hierarchically structured. This assumption should not be controversial: the importance of hierarchy in musical ideation is widely acknowledge (see §4.2.3). This inherent hierarchy is characterised by the relationship between *part* and *whole* i.e., a musical object can be viewed as a composite of smaller structural units at lover levels of abstraction or alternatively (and simultaneously) as a component of a larger structural unit at some higher level of abstraction. In either case, the existence and identity of a musical object is inextricably tied to the existence and identity of its parts; the whole is understood to exist consequently, born from the individual characteristics and mutual juxtaposition of the parts. For example, a perceived sound event might be regarded as a part of a larger perceived or cognitively ordained structure such as a melodic phrase or harmonic texture, or as a fusion or smaller perceptual contributions. A musical glyph might be decomposed into a collection of smaller graphemes, or regarded as a component of a higher level syntagmatic unit such as a bar. A physical sound might be decomposed into a simultaneity of separate sound sources or succession of smaller segments, or regarded as an element of a larger structure such as a multi-track recording or digital music archive. We assume nothing about this hierarchy, other than it is acyclic i.e., we assume that no musical object can be part of itself.

Musical Spaces

Third, we presuppose that musical objects are *understood* through their identification with points in 'musical spaces'. We use the term 'musical space' here in the broadest possible sense: pitch, duration, loudness, frequency, amplitude, waveform, chord type, music glyph, melodic contour, style, genre; these are all examples suggestive of musical spaces, or particular dimensions of composite spaces, wherein musical objects can be located. For example, physical sound objects may be identified with points in frequency or waveform space. Perceived sound objects may be identified with points in pitch or timbre space. A musical glyph may be identified with a point in a space of symbols belonging to a certain notational convention or in a Cartesian plane representing its location on the page. We emphasis 'understanding' for the role of musical spaces to highlight the descriptive (rather than definitive) nature of this conceptualisation.

Reasoning

Fourth, we presuppose that the structure of musical spaces forms the basis for reasoning and manipulation of the musical objects which they describe. As such, we regard musical spaces as 'theories'; particular abstract models of some aspect of the musical universe which serve as conceptual foundations for the formal analysis of objects. For example, the space of perceived pitch may have an interval structure, allowing pairs of points to be uniquely identified with a pitch interval. A space of musical symbols might be regarded purely as a discrete space with no structure other than nominal identification of points. We do not presume that musical spaces necessarily coincide with any particular mathematical spaces. For example, musical spaces need not be metric or ordered (though some may be). We only assume, as a minimum requirement, that their points be nominally identifiable so that musical objects identified with equal points can be judged equivalent in some respect.

5.3 Concepts and Terminology

Having outlined our ontological assumptions, we now informally describe the constituent structure representation by introducing the terminology and concepts which realise the conceptual basis. In particular we introduce the concept of a constituent. The term 'constituent' is taken from the work on CHARM (Wiggins et al., 1989), described in §4.5, but is defined slightly differently here. A detailed comparison of the constituent structure representation and CHARM is given in §5.6.

Constituents

Our representation treats the melange of latent hierarchically related musical objects in a completely general and uniform way. Every extant musical object is designated a representative called a *Constituent*. Constituents are formed from finite collections of other constituents to form multiple-hierarchical structures. We place no restriction on the sorts of musical objects that constituents can represent. Anything identifiable in the process of human conceptualisation should be representable. For example, a written text 'about' music, a compositional germ imagined in the mind of a composer, an emotion elicited in the mind of a listener, the physical movements of performers, the molecules of an instrument; these could all be constituents. Although we know of no representation system which handles, or even aims to handle, such diverse entities, it is clear that the human mind does so with ease, conjuring musical objects whenever the discourse requires them. The generality and flexibility of constituents is intended to capture the ad hoc nature of human conceptualisation of music. We prescribe no essential characteristics or ontological classification of constituents or the relationships which exist between them (other than the fundamental hierarchical relation); they are raw entities composed of other entities that can be refined with additional information.

Abstract Data Types

Musical spaces are given a similarly general treatment: each space is represented by an algebraic specification of an *abstract data type* (Dale and Walker, 1996). The points in the space are the values of the type, and any additional structure that the space bares must be formally captured in the specification by adequate operations, predicates and axioms. Our system does not prescribe any fixed set of abstract data types, nor any operations which they must have, other than a decidable equality operation. The purpose of the abstract data type specifications is two-fold. Firstly, the specification provides an abstract interface which separates the logical and computational properties of the data from the irrelevant (and possibly complex) implementation details. This allows applications which use the interface to operate on different underlying encoding formats. Secondly, the specifications provides formal documentation of the specific 'theory' which is being used to describe musical objects. Making this theory explicit affords greater transparency and conceptual interoperability between systems which process the information.

Attributes

Constituents are identified with elements of abstract data types via *Attributes*. An attribute is a named connection between a constituent and a value. The name of the attribute connection identifies a particular aspect or quality of the represented object.

Each attribute name is uniquely associated with an abstract type representing a musical space. The value of an attribute connection is an element of the abstract type associated with the attribute name. Attributes can be thought of as explicitly representing a projection of a musical object into a musical space. We take the view that the piecemeal ascription of attributes to extant musical objects is sensible as an approximation of human conceptualisation of music, which is fluid and ad hoc, depending on context, musical aptitude, cultural sensibilities, specific activity and the passage of time.

Constituent Structures

A hierarchical collection of constituents along with their attribute connections is a *Constituent Structure* and forms the basis of our representation framework. The modes of analysis and manipulation of a constituent structure are determined by the attributes of the constituents and the algebraic properties of the types of their values. The generality of this structure is an acknowledgement of the multiplicity of potential music conceptualisations, allowing them to coexist and be applied in a modular way to specialise the representation depending on the requirements of the user.

5.4 Structural Specification

In this section we give a structural description of the constituent structure representation. The constituent structure representation is described as a class of set-theoretic structures which capture the fundamental properties of the conceptualisation. Our aim is to make explicit the semantic basis for the representation in a static way. This will support and aid the understanding of the functional model given in the following section, which aims to capture the computational behaviour of the representation.

The components of a constituent structure form a tuple $\langle C, H, A, R, M \rangle$, where C is the set of constituent musical objects, H is the hierarchical relation between the constituents, A is a set of attribute names, M is a collection of musical spaces and R is a map from constituents to tuples of points in musical spaces.

$H \subset C \times C$ is a binary relation on C, such that (C, H) forms a (simple) directed acyclic graph (DAG). The graph (C, H) captures the hierarchical structure of the domain, where $(c, c') \in H$ indicates that c' is a *part of* c. We call the set $P(c) := \{p \in C | (c, p) \in H\}$ the *particles* of c. The set $Sub(c) := \{c' \in C | (c, c') \in H^+\}$ is the set of *sub-constituents* of c, where H^+ is the transitive closure of H. The exclusion of directed cycles reflects the ontological premise that no musical object can be a sub-constituent of itself.

The set A of attribute names can be thought of as the set of perspectives which link constituents to their attributes. Alternatively we think of A as the set of keys that can be used for attribute key-value pairs. The set of locations in musical space $M = (M_a)_{a \in A}$ is a family of sets indexed by A, where M_a is the subspace or dimension indicated by the perspective a. Alternatively M_a is the set of permissible values for attribute key-value pairs whose key is $a \in A$. Attribute key-value pairs are therefore elements of the disjoint union $\coprod_{a \in A} M_a := \{(a, v) | a \in A \wedge v \in M_a\}$.

The map $R : C \to \prod_{a \in A} M_a \cup \{null\}$ assigns to each element $c \in C$ a tuple $R(c) = (v_a)_{a \in A}$ where each v_a is either an element of M_a or is $null$. We can think of R in a number of different but equivalent ways: (*i*) $R(c) \subset \coprod_{a \in A} M_a$ can be thought of as a set of attribute key-value pairs such that no two elements have the same key. (*ii*) R can be thought of as a family of partial maps $(get_a)_{a \in A}$ where $get_a : C \rightharpoonup M_a$ maps a subset of C to values $v \in M_a$ (intuitively, a projection operation). (*iii*) R can be thought of as a table with a row for each $c \in C$ and a column for each attribute $a \in A$, in which each cell $R_{c,a}$ is either an element of M_a or is $null$.

This model is useful for understanding the semantic identity of the constituent structure representation, however it is insufficient as a basis for a computer implementation of the system. In practise we need a symbolic representation for the components of the structure as well as operations for manipulating them in application software. In particular, computers do not deal well with sets and set maps. We must therefore consider how this semantic structure may be adequately implemented as a concrete information structure in a computer system. For this we observe the following: any constituent $c \in C$ can be represented as a tuple

$$\langle \textit{identifier, particles, attributes} \rangle,$$

where $\textit{identifier} = c$, $\textit{particles} = P(c)$ and $\textit{attributes} = R(c)$. Any constituent structure can therefore be represented as set of such tuples. This representation of constituents is a simpler version to that presented by Harris et al. (1991) and captures the identity and attributes of constituents. A constituent identifier will typically be a 'string' or 'database id'. Therefore, a constituent structure can be thought of as an associative array (finite map) from identifiers to pairs $\langle \textit{particles, attributes} \rangle$ which satisfies some extra constraints. This perspective forms the basis of the functional model of constituent structures given in the next section, where constituents and structures are defined as abstract types.

5.5 Functional Specification

In this section we give a function specification of the constituent structure representation in dependent type theory. Our aim is to capture the computational behaviour of the representation at an abstract level which can be used as a specification for implementations of the system. The specification is given in the Calculus of Inductive Constructions using the syntax and notational conventions described in §2.2.3. The core of the model are four abstract types: UCI of constituent identifiers, AN of attribute names, COBJ of constituent objects and STRUC of constituent structures. For each of these types we specify functional interfaces, including operations and logical axioms which constrain their behaviour.

5.5.1 Constituent Identifiers

$\mathsf{UCI} : \mathsf{Set}$ is the type of universal constituent identifiers. Terms of this type $x : \mathsf{UCI}$ are symbolic names which can be used to uniquely identify constituents. Ontologically, we think of identifiers as references to extant musical objects. Computationally, we think of this type as a type of *pointers* to data structures containing information about these objects. Identifiers can be freely duplicated and stored in data structures, however simple existence of an identifier is not sufficient for it to be dereferenced. Dereferencing is specified as an operation in the constituent structure interface (§5.5.4). We do not define any operations on identifiers except testing for equality which is a pure computation. The set of constituents C from the previous section is represented functionally in the type $\mathsf{fset}(\mathsf{UCI})$.

5.5.2 Attribute Names and Types

$\mathsf{AN} : \mathsf{Set}$ is the type of *attribute names* and is the functional analogue of the set A in the previous section. Elements of the type are symbolic names which are used to identify attribute projections for constituents. In addition we define the function $\mathsf{typ} : \mathsf{AN} \rightarrow \mathsf{AT}$ which maps attribute names $a : \mathsf{AN}$ to attribute types $\mathsf{typ}(a) : \mathsf{AT}$ where the type AT is defined to be the type universe Set. In this way the family M becomes the type universe Set with the indexing captured by the function typ. Attribute key-value pairs can therefore be represented in the dependent sum type $\Sigma a : \mathsf{AT}.\mathsf{typ}(a)$.

5.5.3 Constituent Objects

$\mathsf{COBJ} : \mathsf{Set}$ is the type of *constituent objects*. Terms of this type $c : \mathsf{Con}$ are symbolic representations of pairs $\langle$*particles, attributes*$\rangle$. Ontologically, we think of a constituent object as consisting of a set of other constituents and a collection of attributes. Computationally, we think of the type as the type of data structures which are the returned as a result of dereferencing an identifier.

Operations

The operations defined on constituent objects are shown in Table 5.1. The constructor $\mathsf{delimit}$ takes a set of identifiers and returns a constituent object. The operation $\mathsf{particles}$ extracts the set of particle identifiers of the constituent object. The operations get and set lookup and modify respectively the constituent's attribute values. These operations are dependently typed; the argument $a : \mathsf{AN}$ determines the type of values which are retrieved or set. The partiality of the get operation is captured by the option type constructor. Note that the set operation is a pure function i.e. it returns a new constituent object with a modified environment.

Table 5.1: Constituent object operations

$$\begin{aligned}
\mathsf{delimit} &: \mathsf{fset}(\mathsf{UCI}) \to \mathsf{COBJ} \\
\mathsf{particles} &: \mathsf{COBJ} \to \mathsf{fset}(\mathsf{UCI}) \\
\mathsf{get} &: \Pi a : \mathsf{AN}.\mathsf{COBJ} \to \mathsf{option}(\mathsf{typ}(a)) \\
\mathsf{set} &: \Pi a : \mathsf{AN}.\mathsf{typ}(a) \to \mathsf{COBJ} \to \mathsf{COBJ}
\end{aligned}$$

Axioms

Table 5.2 specifies the axioms which the operations in Table 5.1 must satisfy. $\mathsf{particles\_delimit}$ specifies that the set of identifiers delimited by a constituent object *are* the particles of that object. $\mathsf{get\_delim}$ specifies that a newly delimited set of identifiers has no attributes. $\mathsf{particles\_set}$ specifies that setting attributes does not modify the particles. $\mathsf{get\_set\_same}$ specifies that, for two equal attribute names $a = a'$, getting a after setting a' returns the value that was just set. $\mathsf{get\_set\_other}$ specifies that, for two different attribute names $a \neq a'$, getting a after setting a' is the same as just getting a. These axioms ensure that the operations respect the structure of $R(c)$ described in §5.4.

Table 5.2: Constituent object axioms

$$\begin{aligned}
\mathsf{particles\_delimit} &: \forall ps.\mathsf{particles}(\mathsf{delimit}(ps)) = ps \\
\mathsf{particles\_set} &: \forall a\ v\ c.\mathsf{particles}(\mathsf{set}(a, v, c)) = \mathsf{particles}(c) \\
\mathsf{get\_delimit} &: \forall \mathrm{ps\ a.}\ \mathsf{get}(\mathrm{a}, \mathsf{delimit}(\mathrm{ps})) = \mathsf{None} \\
\mathsf{get\_set\_same} &: \forall a\ a'\ v\ c.\ a = a' \to \mathsf{get}(a, \mathsf{set}(a', v, c)) = \mathsf{Some}(v) \\
\mathsf{get\_set\_other} &: \forall a\ a'\ v\ c.\ a \neq a' \to \mathsf{get}(a, \mathsf{set}(a', v, c)) = \mathsf{get}(a', c)
\end{aligned}$$

5.5.4 Constituent Structures

STRUC : Set is the type of *constituent structures*. Terms of the type s : STRUC are symbolic representations constituent structures. Ontologically, we think of a constituent structure as describing a collection of musical objects, their attributes and hierarchical relationships. Computationally, they are associative arrays: assignments or bindings of constituent identifiers to objects.

Operations

The operations defined for structures are shown in Table 5.3. The constant empty denotes the empty structure. The operation insert takes an identifier, a constituent object and a structure and returns a new structure in which the identifier is bound to the object. The operation lookup takes an identifier and a structure and returns either some constituent object or None. The operation domain takes a structure and returns the finite set (C) of identifiers on which it is defined. Note that the insert operation is pure and overwrites any existing constituent binding.

Axioms

Table 5.4 defines the axioms which the operations in Table 5.3 must satisfy. lookup_empty specifies that the empty structure empty contains no constituents. lookup_insert_same and lookup_insert_other are similar to those for get and set in Table 5.2 and express the compatibility of the lookup and insert with the view of struc-

Table 5.3: Constituent structure operations

$$\begin{aligned}\mathsf{empty} &: \mathsf{STRUC}\\ \mathsf{insert} &: \mathsf{UCI} \to \mathsf{COBJ} \to \mathsf{STRUC} \to \mathsf{STRUC}\\ \mathsf{lookup} &: \mathsf{UCI} \to \mathsf{STRUC} \to \mathsf{option}(\mathsf{COBJ})\\ \mathsf{domain} &: \mathsf{STRUC} \to \mathsf{fset}(\mathsf{UCI})\end{aligned}$$

tures as finite maps. domain_lookup specifies that the set of identifiers returned by the operation domain contains only those identifiers which are bound to values.

Table 5.4: Constituent structure axioms

$$\begin{aligned}\mathsf{lookup\_empty} &: \forall x.\ \mathsf{lookup}(x, \mathsf{empty}) = \mathsf{None}\\ \mathsf{lookup\_insert\_same} &: \forall x, x', c, s.\ x = x' \to \mathsf{lookup}(x, (\mathsf{insert}(x, c, s))) = c\\ \mathsf{lookup\_insert\_other} &: \forall x, x', c, s.\ x \neq x' \to \mathsf{lookup}(y, \mathsf{insert}(x, c, s)) = \mathsf{lookup}(y, s)\\ \mathsf{domain\_lookup} &: \forall x, s.\ x \in \mathsf{domain}(s) \leftrightarrow \mathsf{lookup}(x, s) \neq \mathsf{None}\end{aligned}$$

Note that the hierarchical relation H is only implicit in this model, arising from the construction and insertion of constituents. In addition, the specification does not enforce the acyclicity of constituent structures, nor does it enforce that constituent structures contain complete information i.e., that constituents inserted into a structure have particles that are already bound. The reason for this is that, in distributed information systems, individual documents may only contain partial information, relying on the universality of identifiers to link them. It is therefore desirable that we are able to consider such partial information as well-typed. In §6.2.3 we specify the acyclically and completeness of structures as a logical invariant over constituent structures.

5.6 Summary and Conclusions

In this chapter we have formally defined a representation of music. We have described the conceptual basis and given both a structural and functional specification of the representation. We now summarise the core features.

The representation is abstract: The functional specification uses abstract data types, defined by their functional behaviour. This means that any concrete encoding format can be used to implement the representation so long as it can be adequately interpreted via the structural description (§5.4) and supports implementations of the functional behaviour (§5.5).

The representation is music-generic: The conceptual basis of the representations is entirely general, subsuming many exiting representations of music. It can be seen as a meta-conceptual framework, into which domain-specific representations can exist side by side.

The representation is extensible: The representation is parametric in the types of constituent attribute information. As such, the different representational requirements of music researchers are accommodated through the introduction of new ADT specifications.

The representation is highly expressively complete: The ability to accommodate arbitrary data types means that the representation is as expressive as CIC itself.

The representation is highly structurally general: The ability to represent multiple hierarchies of musical objects with attribute data makes the model capable of expressing arbitrarily complex structures.

The representation generalises many existing approaches. For example, the attribute mechanism separates the name of the attribute from its value (or filler) in the style of KL-ONE ontology languages (Brachman and Schmolze, 1985) and the attributive/referential distinction (Donnellan, 1966). In addition, attributes can be seen as viewpoints in the manner of Conklin and Witten (1995), with canonical projections given by the functional interface. The abstract data type approach subsumes the conceptual space approach (Gärdenfors, 2000), as conceptual spaces and their components can be defined as algebraic specifications and encoded in CIC.

An important aspect of the type-based approach is that it supports the axiomatisation of further operations on constituent structures. Some examples of additional operations might include equality, subsumption, merging and intersecting, retrieving all sub-constituent identifiers of a given constituent, or extracting the downward closure of a set of identifiers. These and many more could be adapted from the Coq standard library module for finite maps[1].

The crucial aspect of the representation which sets it apart from existing methods is the functional specification given in CIC, which captures the algebraic properties of the representation. This can be used as a basis for integrating the representation into functional programming languages. For example, we could write high-level functional programs for manipulating constituent structures using state and option monads (Wadler, 1992). CIC supports the representation of functional data types such as those of Hudak (2011) and grammars such as Bel and Kippen (1992) using higher-order abstract sytanx. Having a functional specification of constituent structures makes it possible to define mappings to and from these other representations.

The functional model can be viewed as a low-level basis upon which more sophisticated data manipulation languages can be defined. It is also possible to define higher-level languages for manipulation of imperative constituents structures using higher-order abstract syntax in CIC, and define their operational semantics in terms of the functional model. In this approach the functional model is used as the type of system or memory states. This is the approach taken by Chlipala (2011).

The formal and technical advantages of using type theory stem the property of equational reasoning. In particular, this will be used in the next chapter as the basis of a specification logic.

We conclude this chapter with a comparison of our representation with the CHARM system, its closest conceptual neighbour. The key similarities are in the use of abstract

[1]https://coq.inria.fr/library/Coq.FSets.FMapInterface.html

data types for modelling musical parameter spaces and the representation of multiple hierarchies of constituent entities. Our main addition to the original work is the fully specified functional model of constituent structure behaviour. The core difference of our system is that it uses type theory as a specification language and logical basis. This does not make it incompatible with the original work, but rather generalises it. In addition, there are a number of minor but fundamental conceptual differences which are as follows:

1. Our specification is given in dependent type theory with emphasis on the functional behaviour of components. This is deliberately to encourage the full separation of logical specification and implementation. The previous work emphasises the structure of entities which influences the implementation. By fully specifying the functional behaviour of entities we can recover the original structural and syntactic notions.

2. We do not differentiate between a basic representation and the constituent mechanism. Instead, every entity is a constituent with a possibly empty set of particles. The reason for this is that is that we do not wish to prescribe any kind of atomic entity. This allows the user of the representation greater flexibility in tailoring the representation to their needs as well as inviting greater integration of representations at different levels of detail.

3. We do not prescribe any inherent attributes of constituents. Instead, any constituent can be fitted with any attribute. This is in contrast to the original in which a fixed abstraction was used to represent atomic entities. The reason for this is firstly that it allows for the integration of representations at varying degrees of expressiveness without the need for conversion tools, and secondly to allow for the representation of partial information in which certain attributes might be unknown.

4. Instead of default values we use the `option` type constructor to capture the par-

tiality of operations on representation components. We argue that this is a more rigorous approach to the design of information systems and a more natural way of specifying systems behaviour in the vast majority of concrete cases.

Chapter 6

A Specification Logic for Constituent Structures

6.1 Introduction

In this chapter we define a specification logic for expressing the properties of constituent structures and their components. The language constructs are defined in CIC and depend on the functional model of constituent structures given in Chapter 5. We define three types which capture the different kinds of expressions in the language: SPROP of structure invariants, CSPEC of constituent specifications and CREL of constituent relations.

Each of these types captures a kind of constructive predicate i.e., a type of Prop-valued functions. SPROP is the type of unary predicates over constituent structures and captures equational invariants of constituent structures. We define a number of operations for combining SPROPs which together form an expressive logical language. This language of invariants forms the basis for defining the types CSPEC and CREL of constituent specifications and relations, respectively. CSPEC is the type of unary SPROP-valued predicates over constituents and CREL is the type of binary SPROP-valued predicates (relations) over constituents. These types capture intrinsic structural properties of named constituents and their relationships in a formal way.

The logical symbols of the language are defined using the notion of *pointwise lifting* of an operation. Specifically, given two types A and B and some n-ary $m : B^n \to B$, we can *lift* the operation m to create an n-ary operation $[m]_{\mathrm{A}} : (A \to B)^n \to (A \to B)$ on functions f from A to B. This operation is defined *pointwise* i.e. $[m]_{\mathrm{A}}(f_1, ..., f_n) = \lambda a.m(f_1(a), ..., f_n(a))$. We denote lifting with square brackets and also use subscripts $_{\mathrm{A}}$ to indicate the type of functions to which the operation has been lifted. In addition, elements (nullary operations) $b : B$ can be lifted to constant functions $[b]_{\mathrm{A}} := \lambda\_.b$. We use underscores _ instead of variables to indicate that the function is constant i.e., the abstracted variable is not free in the body of function. Our specific use of pointwise lifting is in regard to logical predicates $P : A \to \mathsf{Prop}$, where the operations on Prop are the standard inductive definitions of the logical connectives in CIC.

6.2 Structure Invariants

$\mathsf{SPROP} := \mathsf{STRUC} \to \mathsf{Prop}$ is the type of *structure invariants* (unary predicates over structures). These predicates express properties of the information contained in constituent structures via the equational properties of the functional model (§5.5). Elements of the type $\phi : \mathsf{SPROP}$ are lambda terms and applications $\phi(s)$ are logical propositions (dependent types of sort Prop) corresponding with the assertion that the invariant ϕ holds of the structure s.

6.2.1 Atomic Structure Invariants

Table 6.1 defines two atomic structure invariants. $x \overset{a}{\mapsto} v$ asserts that a structure contains a constituent named x with attribute $a : \mathsf{AN}$ of value $v : \mathsf{typ}(a)$. $x \ll y$ asserts that a structure contains a constituent named x whose set of particle identifiers contains y. We also write $y \gg x$ to mean y is part of x as well as $(p \gg)$ to mean $\lambda x.p \gg x$.

Table 6.1: Atomic structure invariants

$$x \overset{a}{\mapsto} v : \mathsf{SPROP} := \lambda s.\exists c.\mathsf{lookup}(x, s) = \mathsf{Some}(c) \land \mathsf{get}(a, c) = \mathsf{Some}(v)$$
$$x \ll y : \mathsf{SPROP} := \lambda s.\exists c.\mathsf{lookup}(x, s) = \mathsf{Some}(c) \land y \in \mathsf{particles}(c)$$

6.2.2 Logical Operations on Structure Invariants

We can compose structure invariants by pointwise lifting of the propositional connectives defined in Prop. We use the subscript $_\mathrm{S}$ to indicate the type of these lifted operations. Table 6.2 defines the lifted connectives. $[P]_\mathrm{S} : \mathsf{SPROP}$ is the lifting of a proposition $P : \mathsf{Prop}$ to the constant function $\lambda\_.P$. The quantification operations $[\forall]_\mathrm{S}, [\exists]_\mathrm{S} : (T \to \mathsf{SPROP}) \to \mathsf{SPROP}$ each take a T-parametrized invariant $\varphi : T \to \mathsf{SPROP}$ and return an invariant in which the formal parameter of φ is bound by the $\forall$ and $\exists$ type constructors, respectively. For readability, we sometimes omit the subscript $_\mathrm{S}$ when the type of the operation is clear from the context. In addition we write $[\forall]x.\phi$ as a short-hand for $[\forall]_\mathrm{S}(\lambda x.\phi)$.

Table 6.2: Logical operations on structure invariants

$$\begin{aligned}
[\top]_\mathrm{S} &:= \lambda\_.\top \\
[\bot]_\mathrm{S} &:= \lambda\_.\bot \\
\phi[\land]_\mathrm{S}\psi &:= \lambda s.\phi(s) \land \psi(s) \\
\phi[\lor]_\mathrm{S}\psi &:= \lambda s.\phi(s) \lor \psi(s) \\
\phi[\to]_\mathrm{S}\psi &:= \lambda s.\phi(s) \to \psi(s) \\
[\neg]_\mathrm{S}\phi &:= \lambda s.\neg\phi(s) \\
[\forall]_\mathrm{S}\varphi &:= \lambda s.\forall t.\varphi(t)(s) \\
[\exists]_\mathrm{S}\varphi &:= \lambda s.\exists t.\varphi(t)(s)
\end{aligned}$$

6.2.3 Well-Formedness of Constituent Structures

Table 6.3 defines the invariant $\mathsf{WF} : \mathsf{SPROP}$ which captures two specific properties of constituent structures: acyclicity and completeness. Acyclicity means the constituent hierarchy contains no directed cycles i.e., it ensures (C, H) is a DAG. Completeness means that every constituent identifier appearing in the particles of bound constituent objects are also bound in the structure. The well-formedness property is defined inductively by the axioms $\mathsf{WF\_empty}$ and $\mathsf{WF\_insert}$. $\mathsf{WF\_empty}$ states that the empty structure is well formed. $\mathsf{WF\_insert}$ states that inserting the pair (x, c) into any well formed structure s yields a well formed structure if the identifier x is unbound in s and the particles of c are all bound in s.

Table 6.3: Well-formedness of constituent structures

$$\begin{aligned}
\mathsf{WF} &: \mathsf{SPROP} \\
\mathsf{WF\_semp} &: \mathsf{WF}(\mathsf{semp}) \\
\mathsf{WF\_insert} &: \forall x\ c\ s. \mathsf{WF}(s) \to \mathsf{particles}(c) \subseteq \mathsf{domain}(s) \\
&\quad \to x \notin \mathsf{domain}(s) \to \mathsf{WF}(\mathsf{insert}(x, c, s))
\end{aligned}$$

6.3 Constituent Specifications

$\mathsf{CSPEC} := \mathsf{UCI} \to \mathsf{SPROP}$ is the type of *constituent specifications*: unary SPROP-valued predicates over constituents. Elements of the type $\Phi : \mathsf{CSPEC}$ are lambda terms and applications $\Phi(x)$ are structure invariants (themselves lambda terms). We can think of constituent specifications as defining classes (unary predicates) of constituents, for which membership is specified in terms of structure invariants.

6.3.1 Atomic Constituent Specifications

Atomic constituent specifications can be constructed by simple lambda abstraction over identifiers appearing in structure invariants, for example, $\lambda x.x \overset{a}{\mapsto} v$. Table 6.4 defines two atomic specifications. $\langle a \rangle$ asserts that a constituent contains some value for the attribute named a. $\mathsf{element}$ asserts that the constituent has an empty set of particles.

Table 6.4: Atomic constituent specifications

$$\begin{aligned}\langle a \rangle : \mathsf{CSPEC} &:= \lambda x.[\exists]v.x \overset{a}{\mapsto} v \\ \mathsf{element} : \mathsf{CSPEC} &:= \lambda x.[\forall]p.[\neg]x \ll p\end{aligned}$$

6.3.2 Logical Operations on Constituent Specifications

As before, we can compose constituent specifications by pointwise lifting of the logical connectives defined on structure invariants. We use subscript $_{\mathrm{C}}$ to indicates the type of the lifted operations, but sometimes omit this when the type is clear from the context. $[\phi]_{\mathrm{C}} : \mathsf{CSPEC}$ is the lifting of a structure invariant $\phi : \mathsf{SPROP}$ to the constant function $\lambda\_.\phi$. Table 6.5 defines the lifted logical operations for constituent specifications.

Table 6.5: Logical operations on constituent specifications

$$\begin{aligned}[\top]_{\mathrm{C}} &:= \lambda\_.[\top]_{\mathrm{S}} \\ [\bot]_{\mathrm{C}} &:= \lambda\_.[\bot]_{\mathrm{S}} \\ \Phi[\wedge]_{\mathrm{C}}\Psi &:= \lambda x.\Phi(x)[\wedge]_{\mathrm{S}}\Psi(x) \\ \Phi[\vee]_{\mathrm{C}}\Psi &:= \lambda x.\Phi(x)[\vee]_{\mathrm{S}}\Psi(x) \\ \Phi[\rightarrow]_{\mathrm{C}}\Psi &:= \lambda x.\Phi(x)[\rightarrow]_{\mathrm{S}}\Psi(x) \\ [\neg]_{\mathrm{C}}\Phi &:= \lambda x.[\neg]_{\mathrm{S}}\Phi(x) \\ [\forall]_{\mathrm{C}} &:= \lambda \mathit{\Phi}\ x.[\forall]_{\mathrm{S}}t.\mathit{\Phi}(t)(x) \\ [\exists]_{\mathrm{C}} &:= \lambda \mathit{\Phi}\ x.[\forall]_{\mathrm{S}}t.\mathit{\Phi}(t)(x)\end{aligned}$$

6.3.3 Quantification Over Constituent Particles

Often it is desirable to capture properties of constituents which quantify over the elements of their particle sets (rather than a type). This can be achieved using the operations already defined as follows: $[\forall]p.(p \gg)[\rightarrow][\Phi(p)] : \mathsf{CSPEC}$ is a specification which asserts that every particle p of a constituent satisfies some specification $\Phi : \mathsf{CSPEC}$. However, specifications of this kind do not compose very easily, as each nested quantifier must be qualified by $(p \gg)[\rightarrow]....$

Table 6.6 defines two operations $\forall_{\text{Parts}}$ and $\exists_{\text{Part}}$ specially for quantification over particles which hide the $(p \gg)[\rightarrow]$ component. In this way we can write specifications in a more concise and natural way. For example,

$$\forall_{\text{Parts}}\ p_1.[\neg]\exists_{\text{Part}}\ p_2.[[p_1 \neq p_2]_{\text{S}}[\wedge]_{\text{S}}R(p_1, p_2)]_{\text{C}} : \mathsf{CSPEC}$$

is a specification with nested particle quantification which asserts that no two distinct particles of a constituent are related by R. Note that we write $\forall_{\text{Parts}}\ p_1.\Phi$ as a shorthand for $\forall_{\text{Parts}}(\lambda p_1.\Phi)$.

Table 6.6: Operations for the quantification over constituent particles

$$\forall_{\text{Parts}} : (\mathsf{UCI} \rightarrow \mathsf{CSPEC}) \rightarrow \mathsf{CSPEC} := \lambda \varPhi.[\forall]p.(p \gg)[\rightarrow]\varPhi(p)$$
$$\exists_{\text{Part}} : (\mathsf{UCI} \rightarrow \mathsf{CSPEC}) \rightarrow \mathsf{CSPEC} := \lambda \varPhi.[\exists]p.(p \gg)[\wedge]\varPhi(p)$$

6.3.4 Relations Between Constituents

$\mathsf{CREL} : \mathsf{UCI} \rightarrow \mathsf{UCI} \rightarrow \mathsf{SPROP}$ is the type of *constituent relations*: binary SPROP-valued predicates over constituents. One such relation we have already introduced: the atomic hierarchical relation $(\ll) : \mathsf{CREL}$ between constituents is defined in terms of the basic operations on constituent structures. Other relations can be formed by nested lambda abstractions over identifier variables in arbitrary structure invariants.

Table 6.7: Atomic constituent relations

$$\begin{aligned}
\mathsf{AttRel}(a)(R) : \mathsf{CREL} :=&\ \lambda x\ y.[\exists] v_1\ v_2.x \overset{a}{\mapsto} v_1 [\wedge] y \overset{a}{\mapsto} v_2 [\wedge] [R(v_1, v_2)] \\
\mathsf{PartRel}(R) : \mathsf{CREL} :=&\ \lambda x\ y.[\exists] l_1\ l_2.\mathsf{elements}(x, l_1) [\wedge] \\
&\qquad\qquad\quad \mathsf{elements}(y, l_y) [\wedge] \\
&\qquad\qquad\quad \mathsf{PairWise}(R, l_1, l_2)
\end{aligned}$$

Table 6.7 defines two atomic constituent relations. $\mathsf{AttRel}(a) : (\mathsf{typ}(a) \to \mathsf{typ}(a) \to \mathsf{Prop}) \to \mathsf{CREL}$ is an atomic relation constructor which takes a relation R on attribute values of type $\mathsf{typ}(a)$ and returns a constituent relation which asserts that two constituents x and y have attribute values v_1 and v_2 which are related by R. $\mathsf{PartRel} : \mathsf{CREL} \to \mathsf{CREL}$ is a relation constructor which takes a constituent relation R and returns a new constituent relation which asserts that the particles of the constituents can be represented as lists which are pairwise related by R. The structure invariant $\mathsf{elements}(x, l)$ asserts that the particles of a constituent x can be represented by the functional list $l : \mathsf{list}(\mathsf{UCI})$, and the structure invariant $\mathsf{PairWise}(R, l_1, l_2)$ asserts that the elements of two lists of identifiers are pairwise related by the constituent relation R. The definitions of these two structure invariants are defined using the inductive structure of functional lists. However, their definitions are omitted as they are not essential in understanding the definition of $\mathsf{PartRel}$. As before, constituent relations can be composed by lifting structure invariants (and constituent specifications) and their corresponding connectives. We use the subscript $_{\mathrm{R}}$ to denote the pointwise operations on relations.

6.4 Summary and Conclusions

In this chapter we have defined a specification logic for constituent structures. It constitutes a highly expressive language for giving structural descriptions of constituent structures. The origin of its expressive power is the underlying constructive logic of CIC and in the next chapter it will be used to axiomatise sophisticated musical structures.

The expressive power of the language comes at a cost from the perspective of the knowledge engineer. Firstly, as with any expressive language, there is more than one way to construct equivalent statements. This fosters potential overlap and redundancy in knowledge bases which use the language. Secondly, the kind of global, fully automated reasoning, such as that afforded by description logic languages, is not possible.

The former issue is mitigated by the fact that the logic is constructive with a proof-theoretic semantics. This means that it is possible to construct proofs of the equivalence of two statements. These proofs, being functional programs, can be used as functions which actually transform a proof of one statement into a proof of the other.

The later issue requires paradigm shift in the practise of knowledge representation and reasoning. Traditionally, knowledge representation languages are created in order to support off-the-shelf reasoning procedures (such as classifiers), often at the expense of the expressive power of the language. In the type-theoretic setting, the process of reasoning is equatable to type-based theorem proving. As such, in the constructive setting the knowledge engineer must take up responsibility for defining automated procedures for constructing and manipulating proofs. Rather than using black box reasoners with complex implementations, the knowledge engineer can use proof assistants (such as Coq) to implement functional decision procedures or perform tactic-based proof search.

This shift in thinking, from set-theoretic, model-based systems to constructive type-theoretic, proof-based systems, requires a significant level of commitment from the research community. However, the advantages of such an approach are are being demonstrated in the verified-software and programming language communities with the development of ecosystems for software development and formal verification (Chlipala, 2013).

Chapter 7

Extensions to the Framework

7.1 Introduction

In this chapter we present six extensions to the representation framework described in the Chapters 5 and 6. The purpose of these extensions is to demonstrate how the type-based framework supports two crucially important aspects of our approach to music representation: Firstly, how knowledge engineers may use the type-based method of abstract representation to design representations to suit their specific conceptualisation of musical aspects. Secondly, how constituent structures and the specification logic can be extended with new operations and definitions to allow for the expression of additional representational features.

Chapter Outline

§7.2 describes how note-like events can be represented via specification of abstract data types for pitch and time, as well as higher-level temporal structures. §7.3, §7.4 and §7.5 describe extensions whereby particular properties and characteristics of constituents can be explicitly represented and reasoned with. §7.6 describes how arbitrary associations between constituents may be represented. §7.7 describes an extension to the syntax of the specification logic which allows for concise descriptions of structured objects.

7.2 Representing Note-Like Events

In this section we propose abstract data type representations of pitch and time and show how they can be used in specifications for higher-level musical structures. Pitch and time are two fundamental dimensions of musical structure and are used in many applications. Conceptually, we regard these aspects of music as musical spaces. The abstract data type representation constitutes a formal theory of the nature of these spaces. It is important to stress here, that the abstract representations presented here are not intended to be definitive of either pitch or time, but are merely indented to illustrate the method by which specific conceptualisations of these aspects of music can be can be made formally explicit.

7.2.1 Abstract Data Type Specifications for Pitch and Time

Our specifications follows that of Wiggins et al. (1989), in which pitch and time are each considered to be linearly ordered interval dimensions characterised by a type of *points* and a type of *intervals*. Intervals in each space form linearly ordered abelian groups under addition, and can act on points as a mode of transformation. This representation is close to the mathematical notion of an affine space and forms the basis of the generalised interval system of Lewin (1987). In this section we present the operations, predicates and axioms for the pitch abstract data type. The time abstract data type specification is taken to be the same modulo the renaming of the components (Wiggins et al., 1989; Harris et al., 1991).

Operations

Table 7.1 shows the types and operations for the pitch abstract data type. The specification includes the types Pitch and $\mathsf{Interval}$ for pitches and intervals, respectively, boolean comparison operations for equality and ordering on both pitches and intervals, the group operations on intervals, and operations diff_p and shift_p for calculating the interval between two pitches, and shifting a pitch by an interval, respectively. We use the subscripts $_p$ and $_i$ to indicate that these operations apply to pitches or intervals.

Table 7.1: Types and operations of the pitch abstract data type

Pitch	$: \mathsf{Set}$
$\mathsf{Interval}$	$: \mathsf{Set}$
eqb_p	$: \mathsf{Pitch} \to \mathsf{Pitch} \to \mathsf{bool}$
eqb_i	$: \mathsf{Interval} \to \mathsf{Interval} \to \mathsf{bool}$
lteb_p	$: \mathsf{Pitch} \to \mathsf{Pitch} \to \mathsf{bool}$
lteb_i	$: \mathsf{Interval} \to \mathsf{Interval} \to \mathsf{bool}$
add_i	$: \mathsf{Interval} \to \mathsf{Interval} \to \mathsf{Interval}$
inv_i	$: \mathsf{Interval} \to \mathsf{Interval}$
zero_i	$: \mathsf{Interval}$
diff_p	$: \mathsf{Pitch} \to \mathsf{Pitch} \to \mathsf{Interval}$
shift_p	$: \mathsf{Interval} \to \mathsf{Pitch} \to \mathsf{Pitch}$

Predicates

In addition to the operations, we require type-level predicates for capturing equations and inequalities between pitches and intervals. For this we require Prop-valued relations on Pitch and $\mathsf{Interval}$ for equality and ordering. Table 7.2 shows these relations. These predicates can be used to define other relations amongst pitches and intervals. For example, $\mathsf{gt}_p := \lambda p_1\ p_2.\neg\mathsf{lte}_p(p_1, p_2)$ is the 'greater than' relation on pitches, defined as the logical negation of 'less than or equal'.

For readability we use standard symbols for the equality and ordering relations of types with a subscript to indicate their type. For example, we write $p_1 =_p p_2$ as short-hand for $\mathsf{eq}_p(p_1, p_2)$, and $p_1 \leqslant_p p_2$ for $\mathsf{lte}_p(p_1, p_2)$.

Axioms

Table 7.3 shows the axioms that the operations and predicates must satisfy. The equality relations on pitches and intervals must be equivalence relations: they must be reflex-

Table 7.2: Predicates of the pitch abstract data type

$$
\begin{aligned}
\mathsf{eq}_p &: \mathsf{Pitch} \to \mathsf{Pitch} \to \mathsf{Prop} \\
\mathsf{eq}_i &: \mathsf{Interval} \to \mathsf{Interval} \to \mathsf{Prop} \\
\mathsf{lte}_p &: \mathsf{Pitch} \to \mathsf{Pitch} \to \mathsf{Prop} \\
\mathsf{lte}_i &: \mathsf{Interval} \to \mathsf{Interval} \to \mathsf{Prop}
\end{aligned}
$$

ive, symmetric and transitive (axioms 7.3.1 to 7.3.6). The ordering relations on pitches and intervals must be total orders: they must be reflexive, transitive and antisymmetric (axioms 7.3.7 to 7.3.12). In addition we require that the boolean comparison operations be compatible with type-level relations (axioms 7.3.13 to 7.3.16). This ensures that ordering relations are total and that equality is decidable. The operations on intervals must satisfy the axioms of an abelian group: the add_i operation must be commutative and associative; the constant zero_i must be the unit for the add_i operation; and the inv_i operation must return the inverse interval (axioms 7.3.17 to 7.3.20). Finally, we require that the diff_p and shift_p operations behave as expected with respect to the group structure of intervals (axioms 7.3.21 to 7.3.24).

7.2.2 Note-Like Constituents

Armed with these abstract data types for pitch and time, we can define constituent specifications in the specification logic which capture note-like constituent objects. We assume that the type AN is inhabited by three constants *Onset*, *Duration* and *Pitch*, with $\mathsf{typ}(\mathit{Onset}) = \mathsf{Time}$, $\mathsf{typ}(\mathit{Duration}) = \mathsf{TimeInterval}$ and $\mathsf{typ}(\mathit{Pitch}) = \mathsf{Pitch}$. We can now define constituent specifications which describe constituents which have these particular attributes as follows:

$$
\begin{aligned}
\mathsf{Temporal} : \mathsf{CSPEC} &:= \langle \mathit{Onset} \rangle [\wedge]_{\mathrm{C}} \langle \mathit{Duration} \rangle \\
\mathsf{Note} : \mathsf{CSPEC} &:= \mathsf{Temporal} [\wedge]_{\mathrm{C}} \langle \mathit{Pitch} \rangle .
\end{aligned}
$$

$\mathsf{Temporal}$ captures constituents with onset and duration attributes, while Note cap-

Table 7.3: Axioms for the pitch abstract data type

$$\mathsf{eq}_p\_\_\mathsf{reflexive} : \forall p.p =_p p \tag{7.3.1}$$
$$\mathsf{eq}_p\_\_\mathsf{symmetric} : \forall p\ q.p =_p q \to q =_p p \tag{7.3.2}$$
$$\mathsf{eq}_p\_\_\mathsf{transitive} : \forall p\ q\ r.p =_p q \to q =_p r \to p =_p r \tag{7.3.3}$$

$$\mathsf{eq}_i\_\_\mathsf{reflexive} : \forall i.i =_i i \tag{7.3.4}$$
$$\mathsf{eq}_i\_\_\mathsf{symmetric} : \forall i\ j.i =_i j \to j =_i i \tag{7.3.5}$$
$$\mathsf{eq}_i\_\_\mathsf{transitive} : \forall i\ j\ k.i =_i j \to j =_i k \to i =_i k \tag{7.3.6}$$

$$\mathsf{lte}_p\_\_\mathsf{reflexive} : \forall p.p \leqslant_p p \tag{7.3.7}$$
$$\mathsf{lte}_p\_\_\mathsf{transitive} : \forall p\ q\ r.p \leqslant_p q \to q \leqslant_p r \to p \leqslant_p r \tag{7.3.8}$$
$$\mathsf{lte}_p\_\_\mathsf{antisymmetric} : \forall p\ q.p \leqslant_p q \to q \leqslant_p p \to p =_p q \tag{7.3.9}$$

$$\mathsf{lte}_i\_\_\mathsf{reflexive} : \forall i.i \leqslant_i i \tag{7.3.10}$$
$$\mathsf{lte}_i\_\_\mathsf{transitive} : \forall i\ j\ k.i \leqslant_i j \to j \leqslant_i k \to i \leqslant_i k \tag{7.3.11}$$
$$\mathsf{lte}_i\_\_\mathsf{antisymmetric} : \forall i\ j.i \leqslant_i j \to j \leqslant_i i \to i =_i j \tag{7.3.12}$$

$$\mathsf{eqb}_p\_\_\mathsf{compat} : \forall x\ y.x =_p y \leftrightarrow \mathsf{eqb}_p(x, y) = \mathsf{true} \tag{7.3.13}$$
$$\mathsf{eqb}_i\_\_\mathsf{compat} : \forall x\ y.x =_i y \leftrightarrow \mathsf{eqb}_i(x, y) = \mathsf{true} \tag{7.3.14}$$
$$\mathsf{lteb}_p\_\_\mathsf{compat} : \forall x\ y.x \leqslant_p y \leftrightarrow \mathsf{lteb}_p(x, y) = \mathsf{true} \tag{7.3.15}$$
$$\mathsf{lteb}_i\_\_\mathsf{compat} : \forall x\ y.x \leqslant_i y \leftrightarrow \mathsf{lteb}_i(x, y) = \mathsf{true} \tag{7.3.16}$$

$$\mathsf{add}_i\_\_\mathsf{commutative} : \forall x\ y.\mathsf{add}_i(x, y) = \mathsf{add}_i(y, x) \tag{7.3.17}$$
$$\mathsf{add}_i\_\_\mathsf{associative} : \forall x\ y\ z.\mathsf{add}_i(x, \mathsf{add}_i(y, z)) = \mathsf{add}_i(\mathsf{add}_i(x, y), z) \tag{7.3.18}$$
$$\mathsf{zero}_i\_\_\mathsf{identity} : \forall x.\mathsf{add}_i(x, \mathsf{zero}_i) = x \tag{7.3.19}$$
$$\mathsf{inv}_i\_\_\mathsf{inverse} : \forall x.\mathsf{add}_i(x, \mathsf{inv}_i(x)) = \mathsf{zero}_i \tag{7.3.20}$$

$$\mathsf{shift}_p\_\_\mathsf{identity} : \forall p.\mathsf{shift}_p(\mathsf{zero}_i, p) = p \tag{7.3.21}$$
$$\mathsf{shift}_p\_\_\mathsf{associativity} : \forall i_1\ i_2\ p.\mathsf{shift}_p(i_2, (\mathsf{shift}_p(i_1, x)) = \mathsf{shift}_p(\mathsf{add}_i(i_1, i_2), p) \tag{7.3.22}$$
$$\mathsf{weyl}_p\_\_1 : \forall x\ y.\mathsf{shift}_p(\mathsf{diff}_p(x, y), x) = y \tag{7.3.23}$$
$$\mathsf{weyl}_p\_\_2 : \forall x\ y\ z.\mathsf{add}_i(\mathsf{diff}_p(a, b), \mathsf{diff}_p(b, c)) = \mathsf{diff}_p(a, c) \tag{7.3.24}$$

tures pitched temporal entities.

We can also define more expressive structure invariants as SPROP-valued relations between constituent identifiers and attribute values as follows:

$$\begin{aligned}\mathsf{event}(x,t,d) : \mathsf{SPROP} :=& x \overset{Onset}{\mapsto} t [\wedge] x \overset{Duration}{\mapsto} d\\ \mathsf{note}(x,t,d,p) : \mathsf{SPROP} :=& \mathsf{event}(x,t,d)[\wedge] x \overset{Pitch}{\mapsto} p\end{aligned}$$

This kind of representation is close to the logical relation representation suggested in Harris et al. (1991). The difference is that the expressions capture structure invariants whose truth is associated with a constructive proof, rather than being asserted as a logical axiom into a knowledge base.

7.2.3 Temporal Relations

We can use these invariants to capture the basic temporal relations defined by Allen (1984). We begin by defining an operation which constructs a period relation from four point relations, which specify the pairwise order between onsets and end points of the participating events. The operation is defined as follow:

$$\begin{aligned}\mathsf{TempRel}(a,b,c,d) : \mathsf{CREL} := \lambda x\ y. [\exists] t_1\ t_2\ d_1\ d_2.&\\ &\mathsf{event}(x,t_1,d_1)[\wedge]\mathsf{event}(y,t_2,d_2)[\wedge]\\ &[\ a(t_1,t_2)\wedge\\ &\ \ b(t_1,\mathsf{shift_t}(d_2,t_2))\wedge\\ &\ \ c(\mathsf{shift_t}(d_1,t_1),t_2)\wedge\\ &\ \ d(\mathsf{shift_t}(d_1,t_1),\mathsf{shift_t}(d_2,t_2))\ ],\end{aligned}$$

where a, b, c and d are of type TimePoint $\rightarrow$ TimePoint $\rightarrow$ Prop. The TempRel operation can be used to construct the period relations of Allen (1984) in a manner akin to the 'point notation' used by Marsden (2000, p. 59), as follows:

$$\begin{aligned}
\mathsf{equal} &:= \mathsf{TempRel}(=_t, <_t, >_t, =_t)\\
\mathsf{precedes} &:= \mathsf{TempRel}(<_t, <_t, <_t, <_t)\\
\mathsf{meets} &:= \mathsf{TempRel}(<_t, <_t, =_t, <_t)\\
\mathsf{overlaps} &:= \mathsf{TempRel}(<_t, <_t, >_t, <_t)\\
\mathsf{starts} &:= \mathsf{TempRel}(=_t, <_t, >_t, <_t)\\
\mathsf{during} &:= \mathsf{TempRel}(>_t, <_t, >_t, <_t)\\
\mathsf{finishes} &:= \mathsf{TempRel}(>_t, <_t, >_t, =_t)
\end{aligned}$$

The advantage of having the relations defined in this way, is that we can easily construct new relations, for example, a relation which captures any non-overlapping pair of events symmetrically.

7.2.4 Temporal Structures

We now demonstrate how the temporal relations can be used to define various higher-level temporal structures. We define three such kinds of structure: *sequence*, *stream* and *chain*. A sequence is a temporal collection whose particles are totally ordered by their onset attributes. A stream is a sequence which, in addition, contains no particles which overlap in time. Finally, a chain is a stream in which the end time of each particle is equal to the onset of the next. These specifications are defined as follows:

$$\begin{aligned}
\mathsf{Sequence} : \mathsf{CSPEC} :=& \mathsf{TemporalCollection}[\wedge]\\
& \forall_{\text{Parts}} p_1 .\\
& [\neg]\exists_{\text{Part}} p_2 .\\
& [p_1 \neq p_2][\wedge]\mathsf{starts}(p_1, p_2)\\
\mathsf{Stream} : \mathsf{CSPEC} :=& \mathsf{Sequence}[\wedge]\\
& \forall_{\text{Parts}} p_1 .\\
& [\neg]\exists_{\text{Part}} p_2 .\\
& [p_1 \neq p_2][\wedge]\mathsf{overlaps}(p_1, p_2)\\
\mathsf{Chain} : \mathsf{CSPEC} :=& \mathsf{Stream}[\wedge]\\
& \forall_{\text{Parts}} p_1\ p_2 . \mathsf{precedes}(p_1, p_2)[\rightarrow]\\
& \exists_{\text{Part}} p_3\ p_4 . \mathsf{meets}(p_1, p_3)[\wedge]\mathsf{meets}(p_4, p_2)
\end{aligned}$$

Here, we have demonstrated how various abstractions of musical objects can be described formally using the specification logic, in conjunction with appropriate specifications of abstract data types. In particular, the Stream specification is logically equivalent to the definition given in Harris et al. (1991, p. 9). The use of these different abstractions will depend on the particular context. Many representations of music explicitly represent chains as syntactic sequences of symbols or specific data structures (Cuthbert and Ariza, 2010), whist streams are often required for particular analysis methods (Harris et al., 1991).

7.3 Representing Extrinsic Properties of Constituents

There exist properties of constituents which may not be defined in terms of invariants over structures. Rather, they are true by fiat and assigned by the user of the representation. For accommodating this kind of property, we introduce the type $\mathsf{EP} : \mathsf{Set}$ of *extrinsic properties*. This type is a parameter to the existing representation framework. Elements of the type $e : \mathsf{EP}$ are symbolic labels which can be explicitly attached to constituent objects. Constituents may be annotated with multiple such properties. We do not define any operations on the type EP except equality checking.

Constituent object are associated with extrinsic properties by extending the constituent object interface (Table 5.1) to include operations for adding and retrieving labels. Table 7.4 gives these operations. $\mathsf{add\_ep}$ adds an extrinsic property label to the constituent object, while $\mathsf{get\_ep}$ retrieves the set of extrinsic properties which are associated with a constituent object.

Table 7.4: Operations for extrinsic properties

$$\mathsf{add\_ep} : \mathsf{EP} \to \mathsf{COBJ} \to \mathsf{COBJ}$$
$$\mathsf{get\_ep} : \mathsf{COBJ} \to \mathsf{fset}(\mathsf{EP}).$$

Table 7.5 gives the axioms which these operations must satisfy. $\mathsf{get\_ep\_delimit}$ asserts that a newly delimited set of particles has no assigned extrinsic properties. $\mathsf{get\_ep\_add\_ep}$ asserts that adding an extrinsic property e to a constituent object c, adds the value e to the set of properties $\mathsf{get\_ep}(c)$. $\mathsf{get\_ep\_set}$ asserts that modifying a constituents attributes does not affect the set of extrinsic property labels associated with it.

Table 7.5: Axioms for extrinsic properties

$$\begin{aligned}
\mathsf{get\_ep\_delimit} &: \forall ps.\ \mathsf{get\_ep}(\mathsf{delimit}(ps)) = \emptyset \\
\mathsf{get\_ep\_add\_ep} &: \forall e\ c.\ \mathsf{get\_ep}(\mathsf{add\_ep}(e, c)) = \mathsf{get\_ep}(c) \cup \{e\} \\
\mathsf{get\_ep\_set} &: \forall a\ v\ c.\ \mathsf{get\_ep}(\mathsf{set}(a, v, c)) = \mathsf{get\_ep}(c)
\end{aligned}$$

Extrinsic properties can be incorporated into the description language by adding the following definition of an atomic structure invariant:

$$x =_{\text{def}} e : \mathsf{SPROP} := \lambda s.\exists c.\mathsf{lookup}(x, s) = \mathsf{Some}(c) \wedge e \in \mathsf{get\_ep}(c)$$

This invariant asserts that the constituent named x is labelled with the extrinsic property e. We can now incorporate information about the explicit labelling of extrinsic properties of constituents with the structural specifications of their intrinsic properties.

7.4 Explicit Annotation of Musical Surface

Here, we propose an extension to the representation which introduces the notion a musical surface. Wiggins (2000) suggests that explicit representation of musical surface allows collections of kindred entities to be grouped together to identify a common basis for analysis. In this section we propose such a mechanism and suggest how it might be used to support the kinds surface-based reasoning suggested by Wiggins (2000).

We start by introducing the abstract type $\mathsf{MS} : \mathsf{Set}$ of musical surfaces as a parameter to the representation system. Elements of the type $m : \mathsf{MS}$ are symbolic labels which can be explicitly attached to constituent objects. As such, the operations and axioms for musical surfaces follow the exact pattern as those of extrinsic properties, with operations for adding and retrieving surface annotations, and axioms to ensure the operations behave as expected.

Table 7.6 defines three atomic specifications for musical surfaces. $\mathsf{AtSurface}(x, m) : \mathsf{SPROP}$ asserts that the constituent named x is labelled with the musical surface m. The constituent specification $\mathsf{At}(m) : \mathsf{CSPEC}$ asserts that a constituent is labelled with the musical surface m. The constituent specification $\mathsf{SurfaceStructure}(m) : \mathsf{CSPEC}$ asserts that all the particles of a constituent are at a particular musical surface m.

Table 7.6: Atomic specifications for musical surfaces

$$\begin{aligned}\mathsf{AtSurface}(x, m) : \mathsf{SPROP} &:= \lambda s.\exists c.\mathsf{lookup}(x, s) = \mathsf{Some}(c) \land m \in \mathsf{get\_ms}(c)\\ \mathsf{At}(m) : \mathsf{CSPEC} &:= \lambda x.\mathsf{AtSurface}(x, m)\\ \mathsf{SurfaceStructure}(m) : \mathsf{CSPEC} &:= \forall_{\text{Parts}}\mathsf{At}(m)\end{aligned}$$

These extensions to the specification language allow for two additional representational features to be captured that allow for more sophisticated reasoning with musical surfaces. Firstly, the type of musical surfaces may have more structure than simply discrete elements. For example, there may exist a relation on musical surfaces which captures how entities on different surfaces can be hierarchically related. Given such a relation $< : \mathsf{MS} \to \mathsf{MS} \to \mathsf{Prop}$ as a parameter, we can construct a specification that captures when a constituent adheres to the specific relation on musical surfaces, as follows:

$$\mathsf{At}(m)[\land]\mathsf{SurfaceStructure}(m')[\land][m < m'].$$

Secondly, musical surfaces may be explicitly associated with specific attribute types which capture how the surface relates to musical spaces. Given an operation $\mathsf{SAtts} : \mathsf{MS} \to \mathsf{fset}(\mathsf{AN})$ mapping musical surfaces to finite sets of attribute names, we can construct a specification which captures when a constituent realises this interpretation of musical surface, as follows:

$$\mathsf{At}(m)[\wedge][\forall]a.a \in \mathsf{SAtts}(m)[\rightarrow]\langle a\rangle(x).$$

Specifications such as these can be combined to describe sophisticated music information structures, distributed across different explicitly defined musical surfaces.

7.5 Representing The Ontological Domain of Constituents

As discussed in Chapter 4, the notion of ontological domain of musical objects is a useful tool for sharpening the discourse of music. In this section we propose a representation of ontological domain of constituents which allows for the generation of specifications which capture user defined ontological axioms.

We start by introducing the abstract type $\mathsf{OD} : \mathsf{Set}$ of ontological domains. Elements of this type $o : \mathsf{OD}$ are symbolic labels which can be used to explicitly label constituent objects. Again, the requisite extension to the constituent object interface follows the exact same as for extrinsic properties and musical surfaces.

Table 7.7 defines three atomic specifications which capture the description of ontological domain, and follow the exact pattern as for musical surfaces.

Table 7.7: Atomic specifications for ontological domains

$$\begin{aligned}
\mathsf{InDomain}(x, o) : \mathsf{SPROP} &:= \lambda s.\exists c.\mathsf{lookup}(x, s) = \mathsf{Some}(c) \wedge o \in \mathsf{get\_od}(c)\\
\mathsf{In}(o) : \mathsf{CSPEC} &:= \lambda x.\mathsf{InDomain}(x, o)\\
\mathsf{DomainStructure}(o) : \mathsf{CSPEC} &:= \forall_{\text{Parts}}\mathsf{In}(o)
\end{aligned}$$

We can use these language extensions to incorporate specific kinds of constituent properties. In particular, it may be useful to capture a constraint which asserts that hierarchically related constituents inhabit the same domain. This can be captured by the following specification:

$$\mathsf{DomainStructure}(o)[\wedge]\mathsf{In}(o).$$

7.6 Representing Associations Between Constituents

As well as representing multiple hierarchies of discrete constituent entities, we would also like to be able to annotate these structures with arbitrary connections between nodes in the hierarchy. Here we describe a simple extension to our framework which introduces a new kind of information object: the *constituent association*. A constituent association is a directed connection between constituents, which can be described by attributes and intrinsic properties in exactly the same way as constituents themselves.

7.6.1 Association Objects

We introduce the types $\mathsf{UAI} : \mathsf{Set}$ and $\mathsf{AOBJ} : \mathsf{Set}$ of universal association identifiers and association information objects, respectively. Elements $l : \mathsf{UAI}$ are symbolic names which are used to uniquely refer to elements $q : \mathsf{AOBJ}$. These types are the exact analogue of those for constituents.

Operations

Table 7.8 shows the operations defined on constituent association objects. The constructor assoc takes a pair of constituent identifiers and returns an association object between them. The operations source and target extract the source and target constituent identifiers respectively, and the get and set operations are the analogue of those for constituent objects from Table 5.1. Like constituent objects, the specification for association objects is parametric in the types AN and AT, and the function typ.

Table 7.8: Association object operations

$$\begin{aligned}
\mathsf{assoc} &: \mathsf{UCI} \to \mathsf{UCI} \to \mathsf{AOBJ} \\
\mathsf{source} &: \mathsf{AOBJ} \to \mathsf{UCI} \\
\mathsf{target} &: \mathsf{AOBJ} \to \mathsf{UCI} \\
\mathsf{get} &: \Pi a : \mathsf{AN}.\mathsf{AOBJ} \to \mathsf{option}(\mathsf{typ}(a)) \\
\mathsf{set} &: \Pi a : \mathsf{AN}.\mathsf{typ}(a) \to \mathsf{AOBJ} \to \mathsf{AOBJ}
\end{aligned}$$

Axioms

Table 7.9 shows the axioms which the operations of Table 7.8 must satisfy. source_assoc and target_assoc assert that the operations source and target behave as expected with respect to the constructor assoc. The other three axioms are the analogues of those for constituent objects from Table 5.2.

Table 7.9: Association object axioms

$$\begin{aligned}
\mathsf{source\_assoc} &: \forall x\ y.\ \mathsf{source}(\mathsf{assoc}(x, y)) = x \\
\mathsf{target\_assoc} &: \forall x\ y.\ \mathsf{target}(\mathsf{assoc}(x, y)) = y \\
\mathsf{get\_assoc} &: \forall x\ y\ a.\ \mathsf{get}(a, \mathsf{assoc}(x, y)) = \mathsf{None} \\
\mathsf{get\_set\_same} &: \forall a\ a'\ v\ q.\ a = a' \to \mathsf{get}(a, \mathsf{set}(a', v, q)) = \mathsf{Some}(v) \\
\mathsf{get\_set\_other} &: \forall a\ a'\ v\ q.\ a \neq a' \to \mathsf{get}(a, \mathsf{set}(a', v, q)) = \mathsf{get}(a', q)
\end{aligned}$$

7.6.2 Constituent Structures with Associations

An extension to the constituent structure interface (Table 5.3) to include operations for inserting and retrieving associations would follows the exact pattern as for constituents. We must also include an operation analogous to the domain operation which returns the finite set of association identifiers bound in the structure, as well as axioms which which ensure that constituent and association operations behave together as expected.

With this extended constituent structure interface we can extend our description language with two additional atomic structure invariants involving associations. Table

7.10 shows the definitions of these invariants. $x \overset{l}{\Rightarrow} y$ asserts that association l is a connection between constituents x and y. $l \overset{a}{\mapsto} v$ asserts that the association has a value v for the attribute named a.

Table 7.10: Atomic structure invariants for associations

$$x \overset{l}{\Rightarrow} y : \mathsf{SPROP} := \lambda s.\exists q.\mathsf{lookup}(l, s) = \mathsf{Some}(q) \wedge \mathsf{source}(q) = x \wedge \mathsf{target}(q) = y$$
$$l \overset{a}{\mapsto} v : \mathsf{SPROP} := \lambda s.\exists q.\mathsf{lookup}(l, s) = \mathsf{Some}(q) \wedge \mathsf{get}(a, q) = \mathsf{Some}(v)$$

7.6.3 Association Specifications

Logical specifications for associations can be defined in the same way as for constituents. $\mathsf{ASPEC} := \mathsf{UAI} \rightarrow \mathsf{SPROP}$ is the type of association specifications. As we have already developed adequate machinery for specifying relations between constituent (§6.3.4), we need only define an operation which constructs association specifications from constituent relations $R : \mathsf{CREL}$. We define this operation as follows:

$$\mathsf{Assoc} : \mathsf{CREL} \rightarrow \mathsf{ASPEC} := \lambda R\ l.[\exists]x\ y.x \overset{l}{\Rightarrow} y[\wedge]R(x, y)$$

Composition of association specifications can be done by pointwise lifting of the structure invariant connectives, using the subscript $_{\text{A}}$ to indicate the type.

7.6.4 Summary

Associations allow us to reify connections or latent relationships that exist between constituents and attach information to them. This is a highly general idea which has many practical representational consequences. We briefly mention two here. Firstly, as with constituents, the representation of associations could be extended to support other types of information such as extrinsic properties, musical surfaces or ontological domains. In particular we might want to identify a specific class of association which represents a connection between different ontological domains, such as the co-reference relationship posited in Wiggins (2000). Secondly, the representation of associations as a distinguished type of information object connecting other objects, could be extended to higher-order; we could, for example, define a type of secondary association which connects (primary) associations. This general idea underpins certain meta-theoretical approaches to musical structure such as Ockelford (2005).

7.7 Representing Structured Objects: Layouts

Our final extension to the framework demonstrates how new logical connectives can be defined to enable more concise descriptions of structures. Layouts capture the type of predicates over finite sets constituent identifiers. The principal layout connective is a conjunction operation which assures that each of the conjuncts holds for disjoint parts of the set, similar to the separating conjunction from separation logic (Reynolds, 2002).

For example, there exist certain kinds of musical structure which consist of a finite number of particle elements lying in specific relation to one another. Consider the definition of *dactyl* from Harris et al. (1991). A specification for this structure in our description language might be as follows:

$$\begin{aligned}\mathsf{dactyl} : \mathsf{CSPEC} := \exists_{\text{Part}}\ & p_1\ p_2\ p_3. \\ & [\ \mathsf{meets}(p_1, p_2)\ [\wedge] \\ & \quad \mathsf{meets}(p_2, p_3)\ [\wedge] \\ & \quad \mathsf{AttRel}(\mathit{Duration}, >_d, p_1, p_2)\ [\wedge] \\ & \quad \mathsf{AttRel}(\mathit{Duration}, =_d, p_2, p_3)\ ]\ [\wedge]_{\text{C}} \\ & \quad \forall_{\text{Parts}}\ p_4.\ [p_4 = p_1 \vee p_4 = p_2 \vee p_4 = p_3\ ]\end{aligned}$$

This definition uses the attribute name *Duration*, the relations $>_d$ and $=_d$ on $\mathsf{typ}(\mathit{Duration})$, and the constituent relation $\mathsf{meets} : \mathsf{CREL}$, defined in §7.2.

Specifications like this do not easily scale, as given n existentially quantified particles we must universally quantify over an $(n + 1)$th and assert the disjunction of $\binom{n}{2}$

equalities. In other words a specification for some hex-chord would need to include 15 equality assertions. This problem occurs when quantification allows for the aliasing of logical variables which range over the type of identifiers. To avoid this we introduce the type $\mathsf{Layout} := \mathsf{fset}(\mathsf{UCI}) \to \mathsf{SPROP}$ of SPROP-valued predicates over finite sets of constituent identifiers. Table 7.11 defines three layout constructors. Emp asserts that a set of identifiers is empty. $x <: \Phi$ asserts that a set of identifiers contains only one identifier x which satisfies the specification Φ. $L_1 * L_2$ is the layout conjunction operation, which asserts that a set of identifiers u can be split in to two disjoint parts u_1 and u_2 which satisfy L_1 and L_2 respectively. The predicate $\mathsf{splits}(u_1, u_2, u) := u_1 \cap u_2 = \varnothing \wedge u_1 \cup u_2 = u$ captures the notion of splitting of a finite set. The other logical operations on layouts can be defined via pointwise lifting, and use the subscript $_{\mathrm{L}}$.

Table 7.11: Layouts constructors

$$\begin{aligned}
\mathsf{Emp} : \mathsf{Layout} &:= \lambda u.[u = \varnothing] \\
x <: \Phi : \mathsf{Layout} &:= \lambda u.[u = \{x\}][\wedge]\Phi(x) \\
L_1 * L_2 : \mathsf{Layout} &:= \lambda u.[\exists] u_1\ u_2.[\mathsf{splits}(u_1, u_2, u)][\wedge] L_1(u_1)[\wedge] L_2(u_2)
\end{aligned}$$

Finally we need an operation which turns a layout into a constituent specification. For this we define $\{\mathrm{L}\} : \mathsf{CSPEC} := \lambda x\ s.\exists c.\mathsf{lookup}(x, s) = \mathsf{Some}(c) \wedge L(\mathsf{particles}(c), s)$ which asserts that the layout L holds of the particles of a constituent x. We can now rewrite the specification of dactyl from above in a more concise form using layouts:

$$\begin{aligned}
\mathsf{dactyl} : \mathsf{CSPEC} := [\exists]\ & p_1\ p_2\ p_3. \\
& \{\ p_1 <: [\top] \\
& * p_2 <: \mathsf{meets}(p1)[\wedge]\ \mathsf{AttRel}(\mathit{Duration}, >_d, p1) \\
& * p_3 <: \mathsf{meets}(p2)[\wedge]\ \mathsf{AttRel}(\mathit{Duration}, =_d, p2)\ \}
\end{aligned}$$

7.8 Summary and Conclusions

In this section we have outlined six extensions to the constituent structure representation from Chapter 5 and constituent specification logic from Chapter 6. These extensions were specifically motivated by the requirements identified in §4.2 and demonstrate how the constituent structure representation and specification logic provide a strong basis for developing more precise representational features. The features suggested in this sections are not to be viewed as proposals for a model of musical ontology, but rather as a demonstration of the flexibility and power of the type-based framework. In particular, specifications of abstract data types in CIC generalise the algebraic theories of Ologs (Spivak and Kent, 2012), the dimensions of conceptual spaces (Gärdenfors, 2000), and the data types of many other data types of many other formats such as XML.

Formally specifying every aspect of the representation in CIC requires a considerable level of commitment from the user of the framework. It results in a rather gradual ascent from basic concepts to more sophisticated musical descriptions. However, we argue that to achieve a state of affairs where computer systems can share and automatically reason with complex musicological descriptions, a high degree of precision is required from the formal specifications. The type-based framework presented in this part provides such a level of precision, as well as a strong logical and computation basis for the design of music representations to suit a wide variety of applications. In the Part III we present an implementation of the framework which allows users to leverage the expressive power and precision of the type-based representation method using mainstream tools and technologies.

Part III

An Implementation of the Framework

Summary

In this part, we present an implementation of the type-based framework presented in Part II, and use it to demonstrate the utility of the constituent structure representation and specification logic. The implementation consists of three Semantic Web ontologies, which capture the core structural features of the representation, and three JavaScript libraries, which implement the core functionality of the framework. The part is divided into three chapters. Chapter 8 describes the Semantic Web ontologies, each of which captures a different aspect of the representation framework. For each, we give an overview of the model, describe the core classes and relations, and give an example of how it is used. Chapter 9 describes the JavaScript libraries. In particular, it describes how they implement the functionality of the representation framework, including interfacing with linked data structured by the ontologies. Chapter 10 provides an illustration of how the representation framework and implementation support three different music analysis tasks, including pattern search and discovery, paradigmatic analysis and hierarchical set-class analysis. In particular, it describes how the constituent structure model supports representation of both the inputs and outputs of these analyses. Finally, we present an demonstrator web application, built using the JavaScript tools, which performs paradigmatic analysis and visualisation of linked data documents structured by the ontologies presented in Chapter 8.

Chapter 8

Semantic Web Ontologies

8.1 Introduction

In this chapter we describe a number OWL ontologies for representing constituent structures and their specifications on the Semantic Web. The purpose of these ontologies is to provide linked data formats which capture the fundamental structural components of the representation in a way which can be used by systems which implement the framework.

Why Semantic Web?

Our motivation for designing linked data formats for our representation is the desire to integrate our representation framework with existing methods of music representation on the semantic web. The ever increasing popularity of the Semantic Web as a platform for knowledge representation in many research fields means that new representations will be maximally useful if they can be integrated with this growing body of information.

Chapter Outline

§8.2 describes the *Constituent Structure Ontology* which provides a linked data format for the constituent structure representation presented in Chapter 5. §8.3 describes the

Dependent Type Ontology which provides a linked data format for the specifications abstract data types, such as those described in §7.2. §8.4 describes the *Constituent Structure Specification Ontology* which provides linked data formats for capturing expressions of the specification logic presented in Chapter 6.

8.2 The Constituent Structure Ontology

In this section we describe the *Constituent Structure Ontology* (prefix: `cso`). The ontology contains OWL classes and relations which capture the constituent structure representation presented in Chapter 5. The purpose of the ontology is to provide a linked data format which captures the structural properties of the constituent structure representation, presented in §5.4, which is compatible with implementations of the functional model, presented in §5.5.

8.2.1 Core Classes and Relations

The core classes and relations of the ontology are shown in Figure 8.1. The `cso:Constituent` class captures the constituent musical objects of the representation. The identity of musical objects is associated with the resource identifier used to represent them. Constituents are associated with their particles via the `cso:hasPart` property, with inverse defined as `cso:isPartOf`. Constituents are associated with attributes and properties via the `cso:hasAttribute` and `cso:hasProperty` object properties, respectively. Attributes are arbitrary resources constructed to reify the connection between constituents and points in musical spaces, while properties are resources which represent formal properties of constituents. We define two subclasses of `cso:Property`: `cso:IntrinsicProperty` and `cso:ExtrinsicProperty`. Intrinsic properties represent properties of constituents which require checking, such as constituent specifications, while extrinsic properties are true by definition. An attribute is associated with an `cso:AttributeName` and an `cso:AttributeValue` via the functional object properties

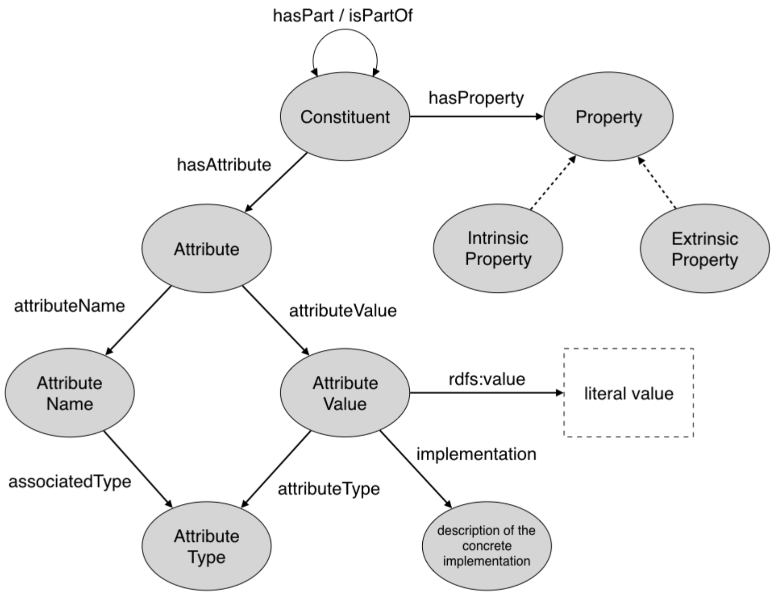


Figure 8.1: The core classes and relations of the Constituent Structure Ontology. Grey ovals indicate OWL classes, solid arrow indicate OWL object properties, dotted arrows indicate the subclass relation, and dotted rectangles represent RDF literals.

`cso:attributeName` and `cso:attributeValue`, respectively. Attribute names are resources which explicitly identify the particular aspect of the constituent described by the attribute. Attribute values are arbitrary resources constructed to represent points in musical spaces. Both attribute names and values are associated with instances of `cso:AttributeType` via the `cso:associatedType` and `ch:attributeType` functional object properties, respectively. Attribute types identify abstract data types which represent musical spaces. Attribute values can be associated with a literal value via the `rdfs:value` property as well as information about the concrete implementation of the type via the `cso:implementation` property.

8.2.2 Example

Here, we give an example of how the ontology can be used to represent a specific piece of musical material. The example is taken from Harris et al. (1991, p. 8) and uses the first four bars of Webern's Variations for Piano, Op 27, show in Figure 8.2. Figure 8.3 shows a linked data description of the same four bars structured

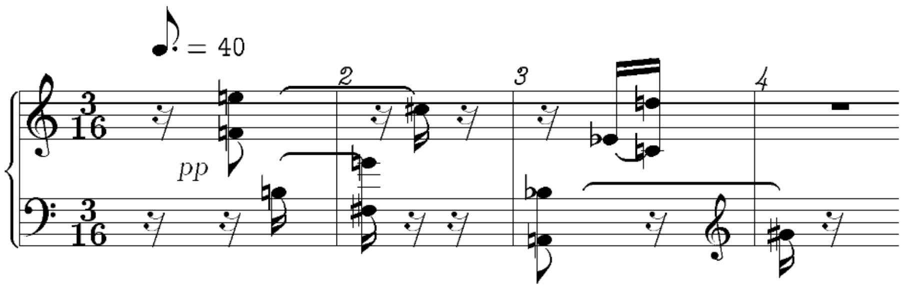


Figure 8.2: Webern, Op 27, bars 1-4.

```
{"@context": ":ctx.json"
 "@graph:"[
    {"@id": ":e00",
        "@type":"cso:Constituent",
        "cso:hasAttribute": [":e00p", ":e00o", ":e00d"]},
       {"@id": ":e01",
        "@type":"cso:Constituent",
        "cso:hasAttribute":[":e01p", ":e01o", ":e01d"]},
       {"@id": ":e02",
        "@type":"cso:Constituent",
        "cso:hasAttribute":[":e02p", ":e02o", ":e02d"]},
       ... ,
       {"@id": ":e11",
        "@type":"cso:Constituent",
        "cso:hasAttribute":[":e11p", ":e11o", ":e11d"]},
       ... ]}
```

Figure 8.3: The note events of Webern, Op 27, bars 1-4 represented using JSON-LD and structured using the Constituent Structure Ontology.

using the Constituent Structure Ontology. The concrete syntax used is JSON-LD, with '...' used to indicate elided material. Each note event is represented as an instance of `cso:Constituent` and associated with three, arbitrarily named, attributes via the `cso:hasAttribute` property. Figure 8.4 shows the pitch attribute, `:e00p`, of the first note of the excerpt, `:e00`. The attribute is linked with a name and value via the `cso:attributeName` and `cso:attributeValue` properties, respectively. The value is linked with an abstract type via the `cso:attributeType` property, and the `cso:implementation` property is used to indicate the concrete format used to encode the literal value, in this case `:string_pitch`. Figure 8.5 shows two examples of higher-level structural annotations, again taken from Harris et al. (1991, p. 13), which capture particular groupings of the basic musical material. The first, `:c00`, is linked with an instance of `cso:ExtrinsicProperty`, identifying that the constituent is a 'subject' of the piece. The second, `:c08`, is linked with an instance of `cso:IntrinsicProperty`, which indicates that the grouping is a 'triple'. Each constituent is linked with its particles using the `cso:hasPart` property.

```
...
{"@id": ":e00p",
 "@type": "cso:Attribute",
 "cso:attributeName": "an:pitch",
 "cso:attributeValue": {
     "@type": "cso:AttributeValue",
     "rdfs:value": "f4",
     "cso:attributeType": "adt:Pitch",
     "cso:implementation": "string_pitch" }},
...
```

Figure 8.4: An example of an attribute; the pitch attribute of the first note of Webern, Op 27, represented in JSON-LD and structured using the Constituent Structure Ontology.

```
...
{"@id": ":c00",
 "@type": "cso:Cosntituent",
 "cso:hasProperty": {
     "@id": ":subject",
     "@type": "cso:ExtrinsicProperty" },
 "cso:hasPart": [":e00", ":e01", ... , ":e11"]},
 ...
{"@id": ":c08",
 "@type": "cso:Constituent",
 "cso:hasProperty": {
     "@id": ":triple",
     "@type": "cso:IntrinsicProperty"},
 "cso:hasPart": [":e00", ":e01", ":e02"]},
 ...
```

Figure 8.5: Two examples of constituents which capture structural groupings represented in JSON-LD and structured using the Constituent Structure Ontology.

8.2.3 Extending the ontology in OWL

Simple extensions to the ontology can admit more domain-specific knowledge modelling. In this section we give examples of this by defining new OWL classes. For these examples we use the OWL functional syntax[1] to write class definitions.

Firstly, the `cso:AttributeValue` class can be extended to capture classes of attribute values of a particular abstract type. Below we define the class `:PitchValue`, where `:pitch` is a resource indicating the abstract type of pitch from §7.2.

```
EquivalentClasses(
  :PitchValue
  ObjectHasValue( cso:attributeType :pitch ))
```

The `cso:Attribute` class can be extended to capture attributes whose value is of a certain type. Below we define the class `:PitchAttribute` in this way, using the `ObjectSomeValuesFrom` class construction.

```
EquivalentClasses(
  :PitchAttribute
  ObjectSomeValuesFrom( cso:attributeValue :PitchValue ))
```

The `cso:hasAttribute` property can be extended to reflect the greater precision of these constituent-attribute relations. Below we define the object property `:hasPitch` which connects a `cso:Constituent` with a `:PitchAttribute`.

```
SubObjectPropertyOf( :hasPitch cso:hasAttribute )
ObjectPropertyRange( :hasPitch :PitchAttribute )
```

We can give similar treatment to onset and duration attributes. Now, the `cso:Constituent` class can be extended to define subclasses of constituents which have specific attributes. Below, we define the classes `:Event` and `:Note`. `:Event` classifies all those constituents that have onsets and durations while `:Note` classifies constituents which are events and have pitches.

[1]https://www.w3.org/TR/owl2-syntax/

```
EvivalentClasses(
  :Event
  ObjectIntersectionOf(
    ObjectSomeValuesFrom( :hasOnset :TimeAttribute )
    ObjectSomeValuesFrom( :hasDuratation :DurationAttribute )))

EvivalentClasses(
  :Note
  ObjectIntersectionOf(
    :Event
    ObjectSomeValuesFrom( :hasPitch :PitchAttribute )))
```

Finally, these classes can be used to define simple descriptions of higher-level structures. Below, we define the class of constituents that contain only notes as particles, called `:NoteCollection`.

```
EquivalentClasses(
  :NoteCollection
  ObjectIntersectionOf(
    ObjectHasSomeValuesFrom( cso:hasPart :Note )
    ObjectAllValuesFrom( :hasPart :Note )))
```

These example classes illustrate how the model captured by the Constituent Structure Ontology is sufficiently general that more specialised representations can be can be described through extension of the classes within the OWL language. These extensions can provide greater precision when representing specific types of musical knowledge, as well as greater support for OWL reasoners. The advantage of having a highly general, top-level ontology is that domain-specific extensions can be more easily integrated, and so are more conceptually interoperable. The Constituent Structure Ontology provides such an upper-level model, as well as the formal strength of the type-based framework.

8.2.4 Discussion

The Constituent Structure ontology provides a general purpose model for representing multiple-hierarchical musical structures on the Semantic Web and acts as an anchor point for more domain-specific knowledge representation. In addition, the ontology can been seen as generalising many approaches to music representation found in existing ontologies. For example, the `cso:AttributeValue` and `cso:AttributeType` classes can be seen as generalisations of the `tl:Instant`/`tl:Interval` and `tl:Timeline` classes from the Timeline Ontology (Abdallah et al., 2006). This generalisation views time lines as just a specific kind of 'type' or 'space' representation, with its inhabitants viewed as abstract 'values' or 'points'. In addition, the `cso:implementation` property can be seen as generalising the notion of namespace from the JAMS specification (Humphrey et al., 2014); any concrete representation of an attribute value can be explicitly linked to information about the specific way in which it is encoded. Finally, the `cso:Constituent` class can be seen as a generalisation of the `event:Event` class; events can be seen as a particular instances of constituents whose attributes include temporal information.

On its own, the ontology affords limited reasoning through the standard OWL reasoners. In particular, the axioms of the ontology do not fully constrain the typing inherent in the functional model of constituent structures given in §5.5. The ontology can be extended in OWL to admit more precise descriptions of musical structures. However, the expressive power of OWL is limited compared to that of the specification logic described in §6.

8.3 The Dependent Type Ontology

In this section we describe the *Dependent Type Ontology* (prefix: `dto`). The ontology contains classes and relations which capture the abstract syntax of the terms of an arbitrary pure type system (see §2.2), extended with classes which capture the sorts of CIC (Bertot and Castéran, 2004), and a number of specific classes which capture particular inductive definitions as syntactic extensions to the term language. The purpose of the ontology is to provide a linked data format for representing formal specifications of abstract data types, such as those presented in §7.2.

8.3.1 Core Classes and Relations

The core of the ontology is a single top class `dto:Term`, which captures expressions of the term language. Subclasses are used to distinguish the various syntactic classes of terms. For example, `dto:Variable`, `dto:Sort`, `dto:Lambda`, `dto:Pi` and `dto:Application` represent variables, sorts, lambda abstractions, dependent products and function applications, respectively. Object properties are defined which link instances of a particular subclass to its sub-terms. For example, the object property `dto:lambdaBinds` links an instance of `dto:Lambda` to an instance of `dto:Variable` representing the variable which is bound in the body of the lambda term. In this way, the ontology captures terms of dependent type theory as abstract syntax trees. In addition to the basic terms of dependent theory, the ontology includes additional subclasses of `dto:Term` which capture additional types such as dependent sums, simple types, such as non-dependent functions, binary products and coproducts, and logical formula. Each of these syntactic extensions can be interpreted as an inductive definition in CIC. Finally, the ontology includes a small vocabulary of terms for defining the *signatures* of abstract data types. For this vocabulary we use a distinguished prefix, `sig`. It allows the modular organisation of specification components, and includes the classes `sig:AtomicType`, `sig:Operation`, `sig:Predicate` and `sig:Axiom` for representing abstract atomic types, operations, predicates and axioms, respectively. These components can be associated with instances of the class `sig:Signature` via the properties `definesType`, `sig:definesOperation`, `sig:definesPredicate` and `sig:definesAxiom`, respectively. Components of specifications can be associated with a type specification via the property `sig:hasType`. The types of specification components are instances the class `dto:Term`.

8.3.2 Example

Figure 8.6 shows an example of how the ontology can be used to capture specifications of abstract data types. The example is written in JSON-LD and encodes the abstract data type specification for pitch, described in §7.2. `:PitchSig` is an instance of `sig:Signature`, and is associated with its components via the relevant properties. Figure 8.7 shows the `:pitch_shift` component of this specification; an encoding of the pitch_shift operation defined in Table 7.1. It is an instance of `sig:Operation` and is linked to its type specification via the `sig:hasType` property. The type is a linked data encoding of the term "Interval $\rightarrow$ Pitch $\rightarrow$ Pitch". It is an instance of `dto:Arrow`, a subclass of `dto:Term`, which captures the syntactic class of non-dependent functions type expressions. The properties `dto:domain` and `dto:codomain` associate instances of `dto:Arrow` with sub-expressions corresponding to their domain and codomain, respectively.

```
{"@id": ":PitchSig",
 "@type": "sig:Signature",
 "sig:definesType": [
    ":Pitch",
    ":Interval" ],
 "sig:definesOperation": [
    "pitch_eqb",
    "interval_eqb",
    ...
    ":pitch_diff",
    ":pitch_shift" ],
 "sig:definesPredicate": [
    "pitch_eq",
    "interval_eq",
    "pitch_lte",
    "interval_lte" ],
 "sig:definesAxiom": [
    ":pitch_total_order",
    ":interval_total_order",
    ...
    ":shift_associativity",
    ":shift_identity",
    ... ]}
```

Figure 8.6: A linked data description of the abstract data type specification for pitch, described in §7.2, represented using JSON-LD and structured using the Dependent Type Ontology.

```
{"@id": ":pitch_shift",
 "@type": "sig:Operation",
 "sig:hasType": {
    "@type": "dto:Arrow",
    "dto:domain": ":PitchInterval",
    "dto:codomain": {
        "@type": "dto:Arrow",
        "dto:domain": ":Pitch",
        "dto:codomain": "Pitch" }}}
```

Figure 8.7: A linked data description of the specification for the `pitch_shift` operation, described in §7.2, represented using JSON-LD and structured using the Dependent Type Ontology.

8.3.3 Discussion

The Dependent Type Ontology can be used to augment the Constituent Structure Ontology, presented in §8.2, by linking `cso:AttributeName`s and `cso:AttributeValue`s with the appropriate type information. Being able to represent algebraic specifications of data types on the semantic web is an important step in enabling the integration of web-based knowledge representation and programming languages. Music is one domain in which representations must support both reasoning applications and user-defined algorithms for specific analysis methods.

It is important to clarify that the axioms of the Dependent Type Ontology do not capture or constrain the well-typedness or convertability of terms. OWL reasoners may be used to detect syntactic errors in the linked data descriptions of expressions. However, the user must rely on additional external tools for type checking. Such a tool is presented in the following chapter.

A core advantage of the ontology, is that it allows for the modular extension of the term language via specific subclasses of `dto:Term`. Rather than encode the syntax of general inductive definitions, new type and term expressions are introduced as *syntactic* extensions to the language. It is important to notice, however, that the soundness of these extensions is not controlled by the axioms of the ontology. It is the responsibility of the user to ensure that any syntactic extensions defined as new subclasses of `dto:Term` constitute correct inductive definitions in CIC.

The approach of reifying logical formulae in Semantic Web languages is not entirely novel. For example, McDermott and Dou (2002) propose a system for embedding certain FOL formulae in RDF. In addition, the N3-Tr method (Raimond, 2008, see §4.4) of repressing music processing algorithms, embeds formulae of a concurrent transaction logic in RDF. However, no previous attempt has utilised the full expressive power and formal rigour of dependent type theory.

8.4 The Constituent Structure Specification Ontology

In this section we describe the *Constituent Structure Specification Ontology* (prefixes: `csso`, `sprop`, `cspec` and `crel`). The ontology contains classes and relations which capture the abstract syntax of the specification logic described in Chapter 6. The purpose of the ontology is to provide a linked data format for capturing logical specifications which can be linked to representations of constituent structures.

8.4.1 Core Classes and Relations

The core of the ontology are three disjoint subclasses of `dto:Term`: `csso:StructureInvariant` `csso:ConstituentSpecification` and `csso:ConstituentRelation`, which capture the syntactic classes of specifications of type SPROP, CSPEC and CREL, respectively. As in the Dependent Type Ontology, subclasses are used to capture specific syntactic classes of specifications, and specific object properties are used to link an expression to its sub-expressions. For example, the class `sprop:Conjunction` is the syntactic class of structure invariants of the form $\phi[\wedge]_{\mathrm{s}}\psi$ (see Table 6.2). The object properties `sprop:leftConjunct` and `sprop:rightConjunct` are used to connect these conjunction expressions with their conjuncts. In this way, the Constituent Structure Specification Ontology can be seen as a syntactic extension to the language captured by the Dependent Type Ontology, presented in §8.3, in accordance with the definitions given in Chapter 6.

8.4.2 Example

Figure 8.8 shows an example of how the ontology can be used to build linked data descriptions of constituent specifications. The example shows the Sequence constituent specification, defined in §7.2, represented in JSON-LD. `:Sequence` is an instance of the class `cspec:Conjunction`, and is associated with its left and right conjuncts via the properties `cspec:leftConjunct` and `cspec:rightConjunct`, respectively. Each nested expression is annotated with its syntactic class and linked with its sub-expressions via properties from the ontology. In addition, the specification makes reference to resources which represent the constituent specification `:TemporalCollection`, the equality relation on constituent identifiers `:uci_eq`, and the temporal constituent relation `:starts`, defined in §7.2.

8.4.3 Discussion

The Constituent Structure Specification Ontology captures the syntax of constituent structure specifications as defined in Chapter 6 by extending the Dependent Type Ontology. This allows instances of `cso:Constituent` to be linked with instances of `csso:ConstituentSpecification` via the `cso:hasProperty` property. In this way, the class `csso:ConstituentSpecification` is a subclass of `cso:IntrinsicProperty`. However, the axioms of the ontologies do capture whether such a statement is provable in the underlying type theory. This is a job for external tools which perform proof automation and checking. As with the Dependent Type Ontology, the axioms of the Constituent Structure Specification Ontology do not capture the well-typedness of expressions, but do make possible the detection of syntactic errors. Syntactic expressions represented using the ontology can become quite large, by comparison to typical

```
{"@id": ":Sequence",
 "@type": "cspec:Conjunction",
 "cspec:leftConjunct": ":TemporalCollection",
 "cspec:rightConjunct": {
    "@type": "cspec:ForallParts",
    "cspec:fpBinds": ":p1",
    "cspec:fpBody": {
        "@type": "cspec:Not",
        "cspec:notBody": {
            "@type": "cspec:ExistsPart",
            "cspec:epBinds": ":p2",
            "cspec:epBody": {
                "@type": "cspec:Conjunction",
                "cspec:leftConjunct": {
                    "@type": "cspec:LiftProp",
                    "cspec:liftProp": {
                        "@type:": "prop:Not",
                        "prop:notBody": {
                            "@type": "dto:Application",
                            "dto:appHead": {
                                "@type": "dto:Application",
                                "dto:appHead": ":uci_eq",
                                "dto:appParameter": ":p1" },
                            "dto:appParameter": ":p2" }}},
                "cspec:rightConjunct": {
                    "@type": "cspec:LiftSprop",
                    "cspec:liftSprop": {
                        "@type": "dto:Application",
                            "dto:appHead": {
                                "@type": "dto:Application",
                                "dto:appHead": ":starts",
                                "dto:appParameter": ":p1" },
                            "dto:appParameter": ":p2" }}}}}}}
```

Figure 8.8: The Sequence constituent specification, defined in §7.2, represented in JSON-LD and structured using the Constituent Structure Specification Ontology.

OWL class axioms. However, this is mitigated by the ability to incrementally construct specifications by reusing identifiers which represent existing descriptions.

8.5 Summary and Conclusions

The three ontologies described in this chapter together form the basis for a full Semantic Web realisation of the type-based framework, presented in Part II. The Constituent

Structure Ontology captures the core structure of the representation, while the Dependent Type and Constituent Structure Specification ontologies can be used to link constituents and attributes with rich type information. However, the ontologies do not capture the well-typedness of such descriptions. In order for the ontologies to be incorporated into an implementation of the framework, it is necessary to develop complementary tools which implement the functionality of the representation and perform the necessary type checking and proof automation. In the next chapter we describe work that has been done towards providing such tool support.

Chapter 9

JavaScript Tools

9.1 Introduction

In this chapter we describe a number of JavaScript modules which have been developed to support the use of the constituent structure representation and specification language defined in Part II. The purpose of the tools is two-fold. Firstly, they are intended to implement the functionality of the representation framework, and provide users with a programming interface for working with linked data structured by the ontologies presented in the previous chapter. Secondly, they are intended as a basis for the development of formally specified, interoperable software tools for computer music research.

Why JavaScript?

The tools are developed in JavaScript using Node.js. The choice of JavaScript as an implementation language was made for two reasons. Firstly, JavaScript is a mainstream language which is used by the majority of web developers. Its widespread use for building web applications makes it an obvious choice for the development of web-based tools for digital musicology which process linked data. Secondly, it is a highly flexible language. Although it apparently lacks many of the features necessary for implementing type-based specifications (specifically, static type checking), it has a highly flexible object mechanism, as well as higher-order functions. This flexibility makes it possible to incorporate typically functional features by designing data structures which contain run-time type information.

Chapter Outline

§9.2 describes a JavaScript implementation of the constituent structure representation, presented in Chapter 5. §9.3 describes JavaScript tools which enable the user to build and use specifications of abstract data types in conjunction with constituent structures. §9.4 describes an implementation of the specification logic, presented in Chapter 6, which allows specifications to be constructed and used to check the intrinsic properties of constituents.

9.2 An Implementation of the Constituent Structure Model

In this section we describe a JavaScript module, `cs.js`, which implements the constituent structure representation, presented in Chapter 5. The library includes and in-memory implementation of constituent structures as JavaScript objects, whose structure is controlled by the prototype mechanism.

The library defines a JavaScript class for each of the abstract types of the functional model, given in §5.5. The construction and manipulation of constituent objects and structures is controlled by methods which perform run-time type checking. Constituent identifiers are implemented using strings. The library includes both a functional and a imperative interface for manipulating constituent structures. That is, object's include methods which mutate the original object, as well as methods which return a new deep copy of the object, with the requisite modification. The library also includes operations for interfacing with linked data documents structured by the Constituent Structure

Ontology, presented in §8.2. It allows for JSON-LD documents to be imported and used to construct internal representations of constituent structures. The operation performs checking of the correctness of linked data descriptions, including the typing of attributes. Finally, JavaScript objects representing constituents and structures can be serialised to JSON-LD.

9.3 Tools Support for Abstract Data Types

In this section we describe a JavaScript module, `dt.js`, which implements the term language captured by the Dependent Type Ontology, presented in §8.3. The implementation includes a JavaScript class for each class of the ontology with methods which control the creation and manipulation of term objects. The structure of term objects follows the exact structure of the ontology. Term objects include methods which perform substitution, beta-reduction and convertibility checks on terms. In addition, we implement a system of type contexts, which are run-time structures containing lists of type declarations. These are used to type-check term objects at run-time, by passing a context object to the type checking method.

Specifications of abstract data types can be constructed as special instances of context objects which include a declaration for each component of the specification. These objects can be imported by the `cs.js` module, extending the constituent structure interface with abstract operations for manipulating constituent attribute data. Concrete implementations of these abstract interfaces, in the form of objects containing native JavaScript functions, can be 'plugged-in' to the abstract interface. This produces special term objects which includes a 'type-safe' JavaScript function. Applications of these terms to attribute values type check if the attribute value's abstract type and implementation information match. Performing beta-reduction on these term objects produces attribute values whose literal value is the result of applying the native JavaScript function to the literal values of the arguments. It is important to note, that currently the implementation is not able to verify whether a concrete implementation of an abstract data type satisfies the axioms of the specification; this is the responsibility of the implementer. Finally, the module includes operations for interfacing with linked data, allowing JSON-LD documents to be imported and generated from term objects.

A mechanism of dynamic dispatch is implemented, whereby attribute values are labelled with the name of a JavaScript module which contains a concrete implementation. The beta-reduction algorithms reads this information, automatically requires the module and locates the appropriate concrete operation, applying it to the literal arguments.

9.4 Tool Support for Constituent Specifications

In this section we describe a JavaScript module, `css.js`, which implements the specification language, defined in Chapter 6. The library enables users to construct structure invariants, constituent specifications, and constituent relations, and use them to check properties of constituent structures.

Expression of the specification language are implemented as JavaScript objects. The methodology follows exactly that of the `dt.js` module, with the structure of in-memory specification objects corresponding exactly with the structure of the ontology. Type checking is performed in exactly the same way. In addition, a partial decision procedure has been implemented which takes a constituent structure object and a structure invariant specification, and checks whether the structure satisfies the invariant. This procedure is defined recursively according to the structure of specification expressions. Atomic propositions are decided using built-in boolean operations or user-defined boolean operations from implementations of abstract data type specifications. The procedure works for a subset of specification expressions, described in Chapter 6, which does not include quantification over arbitrary abstract types.

9.5 Summary and Conclusions

In this chapter we have briefly described three JavaScript modules which implement the key aspects of the representation framework presented in Part II, and provide tool support for working with linked data documents structured by the ontologies presented in Chapter 8. Their purpose is to aid in the development of formally specified, interoperable software tools for computational musicology. The core contribution of this chapter is to demonstrate that, despite the apparent complexity of the type-based specification and its detachment from practical software tools, it is possible to implement its the functionality in a way which hides the complexity of the underlying theory from the user. In doing so the user is free to focus on data modelling and application development for his or her intended problem.

Chapter 10

Demonstration Analysis Applications

10.1 Introduction

In this chapter we describe how our representation framework and implementation can be used for the development of software tools for digital musicology. The purpose of this chapter is to demonstrate the utility of the proposed representation and the applicability of type-based methods of specification.

We examine three analysis tasks: pattern search and discovery, paradigmatic analysis and hierarchical set-class analysis. For each of these tasks we show how our framework supports representation of the input data and the output of the analysis. We focus on abstract, simplified examples of data for clarity. In addition, we characterise the stages of the analysis algorithms in terms high-level operations on abstract types. For the purposes of this thesis, we do not give formal definitions and/or specifications for these abstract types and operations. To do so would require extending the specification presented in §5.5 to capture the behaviour of constituent structures under certain mapping and folding operations. This is left for future work and discussed in §12.1. The implementations of the analysis algorithms instead manipulate JavaScript objects (representing constituent structures) using implicit knowledge about the way those objects are structured and the abstract data interfaces. Focus is concentrated on representation and specification of the inputs and outputs of the analysis, which are subject to type-checking as described in Chapter 9.

10.2 Pattern Search and Discovery

An important part of structural analysis of music is the identification of repeated patterns in low-level musical data. The SIA family of algorithms (Meredith et al., 2002) were developed to do just that. In this section we describe how our representation supports these analysis methods.

10.2.1 Representing the Input

The SIA algorithm takes as input a multi-dimensional dataset. This dataset is represented as a constituent whose particles represent the points of the data set. The coordinates of the points are defined via attributes. A k-dimensional dataset of size n is represented as a constituent structure s as follows:

$$\begin{aligned} s : \mathsf{STRUC} := [\quad & \mathsf{dataset} \mapsto (\{p_1,\ ...\ ,\ p_n\}, \{\}),\\ & p_1 \mapsto (\{\}, \{a_1 := v_{11},\ ...\ ,\ a_k := v_{1k}\}),\\ & \quad ...\\ & p_n \mapsto (\{\}, \{a_1 := v_{n1},\ ...\ ,\ a_k := v_{nk}\}) \quad], \end{aligned}$$

where the notation $(ps, \{a_1 := v_1,\ ...\ ,\ a_n := v_n\})$ is a syntactic shorthand for the functional expression $\mathsf{set}(a_n, v_n, (\ ...\ \mathsf{set}(a_1, v_1, \mathsf{delimit}(ps)))\ ...\ )$, and $[x_1 \mapsto c_1,\ ...\ ,\ x_n \mapsto c_n]$ is a syntactic shorthand for the functional expression $\mathsf{insert}(x_1, c_1, (...\mathsf{insert}(x_n, c_n, \mathsf{empty}))...)$. We also define a constituent specification which captures the structure of such multidimensional datasets as follows:

$$\mathsf{SIA\_Dataset} : \mathsf{CSPEC} := \forall_{\mathrm{Parts}}(\langle a_1\rangle[\wedge]...[\wedge]\langle a_k\rangle)$$

In addition, we require that the type of each attribute dimension $\mathsf{typ}(a_i)_{1<=i<=k}$ has an ordered interval structure. Typically the dimensions of the dataset will include pitch and onset. The abstract data type specifications for pitch and time given in §7.2 provide adequate functionality to perform this analysis, as they include ordering relations on points and intervals and functions for calculating the intervals between pairs of points.

Figure 10.1 shows the k-dimensional dataset constituent structure encoded in JSON-LD using the Constituent Structure Ontology. The dataset constituent is associated with the intrinsic property `:SIA_Dataset` to indicate its suitability for application of the SIA algorithm. Figure 10.2 shows this specification encoded in JSON-LD using the Constituent Structure Specification Ontology. The JavaScript tools are used to type check the dataset representation, as well as check that that is satisfies the specification indicated by its intrinsic property.

```
{"@context": ":ctx.json",
 "@id": ":dataset",
 "@type": ":Constituent",
 "cso:intrinsicProperty": ":SIA_Dataset",
 "cso:hasPart": [
    {"@id": ":p1",
     "@type":"cso:Constituent",
     "cso:hasAttribute": [":p1_a1", ... , ":p1_ak"]},
     ...
     {"@id": ":pn",
      "@type":"cso:Constituent",
      "cso:hasAttribute":[":pn_a1", ... , ":pn_ak"]}]}
```

Figure 10.1: A k-dimensional dataset of size n, represented in JSON-LD and structured using the Constituent Structure Ontology.

```
{"@id": ":SIA_Dataset",
 "@type": "cspec:ForallParts",
 "cspec:fpBody": {
    "@type": "cspec:Conjunction",
    "cspec:leftConjunct": {
        "@type": "cspec:hasAttribute",
        "cspec:attribute": ":a1"},
    "cspec:rightConjunct": {
        "@type": "cspec:Conjunction",

            ...

            "cspec:rightConjunct": {
                "@type": "cspec:hasAttribute",
                "cspec:attribute": ":ak"} ... }}}
```

Figure 10.2: The SIA input dataset specification represented in JSON-LD and structured using the Constituent Structure Specification Ontology.

10.2.2 The Algorithm

In this section we describe the steps of the SIA algorithm and characterise the operations abstractly as functional operations on abstract types.

1. The data points are sorted. This operation takes as input the dataset constituent structure and an ordering relation on the points, in the form of a constituent relation. This relation is defined in terms of the ordering and equality relations on the abstract types of the values. In the case of pitch and onset dimensions, this relation is defined as

$$\begin{aligned}\lambda x\ y.\exists t_x\ p_x\ t_y\ p_y.x &\overset{Onset}{\mapsto} t_x[\wedge]x \overset{Pitch}{\mapsto} p_x[\wedge]\\ &y \overset{Onset}{\mapsto} t_y[\wedge]y \overset{Pitch}{\mapsto} p_y[\wedge]\\ &[t_x <_t t_y \vee (t_x =_t t_y \wedge p_x \leqslant_p p_y)]\end{aligned}$$

   using the ordering and equality relations $<_t$, $=_t$ and $\leqslant_p$. The sorting operation produces an ordered list of constituent identifiers.

2. The vector table is computed. Translation vectors between points of the data set are represented as tuples of their respective interval types. For example, vectors between points in pitch and time would be of the product type $\mathsf{Vector} := \mathsf{Interval} \times \mathsf{TimeInterval}$. This stage of the algorithm takes as input the ordered list of constituent identifiers and returns a vector table. At the abstract level, a vector table is merely a finite map from pairs of identifiers to vectors.

3. The maximal translatable pattern (MTP) is computed for each vector. This stage of the analysis involves mapping over the vector table and grouping together constituent identifiers which are translatable by the same vector. This produces a finite map from vectors to finite sets of identifiers. The set of identifiers associated with each vector is an MTP.

SIATEC in addition finds all occurrences of each MTP. Additional heuristics such as compactness and coverage can be used to isolate only the interesting or significant patterns, however we do not include these aspects here for simplicity.

10.2.3 Representing the Output

We now describe how the results of the SIA analysis algorithm may be represented using constituent structures. MTPs are represented as constituents whose particles are the set of identifiers associated to a particular vector. In addition, we attribute to each MTP, the specific translation vector from which it arises. Each MTP constituent is annotated with an extrinsic property indicating that it *is* a discovered pattern. In the case of SIATEC we represent all occurrences of a particular pattern with a constituent. Finally we group together all the pattern classes in a single constituent representing the overall output of the analysis procedure. Figure 10.3 shows an example of the output of the method represented in JSON-LD using the Constituent Structure Ontology. `:mtp1` is an MTP consisting of two points, `:p1` and `:p2`, which is translatable by the abstract value `:vector1`, `:pattern1` is a constituent representing the pattern class of all occurrences of the MTP `:mtp1`, and `:results` groups together all such pattern classes.

```
...
{"@id": ":mtp1",
 "@type": "cso:Constituent",
 "cso:extrinsicProperty": ":MTP",
 "cso:hasAttribute": {
    "cso:attributeName": ":SIATranslationVector",
    "cso:attributeValue": ":vector1"},
 "cso:hasPart": [":p1", ":p2" ]}
...
{"@id": ":pattern1",
 "@type": "cso:Constituent",
 "cso:extrinsicProperty": ":PatternClass",
 "cso:hasPart": [":mtp1", "mpt1_1"]}
...
{"@id": ":results",
 "@type": "cso:Constituent",
 "cso:hasPart": ["pattern1", "pattern2"]}
...
```

Figure 10.3: An example MTP computed by SIA, represented in JSON-LD and structured using the Constituent Structure Ontology.

Associations can be used to annotate these output hierarchies. For example, translations between occurrences of patterns may be annotated with associations indicating the vector which relates them. In addition, an analyst may wish to annotate the workflow from input to output, in the manner of Raimond (2008), by associating with the dataset constituent the set of pattern classes which were computed from it. The use of associations for such annotations is described in §10.3.

10.3 Paradigmatic Analysis

An important aspect of music analysis is similarity between structural elements of a piece of music. The SIA method of discovering patterns, described in §10.2, regards translation as a primitive form of similarity between points in a multidimensional dataset. In this section we describe how more general notions of similarity can be captured as constituent relations in the specification logic, and how similarity annotations can be represented in constituent structures using associations. Specifically, we examine the paradigmatic analysis algorithm of Ruwet (1972) adapted from Smaill et al. (1993).

10.3.1 Representing the Input

This analysis procedure takes as input a monophonic line of musical notes. We define a specification for this input as follows:

$$\mathsf{SIM\_Dataset} : \mathsf{CSPEC} := \mathsf{Stream}[\wedge]\forall_{\text{Parts}}(\langle Pitch \rangle)$$

This specification asserts that a constituent is a stream, i.e. has no parts which overlap in time, and all of its parts have a pitch attribute. In addition, the analysis procedure takes a number of similarity relations which are used to determine the important structure within the piece. The four similarity relations considered by Smaill et al. (1993) can be formalised in the specification logic as constituent relations as follows:

$$\begin{aligned}
\mathsf{Identity} :=& \mathsf{PartRel}(\mathsf{NoteRepeat})\\
\mathsf{LongIdentity} :=& \lambda x\ y.[\exists]p_1\ p_2\ ps_1\ ps_2.\\
& \mathsf{MString}(p_1 :: ps_1, x)[\wedge]\mathsf{MString}(p_2 :: ps_2, y)[\wedge]\\
& \mathsf{AttRel}(Pitch, =_p, p_1, p_2)[\wedge]\mathsf{PairWise}(\mathsf{NoteRepeat}, ps_1, ps_2)\\
\mathsf{Transpose} :=& \lambda x\ y.[\exists]i.\mathsf{PartRel}(\mathsf{PInt}(i), x, y)\\
\mathsf{LooseTranspose} :=& \lambda x\ y.[\exists]p_1\ p_2\ ps_1\ ps_2.\\
& \mathsf{MString}(p_1 :: ps_1, x)[\wedge]\mathsf{MString}(p_2 :: ps_2, y)[\wedge]\\
& [\exists]i.\mathsf{PInt}(i, x, y)[\wedge]\mathsf{PairWise}(\mathsf{PInt}(i), ps_1, ps_2)
\end{aligned}$$

Identity identifies an exact repeat of a phrase. The constituent relation NoteRepeat captures when one note is a repeat of another, i.e. has equal pitch and duration attributes. LongIdentity identifies an exact repeat where the first note of the phrase is longer. The specification MString(l) captures when a constituent's particles form a stream of notes represented by the list of identifiers l. Transpose identifies a transposition of the original phrase by a fixed pitched interval i. The constituent relation $\mathsf{PInt}(i)$ captures when the pitch interval between two notes is i. $\mathsf{LooseTranspose}$ identifies a transposition by a fixed pitch interval, where the durations of the initial note might be different. The precise details of these relation specifications is unimportant, however we aim to emphasise that the specification logic is sufficiently expressive to formally capture the requisite relations. The dataset, its logical specification and the similarity relations are represented using the ontologies as was done for the SIA implementation in §10.2.

10.3.2 The Algorithm

The basic algorithm is as follows:

1. The piece is scanned for significant phrases. This is achieved by applying a modified version of the SIA algorithm which selects only translation vectors in which all but the time interval component is zero. This stage of the analysis takes as input the constituent structure dataset and produces an new constituent structure which includes constituents for each of the MTPs.

2. The piece is then rescanned for phrases which are similar to each of the MTPs using the similarity relations. This involves generating candidate constituents and checking whether the particular constituent relation holds between them. This stage of the analysis takes as input the set of MTPs and produces a new constituent structure which includes constituents which represent each similar phrase.

10.3.3 Representing The Output

The output of the algorithm is a set of constituents which are related to one another by the similarity relations. The representation of this hierarchical structure is done in the same way as described in §10.2. Figure 10.4 shows an example of the output of the analysis encoded in JSON-LD and structured using the Constituent Structure Ontology. `:phrase1` is a constituent representing the first occurrence of a musical phrase. It is given the extrinsic property `:motif` to indicate it was identified as such by the analysis procedure. `:phrase1_repeat1` is a constituent representing a repeat of the phrase. It is given the intrinsic property $\mathsf{Identity}(\texttt{:phrase1}) : \mathsf{CSPEC}$ to indicate the motif and similarity relation with which it is deemed structurally relevant. `:sim1` is an association which explicitly represents the analytic connection between the constituents. Here we use a simple Semantic Web vocabulary for representing constituent associations. It uses the namespace prefix `aso` and includes the class `aso:Association` of associations, the properties `aso:source` and `aso:target` for relating associations with their source and target constituents, respectively, and the property `aso:intrinsicProperty` for linking associations with association specifications. In this case, the intrinsic property of the association is the association specification $\mathsf{Assoc}(\mathsf{Identity}) : \mathsf{ASPEC}$. `:wf1` is an association linking the input dataset and the results of the analysis. It is annotated with an extrinsic property identifying it as a similarity analysis connection.

10.4 Hierarchical Set-Class Analysis

In this section we examine the hierarchical set-class analysis method of Martorell (2015). Pitch set classes are a way of abstractly describing the harmonic content of musical structures. The output of this analysis method is different in nature to that of the previous analysis examples in that it generates a new kind of analytic object: a segment. We describe how constituent structures can be usefully employed to represent such objects in a uniform way. In addition, the analysis involves new kinds of data values, specifically pitch classes and pitch set classes. A such, it serves as a good example for demonstrating how the type-based framework accommodates varied representations through the introduction of new abstract data types.

```
...
{"@id": ":phrase1",
 "@type": "cso:Constituent",
 "cso:extrinsicProperty": ":motif",
 "cso:hasPart": [ ... ]},
...
{"@id": ":phrase1_repeat1",
 "@type": "cso:Constituent",
 "cso:hasPart": [ ... ],
 "cso:extrinsicProperty": ":motif",
 "cso:intrinsicProperty": {
    "@type": "dto:Application",
    "dto:appHead": ":Identity",
    "dto:appParameter": ":phrase1"}},
...
{"@id": ":sim1",
 "@type": "aso:Association",
 "aso:source": ":phrase1",
 "aso:target": ":phrase1_repeat1",
 "aso:intrinsicProperty": {
    "@type": "aspec:Assoc",
    "aspec:crel": ":Identity"}}
...
{"@id": ":wf1",
 "@type": "aso:Association",
 "aso:source": ":sim_dataset",
 "aso:target": ":sim_results",
 "aso:extrinsicPropery": ":sim_analysis"}
...
```

Figure 10.4: Example output of the paradigmatic analysis procedure represented in JSON-LD and structured using the Constituent Structure Ontology.

10.4.1 Representing the Input

The analysis takes as input an arbitrarily complex musical surface of notes. This is represented as a constituent structure as in §10.2, where the attributes must include *Pitch*, *Onset* and *Duration*. We define a constituent specification for this class of inputs as follows:

$$\mathsf{SC\_Dataset} : \mathsf{CSPEC} := \forall_{\mathrm{Parts}}(\mathsf{Note}).$$

In addition, we must have available to us, data type specifications for pitch class, interval class, pitch class set and set class. These types are defined by extending the abstract data type specification for pitch from §7.2 with additional types and operations. The necessary operations include pitch and interval constants octave and semitone, a mapping from pitches to pitch classes, and a mapping from pitch class sets to set classes. The details of the abstract data type specifications are left for future work.

10.4.2 The Algorithm

The basic analysis method is as follows:

1. The musical input is systematically segmented. This involves creating a new segment for every possible pair of time points corresponding to either the beginning or end of a note. This method exhausts all segmentation possibilities.

2. The *class-scape* (Martorell, 2015, p. 5) is computed by assigning a set class to each segment. This is done by first computing the pitch class set of the segment; the pitch class material of a segment is defined as the set of pitch classes associated with notes which overlap the segment. The set class of a pitch class set is calculated as the cardinality-ordinal number of Forte (1973). The class scape is the collection of segments of the piece indexed by their onset, duration and set-class.

3. Subsequent analysis of the class-scape representation involves filtering or otherwise reducing the information for the purposes of visualisation, or using set-class analysis methods such as similarity measures (for example, Lewin (1979)) to view segments in terms of their similarity to some other judiciously selected set class.

10.4.3 Representing the Output

In this section we describe three possible methods of using constituent structures to represent the class-scape information. The specific method used will depend on the analytic requirements of the user of the information. A thorough examination of these requirements is beyond the scope of this thesis. Instead we aim to emphasise that the constituent structure representation is sufficiently general and flexible so as to accommodate multiple perspectives on the same data simultaneously and in a uniform way. The hierarchical aspects of the proposed representations are general to any temporal segment based analysis of music, although we use the set class analysis context to consider the utility of each representation.

The simplest way of representing class scape information with constituent structures is to assign an elementary constituent to each segment, attributed with onset, duration and set-class values. This essentially amounts to a flat hierarchical structure. Whist this may be the most computationally efficient for certain processing tasks, in the case of set class analysis, it does not explicitly represent the hierarchical relationships between set class material required for harmonic analysis (Martorell, 2015).

One method for capturing the hierarchical relationships between segments is to generate a constituent structure with a constituent for every segment, whose particles include every segment which is temporally included within it. This provides the analyst with explicit representation of the containment ordering between segments of a piece. However, this hierarchical structure implicitly contains a high degree of redundancy, as the particles of any given segment constituent are themselves hierarchically related.

A second method for capturing hierarchy between segments mitigates this redundancy. It involves constructing a stratified hierarchy, whereby each segment is indexed by the number of *atomic* segments that it spans. Atomic segments are defined by adjacent time points. Given a musical surface with n significant time points, there are $n-1$ atomic (1-)segments, $n-2$ 2-segments, and so on up to the entire piece, a single $(n-1)$-segment. The stratified hierarchy therefore consist of constituents representing i-segments whose particles are the two $(i-1)$-segments which are included in it. This representation explicitly captures the fine-grained detail of the hierarchical relationships between set classes in a uniform structure. Figure 10.5 shows a stratified hierarchy representation of a class-scape with ten segments (four atomic segments). `:s1` and `:s2` are atomic segments with no particles. `:s5` is the 2-segment which contains both `:s1` and `:s2`. `:s9` is the 3 segment which contains the 2-segments `:s6` and `:s7`, and `:s10` is the 4-segment spanning the entire piece and contains the 3-segments `:s8` and `:s9`. Each segment is annotated with an extrinsic property indicating its level of the hierarchy and given attributes representing its onset, duration and set class values.

```
...
{"@id": ":s1",
 "@type": "cso:Constituent",
 "cso:extrinsicProperty": ":1_segment",
 "cso:hasAttribute": [":s1_o", "s1_d", "s1_sc"]},
{"@id": ":s2",
 "@type": "cso:Constituent",
 "cso:extrinsicProperty": ":1_segment",
 "cso:hasAttribute": [":s2_o", "s2_d", "s2_sc"]},
...
{"@id": "s5",
 "@type": "cso:Constituent",
 "cso:extrinisicProperty": ":2_segement",
 "cso:hasAttribute": [":s5_o", "s5_d", "s5_sc"],
 "cso:hasPart": [":s1", "s2"]},
...
{"@id": ":s9",
 "@type": "cso:Constituent",
 "cso:extrinisicProperty": "3_segment",
 "cso:hasAttribute": [":s9_o", "s9_d", "s9_sc"]},
 "cso:hasPart": [":s6", ":s7"]},
{"@id": "s10",
 "@type": "cso:Constituent",
 "cso:extrinisicProperty": "4_segment",
 "cso:hasAttribute": [":s10_o", "s10_d", "s10_sc"]},
 "cso:hasPart": [":s8", ":s9"]},
...
```

Figure 10.5: A stratified hierarchy representation of class-scape information, encoded in JSON-LD and structured using the Constituent Structure Ontology.

The choice of constituent structure representation for complex hierarchies is left to the user. The set class example might be best represented as a stratified hierarchy of segments. However, a different kind of segment analysis, such as key estimation or audio feature analysis, might be best served by a different structure. The relation between analytic requirements and structural representation has not been addressed in this thesis. However, we again emphasis the key aspect of our approach: the accommodation of multiple perspectives without prescribing the kinds of hierarchy available.

10.5 Demonstrator Application

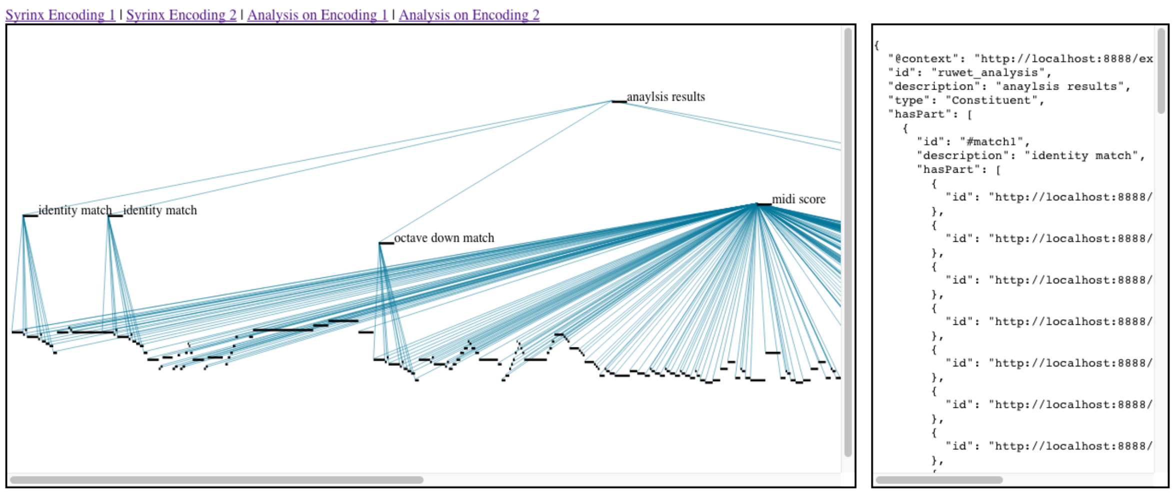
In this section we present a demonstrator application which performs simple analysis and visualisation of linked data documents structured by the constituent structure ontology (§8.2). The application is implemented as a Node.js web application using the JavaScript modules defined in Chapter 9. It implements the paradigmatic analysis algorithm described in §10.3.

10.5.1 Implementation

The application is implemented JavaScript using the modules described in Chapter 9. The analysis algorithm is a simplification of the method described in §10.3. The simplification involves explicitly passing, as a parameter to the procedure, a specific motif to which similar motifs will be found. The algorithm takes as input a constituent structure and two constituent identifiers. The first identifier represents the constituent whose particles are the musical material to be analysed. The second identifier identifies a constituent which represents a specific motif of interest. It begins by checking that both identifiers represent constituents of the structure and that they both represent monophonic lines. The algorithm then extracts from the structure, ordered collections of objects containing identifiers and attributes. The ordering is performed by comparison of the onset, pitch and duration attributes, respectively, of the constituents. This intermediate representation is used by the algorithm for efficient scanning of the material. The algorithm then uses the four similarity relations from §10.3 to find similar motifs. It iterates the second particle set over the first, generating temporary candidate constituents and checking whether each relation holds between them. If it does, a new constituent identifier is minted and used to insert an new constituent which delimits that particle subset. This constituent is then given a label which describes its structural relevance. Once all the relations have been checked, the new constituents are grouped into a further constituent labelled as the analysis results.

10.5.2 Analysis of Syrinx

The implementation of the algorithm is used in a simple web application which reads a linked data document from a local server, performs the analysis and displays the results using a simple visualisation tool. Figure 10.6 shows a screen shot from the application in which the analysis has been applied to an encoding of Debussy's Syrinx. The left hand panel of the window displays a visualisation the hierarchical constituent structure which is produced. Black rectangles indicate constituents, while blue lines indicate the particular relationship. The right hand panel of the window displays the analysis results in JSON-LD form.


Syrinx Encoding 1 | Syrinx Encoding 2 | Analysis on Encoding 1 | Analysis on Encoding 2
anaylsis results
identity match
identity match
octave down match
midi score

```
{
  "@context": "http://localhost:8888/ex
  "id": "ruwet_analysis",
  "description": "anaylsis results",
  "type": "Constituent",
  "hasPart": [
    {
      "id": "#match1",
      "description": "identity match",
      "hasPart": [
        {
          "id": "http://localhost:8888/
        },
        {
          "id": "http://localhost:8888/
        },
        {
          "id": "http://localhost:8888/
        },
        {
          "id": "http://localhost:8888/
        },
        {
          "id": "http://localhost:8888/
        },
        {
          "id": "http://localhost:8888/
        },
        {
          "id": "http://localhost:8888/
        },
```

Figure 10.6: Screenshot from the demonstrator application.

The application also demonstrates how the implementation supports generic processing of musical documents by applying the same analysis algorithm to two different underlying encodings of Syrinx. The underlying implementation of the framework uses dynamic dispatch to select, at run-time, the correct implementation of the abstract data types according to the data it is presented.

10.6 Summary and Conclusions

In this chapter we have demonstrated how our representation framework supports three different analysis tasks. We have shown how the framework supports the representation the low-level input musical data, as well as the logical properties of this data and logical relations required for analysis. We have shown how the framework supports the representation of the information produced by these analysis methods as output, including sophisticated structural annotations and workflow relationships. Finally, we have described how the analysis algorithms may be viewed in terms of high-level operations on abstract types of the representation. Although, the existing implementations do not entirely follow these high-level descriptions, they are important in motivating future work on the formal specification of music processing methods, discussed in §12.2.

Representation of the output of analysis methods is something which is often neglected by music representations systems. We argue that it is an essential requirement of music information systems, in which any analysis performed produces data of potential relevance to another music research question. The ability to accurately record, share and reproduce analysis is an essential part of any scientific research field. We believe that, in this chapter, we have giving sufficient preliminary illustration of how the type-based framework, presented in Part II, provides a formal basis which facilitates these aspects of music research.

Part IV

Conclusions and Future Work

Summary

In this, the final part of the thesis, we give the conclusions of the research and propose a number of directions for potential future work on the topic. Chapter 11 provides a summary of the work and highlights the major findings and contributions, as well as giving a high-level discussion of the ultimate goals and status of the wider project. Chapter 12 discusses future work and identifies three directions in which it might proceed.

Chapter 11

Conclusions

11.1 Summary

In this thesis we have addressed the problem of representing music and musical information in computer systems. In Part I we surveyed the literature on the topics of knowledge representation and music representation, and in doing so identified the requirements of a general purpose music knowledge representation system. In Part II we gave a abstract specification for a music representation framework which meets these requirements. The specification was given in dependent type theory and consisted of a multiple-hierarchical information model and a specification logic. In Part III we gave an implementation of the specification using Semantic Web ontologies and JavaScript, and demonstrated how it could be used to support the development of software tools for music analysis.

Figure 11.1 gives a diagrammatic overview of the work, indicating how the framework and implemented tools fit into a broader setting of computer-based music research. It shows the different levels of abstraction in an idealised music research pipeline, with the framework acting as an abstract interface between user applications and musical data. This overview provides a point for reference for the discussion of the rest of this chapter and the following one.

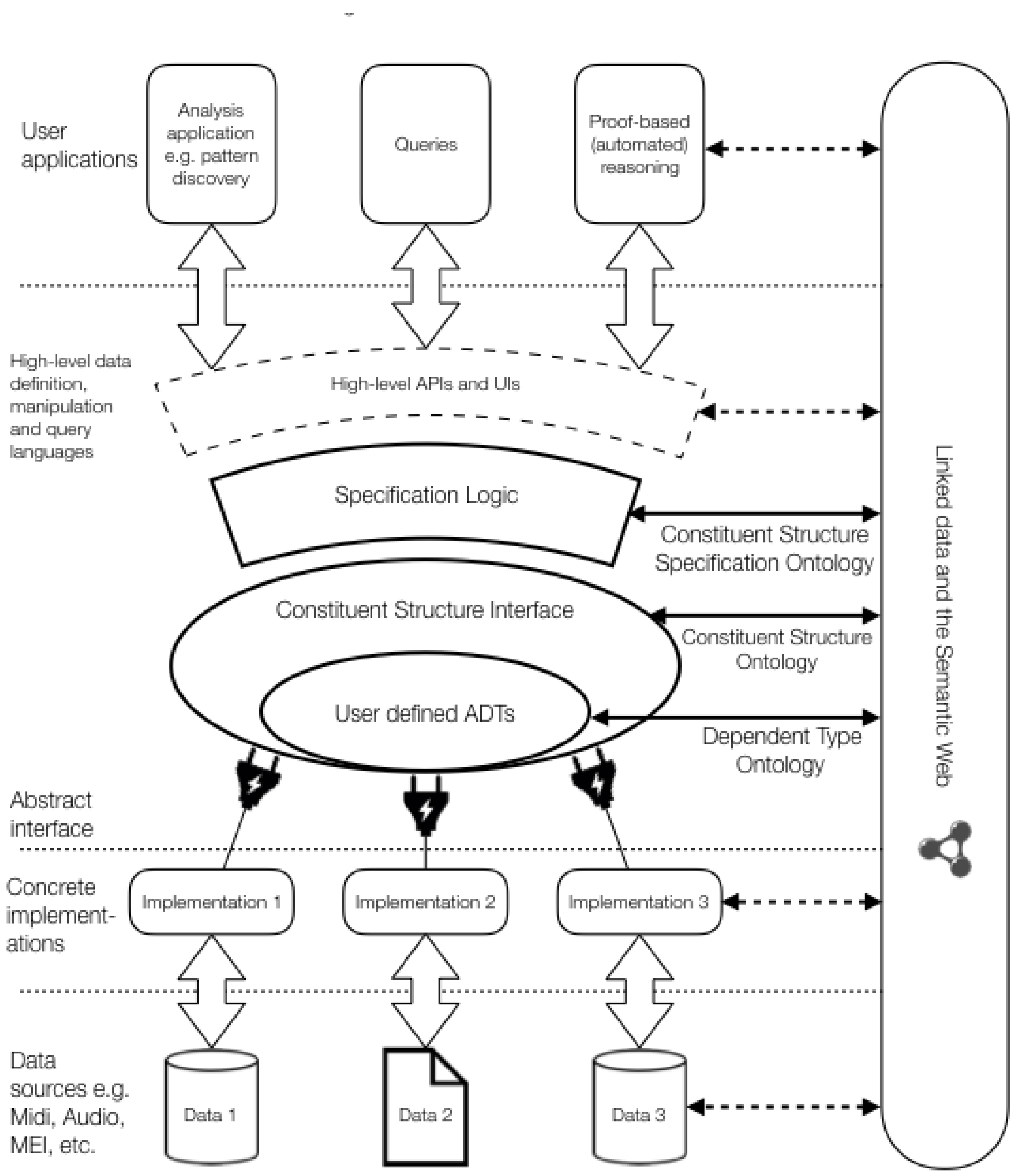

User applications
Analysis application e.g. pattern discovery
Queries
Proof-based (automated) reasoning
High-level data definition, manipulation and query languages
High-level APIs and UIs
Specification Logic
Constituent Structure Specification Ontology
Constituent Structure Interface
Constituent Structure Ontology
User defined ADTs
Dependent Type Ontology
Abstract interface
Concrete implement-ations
Implementation 1
Implementation 2
Implementation 3
Data sources e.g. Midi, Audio, MEI, etc.
Data 1
Data 2
Data 3
Linked data and the Semantic Web

Figure 11.1: The different levels of abstraction in an idealised music research pipeline: The knowledge representation framework forms the core of the abstract interface layer, which connects user applications to data sources via dedicated concrete implementations. On the right hand side, Semantic Web ontologies, represented by arrows, provide linked data formats for the different levels of abstraction. Solid arrows denote ontologies presented in the thesis, whilst dotted arrows denote ontologies founds elsewhere or designated future work.

11.2 Contributions

The core contributions of the thesis are as follows:

1. A detailed literature review examining the issues and requirements of music knowledge representation.
2. An abstract specification for a general purpose music knowledge representation framework in dependent type theory.
3. An implementation of the framework using Semantic Web technologies and JavaScript, and demonstrator application for music analysis and visualisation of musical data.

11.3 Conclusions

In this section we summarise the conclusions of the thesis:

Part I: Using type theory for music knowledge representation is sensible and necessary. Music is a highly complex subject and so is demanding from the perspective of knowledge representation. Type theory is a powerful tool for logic and programming which meets these demands.

Chapter 2: Type theory is a powerful alternative to classical FOL as a basis for knowledge representation.

Chapter 3: Knowledge representation languages can be made more expressive, precise and interoperable with types.

Chapter 4: Musical knowledge requires expressive, precise and interoperable representation languages. Knowledge representation languages based on singly-typed classical FOL are not suitable for music representation.

Part II: Type theory, specifically the Calculus of Inductive Constructions, can be used to give a formal specification for a general purpose representation of music, at an abstract level. Separating the information structure from the language of invariants used to describe it, enables the representation to remain entirely general.

Chapter 5: The constituent structure representation satisfies the requirements of a general purpose music representation system, and generalises many existing methods. The use of algebraic specifications of abstract data types provides greater precision and modularity.

Chapter 6: The constituent structure specification logic is highly expressive and is able to accommodate the vast majority of data description and conceptual modelling needs in the domain of music.

Chapter 7: The representation can be extended to incorporate the diverse requirements of music representation.

Part III: The type-based specification can be implemented using mainstream technologies, namely Semantic Web ontology languages and JavaScript. Such implementations are necessary in affording users of the framework the power of linked open data and web applications.

Chapter 8: The representation framework can be deployed on the Semantic Web. Semantic Web ontology languages are capable of capturing the structural properties of constituent structures, as well as the syntax of expression of the specification logic.

Chapter 9: The representation framework can be operationalised using JavaScript. JavaScript provides enough flexibility to allow for the incorporation of type information and performance of type checking at runtime.

Chapter 10: The implementation supports the development of tools for music analysis. The representation requirements of many music analysis tasks can be formalised in the framework. The demonstrator application is a proof of concept. It shows that the abstract specification can be implemented using mainstream tools allowing developers and researchers to readily adopt it.

11.4 Discussion

The overall goal of this kind work on representation is a future state of affairs in which computer systems for computational musicology are entirely conceptually interoperable and highly automated. From the perspective of software development this involves the formal specification of not only the information and knowledge content, but also the logical and computational properties of representations. This level of specification allows developers to use high-level, domain-specific languages to create precise, modular and automated software tools which in turn provide musicologists with composable methods of analysis and comparison.

The work presented in this thesis is a first step in achieving this goal. The selection of type theory as a basis for both software development and knowledge representation leads to the application of type-based methods of verified software to music knowledge representation. The music representation framework presented in this thesis constitutes a formal basement for future research into music knowledge representation.

Using type theory for knowledge representation constitutes a significant change from the traditional methods involving set theory and FOL. Chief among the consequences are that knowledge representation languages, conceptual models and ontologies must be developed in concert with appropriate methods of automated reasoning. In the type-theoretic or constructive setting, proofs are the method by which knowledge is represented and communicated. We take the view that this is a highly natural and intuitive way to approach knowledge representation. Intuitionistic logic takes its name from Brouwer's program of intuitionism, in which all mathematical objects, including proofs and programs, are regarded as kinds of construction which be shared, manipulated and verified by machines. This work is just the beginning in exploring these connections.

Chapter 12

Future Work

12.1 Developing the Specification in Coq

The Coq proof assistant is a powerful tool for developing type-based specifications, and includes an imperative language $\mathrm{L}_{\mathrm{tac}}$ for automating proof construction. Possible future work could be the use of $\mathrm{L}_{\mathrm{tac}}$ to develop proof automation methods for the specification logic presented in Chapter 6. Such proof automation could involve proof by reflection, whereby a syntactic subset of the logic is represented as a functional data structure and used to develop decision procedures.

Coq has many other features which assist in proof automation and knowledge management. Spitters and van der Weegen (2011) uses *type classes* and instance resolution formalise various aspects of mathematics and universal algebra in Coq. This constitutes a flexible and intuitive approach to proof management and the structuring of mathematical knowledge. Similar techniques could be used in the management of abstract data type specifications for the music representation framework.

Proof assistants such as Coq are often used to formalise and verify properties of programming languages using higher-order abstract syntax. The statics and dynamics of these languages are then defined by logical relations over abstract syntax trees. These techniques could be used to design high-level languages for data querying, data manipulation and data description based on constituent structures.

There is scope for the development constituent structure specification to include operations for mapping and folding over the finite data structures involved, such as the constituents of a structure or the particles of a constituent. The requisite extensions could borrow almost entirely from the Coq standard library for lists. This aspect of the specification has been mentioned in Chapter 10, and is, in general, required in order to give full logical specifications of analysis algorithms at the specification level. This would allow analysis algorithms, such as those described in Chapter 10, to be implemented only in terms of the abstract type and interface operations of the specification, ultimately leading to more conceptually interoperable tools for musicology.

12.2 Developing the Implementation

Further work is required on both the Semantic Web ontologies and JavaScript Modules in order to make them publishable for use by other researchers and developers.

We have demonstrated ways in which the Constituent Structure Ontology can be extended with OWL axioms which capture more domain-specific music knowledge representation. Future research could explore these extensions more thoroughly and work towards using the ontology to unify and integrate existing music ontology projects.

The Dependent Type Ontology could be extended to develop high-level languages for the modular organisation of data type specification. Such languages could be based on category theory such as Ologs (Spivak and Kent, 2012) or using distinct levels of abstraction such as those of the Audio Features Ontology (Allik and Sandler, 2016).

There are many possible ways in which the JavaScript tools may be developed and extended. In particular, it would be highly beneficial to developers and musicologists if there existed tools which could automatically generate code from high-level descriptions of music analysis workflows. Such tools could be developed in conjunction with additional ontologies for capturing these high-level description. Implementations if these kinds of tools would most likely require the extensions to the constituent structure specification mentioned in §12.1. Implementations of an interface which included mapping and reducing functionality on both constituent structures and constituent objects would allow analysis algorithms to be constructed at the abstract level by composition of atomic operations from the specifications, and used to automatically generate implementations of the analysis procedure given the requisite implementations of abstract data types.

12.3 Type Theories for Knowledge Representation

A final direction of future work could explore the application of different type theories and systems to knowledge representation. In this thesis we have used CIC to give formal specifications for the representation components. Alternative type systems with different computational and logical properties could prove more suitable for certain aspects of knowledge representation systems. For example, sub-structural type systems capture various different notions of computation by omitting one or more of the structural rules. In particular, the contexts of *linear* type systems capture the notion of memory states, or resources, and could be used to model knowledge bases which capture the existence of certain objects. Krishnaswami et al. (2015) develop a type system which integrates linear and dependent types, and shows how it can be used to give formal specifications of type-safe imperative programs. Such a type system could be applicable to the design of knowledge representation or database languages. Another possibility is to explore the use of more exotic type systems. Benzaken et al. (2013) present a type system for No-SQL-style languages. It focusses on the typing of programs designed to manipulate semi-structured data, such as that of JSON documents. Such languages could be widely applied to music knowledge representation, as they would afford the smooth integration of conceptual modelling and programming languages.

Bibliography

Abdallah, S., Y. Raimond, and M. Sandler

2006. An ontology-based approach to information management for music analysis systems. In *120th Convention of the Audio Engineering Society*, Pp. 1–10.

Ackoff, R. L.

1989. From data to wisdom. *Journal of Applied Systems Analysis*, 16(1):3–9.

Adams, B. and M. Raubal

2009a. A Metric Conceptual Space Algebra. In *Spatial information theory*, Pp. 51–68. Springer.

Adams, B. and M. Raubal

2009b. Conceptual Space Markup Language (CSML): Towards the cognitive SemanticWeb. *ICSC 2009 - 2009 IEEE International Conference on Semantic Computing*, Pp. 253–260.

Allen, J. F.

1984. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154.

Allik, A. and M. Sandler

2016. An Ontology for Audio Features. In *Ismir 2016*.

Alvaro, J., E. Miranda, and B. Barros

2005. EV Ontology: Multilevel Knowledge Representation and Programming. In *Proceedings of the 10th Brazilian Symposium of Musical Computation (SBCM). Belo Horizonte, Brazil*, Pp. 36–47.

Arp, R., B. Smith, and A. D. Spear

2015. *Building Ontologies With Basic Formal Ontology.*

Awodey, S.

2010. *Category Theory*, 2 edition. Oxford: Oxford University Press.

Baader, F., I. Horrocks, and U. Sattler

2007. Description Logics. In *Handbook of Knowledge Representation*, F. van Harmelen, V. Lifschitz, and B. Porter, eds., chapter 3, Pp. 135–180. Elsevier.

Babbitt, M.

1965. The Use of Computers in Musicological Research. *Perspectives of New Music*, 3(2):74–83.

Balaban, M.

1988. A music-workstation based on multiple hierarchical views of music. In *14th International Computer Music Conference*, Pp. 56–65.

Balaban, M.

1996. The music structures approach in knowledge representation for music processing. *Computer Music Journal*, 20(2):96–111.

Barendregy, H.

1991. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154.

Barlatier, P. and R. Dapoigny

2012. A type-theoretical approach for ontologies: The case of roles. *Applied Ontology*, 7(3):311–356.

Barthes, R.

1967. *Elements of Semiology.*

Barwise, J. and J. Seligman

1997. Information Flow: The Logic of Distributed Systems. *Cambridge Tracts in Theoretical Computer Science*, 44.

Bates, R. and R. MacGregor

1987. The Loom Knowledge Representation Language. Technical report.

Bel, B. and J. Kippen

1992. Bol Processor Grammars. In *Understanding Music with AI – Perspectives on Music Cognition*, Pp. 366–401.

Benzaken, V., G. Castagna, K. Nguyen, and J. Siméon

2013. Static and dynamic semantics of NoSQL languages. *POPL: Principles of Programming Languages*, Pp. 101–114.

Bertot, Y. and P. Castéran

2004. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions.*

Boley, H., S. Tabet, and G. Wagner

2001. Design Rationale of RuleML: A Markup Language for Semantic Web Rules. *The First International Semantic Web Working Symposium*, 1:381–401.

Bonner, A. J. and M. Kifer

1994. An overview of transaction logic. *Theoretical Computer Science*, 133(2):205–265.

Brachman, R. and J. Schmolze

1985. An Overview of the KLONE Knowledge Representation System. *Cognitive science*, 216:171–216.

Brachman, R. J.

1978. A Structural Paradigm for Representing Knowledge. Technical report.

Brachman, R. J. and H. J. Levesque

2004. *Knowledge Representation and Reasoning.* Elsevier.

Bratko, I.

2012. *PROLOG Programming for Artificial Intelligence*, 4 edition. New York: AddisonWesley.

Cafezeiro, I. and E. H. Haeusler

2007. Semantic interoperability via category theory. In *Tutorials, posters, panels and industrial contributions at the 26th international conference on Conceptual modeling*, volume 83, Pp. 197–202.

Camurri, A. and M. Frixione

1992. A model representation and communication of music and multimedia knowledge. *Proceedings of the 10th ....*

Camurri, A., M. Frixione, and C. Innocenti

1994. A cognitive model & knowledge representation system for music & multimedia. *Journal of New Music Research*, 23(4):317–347.

Chella, A.

2015. A cognitive architecture for music perception exploiting conceptual spaces. In *Applications Of Conceptual Spaces: The Case For Geometric Knowledge Representation*, Pp. 187–206. Cham: Springer International Publishing.

Chlipala, A.

2011. Mostly-automated verification of low-level programs in computational separation logic. *ACM SIGPLAN Notices*, 46(6):234–245.

Chlipala, A.

2013. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. *... international conference on Functional programming*, Pp. 391–402.

Chlipala, A. J.

2007. *Implementing Certified Programming Language Tools in Dependent Type Theory.* PhD thesis.

Church, A.

1936. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58(2):345.

Church, A.

1940. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(2):56–68.

Conklin, D. and I. H. Witten

1995. Multiple viewpoint systems for music prediction. *Journal of New Music Research*, 24(1):51–73.

Cook, W. R. and A. H. Ibrahim

2006. Integrating programming languages and databases: What is the problem. *ODBMS ORG Expert Article*, (0448128).

Coquand, T. and G. Huet

1988. The calculus of constructions. *Information and Computation*, 76(2-3):95–120.

Corcho, O. and A. Gomez-Perez

2000. Evaluating knowledge representation and reasoning capabilites of ontology specification languages. *In Proceedings of the ECAI 2000 Workshop on Applications of Ontologies and Problem-Solving Methods.*

Crawford, T. and R. Lewis

2016. Review: Music Encoding Initiative. *Journal of the American Musicological Society*, 69(1):273–285.

Curry, H. B. and R. Feys

1958. *Combinatory Logic. Vol. I.* Amsterdam: North Holland: [publisher unknown].

Cuthbert, M. S. and C. Ariza

2010. Music21 A Toolkit for Computer-Aided Musicology and Symbolic Music Data. In *11th International Society for Music Information Retrieval Conference(ISMIR 2010)*, Pp. 637–642.

Dale, N. and H. M. Walker

1996. *Abstract Data Types: Specifications, Implementations, and Applications.* Jones & Bartlett Learning.

Dannenberg, R. B.

1993. A Brief Survey of Music Representation Issues, Techniques, and Systems. Technical Report 3.

Dapoigny, R. and P. Barlatier

2010a. Modeling contexts with dependent types. *Fundamenta Informaticae*, 104(4):293–327.

Dapoigny, R. and P. Barlatier

2010b. Towards ontological correctness of part-whole relations with dependent types. In *Frontiers in Artificial Intelligence and Applications*, volume 209, Pp. 45–58.

Davis, R., H. Shrobe, and P. Szolovits

1993. What is Knowledge Representation. *AI Magazine*, 14(1):17–33.

Delaware, B., C. Pit-Claudel, J. Gross, and A. Chlipala

2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. *SIGPLAN Not.*, 50(1):689–700.

Donnellan, K. S.

1966. Reference and Definite Descriptions. *The Philosophical Review*, 75(3):281.

Dumbrava, S.-G.

2016. *A Coq Formalization of Relational and Deductive Databases and Mechanizations of Datalog.* PhD thesis.

Erickson, R. F.

1975. "The darms Project": A Status Report. *Computers and the Humanities*, 9(6):291–298.

Fields, B., K. Page, D. De Roure, and T. Crawford

2011. The segment ontology: Bridging music-generic and domain-specific. In *Proceedings - IEEE International Conference on Multimedia and Expo*, Pp. 1–6.

Fisher, K., Y. Mandelbaum, and D. Walker

2006. The next 700 data description languages. *ACM SIGPLAN Notices*, 41(1):2–15.

Forte, A.

1973. *The Structure of Atonal Music*, volume 61. London: Yale University Press.

Forth, J., G. A. Wiggins, and A. McLean

2010. Unifying Conceptual Spaces: Concept Formation in Musical Creative Systems. *Minds and Machines*, 20(4):503–532.

Fridman, N. and M. Musen

2000. PROMPT: Algorithm and tool for automated ontology merging and alignment. *Proc. AAAI'00*, Pp. 450–455.

Gangemi, A., N. Guarino, C. Masolo, A. Oltramari, and L. Schneider

2002. Sweetening ontologies with DOLCE. *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web, Lecture Notes in Computer Science, vol. 2473*, Pp. 223–233.

Gärdenfors, P.

2000. *Conceptual Spaces: The Geometry of Thought*. Cambridge, MA: MIT Press.

Genesereth, M. R.

1991. Knowledge Interchange Format. *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, (June):238–249.

Genesereth, M. R. and N. J. Nilsson

1987. Logical Foundations of Artificial Intelligence. In *The Journal of Symbolic Logic*, volume 55, P. 405.

Goguen, J. A.

1991. Types as Theories. In *Topology and category theory in computer science*, Pp. 357–385. New York: Oxford University Press.

Good, M.

2001. MusicXML: An internet-friendly format for sheet music. *XML Conference and Expo*, Pp. 1–12.

Gruber, T.

1992. Ontolingua: A mechanism to support portable ontologies. Technical report.

Gruber, T. R.

1993. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220.

Guarino, N.

1998. Formal Ontology and Information Systems. (June):3–15.

Guarino, N. and P. Giaretta

1995. Ontologies and Knowledge Bases: Towards a Terminological Clarification. *Towards Very Large Knowledge Bases. Knowledge Building and Knowledge Sharing*, 1(9):25–32.

Guarino, N. and D. Oberle

2009. What is An Ontology? In *Handbook on Ontologies*, Pp. 1–17.

Harris, M., A. Smaill, and G. Wiggins

1991. Representing Music Symbolically. In *IX Colloquio di Informatica Musicale*, Pp. 55–69.

Hoglund, H.

2014. Music Suite: A Family of Musical Representations. In *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*, Pp. 33–34.

Horrocks, I., P. F. Patel-schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean

2004. SWRL : A Semantic Web Rule Language Combining OWL and RuleML. *W3C Member submission 21*, (May 2004):1–20.

Howard, W.

1980. *The formulae-as-types notion of construction*. Academic Press.

Huang, S. S., T. J. Green, and B. T. Loo

2011. Datalog and emerging applications. In *Proceedings of the 2011 international conference on Management of data - SIGMOD '11*, P. 1213.

Hudak, P.

1996. Haskore Music Tutorial. In *Second International School on Advanced Functional Programming*, Pp. 38–68. Springer-Verlag.

Hudak, P.

2011. The Haskell School of Music–From Signals to Symphonies. *Yale University, Department of Computer Science*, 4:353.

Humphrey, E. J., J. Salamon, O. Nieto, J. Forsyth, R. M. Bittner, and J. P. Bello

2014. JAMS: A JSON Annotated Music Specification for Reproducible MIR Research. In *Proceedings of the 15th International Society for Music Information Retrieval Conference*, Pp. 591–596.

Huovinen, E. and A. Tenkanen

2007. Bird's-Eye Views of the Musical Surface: Methods for Systematic Pitch-Class Set Analysis. *Music Analysis*, 26(1-2):159–214.

Jacobs, B.

1999. *Categorical Logic and Type Theory*. Amsterdam: Elsevier.

Jacobson, K., Y. Raimond, and M. Sandler

2009. An Ecosystem for Transparent Music Similarity in an Open World. *ISMIR*, Pp. 2–7.

Kifer, M.

2008. Rule Interchange Format: The Framework. In *International Conference on Web Reasoning and Rule Systems*, D. Calvanese and G. Lausen, eds., volume 5341, Pp. 1–11. Springer, Berlin, Heidelberg.

Kifer, M. and G. Lausen

1989. F-logic: a higher-order language for reasoning about objects, inheritance, and scheme. *ACM SIGMOD Record*, 18(2):134–146.

Kostelic, C.

2017. Applying the Levels of Conceptual Interoperability Model to a Digital Library Ecosystem – a Case Study. *Proc. Int'l Conf. on Dublin Core and Metadata Applications*, Pp. 62–72.

Krishnaswami, N. R., P. Pradic, and N. Benton

2015. Integrating Dependent and Linear Types. *POPL: Principles of Programming Languages*, Pp. 1–65.

Lambek, J.

1980. From lambda calculus to Cartesian closed categories. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. P. Seldin and J. R. Hindley, eds., Pp. 376–402. Academic Press.

Lassila, O. and R. R. Swick

1999. Resource Description Framework (RDF) Model and Syntax Specification.

Lerdahl, F. and R. Jackendoff

1985. *A generative theory of tonal music.* MIT Press.

Lewin, D.

1979. A response to a response: on Pcset relatedness. *Perspectives of new music*, 18(1):498–502.

Lewin, D.

1987. *Generalized Musical Intervals and Transformations.* New York: Yale University Press.

Lewis, D., R. Woodley, T. Crawford, J. Forth, C. Rhodes, and G. Wiggins

2011. Tools for Music Scholarship and their Interactions: A Case Study. In *Supporting Digital Humanities*, Copenhagen, Denmark.

Loy, G. and C. Abbott

1985. Programming languages for computer music synthesis, performance, and composition. *ACM Computing Surveys*, 17(2):235–265.

Luo, Z.

1989. ECC, an Extended Calculus of Constructions. In *Proceedings. Fourth Annual Symposium on Logic in Computer Science*, Pp. 386–395.

Ma, X., J. Erickson, S. Zednik, P. West, and P. Fox

2016. Semantic Specification of Data Types for a World of Open Data. *ISPRS International Journal of Geo-Information*, 5(3):38.

Malecha, G., G. Morrisett, A. Shinnar, and R. Wisnesky

2010. Toward a verified relational database management system. In *POPL*, volume 45, Pp. 237–248.

Marsden, A.

2000. *Representing Musical Time: A Temporal-Logic Approach.* Lisse: Swets & Zeitlinger.

Marsden, A. a.

2005. Generative Structural Representation of Tonal Music. *Journal of New Music Research*, 34(4):409–428.

Martorell, A.

2015. Hierarchical Multi-Scale Set-Class Analysis. *Journal of Mathematics and Music*, 9(1):95–108.

Mazzola, G.

2006. Functors for Music: The Rubato Composer System. In *International Computer Music Conference*, Pp. 83–90.

Mazzola, G.

2012. *The topos of music: geometric logic of concepts, theory, and performance.* Birkhäuser.

McCarthy, J.

1960. Recursive functions symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195.

McDermott, D. and D. Dou

2002. Representing Disjunction and Quantifiers in RDF. *Iswc 2002*, 2342:250–263.

Mcfee, B., E. Humphrey, and J. Urbano

2016. A Plan for Sustainable MIR Evaluation. *International Conference on Music Information Retrieval (ISMIR)*, Pp. 285–291.

McGuinness, D., R. Fikes, J. Rice, and S. Wilder

2000. The Chimaera ontology environment. *Proceedings of the National Conference on Artificial Intelligence*, Pp. 1123–1124.

Meghanathan, N., D. Nagamalai, and N. Chaki

2013. Dynamic Ontology Construction for E-Trading. In *Advances in Computing* *and Information Technology*, N. Meghanathan, D. Nagamalai, and N. Chaki, eds., Pp. 439–449. Springer Berlin Heidelberg.

Meredith, D., K. Lemström, and G. A. Wiggins
2002. Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music. *Journal of New Music Research*, 31(4):321–345.

Milmeister, G.
2014. *The Rubato Composer Music Software*, number 1.

Minsky, M.
1974. A framework for representing knowledge. Technical report.

Motik, B., B. C. Grau, and U. Sattler
2008. Structured Objects in OWL: Representation and Reasoning. *Proceedings of the 17th International World Wide Web Conference (WWW 2008)*, Pp. 555–564.

Motik, B., P. F. Patel-Schneider, B. Parsia, C. Bock, A. Fokoue, P. Haase, R. Hoekstra, I. Horrocks, A. Ruttenberg, U. Sattler, and M. Smith
2009. OWL 2 Web Ontology Language - Structural Specification and Functional-Style Syntax (Second Edition).

Nattiez, J.-J.
1975. *Fondements d'une sémiologie de la musique.* Paris: Union Générale en d'Éditions.

Nieto, O. and J. P. Bello
2015. MSAF: Music Structure Analysis Framework. *Proc. of the 16th International Society for Music Information Retrieval Conference*, Pp. 14–15.

Niles, I. and A. Pease
2001. Towards a standard upper ontology. In *Proceedings of the international conference on Formal Ontology in Information Systems - FOIS '01*, Pp. 2–9, New York, New York, USA. ACM Press.

Ockelford, A.

2005. *Repetition in Music*. London: Routledge.

Orlarey, Y., D. Fober, and S. Letz

1994. Lambda Calculus and Music Calculi. In *Proceedings of the International Computer Music Conference, ICMC'94*, Pp. 243–250.

Parsia, B., E. Sirin, B. Cuenca, E. Ruckhaus, and D. Hewlett

2005. Cautiously Approaching SWRL. Technical report, University of Maryland.

Raimond, Y.

2008. *A Distributed Music Information System for the Degree of Doctor of Philosophy*. PhD thesis, Queen Mary University of London.

Raimond, Y., S. Abdallah, M. Sandler, and F. Giasson

2007. The Music Ontology. In *ISMIR 2007: 8th International Conference on Music Information Retrieval*, volume 8, Pp. 417–422.

Raubal, M., B. Adams, and S. Barbara

2010. The Semantic Web Needs More Cognition. *Semantic Web*, 1(1, 2):69–74.

Rector, A., N. Drummond, M. Horridge, J. Rogers, H. Knublauch, R. Stevens, H. Wang, and C. Wroe

2004. OWL Pizzas: Practical Experience of Teaching OWL-DL: Common Errors & Common Patterns. In *Proceedings of the 14th International Conference on Knowledge Acquisition, Modeling and Management (EKAW 2004)*, Pp. 63–81.

Reynolds, J.

1974. Towards a theory of type structure. *Programming Symposium*, Pp. 408–425.

Reynolds, J. C.

2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*.

Roland, P.

2002. The Music Encoding Initiative (MEI). In *Proceedings of the First International Conference on Musical Applications Using XML*, Pp. 55–59.

Rowley, J.

2007. The wisdom hierarchy: Representations of the DIKW hierarchy. *Journal of Information Science*, 33(2):163–180.

Ruwet, N.

1972. *Langage, musique, poésie.* Paris: Editions du Seuil.

Schmolze, J. G. and D. Isreal

1983. *KL-ONE: Semantics and classification.* Cambridge, MA: Bolt Beranek and Newman.

Schmolze, J. G. and T. a. Lipkis

1983. Classification in the KL-ONE knowledge representation system. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, Pp. 330–332.

Sirin, E., B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz

2007. Pellet: A practical OWL-DL reasoner. *Web Semantics*, 5(2):51–53.

Smaill, A., G. A. Wiggins, and M. Harris

1993. Hierarchical Music Representation for Composition and Analysis. *Journal of Computing and the Humanities*, 27:7–17.

Smith, B., M. Ashburner, C. Rosse, J. Bard, W. Bug, W. Ceusters, L. J. Goldberg, K. Eilbeck, A. Ireland, C. J. Mungall, N. Leontis, P. Rocca-Serra, A. Ruttenberg, S. A. Sansone, R. H. Scheuermann, N. Shah, P. L. Whetzel, and S. Lewis

2007. The OBO Foundry: Coordinated evolution of ontologies to support biomedical data integration.

Sowa, J. F.

1976. Conceptual Graphs for a Data Base Interface. *IBM Journal of Research and Development*, 20(4):336–357.

Sowa, J. F.

2000. *Knowledge Representation: Logistical, Philosophical and Computational Foundations.*

Spitters, B. and E. van der Weegen

2011. Type Classes for Mathematics in Type Theory. *Mathematical Structures in Computer Science*, 21(04):795–825.

Spivak, D. I.

2012. Functorial data migration. *Information and Computation*, 217:31–51.

Spivak, D. I. and R. E. Kent

2012. Ologs: a categorical framework for knowledge representation. *PLoS ONE*, 7(1):e24274.

Steedman, M. J.

1977. The perception of musical rhythm and metre. *Perception*, 6(5):555–569.

Sturm, B. L., R. Bardeli, T. Langlois, and V. Emiya

2014. Formalizing The Problem of Music Description. In *Int. Symposium on Music Information Retrieval (ISMIR).*

Tolk, A. and J. A. Muguira

2003. The Levels of Conceptual Interoperability Model.

Tsarkov, D. and I. Horrocks

2006. FaCT++ Description Logic Reasoner: System Description. Pp. 292–297. Springer, Berlin, Heidelberg.

Tymoczko, D.

2011. *A Geometry of Music: Harmony and Counterpoint in the Extended Common Practice*. New York: Oxford University Press.

Vardi, M. Y.

1996. Why is modal logic so robustly decidable? In *Descriptive Complexity and Finite Models: Proceedings of a DIMACS Workshop*, volume 31, Pp. 149–184.

Wadler, P.

1992. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '92*, Pp. 1–14.

Wang, W., A. Tolk, and W. Wang

2009. The Levels of Conceptual Interoperability Model: Applying Systems Engineering Principles to M&S. *SpringSim '09 Proceedings of the 2009 Spring Simulation Multiconference*.

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.

Wiggins, G. and A. Smail

2000. Musical Knowledge: what can Artificial Intelligence bring to the musician? *Readings in Music and Artificial Intelligence*, Pp. 29–46.

Wiggins, G. A.

2000. Computer-Representation of Music in the Research Environment. In *Modern Methods for Musicology: Prospects, Proposals and Realities*, Crawford and Gibson, eds., Pp. 1–13. Oxford: Ashgate.

Wiggins, G. A.

2007. Models of musical similarity. *Musicae Scientiae*, 11(1 Suppl):315–338.

Wiggins, G. A., M. Harris, and A. Smaill

1989. Representing Music for Analysis and Composition. In *Proceedings of the Second Workshop on AI and Music*, M. Balaban, K. Ebcioglu, O. Laske, C. Lischka, and L. Soriso, eds., number 504, P. 10. Menlo Park, CA: AAAI.

Wissmann, J. S.

2012. *Chord Sequence Patterns in OWL*. Phd, City University London.

Woods, W. a. and J. G. Schmolze

1992. The KL-ONE family. *Computers & Mathematics with Applications*, 23(2-5):133–177.

Zimmermann, A., M. Krotzsch, J. Euzenat, and P. Hitzler

2006. Formalizing Ontology Alignment and its Operations with Category Theory. In *Proc. 4th International conference on Formal ontology in information systems (FOIS)*, volume 150, Pp. 277–288.