ASP, Amalgamation, and the Conceptual Blending Workflow

Manfred Eppe[1 4], Ewen Maclean[2], Roberto Confalonieri[1], Oliver Kutz[3], Marco Schorlemmer[1], and Enric Plaza[1]

[1] IIIA-CSIC, Spain,
{eppe,confalonieri,marco,enric}@iiia.csic.es
[2] University of Edinburgh, UK,
emaclea2@inf.ed.ac.uk
[3] Free University of Bozen-Bolzano, Italy,
oliver.kutz@unibz.it
[4] International Computer Science Institute, Berkeley, USA,
eppe@icsi.berkeley.edu

Abstract. We present a framework 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 Python programming, to query external tools for theorem proving and colimit computation. We exemplify our approach with an example of creativity in mathematics.

1 Introduction, Preliminaries and Motivation

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 [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 combines two input concepts to invent a new one, called the *blend*.
As a classical example of blending, consider the concepts *house* and *boat* (e.g. [7, 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 inhabitant of the house becomes the passenger of the boat. A sub-task of conceptual blending is to find a common ground, called *generic space*, between the 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. 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.
Finding the generic space of two concepts is a non-monotonic search problem, and it is well-known that *Answer Set Programming* (ASP) (see e.g. [5]) is a successful

tool to cope with such problems. In this paper, we present a computational framework for blending, that addresses the following question: *"How can we use ASP as a non-monotonic search engine to find a generic space of 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 [14], which coordinates the generalisation and combination of input concepts.

Concept Blending as Colimit of Algebraic Specifications. Goguen [7] proposes to model the input concepts of blending as *algebraic specifications* enriched by priority information about their elements, which he calls *semiotic systems*. This algebraic view on blending suggests to compute the blend of input specifications as their categorical *colimit* – a general unification operation for categories, similar to the *union* operation for sets. In our case the colimit unifies algebraic signatures (see [12, 17] for category theoretical details). We represent semiotic systems by using the Common Algebraic Specification Language (CASL) [13]. CASL allows us to state first-order logical specifications, which consists of four kinds of elements, namely *sorts*, *operators*, *predicates* and first order logical *axioms*. Operators are functions that map a list of arguments of a certain sort to a range sort, and predicates are functions that map arguments to boolean values. Such a representation language lets us define more than just concepts, namely full first order theories. As an example, consider the following specifications that represent the mathematical theories of natural numbers and lists.

spec NAT =

sort	Nat	p:3
ops	$zero : Nat$;	p:2
	$s : Nat \to Nat$	p:3
	$sum : Nat \to Nat$	p:2
	$qsum : Nat \times Nat \to Nat$	p:2
	$plus : Nat \times Nat \to Nat$	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
	$cons : El \times L \to L$;	p:3
	$app : L \times L \to L$;	p:2
	$rev : L \to L$;	p:2
	$qrev : L \times L \to L$	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

For example, in LIST, the operator $cons$ maps an object of the sort El (element) and an object of the sort L (list) to an object of the sort L. That is, $cons$ constructs lists by appending one element to a list. The rev operator is a recursive reverse function on lists, and the $qrev$ is a tail-recursive version of the reverse function. Similarly, in NAT, s is a successor function, sum denotes a recursive function to obtain the cumulative sum of a number (e.g. $sum(3) = 1 + 2 + 3 = 6$), and $qsum$ is a tail-recursive version of sum. We enrich CASL specifications by considering priority information for the individual

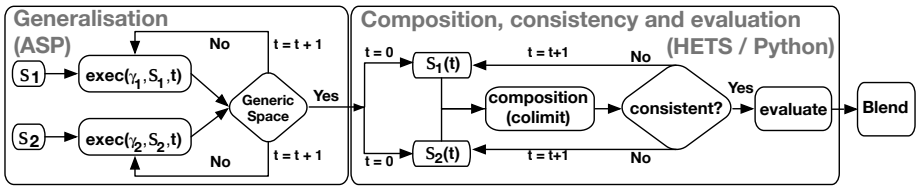

Fig. 1: Amalgamation workflow

elements. We denote such specifications as *prioritised CASL specifications* (PCS). For example, the 'p:3' behind the $cons$ operator declaration denotes that $cons$ has a relatively high priority of 3, and analogously for the other operators, sorts and axioms.

Motivating Example – Discovering Eureka Lemmas by Blending. Of particular interest in the above theories are **(NT)** and **(LT)**. These theorems state that the recursive functions sum and rev are equivalent to the computationally less expensive tail-recursive quick-functions $qsum$ and $qrev$. 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 [9]. 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 a challenging well-known problem (see e.g. [11, 10]), and we demonstrate how blending is used to obtain an analogous eureka lemma for lists as an example application.

2 ASP-driven Blending by Amalgamation

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

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. Then, $\mathfrak{s}_1, \mathfrak{s}_2$ are iteratively generalised by an iterative 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 renames a sort, as described below. The execution of generalisation operators is repeated until the generalised versions of the input specifications have the same sorts, operators, predicates and axioms, 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* [12] 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 logical consistency. To eliminate uninteresting blends from our search process, we consider that the more promising blends 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 so-called *optimality principles* of blending to assess the quality of the blend (due to lack of space, we refer to the literature for details on those principles). Note that different stable models, and therefore different generalisations, can be found by the ASP solver, which lead to different blends.

Modelling algebraic specifications in ASP. First, we translate PCS to ASP facts, with atoms like $sort(\mathfrak{s}, s, t)$ that denote that s is a sort of the specification $\mathfrak{s}$ at a step

t. Operators and predicates are declared similarly. Arguments of operators are defined by atoms $opHasSort(\mathfrak{s}, o, s_i, i, t)$ that denote that an operator o in a specification $\mathfrak{s}$ has the sort s_i as i-th argument. For each element e in a PCS specification $\mathfrak{s}$, we represent its priority v_p as a fact $priority(\mathfrak{s}, e, v_p)$.

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, denoted by rm predicates, and the second kind involves the renaming of an element, denoted by $rename$ predicates. 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. Each generalisation operator is defined via a precondition 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 effect rules model how a generalisation operator changes an input specification. For example, the preconditions for removing and renaming operators are specified by the following rules:

$$poss(rm(e), \mathfrak{s}, t) \leftarrow op(\mathfrak{s}, e, t), exOtherSpecWithoutElem(\mathfrak{s}, e, t), \quad 0\{ax(\mathfrak{s}, A, t) : axInvolvesElem(\mathfrak{s}, A, e, t)\}0 \tag{1a}$$

$$poss(rename(e, e', \mathfrak{s}'), \mathfrak{s}, t) \leftarrow op(\mathfrak{s}, e, t), op(\mathfrak{s}', e', t), not\ op(\mathfrak{s}, e', t), not\ op(\mathfrak{s}', e, t), \quad not\ opSortsNotEquivalent(\mathfrak{s}, e, \mathfrak{s}', e', t), \mathfrak{s} \neq \mathfrak{s}' \tag{1b}$$

For the removal of elements we have a condition $exOtherSpecWithoutElem(\mathfrak{s}, e, t)$, 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 $0\{ax(\mathfrak{s}, A, t) : axInvolvesElem(\mathfrak{s}, A, e, t)\}0$). We also need rules to state when elements remain in a specification. This is expressed via $noninertial/3$ atoms as follows, where (2c) is an examplary case of operator elements of a specification.

$$noninertial(\mathfrak{s}, e, t) \leftarrow exec(rm(e), \mathfrak{s}, t) \tag{2a}$$

$$noninertial(\mathfrak{s}, e, t) \leftarrow exec(rename(e, e', \mathfrak{s}'), \mathfrak{s}, t) \tag{2b}$$

$$op(\mathfrak{s}, e, t+1) \leftarrow not\ noninertial(\mathfrak{s}, e, t), op(\mathfrak{s}, e, t) \tag{2c}$$

For renaming, we also have effect rules that assign the new name for the respective element. For example, for renaming operators we have:

$$op(\mathfrak{s}, e', t+1) \leftarrow exec(rename(e, e', \mathfrak{s}'), \mathfrak{s}, t), op(\mathfrak{s}, e, t) \tag{3}$$

Generalisation search process. ASP is employed 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*, which is generated with the following choice rule:

$$0\{exec(a, \mathfrak{s}, t) : poss(a, \mathfrak{s}, t)\}1 \leftarrow not\ genericReached(t), spec(\mathfrak{s}). \tag{4}$$

Generalisation paths lead from the input specifications to a generic space, which is a generalised specification that describes the commonalities of the input specifications. $genericReached(t)$ atoms determine if a generic space has been reached. This is the case if for two specifications $\mathfrak{s}_1$ and $\mathfrak{s}_2$, at step t, (i) sorts are equal, (ii) operator and

predicate names are equal, and (iii) argument and range sorts of operators and predicates are equal, and (iv) axioms are equivalent.

Composition and evaluation. The next step in the amalgamation process depicted in Figure 1 is to compose generalised versions of input specifications to generate a candidate blend. The key component of this composition process is the categorical colimit [12] of the generalised specifications and the generic space. The colimit is then enriched with the priority information, which we compute as the sum of the priorities of the input elements. The composition is then evaluated according to several factors that reflect the rather informal optimality principles proposed by Fauconnier and Turner [4]. Our formal interpretation of these principles considers logical consistency and the following three evaluation metrics which are based on Fauconnier and Turner [4]'s informal descriptions of certain optimality principles for blending:

a) We support blends that keep as much as possible from their input concepts by using the priority information of elements in the input concepts. This corresponds to *unpacking*, *web* and *integration* principles. Towards this, we compute the *amount of information* in a blend as the sum of the priorities of all of its elements.

b) 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 corresponds to the *vital relations* principle. Maximising common relations raises the *compression of structure* in a composition, which is computed as the sum of priorities of elements in the composition that have counterparts in both input specifications. 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.

c) We support blends where the amount of information from the input specifications is balanced. This corresponds to 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.' Towards this, we consider a *balance penalty* of a blend, which we define as the difference between the amount of information from the input specifications as described in a).

Proof of Concept – Lemma Invention for Theorem Proving. To perform the blend of the theories of naturals and lists discussed in Section 1, our system first generates a generic space, which is achieved with the following generalisation paths:[5]

[5] Note that for this example, we extend the unary constructor $s(n)$ in the naturals by an additional canonical argument c, so that the constructor becomes binary, i.e., $s(c, n)$. This is valid when considering a classical set theoretic construction of the naturals as the cardinality of a set (see [1] for example), where the theory of the naturals corresponds to a theory of lists of the same element.

$P_{\text{NAT}} = \{exec(rename(Nat, L, \text{LIST}), \text{NAT}, 0), exec(rename(zero, nil, \text{LIST}), \text{NAT}, 1),$
$exec(rename(C, El, \text{LIST}), \text{NAT}, 2), exec(rename(s, cons, \text{LIST}), \text{NAT}, 3), \cdots$
$exec(rm(1), \text{NAT}, 9), exec(rm(2), \text{NAT}, 10), exec(rm(c), \text{NAT}, 11), exec(rm(\textbf{NL}), \text{NAT}, 12)\}$

$P_{\text{LIST}} = \{exec(rm(10), \text{LIST}, 0), \cdots, exec(rm(7), \text{LIST}, 3)\}$

With this generalisation path, the sort L is mapped to the sort Nat, the terminal elements nil and $zero$ are mapped to each other, the construction operator s is mapped to $cons$, rev is mapped to sum, $qrev$ is mapped to $qsum$, and app is mapped to $plus$. Note, that the meaning of the $List$-symbols is now much more general because they map to both, the $List$ and the Nat theory, and represent now analogies between both theories. 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)**.

3 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 logically consistent. 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 implementation, 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 [18, 15, 16, 6, 8, 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 [15, 16], 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.

A prototypical implementation of our system can be accessed at `https://github.com/meppe/Amalgamation`. It will be a core part of the bigger computational concept invention framework that is currently being built within the COINVENT project `http://www.coinvent-project.eu`.

Acknowledgements. This work is supported by the 7th Framework Programme for Research of the European Commission funded COINVENT project (FET-Open grant number: 611553). M. Eppe is supported by the German Academic Exchange Service.

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] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice*. Morgan and Claypool, 2012.

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

[8] 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.

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

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

[11] 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.

[12] 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.

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

[14] 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.

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

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

[17] B. Pierce. *Basic category theory for computer scientists*. MIT Press, 1991. ISBN 0262660717.

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