Towards a Computational Framework for Function-Driven Concept Invention

Chapter · July 2016

DOI: 10.1007/978-3-319-41649-6_21

3 authors, including:

Nico Potyka
Universität Osnabrück
34 PUBLICATIONS **64** CITATIONS
SEE PROFILE

Danny Gomez-Ramirez
Universität Osnabrück
6 PUBLICATIONS **2** CITATIONS
SEE PROFILE

Towards a Computational Framework for Function-Driven Concept Invention

Nico Potyka, Danny Gómez-Ramírez, Kai-Uwe Kühnberger

Institute of Cognitive Science, University of Osnabrück

Abstract. We propose a novel framework for computational concept invention. As opposed to recent implementations of Fauconnier's and Turner's Conceptual Blending Theory, our framework simplifies computational concept invention by focusing on concepts' functions rather than on structural similarity of concept descriptions. Even though creating an optimal combination of concepts that achieves the desired functions is NP-complete in general, some interesting special cases are tractable.

1 Introduction

Despite the success of many AI applications, tools, and services, there are some cognitive abilities and phenomena which are hard to model with computational approaches. Classical examples for such shortcomings are creative abilities of cognitive agents, in particular, the ability to create new concepts with new and interesting properties and features based on the available background knowledge of the agent. Particularly, artificial general intelligence (AGI) has an interest to address the problem of developing computational models for certatin aspects of creativity research.

A cognitive theory addressing possibilities to model the invention of new concepts is conceptual blending [6]: In [15], the authors argue for the broad applicability of computational approaches for conceptual blending in the context of concept invention. Conceptual blending can be regarded as the process of combining elements of at least two distinct concepts (or *input spaces*) to get a meaningful new concept (the *blend space*) [6]. For instance, in greek mythology, a centaur is composed of parts of a horse and parts of a human; a faun is composed of a goat and a human. Similar examples can be found in other areas such as mathematics [8] or music [4].

The process of conceptual blending that generates a blend space from given input spaces has been formalized by using tools from various areas such as frame-based knowledge representation [13], algebra [7], quantum theory [1] or analogical reasoning [10]. Many interesting blends can be explained and sometimes automatically generated by these frameworks, but unsupervised blending of concepts can also yield many meaningless results. Quality criteria can be defined to filter the results, but if concept descriptions become large, this approach becomes impractical due to the combinatorial explosion of possible blends. In particular logical approaches that search for maximal consistent blends rely on computational problems that go far beyond NP or are even undecidable.

The authors in [9] propose a goal-oriented view on conceptual blending to further structure the search space and demonstrate the computational benefits with case studies in story generation and pretend play. The three essential components for efficient computation are (see [9] for a detailed discussion):

1. selection of input spaces,
2. selection of elements that should be incorporated in the blend space,
3. stopping criteria for blend elaboration.

Note that the workflow of concept generation will usually apply these components multiple times. Candidate input spaces will be selected (1), combined (2), and evaluated until an acceptable new concept is generated (3). We will follow this philosophy here and assume that our goals can be defined as *functions* that our newly generated concept should satisfy. This view might be less suitable for concept invention in domains like music or poetry, but is well apt to create new physical entities like a houseboat [13] or a monster [12].

In our computational framework, functions will be defined globally and can be satisfied by concepts' *functional units*. By a functional unit, we mean a subset of parts of a concept that achieve a function. For instance, feet, legs, and the pelvis area constitute humans' functional unit for moving the body; hands, arms, and the thorax area humans' functional unit for moving things. In particular, we associate functions with properties that can be used to evaluate the usefulness of functional units. For instance, the functional unit for moving the body can have a property *speed*, the functional unit for moving things can have the property *power*. Roughly speaking, we implement component (1) from [9] by scanning the database for concepts that contribute to satisfying the desired functions. Component (2) and (3) basically consist of combining functional units and evaluating the generated concept with respect to some utility functions. Combination can be guided, for instance, by evaluating functional units' properties while balancing the candidates' contributions to the blend space.

2 Concept Representation

We start our discussion by introducing a functional concept blending framework formally. To this end, let us consider languages $\mathcal{L}_C$ of concepts, $\mathcal{L}_F$ of functions that are associated with concepts and $\mathcal{L}_P$ of properties that functions can have. In our examples, $\mathcal{L}_C$, $\mathcal{L}_F$ and $\mathcal{L}_P$ will be made up of strings like *human* (concept), *move_body* (function) and *speed* (property). Each property p is associated with a domain domain(p) of values it can take. Values can be strings, numerical values, or intervals. For instance, we could specify domain(*speed*) $= \{[b_1, b_2] \mid b_1, b_2 \in \mathbb{R}, 0 \leq b_1 \leq b_2\}$. Intuitively, domain(*speed*) is a set of intervals that represent minimum and maximum speed that a concept can take. We also need some mappings that connect our languages.

– functions : $\mathcal{L}_C \rightarrow 2^{\mathcal{L}_F}$, parents : $\mathcal{L}_C \rightarrow 2^{\mathcal{L}_C}$, components : $\mathcal{L}_C \rightarrow 2^{\mathcal{L}_C}$ associate concepts with the functions they fulfill, with their parents and

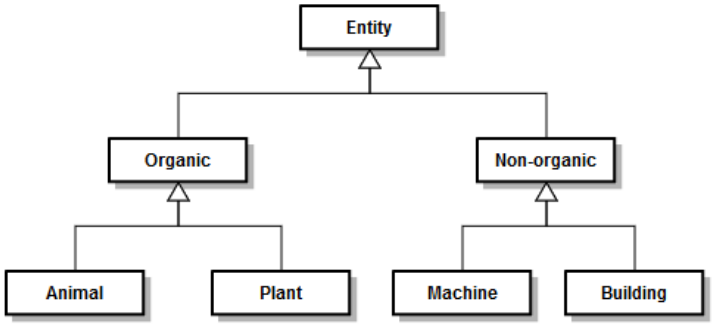

Fig. 1. Hierarchy of abstract concepts.

components. Intuitively, parents correspond to more general concepts (in ontology research often called superconcepts) and serve to arrange concepts hierarchically in a tree-like structure, where child concepts inherit features of their parents. Formally, we demand that

If $p \in \text{parents}(c)$ for some concept c, then $\text{functions}(p) \subseteq \text{functions}(c)$.

For instance, Figure 1 shows a hierarchy, in which the concept Organic might contain functional units like a reproduction system and a metabolic system, which can be overwritten by its childs. Components (in ontology research often associated with concepts standing in the `part_of` relation) are the building blocks that concepts are made of and that can be used to create new concepts in our framework. For instance, we could let $\text{functions}(human) = \{move\_body\}$, $\text{parents}(human) = \{animal\}$, $\text{components}(human) = \{human\_lower\_body, human\_upper\_body, human\_head\}$.

- $\text{funit} : (\mathcal{L}_C \times \mathcal{L}_F) \to 2^{\mathcal{L}_C}$ is a partial mapping that associates concepts and functions with the components that serve to fulfill this function such that
  1. if $f \notin \text{functions}(c)$, then $\text{funit}(c, f) = \bot$ is undefined,
  2. if $f \in \text{functions}(c)$, then $\text{funit}(c, f) \subseteq \text{components}(c)$.
- $\text{eval} : (\mathcal{L}_C \times \mathcal{L}_F \times \mathcal{L}_P) \to \mathcal{L}_V$ is a partial mapping that associates concepts, functions and properties with the value that this property takes for the given function and concept. For instance, we could have $eval(human, move\_body, speed) = [0, 45]$ We demand that
  1. if $f \notin \text{functions}(c)$, then $\text{eval}(c, f, p) = \bot$,
  2. if $f \in \text{functions}(c)$ and $p \notin \text{properties}(f)$, then $\text{eval}(c, f, p) = \bot$,
  3. if $f \in \text{functions}(c)$ and $p \in \text{properties}(f)$, then $\text{eval}(c, f, p) \in \text{domain}(p)$.

Definition 1 (Functional Concept Blending Framework). *A* functional concept blending framework *is a pair* $(\mathcal{C}, \mathcal{F})$*, where* $\mathcal{C} \subseteq \mathcal{L}_C$ *is a set of concepts and* $\mathcal{F} \subseteq \mathcal{L}_F$ *is a set of functions.*

For a discussion of Definition 1 and its relation to classical types of concept blending, please compare Section 4.

Input: Goals $\mathcal{G}$, Concepts $\mathcal{C}'$
Output: Components of new concept
$components \leftarrow \emptyset$;
for $f \in \mathcal{G}$ **do**
  $c \leftarrow \text{bestSatisfies}(\mathcal{C}', f)$;
  $components \leftarrow components \cup \text{funit}(c, f)$;
return *components*;

Algorithm 1: A local algorithm to combine concepts' components.

3 Computing Blends

Given a functional concept blending framework $(\mathcal{C}, \mathcal{F})$, we want to create new concepts. Following [9], our search for interesting candidates to combine will be lead by goals. Our goals are functions that the new concept must satisfy.

The most straightforward way to set the goals is to just let the user select the desired functions. However, if $(\mathcal{C}, \mathcal{F})$ is large in terms of the number of concepts and functions, it is easier to define the desired functions in an implicit way by exploiting the hierarchical structure of the concepts. For instance, given the example from Figure 1, we could just say that we want a concept that features aspects of Organic and Machine and that can move its body in the air. In general, if we select an abstract concept from the hierarchy as a goal, the newly generated concept should satisfy all functions that the abstract concept and its parents satisfy.

So let us assume that we are given a functional concept blending framework $(\mathcal{C}, \mathcal{F})$ and a set of goals $\mathcal{G} \subseteq \mathcal{F}$. Our aim is an algorithm that outputs a new and meaningful concept that fulfills our goals. Similar to [9], we consider three components of the algorithm. Roughly speaking it works by iterating over the database and selecting candidate concepts (1), combining these concepts to a new concept (2), evaluating the concept and maybe restarting the procedure (3). These phases cannot be considered independently of each other. In particular, both the first and second phase depend on the third one. We will first describe combination of concepts and then selection of concepts. Along the way, we will explain how we can evaluate candidates during these phases.

3.1 Concept Combination

Suppose we already selected a subset of components $\mathcal{C}' \subseteq \mathcal{C}$ from which we want to create a new concept. The main task is to select a subset of $\bigcup_{c \in \mathcal{C}'} \text{components}(c)$ that makes up the new concept, let us just call this set *components* . The basic requirement for *components* in our framework is that for each goal $f \in \mathcal{G}$, *components* contains a functional unit that fulfills this function. Algorithm 1 shows a simple algorithm that guarantees this requirement. It just iterates over the concepts and chooses the components of that functional unit that best satisfies the desired function. One naive way to implement bestSatisfies is to select a concept that fulfills the function randomly.

Input: Goals $\mathcal{G}$, Concepts $\mathcal{C}'$
Output: Components of new concept
for $f \in \mathcal{G}$ **do**
  $C_f \leftarrow \{c \in \mathcal{C}' \mid f \in \text{functions}(c)\}$;
$maxUtil \leftarrow -\infty$;
$best \leftarrow \emptyset$;
for *each assignment* $\pi : \mathcal{G} \to \mathcal{C}'$ *such that* $\pi(f) \in C_f$ **do**
  $components \leftarrow \emptyset$;
  for $f \in \mathcal{G}$ **do**
    $components \leftarrow components \cup \text{funit}(\pi(f), f)$;
  $u \leftarrow \text{utility}(\mathcal{G}, \mathcal{C}', components, \pi)$;
  if $u > maxUtil$ **then**
    $best \leftarrow \emptyset$;
    $maxUtil \leftarrow u$;
  if $u = maxUtil$ **then**
    $best \leftarrow best \cup \{components\}$;
return selectBestComponenSet($best$);

Algorithm 2: A global algorithm to combine concepts' components.

A more sophisticated way is to select the concept that best satisfies the function with respect to some *utility* function. To this end, let us assume that for each $f \in \mathcal{F}$ and for each property $p \in \text{properties}(f)$, we have a utility function $u_{f,p} : \text{domain}(p) \to \mathbb{R}$. For instance, for the property speed, we could let $U([b_1, b_2]) = b_2$ be the maximum speed of the concept. Additionally, we might want to reward *diversity* of the new concept. We can do this, for instance, by computing the ratio that the given concept already contributes to the new concept $\text{contribution}(c) = \frac{|\text{components}(c) \cap components|}{|components|}$ and preferring low contributions by multiplying the utility value by a factor like $\frac{3}{1+2^{\text{contribution}(c)}}$. Let $U(\mathcal{C}, \mathcal{G})$ denote the cost of evaluating the utility functions, then bestSatisfiesruns in time $O(|\mathcal{C}'| \cdot U(\mathcal{C}, \mathcal{G}))$. Note that instead of storing the components in *components* explicitly, we can just store a pointer to the fulfilling functional unit for each goal. Then Algorithm 1 runs in time $O(|\mathcal{G}| \cdot |\mathcal{C}'| \cdot U(\mathcal{C}, \mathcal{G}))$.

If we think of our goal as combining concepts' components in a way that maximizes utility, Algorithm 1 corresponds to a local approach that takes only the individual values of concepts' functional units into account, but not their group values. In particular, the influence of the diversity factor depends on the order in which we selected the goals. Algorithm 2 shows an alternative global algorithm. It works by first determining for each goal $f \in \mathcal{G}$ the set of concepts C_f that fulfill this function. Then for each possible assignment $\pi : \mathcal{G} \to \mathcal{C}'$, which determines which goal will be satisfied by which concept's functional unit, the utility value of the corresponding combined concept is determined. From the best component sets, one set is returned. A simple implementation can just select a best component set randomly. It might also be reasonable to add all elements from the intersection $\bigcap_{C \in best} C$ of all best component sets. In order to implement $\text{utility}(\mathcal{G}, \mathcal{C}', components, \pi)$, we need several ingredients. We can evaluate

Input: Concept Blending Framework $(\mathcal{C}, \mathcal{F})$, Goals $\mathcal{G}$
Output: New concept that satisfies goals
$\mathcal{C}' \leftarrow \emptyset$;
while $\mathcal{G} \neq \emptyset$ **do**
  for $c \in \mathcal{C}$ **do**
    if functions$(c) \cap \mathcal{G} \neq \emptyset$ **then**
      $\mathcal{C}' \leftarrow \mathcal{C}' \cup \{c\}$;
      $\mathcal{G} \leftarrow \mathcal{G} \setminus$ functions(c);
$c \leftarrow$ combine$(\mathcal{C}')$;
return$c$;

Algorithm 3: A local algorithm to select concepts.

the usefulness of each individual functional component by using utility functions like in the local approach. We can then combine these individual values by a combination function like a weighted sum. Additionally, we can consider a diversity factor again. In a good combination of concepts, each concept should contribute equally to the new concept. Let us consider again the contribution $\text{contribution}(c) = \frac{|\text{components}(c) \cap components|}{|components|}$ for each concept $c \in \mathcal{C}'$. We want to prefer concepts in which the contributions are distributed more uniformly. We can do this, for instance, by measuring the entropy of the contribution distribution or by measuring its negative euclidean distance to the uniform distribution; and then multiplying this value to the combined utility value. Letting again $U(\mathcal{C}, \mathcal{G})$ denote the maximum cost of evaluating the utility values, Algorithm 2 runs in time $O(|\mathcal{C}'|^{|\mathcal{G}|} \cdot (|\mathcal{G}| + U(\mathcal{C}, \mathcal{G})))$. The runtime is dominated by evaluating all possible assignments π and the worst-case is obtained when all concepts fulfill all functions. However, even if we assume that each function is fulfilled by only 2 concepts, the runtime remains exponential in $|\mathcal{G}|$. Hence, Algorithm 2 is only efficient in its naive form if the number of goals is moderate.

3.2 Selecting Concepts

In order to select concepts from which the new concept shall be generated, we can again follow a local and a global approach. Algorithm 3 shows a simple local algorithm. It works by iterating over the concepts until a concept is found that contributes to the goals. The concept is then added to $\mathcal{C}'$ and all fulfilled goals are removed. This process keeps on until all goals are satisfied. The selected concepts are then combined as explained in the previous section. Algorithm 3 runs in time $O(|\mathcal{G}| \cdot |\mathcal{C}| \cdot |\mathcal{F}| + M(|\mathcal{C}'|)) = O(|\mathcal{C}| \cdot |\mathcal{F}|^2 + M(|\mathcal{C}'|))$, where $M(|\mathcal{C}'|)$ corresponds to the cost of merging the selected concepts. So, for instance, when combining the local algorithms 1 and 3, the overall cost is $O(|\mathcal{C}| \cdot |\mathcal{F}|^2 + |\mathcal{G}| \cdot |\mathcal{C}'| \cdot U(\mathcal{C}, \mathcal{G})) = O(|\mathcal{F}|^2 \cdot |\mathcal{C}| \cdot U(\mathcal{C}, \mathcal{G}))$ and hence polynomial. Note that a naive implementation of Algorithm 3 might yield an old rather than a new concept if there exists a concept that satisfies all goals. However, we can easily handle this case by adding a simple check and if necessary resetting the goals and adding a second concept.

```
Input: Concept Blending Framework (C, F), Goals G
Output: New concept that satisfies goals
maxUtil ← −∞;
best ← ∅;
for k ← 2 to |C| do
    for each k-elementary subset C′ of C do
        c ← combine(C′);
        u ← utility(G, c);
        if u > maxUtil then
            best ← ∅;
            maxUtil ← u;
        if u = maxUtil then
            best ← best ∪ {c};
return select(best);
```

Algorithm 4: A global algorithm to select concepts.

We can modify Algorithm 3 in various ways to iterate over the concepts in more sophisticated ways. One way to do this is to order the concepts in descending order by the key $key(c) = |\,\text{functions}(c) \cap \mathcal{G}|$ for each concept $c \in \mathcal{C}'$. In this way, we prefer concepts that satisfy many goals. The additional computational cost is $O(|\mathcal{C}| \cdot \log |\mathcal{C}|)$ when using efficient sorting algorithms like Quicksort. However, after selecting the first concept, the keys can decrease and the order can change. We can just ignore this fact, resort the concepts periodically or maintain the order by using special data structures like heaps. If we assume that the goals can be satisfied by a small number of concepts, the additional cost is asymtotically negligible. This assumption is in particular satisfied if the goals were defined by selecting a small number of abstract concepts from the hierarchy like Organic and Machine because subconcepts in the hierarchy inherit functions from their parents. We can also think of more sophisticated keys that not only regard the number of functions that are satisfied by a concept, but also how effective the goal is satisfied. To this end, we can apply similar utility functions as explained in the previous section.

Algorithm 4 shows a global algorithm to select concepts. It works by iterating over all subsets $\mathcal{C}'$ of $\mathcal{C}$ that contain at least 2 elements. The concepts in $\mathcal{C}'$ are then combined as explained in the previous section. The new concept is then evaluated and the best concepts are stored. Finally, a best concept is returned. In order to compute utility values and to select a best concept, we can apply similar ideas as explained for Algorithm 2. To this end, we can reuse information from the combine-procedure. However, we can also take additional criteria into account. For instance, we might want to avoid that too many concepts are used and can therefore use a discounting factor like 2^{-k} to decrease utility increases as the size of the subsets increases. The runtime of Algorithm 4 is $O(\sum_{k=2}^{|\mathcal{C}|} \binom{|\mathcal{C}|}{k} \cdot M(k))$, where $M(k)$ again denotes the cost for combining k concepts. Hence, the runtime is exponential in $|\mathcal{C}|$ even if we use the local algorithm for combining concepts. However, usually we do not want to combine an arbitrary number of

concepts, but maybe only two or three. If we consider a bound b on the number of concepts, the runtimes becomes $O(\sum_{k=2}^{b} \binom{|\mathcal{C}|}{k} \cdot M(k)) = O(|\mathcal{C}|^b \cdot M(b))$.

3.3 Computational Results

We proposed a local and a global algorithm to merge and select concepts, respectively. Our global algorithm has an exponential worst-case runtime. Actually, the concept invention problem is inherently difficult if we do not make any assumptions on the maximum number of concepts that may be combined. To make this precise, let us assume that we are given a utility function $u : 2^{\mathcal{C}} \to \mathbb{R}$. This utility function can take all criteria into account that we discussed before. Note that we can also use it to evaluate single concepts $c \in \mathcal{C}$ by letting $u(c) = u(\{c\})$. We call u *polynomial-time computable* iff $u(\mathcal{C}')$ can be computed in time polynomial in $\mathcal{C}$ and $\mathcal{G}$ for all $\mathcal{C}' \subseteq \mathcal{C}$. Let us consider the following decision problem.

> *U-CONCEPT:* Given a concept blending framework $(\mathcal{C}, \mathcal{F})$, a set of goals $\mathcal{G} \subseteq \mathcal{F}$ that is compatible with $(\mathcal{C}, \mathcal{F})$, a polynomial-time computable utility function u and a real number $U \in \mathbb{R}$, decide whether there is a subset $\mathcal{C}' \subseteq \mathcal{C}$ such that $u(\mathcal{C}') > U$.

The following result follows from a guess and check argument and a reduction of 0-1-KNAPSACK. We omit the proof to meet space restrictions.

Proposition 1. *U-CONCEPT is NP-complete.*

However, usually we do not want to combine an arbitrary number of concepts, but combine at most a handful of concepts. If the maximum number of concepts that can be combined is some fixed integer b, our previous results show that a globally optimal solution can be found in time $O(|\mathcal{C}|^b \cdot |\mathcal{C}'|^{|\mathcal{G}|} \cdot (|\mathcal{G}| + U(\mathcal{C}, \mathcal{G}))) = O(|\mathcal{C}|^{b+|\mathcal{G}|} \cdot (|\mathcal{G}| + U(\mathcal{C}, \mathcal{G})))$ (Algorithms 2 and 4). Since we assume that u is polynomial-time computable, this term is polynomial in $\mathcal{C}$, but exponential in $|\mathcal{G}|$. However, if we assume that there is only a small number of goals our algorithm is efficient. In other words, if we want to combine only a small number of concepts, U-CONCEPT is fixed-parameter tractable with parameter $|\mathcal{G}|$ (see [3] for an introduction to parameterized complexity). In particular, by combining the local algorithms 1 and 3, we see that creating a concept that satisfies our goals can be performed in polynomial time. Strictly speaking, we have to assume that there exist indeed b concepts whose functional units can be combined to fulfill all goals. However, it might be reasonable to just introduce a penalty factor in the utility function, which penalizes combinations that ignore goals. We summarize our findings in the following proposition.

Proposition 2. *If there is a subset $\mathcal{C}' \subseteq \mathcal{C}$ of size at most b, whose functional units fulfill all goals in $\mathcal{G}$, then*

1. *U-CONCEPT restricted to combinations of at most b concepts is fixed-parameter tractable with parameter $|\mathcal{G}|$,*
2. *a subset $\mathcal{C}' \subseteq \mathcal{C}$ that satisfies $\mathcal{G}$ can be computed in time polynomial in $|\mathcal{C}|$ and $|\mathcal{G}|$.*

4 Related Work

Conceptual blending (compare, for example, [5] and [6]) is a fundamental cognitive process underlying much of everyday thought and language. It is modeled as a process by which humans combine certain concepts, relations, and properties of originally separate conceptual spaces into a unified space (the blend space), in which new elements and relations emerge, and new inferences can be drawn. In this sense, it can be considered as a source of creativity. Whereas the classical framework of conceptual blending requires two input spaces, a generalization of the input spaces (often called generic space), and a blend space, this paper departs to a certain extent from this framework. Input spaces correspond to concepts (composed of components that fulfill certain functions), the generic space corresponds to goals (functions that shall be fulfilled by the new concept's components) and the blend space corresponds to a new concept (composed of some of the input concepts' components). The present approach can be regarded as a search strategy that can be applied to special instances of the general framework. Other computational models for conceptual blending have been proposed, for instance, in [7] and [9]. There are also relationships to problems like case-based- reasoning [2], predicate invention [11] and concept invention in machine learning [14].

In [12], the authors describe the creation of monsters from a library of descriptions of animals formalized as OWL ontologies. A major difference between [12] and the present approach is the idea of guiding the blending process by functions that should be realized in the blend space. This is related to using priority values for properties to order them according to their importance for their appearance in the blend space. An approach for creating novel musical chord progressions by conceptual blending directed by hand-coded priority values for certain properties that should be realized in the blend space is presented in [4]. A further approach that uses a heuristics in order to guide the blending process in mathematics is presented in [10] where the consistency (or non-consistency) of the resulting blend theory is used as a heuristcs.

5 Conclusions and Future Work

We proposed a computational framework for function-driven concept invention. The search for interesting candidate concepts to combine and the combination of concepts is guided by desirable functions that the new concept should satisfy. Whereas the problem of creating an optimal new concept is NP-complete in general, prohibiting arbitrarily large combinations makes the problem fixed-parameter tractable with respect to the number of goals. Just creating a new concept that satisfies our goals can be performed in polynomial time. We are currently working on basic implementations of the local and global approach.

As in most frameworks for concept invention, our framework's ability to create new concepts strongly depends on the information and structure that we provide. We are planning to apply machine learning tools to automate this process. For instance, we can regard $\mathcal{C}$ as an ontology and we can reuse existing

ontologies from the web or parse existing nomenclatures from online encyclopedia. In order to learn functions of concepts and their functional units, we will try to apply natural language processing tools.

Acknowledgements: Some of the authors acknowledge the financial support of the Future and Emerging Technologies Programme within the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number: 611553 (COINVENT).

References

1. D. Aerts and L. Gabora. A theory of concepts and their combinations ii: A hilbert space representation. *Kybernetes*, 34(1/2):192–221, 2005.
2. T. Besold, K.-U. Kühnberger, and E. Plaza. Analogy, amalgams, and concept blending. In *Proc. ACS 2015*, page 23, 2015.
3. R. G. Downey and M. R. Fellows. *Parameterized complexity*. Springer Science & Business Media, 2012.
4. M. Eppe, R. Confalonieri, E. Maclean, M. A. Kaliakatsos-Papakostas, E. Cambouropoulos, W. M. Schorlemmer, M. Codescu, and K. Kühnberger. Computational invention of cadences and chord progressions by conceptual chord-blending. In *Proc. IJCAI 2015*, pages 2445–2451. AAAI Press, 2015.
5. G. Fauconnier and M. Turner. Conceptual integration networks. *Cognitive Science*, 22(2):133–187, 1998.
6. G. Fauconnier and M. Turner. *The way we think: Conceptual blending and the mind's hidden complexities*. Basic Books, 2008.
7. J. A. Goguen and D. F. Harrell. Style: A computational and conceptual blending-based approach. In *The structure of style*, pages 291–316. Springer, 2010.
8. M. Guhe, A. Pease, A. Smaill, M. Martinez, 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.
9. B. Li, A. Zook, N. Davis, and M. O. Riedl. Goal-driven conceptual blending: A computational approach for creativity. In *International Conference on Computational Creativity*, volume 10, 2012.
10. M. Martinez, U. Krumnack, A. Smaill, T. R. Besold, A. M. Abdel-Fattah, M. Schmidt, H. Gust, K.-U. Kühnberger, M. Guhe, and A. Pease. Algorithmic aspects of theory blending. In *Artificial intelligence and symbolic computation*, pages 180–192. Springer, 2014.
11. S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19:629–679, 1994.
12. F. Neuhaus, O. Kutz, M. Codescu, and T. Mossakowski. Fabricating monsters is hard: towards the automation of conceptual blending. In *Proc. C3GI@ ECAI-14*, volume 1, 2014.
13. F. C. Pereira. *Creativity and artificial intelligence: a conceptual blending approach*, volume 4. Walter de Gruyter, 2007.
14. A. Popescul and L. H. Ungar. Cluster-based concept invention for statistical relational learning. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 665–670. ACM, 2004.
15. M. Schorlemmer, A. Smaill, K.-U. Kühnberger, O. Kutz, S. Colton, E. Cambouropoulos, and A. Pease. Coinvent: Towards a computational concept invention theory. In *Proc. ICCC 2014*, 2014.