ASP, Amalgamation and the Conceptual Blending Workflow

ASP, Amalgamation, and the Conceptual Blending Workflow

paper #60

Abstract. We present an amalgamation technique used for *conceptual blending* – a concept invention method that is advocated in cognitive science as a fundamental, and uniquely human engine for creative thinking. Herein, we employ the search capabilities of ASP to find commonalities among input concepts as part of the blending process, and we show how our approach fits within a generalised conceptual blending workflow. Specifically, we orchestrate ASP with imperative programming languages like Python, to query external tools for theorem proving and colimit computation. We evaluate our system with examples from various domains where creativity is important, in particular mathematics and music.

1 Introduction

Creativity is an inherent human capability, that is crucial for the development and invention of new ideas and concepts [2]. This paper addresses a kind of creativity which Boden [2] calls *combinational*, and which has been studied by Fauconnier and Turner [4] in their framework of *conceptual blending*. In brief, conceptual blending is a process where one comes up with a novel concept, called the *blend*, by combining two familiar input concepts in a serendipitous way. For illustration, consider the classical example of blending the concepts *house* and *boat* (e.g. [5, 4]). A possible result is the invention of a *house-boat* concept, where the medium on which a house is situated (land) becomes the medium on which boat is situated (water), and the resident of the house becomes the passenger of the boat. Another possible blend is the *boat-house*, where the boat becomes the resident of the house.

A computational non-monotonic problem of conceptual blending is to find a common ground, called *generic space*, between the two input concepts [4]. For example, the *house-boat* blend has the generic space of a person using an object which is not situated on any medium. Once the generic space has been identified, one can develop possible blends by specialising the generic space with elements from the input concepts in a meaningful way. However, this is not trivial because the naive 'union' of input spaces can lead to inconsistencies. For example, the medium on which an object is situated can not be land and water at the same time. Hence, before combining the input concepts, it is necessary to generalise, and to remove at least one medium assignment. Another problem is the huge number of possible blends, which are often not meaningful. For example, blending *house* and *boat* such that the house becomes the passenger of the boat is not very convincing. In this work, we present a system that can deal with such problems, and that addresses the following question:

"How can we use ASP as a non-monotonic search engine to find a generic space among input concepts, and how can we orchestrate this search process with external tools to produce meaningful blends within a computationally feasible system?"

Towards this, we use a mixed declarative-imperative *amalgams* process known from case-based reasoning [13], which coordinates the generalisation and the meaningful combination of input concepts.

2 Preliminaries

Goguen [5] proposes to model the input concepts of blending as *semiotic systems*, which are essentially *algebraic specifications* extended by priority information about their elements. The main advantage of this approach is being able to provide a general enough, but computational feasible representation, while being able to resolve inconsistencies. We represent semiotic systems by using the Common Algebraic Specification Language (CASL) [12], and extend it by considering priority information for operators, sorts, predicates and axioms as follows:

Definition 1 (Prioritised CASL specification (PCS)). *A prioritised CASL specification (PCS) is a tuple* $\mathfrak{s} = \langle \mathcal{DST}, \mathcal{ST}, \lesssim, \mathcal{DO}, \mathcal{O}, \mathcal{P}, \mathcal{DA}, \mathcal{A}, prio \rangle$ *with:*

- *a set* $\mathcal{DST}$ *of data-sorts and a set* $\mathcal{ST}$ *of sorts, along with a preorder* $\lesssim$ *that defines a sub-sort relationship;*
- *a set* $\mathcal{DO}$ *of data-operators, and a set* $\mathcal{O}$ *of operators* $o : s_1 \times \cdots \times s_n \mapsto s_d$ *that map zero or more objects of argument sorts* $s_1, \cdots, s_n$ *to a range sort* s_r*;*
- *a set* $\mathcal{P}$ *of predicates* $p : s_1 \times \cdots \times s_n$ *that map zero or more objects of argument sorts* $s_1, \cdots, s_n$ *to Boolean values;*
- *a set* $\mathcal{DA}$ *of data-axioms and a set* $\mathcal{A}$ *of axioms;*
- *a function* $prio : \mathcal{DST} \cup \mathcal{ST} \cup \mathcal{DO} \cup \mathcal{O} \cup \mathcal{P} \cup \mathcal{DA} \cup \mathcal{A} \mapsto \mathbb{N}$ *that assigns a priority to all elements in a specification. The priority for data elements is always 0.*

We refer to the listed constituents of a PCS simply as the elements *of a PCS, denoted by* e*, and we say that two PCS are* compatible *if all of their elements, except the priority function, are equal.*

Data elements are used as a fixed shared background theory of the input specifications [5, Def.1], e.g, a theory about integer numbers. We use set-theoretical notation to denote addition and removal of operators, predicates axioms and sorts of a CASL specification as obvious. For example, let e be an element of a specification $\mathfrak{s}$, then we denote the removal or containment of e in $\mathfrak{s}$ by writing $\mathfrak{s} \setminus e$ or $e \in \mathfrak{s}$ respectively.
Goguen [5]'s algebraic view on blending suggests to compute the blend of input specifications as their categorical *colimit* [11]. The colimit requires *morphisms* to be defined between the signatures of algebraic specifications, in particular between the generic space and the input concepts. We define morphisms between PCS signatures similar to [5, Def.2] as follows:

Definition 2 (Morphisms between PCS). *Given two PCS* $\mathfrak{s}_1, \mathfrak{s}_2$*, a morphism* $m : \mathfrak{s}_1 \mapsto \mathfrak{s}_2$ *is a partial surjective function that maps sorts of* $\mathfrak{s}_1$ *to sorts of* $\mathfrak{s}_2$*, operators of* $\mathfrak{s}_1$ *to operators of* $\mathfrak{s}_2$*, predicates of* $\mathfrak{s}_1$ *to predicates of* $\mathfrak{s}_2$*, such that*

- *for all sorts* $s_a, s_b \in \mathfrak{s}_1$*, if* $s_a \lesssim s_b$*, then* $m(s_a) \lesssim m(s_b)$*,*
- *if* $o : s_1, \ldots, s_n \mapsto s_d$ *is an operator of* $\mathfrak{s}_1$*, then, if defined,* $m(o) : m(s_1), \ldots, m(s_n) \mapsto m(s_d)$ *is an operator of* $\mathfrak{s}_2$*,*
- *if* $p : s_1, \ldots, s_n$ *is a predicate of* $\mathfrak{s}_1$*, then, if defined,* $m(p) : m(s_1), \ldots, m(s_n)$ *is a predicate of* $\mathfrak{s}_2$*,*
- m *is an identity mapping for data sorts and data operators in* $\mathfrak{s}_1$*,*

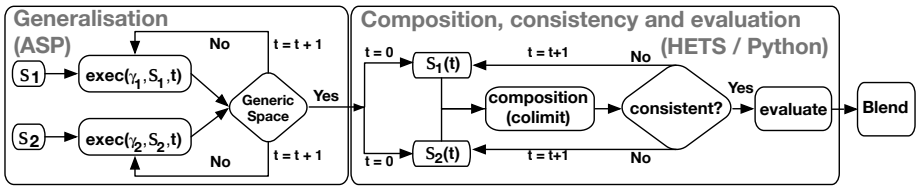

Fig. 1: Amalgamation workflow

3 ASP-driven Blending by Amalgamation

We employ an interleaved declarative-imperative amalgamation process to search for generalisations of input spaces that produce and evaluate consistent blends.

3.1 System Description

The workflow of our system is depicted in Figure 1. First, the input PCS $\mathfrak{s}_1$, $\mathfrak{s}_2$ are translated into ASP facts (see Sec. 3.2). Then, $\mathfrak{s}_1$, $\mathfrak{s}_2$ are iteratively generalised by an ASP solver until a generic space is found. Each generalisation is represented by a fact $exec(\gamma, \mathfrak{s}, t)$, where t is an iterator and γ is a generalisation operator that, e.g., removes an axiom or generalises a sort (see Sec. 3.3). The execution of generalisation operators is repeated until the generalised versions of the input specifications are compatible in the sense of Def.1, i.e., until a generic space is found. We write $\mathfrak{s}(t)$ to denote the t-th generalisation of $\mathfrak{s}$. For example, a first generalisation of the house concept might be the concept of a house that is not situated on any medium. In order to find consistent blends, we apply the category-theoretical *colimit* operation using the HETS toolset [11] to compose generalisations of input specifications. The colimit is applied on different combinations of generalisations, and for each result we query a theorem prover for consistency. An example for a consistent house-boat blend is the combination of the generalised *boat* on the medium water, but without a passenger, and a generalised *house* with a resident, but without a medium.

To eliminate uninteresting blends from our search process, we consider that more promising blends are those that require less generalisations. Consequently, we go from less general generalisations to more general generalisations and stop when a consistent colimit is achieved. Thereafter, the result is evaluated using certain metrics that are inspired by Fauconnier and Turner [4]'s *optimality principles* of blending to assess the quality of the blend (see Sec.3.5). Note that different stable models, and therefore different generalisations, can be found by the ASP solver, which lead to different blends.

3.2 Modelling Algebraic Specifications in ASP

In order to find the generic space and to avoid inconsistencies that arise from the naive combination of input specifications, we relax prioritised CASL specifications using generalisation operators in a step-wise transition process. Generalisation operators modify algebraic specifications by removing operators, sorts, predicates or axioms, or by renaming these elements in a step-wise transition process. In the following, we use t to denote a step-counter that represents the number of modifications made to a specification. We assume sorts, operators and axioms for data as common among input specifications, so that we do not need to consider them in the ASP-based reasoning process. With this, we represent prioritised CASL specifications in ASP as follows:

- For each *sort* s in a specification $\mathfrak{s}$ with and a parent sort s_p we state the facts:

$$sort(\mathfrak{s}, s, t) \tag{1a}$$

$$hasParent(\mathfrak{s}, s, s_p, t) \tag{1b}$$

A fact (1a) assigns a sort s to a specification $\mathfrak{s}$ at a step t, and (1b) assigns a parent sort.

- For each *operator* $o : s_1 \times \cdots \times s_n \mapsto s_r$ in a specification $\mathfrak{s}$ we have:

$$op(\mathfrak{s}, o, t) \tag{2a}$$

$$opHasSort(\mathfrak{s}, o, s_1, 1, t) \cdots opHasSort(\mathfrak{s}, o, s_n, n, t) \tag{2b}$$

$$opHasSort(\mathfrak{s}, o, s_r, rng, t) \tag{2c}$$

Here, and facts (2b),(2c) state the arguments and range sorts of an operator.

- Similarly, for each *predicate* $p : s_1 \times \cdots \times s_n$ in $\mathfrak{s}$ we generate the LP facts:

$$hasPred(\mathfrak{s}, p, t) \tag{3a}$$

$$predHasSort(\mathfrak{s}, p, s_1, 1, t) \cdots predHasSort(\mathfrak{s}, p, s_n, n, t) \tag{3b}$$

- For each *axiom* a with $prio(a) = v_p$ we determine a logical equivalence class of that axiom, denoted eq^a, by passing the axiom to a Python function, which checks for logical equivalence of axioms within all specifications using an external theorem prover. All logically equivalent axioms have the same equivalence class, e.g., $\neg a \vee b$ has the same equivalence class as $a \rightarrow b$. We also determine the elements, i.e, sorts, operators and predicates, that are involved in an axiom. This information are used in the preconditions of removal operators. For example, operator removal has the precondition that there exists no atom that involves the operator. Having computed the equivalence class eq^a and determined n_e elements that are involved in an axiom, we generate the following facts for each axiom a in a specification $\mathfrak{s}$.

$$hasAx(\mathfrak{s}, a, t) \tag{4a}$$

$$axInvolvesElem(\mathfrak{s}, a, e_1, t), \ldots, axInvolvesElem(\mathfrak{s}, a, e_{n_e}, t) \tag{4b}$$

$$axHasEqClass(\mathfrak{s}, a, eq^a, t) \tag{4c}$$

We represent the priority function $prio$ of a PCS as facts $priority(\mathfrak{s}, e, v_p)$ for each element e in a specification $\mathfrak{s}$. Compatibility among two input specifications, as defined in Def. 1, is represented by atoms $incompatible(\mathfrak{s}_1, \mathfrak{s}_2, t)$, which are triggered by additional LP rules if, for $\mathfrak{s}_1$ and $\mathfrak{s}_2$, at step t, (i) sorts or subsort relationships are not equal, or (ii) operator or predicate names are not equal, or (iii) argument and range sorts of operators and predicates are not equal, or (iv) axioms are not equivalent.

3.3 Formalising Generalisation Operators in ASP

For the generalisation of PCS, we consider two kinds of generalisation operators. The first kind involves the removal of an element in a specification, and the second kind involves the renaming of an element. Each generalisation operator is defined via a precondition rule, an inertia rule, and, in case of renaming operations, an effect rule. Preconditions are modelled with a predicate $poss/3$ that states when it is possible to execute a generalisation operation, and inertia is modelled with a predicate $noninertial/3$ that states when an element of a specification stays as it is after the execution of a generalisation operation. Effect rules model how a generalisation operator changes an input specification. We represent the execution of a generalisation operator with atoms $exec(\gamma, \mathfrak{s}, t)$, to denote that a generalisation operator γ was applied to $\mathfrak{s}$ at a step t.

Removal operators. A fact $exec(rm(e), \mathfrak{s}, t)$ denotes the removal of an element e from a specification $\mathfrak{s}$ at a step t. It has different precondition rules for removing axioms (5a), operators (5b), predicates (5c) and sorts (5d):

$$poss(rm(e), \mathfrak{s}, t) \leftarrow ax(\mathfrak{s}, e, t), exOtherSpecWithoutEqivAx(\mathfrak{s}, e, t) \quad (5a)$$

$$poss(rm(e), \mathfrak{s}, t) \leftarrow op(\mathfrak{s}, e, t), exOtherSpecWithoutElem(\mathfrak{s}, e, t), \quad (5b)$$
$$0\{ax(\mathfrak{s}, A, t) : axInvolvesElem(\mathfrak{s}, A, e, t)\}0$$

$$poss(rm(e), \mathfrak{s}, t) \leftarrow pred(\mathfrak{s}, e, t), exOtherSpecWithoutElem(\mathfrak{s}, e, t), \quad (5c)$$
$$0\{ax(\mathfrak{s}, A, t) : axInvolvesElem(\mathfrak{s}, A, e, t)\}0$$

$$poss(rm(e), \mathfrak{s}, t) \leftarrow sort(\mathfrak{s}, e, t), exOtherSpecWithoutElem(\mathfrak{s}, e, t), \quad (5d)$$
$$0\{ax(\mathfrak{s}, A, t) : axInvolvesElem(\mathfrak{s}, A, e, t)\}0$$
$$noOpUsesSort(\mathfrak{s}, e, t), noPredUsesSort(\mathfrak{s}, e, t),$$
$$isNotParentSort(\mathfrak{s}, e, t)$$

The precondition (5a) for removing an axiom from a specification is that an atom $exOtherSpecWithoutEqivAx(\mathfrak{s}, a, t)$ holds. Such atoms are produced, if there exists at least one other specification that does not have an axiom of the same logical equivalence class. For the removal of other elements we have a similar precondition, i.e., $exOtherSpecWithoutElem(\mathfrak{s}, e, t)$, which denotes that an element can only be removed if it is not involved in another specification. Such preconditions are required to allow only generic spaces that are *least general* for all input specifications, in the sense that elements can not be removed if they are contained in all specifications. We also require operators, predicates and sorts not to be involved in any axiom before they can be removed (denoted by $0\{ax(\mathfrak{s}, A, t) : axInvolvesElem(\mathfrak{s}, A, e, t)\}0$). Precondition (5d) for removing sorts has the additional requirement that no operator or predicate with an argument or range of the sort to be removed exists in the specification. Another condition for sort removal is that the sort is not the parent sort of another sort. Consequently, for sort removal, all axioms, operators and predicate that involve the sort must be removed first, and child sorts must also be removed first. The inertia rules for removing elements from a specification are quite simple:

$$noninertial(\mathfrak{s}, e, t) \leftarrow exec(rm(e), \mathfrak{s}, t) \quad (6)$$

$noninertial$ atoms will cause an element e to remain in a specification (see rule (10)).

Renaming operators. A fact $exec(rename(e, e', \mathfrak{s}'), \mathfrak{s}, t)$ denotes the renaming of an element e of a specification $\mathfrak{s}$ to an element e' in a specification $\mathfrak{s}'$. In contrast to removal, renaming can only be applied to predicates, operators and sorts. Axioms are automatically rewritten according to the renamings of the involved elements. Again, we have different preconditions for renaming operators (7a), predicates (7b) and sorts (7c):

$$poss(rename(e, e', \mathfrak{s}'), \mathfrak{s}, t) \leftarrow op(\mathfrak{s}, e, t), op(\mathfrak{s}', e', t), \quad (7a)$$
$$not\ opSortsNotEquivalent(\mathfrak{s}, e, \mathfrak{s}', e', t),$$
$$not\ op(\mathfrak{s}, e', t), not\ op(\mathfrak{s}', e, t), \mathfrak{s} \neq \mathfrak{s}'$$

$$poss(rename(e, e', \mathfrak{s}'), \mathfrak{s}, t) \leftarrow pred(\mathfrak{s}, e, t), pred(\mathfrak{s}', e', t), \quad (7b)$$
$$not\ predSortsNotEquivalent(\mathfrak{s}, e, \mathfrak{s}', e', t),$$
$$not\ pred(\mathfrak{s}, e', t), not\ pred(\mathfrak{s}', e, t), \mathfrak{s} \neq \mathfrak{s}'$$

$$poss(rename(e, e', \mathfrak{s}'), \mathfrak{s}, t) \leftarrow sort(\mathfrak{s}, e, t), sort(spec', e', t), \quad (7c)$$
$$not\ sort(\mathfrak{s}, e', t), not\ sort(\mathfrak{s}', e, t)$$

A common precondition for all three renaming operations is that the element e must exist in the specification $\mathfrak{s}$, and that e' must exist in $\mathfrak{s}'$. Furthermore, it must not be the case that e' is already part of $\mathfrak{s}$, and that e is part of $\mathfrak{s}'$. In case of renaming operators and predicates, the argument and range sorts of e and e' must also be equivalent for the renaming to become possible. For example, an operator $situatedOn : Object \mapsto Medium$ can not be mapped to an operator $usedBy : Object \mapsto Person$, which has a different range sort. The inertia rules for renaming elements e in a specification are analogous to the inertial rule for removing elements:

$$noninertial(\mathfrak{s}, e, t) \leftarrow exec(rename(e, e', \mathfrak{s}'), \mathfrak{s}, t) \quad (8)$$

For renaming, we have the following set of effect rules that assign the new name for the respective element:

$$sort(\mathfrak{s}, e', t+1) \leftarrow exec(rename(e, e', \mathfrak{s}'), \mathfrak{s}, t), sort(\mathfrak{s}, e, t) \quad (9a)$$
$$op(\mathfrak{s}, e', t+1) \leftarrow exec(rename(e, e', \mathfrak{s}'), \mathfrak{s}, t), op(\mathfrak{s}, e, t)$$
$$pred(\mathfrak{s}, e', t+1) \leftarrow exec(rename(e, e', \mathfrak{s}'), \mathfrak{s}, t), pred(\mathfrak{s}, e, t)$$

These rules state, that a specification will contain an element e' at a step $t+1$ if an element e has been renamed to e' at step t.

Inertia. In order to use the inertia rules (6, 8), we need the following rules to state that elements e remain in a specification $\mathfrak{s}$ if they are inertial:

$$sort(\mathfrak{s}, e', t+1) \leftarrow not\ noninertial(\mathfrak{s}, e, t), sort(\mathfrak{s}, e, t) \quad (10a)$$
$$op(\mathfrak{s}, e', t+1) \leftarrow not\ noninertial(\mathfrak{s}, e, t), op(\mathfrak{s}, e, t) \quad (10b)$$
$$pred(\mathfrak{s}, e', t+1) \leftarrow not\ noninertial(\mathfrak{s}, e, t), pred(\mathfrak{s}, e, t) \quad (10c)$$
$$ax(\mathfrak{s}, e', t+1) \leftarrow not\ noninertial(\mathfrak{s}, e, t), ax(\mathfrak{s}, e, t) \quad (10d)$$

Updating Axiom Equivalence. When operators, predicates or sorts that are involved in an axiom are renamed, then the axiom's equivalent class changes. Determining logical equivalence of FOL axioms is a well understood research domain on its own, and we make use of existing theorem proving tools here. Towards this, we use an external Python function $renameEleAndGetNewEqClass$ in rule (11) during the ASP solving process, which updates the equivalence class by querying theorem proving tools that determine a new equivalence class for an axiom if elements are renamed. This happens by accessing an internal database of axioms that is built dynamically during the ASP solving process.

$$axHasEqClass(\mathfrak{s}, a, eq^a_{new}, t+1) \leftarrow axHasEqClass(\mathfrak{s}, a, eq^a, t), \quad (11)$$
$$exec(rename(\mathfrak{s}, e_1, e_2, t), axInvolvesElem(\mathfrak{s}, a, e_1, t), ax(\mathfrak{s}, a, t),$$
$$eq^a_{new} = @renameEleAndGetNewEqClass(eq^a, e_1, e_2)$$

Additional rules that update the $axInvolvesElem$ atoms if elements are renamed are also part of our implementation.

3.4 Generalisation Search Process

The main search process that we use ASP for, is to find a generic space, and generalised versions of the input specifications which lead to a consistent blend. This is done

by successively generating generalisations of the input specifications. A sequence of generalisation operators defines a *generalisation path*.

Definition 3 (Generalisation path). *Let $\mathfrak{s}$ be a specification, let $\{\gamma_1, \ldots, \gamma_n\}$ be the set of generalisation operators for $\mathfrak{s}$ and $t_1 < \cdots < t_n$ be steps. Then we call a set of atoms $P = \{exec(\gamma_1, S, t_1), \cdots, exec(\gamma_n, S, t_n)\}$ a generalisation path of $\mathfrak{s}$.*

Generalisation paths are generated with the following choice rule, that allows one or zero generalisation operations per specification at a time.

$$0\{exec(a, \mathfrak{s}, t) : poss(a, \mathfrak{s}, t)\}1 \leftarrow not\ genericReached(t), spec(\mathfrak{s}). \quad (12)$$

Generalisation paths lead from the input specifications to a generic space, which is a generalised specification that describes the commonalities of the input specifications. A generic space is reached, if the generalised versions of the input specifications are compatible in the sense of Def.1. We use the $incompatible$ predicate introduced in Sec. 3.2, to determine if a generic space has been reached.

$$notGenericReached(t) \leftarrow spec(\mathfrak{s}_1), spec(\mathfrak{s}_2), incompatible(\mathfrak{s}_1, \mathfrak{s}_2, t), \mathfrak{s}_1 \neq \mathfrak{s}_2 \quad (13)$$

$$genericReached(t) \leftarrow not\ notGenericReached(t) \quad (14)$$

Finally, a constraint $\leftarrow notGenericReached(t)$ assures that the generic space is reached in all Stable Models.

3.5 Composition of generalised input spaces

The generalisation part of our framework generates one stable model for each combination of generalisation paths that lead to the generic space. The next step in the amalgamation process is to compose generalised versions of input specifications to generate a candidate blend (see Fig. 1). The key component of this composition process is the categorical colimit [11] of the generalised specifications and the generic space. This requires also the morphisms from the generic space to the input specifications, which are induced by the generalisation path (see Def. 3). Since the colimit of algebraic specification signatures does not consider consistency and priority information, we need to define a composition operation for *prioritised* CASL specification, that is based on the colimit but that also considers priorities and consistency.

Definition 4 (Composition of PCS). *The composition $\mathfrak{c}$ of PCS $\mathfrak{s}_1, \mathfrak{s}_2$, a generic space $\mathfrak{g}$ and total morphisms $m_1 : \mathfrak{g} \mapsto \mathfrak{s}_1$, $m_2 : \mathfrak{g} \mapsto \mathfrak{s}_2$ is defined as follows: Let $\mathfrak{s}_{colimit}$ be the colimit of the PCS, as described in [11], with the morphisms $m_1^c : \mathfrak{s}_1 \mapsto \mathfrak{s}_{colimit}$ and $m_2^c : \mathfrak{s}_2 \mapsto \mathfrak{s}_{colimit}$. Let $prio^1, prio^2$ denote the priority functions of $\mathfrak{s}_1, \mathfrak{s}_2$ respectively. Then the composition $\mathfrak{c}$ of $\mathfrak{s}_1, \mathfrak{s}_2$ is a PCS that is constituted by the colimit $\mathfrak{s}_{colimit}$, enriched with the following priority function for elements e in the composed PCS:*

$$prio(e) = \sum_{(e^s, e) \in m_1} prio^1(e^s) + \sum_{(e^s, e) \in m_2} prio^2(e^s) \quad (15)$$

Hence, to assign the priorities for the elements e in the composition, Equation (15) simply adds up the priorities of the respective source elements e^s in the morphisms.

3.6 Evaluating blends

The next step in the blending process is to evaluate the composition as a whole, according to several factors that reflect the rather informal optimisation criteria proposed by Fauconnier and Turner [4]. Our formal interpretation of these principles considers logical consistency and the following three evaluation metrics:

1. We support blends that keep as much as possible and the most important parts from their input concepts by using the priority information of elements in the input concepts. This supports the *unpacking*, *web* and *integration* principles in [4].
2. We support blends that maximise common relations among input concepts as a means to compress the structure of the input spaces. Relations are made common by appropriate renamings of elements in the input specification. This supports the *vital relations* principles in [4].
3. We support blends where the amount of information from the input specifications is balanced. This supports the *double-scope* property of blends, which is described by Fauconnier and Turner [4] as '... what we typically find in scientific, artistic, and literary discoveries and inventions.'

We now define these metrics formally. The *amount of information* in a PCS is given as:

$$infoValue(\mathfrak{s}) = \sum_{e \in \mathfrak{s}} prio(e) \tag{16}$$

Equation (16) defines the amount of information in a PCS as the sum of the priorities of all of its elements. A measure for the *compression of structure* in a composition $\mathfrak{c}$ with two morphisms $m_1 : \mathfrak{s}_1 \mapsto \mathfrak{c}$ and $m_2 : \mathfrak{s}_2 \mapsto \mathfrak{c}$ is given as:

$$compression(\mathfrak{c}) = \sum_{e \in \mathfrak{c}} : eleComp(e) \qquad \text{where} \tag{17}$$

$$eleComp(e) = \begin{cases} prio(e) & \text{if } \exists e_1^s, e_2^s : (e_1^s, e) \in m_1 \wedge (e_2^s, e) \in m_2 \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

The compression value of the composition $\mathfrak{c}$ is the sum of the compression values of its individual elements (denoted by $eleComp$). The compression value of an individual element is the priority of that element if it has isomorphic counterparts e_1^s, e_2^s in both input specification mappings and 0 if not. For example, consider the predicate $liveIn : Person \times House$ of the $House$ specification and the predicate $ride : Person \times Boat$ of the $Boat$ specification. Both are mapped to the same element in the composition, i.e., the predicate $liveIn : Person \times House$. The $liveIn$ in the composition uses the same symbol as the one in $House$, but it carries more information because due to the renaming it now also represents the $ride$ predicate. We account for this form of compression of information by adding the priority of $liveIn$ to the compression value.

We also account for the *balance of information* from both input specifications. That is, we consider blends to be better where the amount of information from the input specifications is similar. Towards this we define an imbalance penalty as the half of the difference of the amount of information from the input specification as follows:

$$imbalance(\mathfrak{c}) = \frac{abs(infoValue(\mathfrak{s}_1) - infoValue(\mathfrak{s}_2))}{2} \tag{19}$$

Taking only the half of the difference as imbalance penalty turned out to be more useful than taking the full difference, because this still encourages blends which have more information in total, even if they are imbalanced. The final evaluation of a blend is done by summing up the three evaluation metrics and by considering consistency as follows:

$$value(\mathfrak{c}) = \begin{cases} infoValue(\mathfrak{c}) + compression(\mathfrak{c}) - imbalance(\mathfrak{c}) & \text{if } \mathfrak{c} \text{ is consistent} \\ 0 & \text{otherwise} \end{cases} \tag{20}$$

Note that the imbalance penalty can never be bigger than the information value, so that the blend value is always positive.

4 Proof of Concept

To demonstrate our system we consider examples from two real-world domains where creativity is important, namely mathematics and music.

4.1 Lemma invention for theorem proving

As an example, we present a general approach for using blending to exploit existing creative lemmas in a well understood theory, to automatically generate creative lemmas in another less understood theory. For illustration, consider the following prioritised theories of natural numbers and lists:

spec NAT =

sort	*Nat*	p:3
ops	*zero* : *Nat*;	p:2
	s : *Nat* $\rightarrow$ *Nat*	p:3
	sum : *Nat* $\rightarrow$ *Nat*	p:2
	qsum : *Nat* $\times$ *Nat* $\rightarrow$ *Nat*	p:2
	plus : *Nat* $\times$ *Nat* $\rightarrow$ *Nat*	p:1
$\forall$ *x*, *y* : *Nat*		
(0) . *sum*(*zero*) = *zero*		p:2
(1) . *sum*(s(*x*)) = *plus*(s(*x*), *sum*(*x*))		p:2
(2) . *qsum*(s(*x*), *y*) = *qsum*(*x*, *plus*(s(*x*), *y*))		p:2
(3) . *qsum*(*zero*, *x*) = *x*		p:2
(4) . *plus*(*zero*, *x*) = *x*		p:1
(5) . *plus*(s(*x*), *y*) = s(*plus*(*x*, *y*))		p:1
(NT) . *sum*(*x*) = *qsum*(*x*, *zero*)		p:3
(NL) . *plus*(*sum*(*x*), *y*) = *qsum*(*x*, *y*)		p:3

end

spec LIST =

sorts	*El*	p:3
	L	p:3
ops	*nil* : *L*;	p:2
	cons : *El* $\times$ *L* $\rightarrow$ *L*;	p:3
	app : *L* $\times$ *L* $\rightarrow$ *L*;	p:2
	rev : *L* $\rightarrow$ *L*;	p:2
	qrev : *L* $\times$ *L* $\rightarrow$ *L*	p:2
$\forall$ *x*, *y* : *L*; *h* : *El*		
(6) . *rev*(*nil*) = *nil*		p:2
(7) . *rev*(*cons*(*h*, *x*)) = *app*(*rev*(*x*), *cons*(*h*, *nil*))		p:2
(8) . *qrev*(*nil*, *x*) = *x*		p:2
(9) . *qrev*(*cons*(*h*, *x*), *y*) = *qrev*(*x*, *cons*(*h*, *y*))		p:2
(10) . *app*(*nil*, *x*) = *x*		p:1
(11) . *app*(*cons*(*h*, *x*), *y*) = *cons*(*h*, *app*(*x*, *y*))		p:1
(LT) . *rev*(*x*) = *qrev*(*x*, *nil*)		p:3

end

Important elements of these specifications are the constructor operators s and $cons$, so we give them a high priority. Of particular interest here are also the theorems **(NT)** and **(LT)**, which are also given a high priority because they provide important insights about the relation between the tail-recursive functions $qrev$ and $qsum$, and their primitively recursive counterparts rev and sum. Proving such theorems by induction is very hard due to the absence of a universally quantified variable in the second argument of the tail-recursive version [8]. An expert's solution here is to use a lemma that generalises the theorem. An example of such a generalisation is the eureka lemma **(NL)** in the naturals, which we assume to be known in this scenario. Discovering such lemmas is in general a very challenging and well-known problem – see [10, 9] for example. Our goal is to use blending to discover an analogous lemma which facilitates the inductive proof of for **(LT)** in LIST.

Towards this, we first need to find a generic space. However, this is problematic because the constructor $s(n)$ in the naturals is unary, whereas the constructor $cons(h, l)$ in lists is binary. In order to resolve this problem we take inspiration from a classical set theoretic construction of the naturals as the cardinality of a set (see [1] for example) . In our example we can use list to model the notion of set. In this case the theory of the naturals corresponds to a theory of lists of exactly the same element. Moreover there is a generalised motivation to constructing the naturals in this way which is presented

in Sec. 5. We can exploit this to generalise the theory of naturals by adding an extra argument to the successor function, but only computing with the second argument – $s(n)$ becomes $s(c, n)$ where c is some canonical element of a canonical sort C. The step definition of plus, for example, then becomes $plus(s(c, n), m) = s(c, plus(n, m))$ – the computation remains in the second element. This way we obtain a correctly typed generic space by interpreting the renaming and removal operators from the ASP solver, which allows us to associate the constructors of NAT andLIST with the following generalisation paths:

$P_{\text{NAT}} = \{$
$exec(rename(Nat, L, \text{LIST}), \text{NAT}, 0), exec(rename(zero, nil, \text{LIST}), \text{NAT}, 1),$
$exec(rename(C, El, \text{LIST}), \text{NAT}, 2), exec(rename(s, cons, \text{LIST}), \text{NAT}, 3),$
$exec(rename(sum, rev, \text{LIST}), \text{NAT}, 4), exec(rename(qsum, qrev, \text{LIST}), \text{NAT}, 5),$
$exec(rename(plus, app, \text{LIST}), \text{NAT}, 6),$
$exec(rm(4), \text{NAT}, 7), exec(rm(5), \text{NAT}, 8), exec(rm(1), \text{NAT}, 9),$
$exec(rm(2), \text{NAT}, 10), exec(rm(c), \text{NAT}, 11), exec(rm(NL), \text{NAT}, 12)\}$

$P_{\text{LIST}} = \{$
$exec(rm(10), \text{LIST}, 0), exec(rm(11), \text{LIST}, 1), exec(rm(9), \text{LIST}, 2), exec(rm(7), \text{LIST}, 3)\}$

After applying the respective renamings and and removals, a generic space is reached, using the symbols from the $List$ theory. Note, that even though the symbols of the lists theory are used in the generic space, their meaning is now much more general because they map to both, the $List$ and the Nat theory, and represent now analogies between both theories as depicted in Table 1. That is, the generic space is a general theory with

NAT	GENERIC SPACE	LIST
Nat	Constructed datatype	L
C	Constructed datatype element	El
$zero$	terminal element	nil
s	constructor	$cons$
sum	recursive function	rev
$qsum$	tail-recursive function	$qrev$
$plus$	auxiliary function	app

Table 1: The generic space and its mappings to the theories LIST and NAT

sorts for the constructed data types and their elements, a binary constructor, a terminal element, a primitively recursive function, and a tail-recursive function which is defined in terms of the auxiliary function. After finding the generic space, our framework iterates over different combinations of generalised input specifications and computes the colimit. It then checks the colimits consistency and computes the blend value. In this example, the highest composition value for a consistent colimit is 90, where the 4th generalisation of LIST and the 8th generalisation of NAT is used as input. The result is a theory of lists with the newly invented lemma $app(rev(x), y) = qrev(x, y)$ which can be used successfully as a generalisation lemma to prove **(LT)**.

4.2 Harmony invention for music composition

In the music domain, we consider the use of blending to invent novel chord progressions by blending existing ones, and demonstrate how our approach extends the musicological framework proposed in [3]. The authors show how they blend chords to invent novel *cadences* – short chord progressions that can be understood as 'punctuation' within a music piece. While their system is limited to blending single chords, we are able to blend whole chord progressions, as for example the following *perfect cadence* and *Phrygian cadence*, that we represent as algebraic specifications as follows:

spec PERFECTCAD = CHORDPROG **then**

op *c1Perf* : *Chord*	p:10
op *c2Perf* : *Chord*	p:10
. *succ*(*c1Perf*, *c2Perf*)	p:5
. *absNote*(*c1Perf*, 7)	p:2
. *absNote*(*c1Perf*, *11*)	p:3
. *absNote*(*c1Perf*, 2)	p:1
. *absNote*(*c1Perf*, 5)	p:2
. *relNote*(*c1Perf*, 0)	p:3
. *relNote*(*c1Perf*, 4)	p:3
. *relNote*(*c1Perf*, 7)	p:2
. *relNote*(*c1Perf*, *10*)	p:3
. *root*(*c2Perf*) = 0	p:1

end

spec PHRYGCAD = CHORDPROG **then**

op *c1Phryg* : *Chord*	p:10
op *c2Phryg* : *Chord*	p:10
. *succ*(*c1Phryg*, *c2Phryg*)	p:5
. *absNote*(*c1Phryg*, *10*)	p:2
. *absNote*(*c1Phryg*, 1)	p:1
. *absNote*(*c1Phryg*, 5)	p:2
. *relNote*(*c1Phryg*, 0)	p:3
. *relNote*(*c1Phryg*, 3)	p:3
. *relNote*(*c1Phryg*, 7)	p:2
. *root*(*c2Phryg*) = 0	p:1

end

Both specifications are built on a background theory CHORDPROG about chord progressions, which defines the predicate $succ$ to denote the successor relation among chords, the predicate $absNote$ to determine the absolute notes of a chord, and the predicate $relNote$ to determine the notes of a chord relative to the root note. This allows one to define an axiom that defines the relation between absolute and relative notes in the background theory, and that states when a chord is dissonant. Dissonance is is captured via axioms that forbid certain relative note combinations. For example, they express that a chord cannot have a major third (relative note 4) and a minor third (relative note 3) at the same time, i.e., $\forall c : Chord\ .\ \neg(relNote(c, 3) \wedge relNote(c, 4))$. Given a C major key, the Phrygian cadence involves a $B\flat min$ chord followed by a C chord, and the perfect cadence is a $G7$ chord followed by a C. The priorities of the axioms that assign notes to the chords are musicologically justified as described in [3], i.e., the relatives are given a higher priority, and those absolute notes which are salient within the key are also given a higher priority. However, in addition to these axioms, our system also considers the priority of individual chords which are represented as operators $c1Perf$ ($G7$ chord) ,$c2Perf$ (C chord),$c1Phryg$ ($B\flat min$ chord) and $c2Phryg$ (C chord). Our system blends the two cadences and produces a ***Tritone substitution*** cadence as the result with the highest value. The tritone substitution was invented in jazz music decades after the Phrygian and perfect cadence. It takes the $D\flat$ note from the first chord of the Phrygian cadence, specified by $absNote(c1Phryg, 1)$, as root of the first chord of the novel Tritone cadence. The blending also adds the relative seventh of the $G7$ chord of the perfect cadence ($relNote(c1Perf, 10)$), as well as the major third ($relNote(c1Perf, 4)$) and the fifth ($relNote(c1Perf, 7)$) which are present in both chords. The result is a $D\flat 7$ chord as first chord of the Tritone substitution. The system also allows to blend the second chord of one cadence with the first chord of another, so that a novel chord

progression of three notes is produced. Note that, due to the renaming generalisation operators, this can be done on the level of cadences and chord progressions as a whole, and not only on the level of single chords. This makes our system more general than the one proposed by [3], which can only blend single chords.

5 Conclusion

We present a computational approach for conceptual blending where ASP plays a crucial role in finding the generic space and generalised input specifications. We implement the generalisation of algebraic specifications using a transition system semantics of preconditions and postconditions within ASP, which allows us to access generalised versions of the input specifications. These generalised versions of the input specifications let us find blends which are consist. To the best of our knowledge, there exists currently no other blending framework that can resolve inconsistencies and automatically find a generic space, while using a representation language that is similarly expressive as ours. On top of the ASP-based generalisation, we propose metrics to evaluate the quality of blends, based on the cognitive optimality principles by Fauconnier and Turner [4].

A number of researchers in the field of computational creativity have recognised the value of conceptual blending for building creative systems, and particular implementations of this cognitive theory have been proposed [16, 14, 15, 6, 7, 3]. They are, however, mostly limited in the expressiveness of their representation language, and it is in most cases unclear how they deal with inconsistencies and how the generic space is computed. Furthermore, existing approaches lack a sophisticated evaluation to determine formally how 'good' a blend is. An exception is the very sophisticated framework in [14, 15], which also has optimality criteria based on [4]'s theory. However, the authors do not say how to find the generic space automatically and how to deal with inconsistencies.

As future work, we want to generalise our approach do discover creative 'eureka lemmas' in the mathematics to other data structures. For example, a general form of describing a data structure is to define a constructor as $c : list(\tau) \times list(\sigma) \rightarrow \sigma$. This is to say that a constructor can take any number of non-recursive and recursive arguments to form another version of itself. In the example of naturals, the constructor is $s([], [x]) \equiv s(x)$ and for lists $cons([h], [l]) \equiv cons(h, l)$. For binary trees with data at the nodes where the constructor is $t_{::}([h], [l1, l2])$ since there are two recursive arguments. This allows us to find a mapping in the generic space between constructors, and hence to use the techniques expressed in this paper to discover eureka lemmas in new theories. We also want to investigate multi-domain blending of input specifications. For example, blending the theory of lists with chord progressions should result in operations on chord progressions, as for example a reverse operation, which seems to be interesting for applications in automated music composition.

Bibliography

[1] D. Anderson and E. Zalta. Frege, boolos and logical objects. *Journal of Philosophical Logic*, 33:1–26, 2004.

[2] M. A. Boden. Creativity. In M. A. Boden, editor, *Artificial Intelligence (Handbook Of Perception And Cognition)*, pages 267–291. Academic Press, 1996.

[3] M. Eppe, R. Confalonieri, E. MacLean, M. Kaliakatsos, E. Cambouropoulos, M. Schorlemmer, and K.-U. Kühnberger. Computational invention of cadences and chord progressions by conceptual chord-blending. In *IJCAI*, 2015 (to appear).

[4] G. Fauconnier and M. Turner. *The Way We Think: Conceptual Blending And The Mind's Hidden Complexities*. Basic Books, 2002. ISBN 978-0-465-08785-3.

[5] J. Goguen. An introduction to algebraic semiotics, with application to user interface design. *Computation for metaphors, analogy, and agents*, pages 1–39, 1999.

[6] J. A. Goguen and D. F. Harrell. Style: A computational and conceptual blending-based approach. In S. Argamon, K. Burns, and S. Dubnov, editors, *The Structure of Style: Algorithmic Approaches to Understanding Manner and Meaning*, pages 291–316. Springer, 2010. ISBN 978-3-642-12336-8. doi: 10.1007/978-3-642-12337-5_12.

[7] M. Guhe, A. Pease, A. Smaill, M. Martínez, M. Schmidt, H. Gust, K.-U. Kühnberger, and U. Krumnack. A computational account of conceptual blending in basic mathematics. *Cognitive Systems Research*, 12(3-4):249–265, 2011. doi: 10.1016/j.cogsys.2011.01.004.

[8] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996.

[9] M. Johansson, L. Dixon, and A. Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, 47:251–289, 2011.

[10] O. Montano-Rivas, R. McCasland, L. Dixon, and A. Bundy. Scheme-based synthesis of inductive theories. In *MICAI*, volume 6437 of *LNCS*, pages 348–361, 2010.

[11] T. Mossakowski. Colimits of order-sorted specifications. In *Recent trends in algebraic development techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 316–332. Springer, Berlin, 1998.

[12] P. D. Mosses. *CASL Reference Manual – The Complete Documentation of the Common Algebraic Specification Language*. Springer, 2004.

[13] S. Ontañón and E. Plaza. Amalgams: A formal approach for combining multiple case solutions. In I. Bichindaritz and S. Montani, editors, *Case-Based Reasoning. Research and Development, ICCBR*, pages 257–271. Springer, 2010.

[14] F. C. Pereira. *A Computational Model of Creativity*. PhD thesis, Universidade de Coimbra, 2005.

[15] F. C. Pereira. *Creativity and Artificial Intelligence: A Conceptual Blending Approach*. Mouton de Gruyter, 2007.

[16] T. Veale and D. O. Donoghue. Computation and blending. *Cognitive Linguistics*, 11(3-4):253–282, 2000. doi: 10.1515/cogl.2001.016.