Journal of Artificial Intelligence Research 82 (2025) 1961-1997 Submitted 10/2024; published 03/2025

A Divide, Align, and Conquer Strategy For Program Synthesis

Jonas Witt JONAS.WITT.LAB@UNI-BAMBERG.DE
University of Bamberg, Markusplatz 3, 96047 Bamberg, Germany

Sebastijan Dumančić S.DUMANCIC@TUDELFT.NL
TU Delft, Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands

Tias Guns TIAS.GUNS@KULEUVEN.BE
KU Leuven, Celestijnenlaan 200a, 3001 Leuven, Belgium

Claus-Christian Carbon CCC@UNI-BAMBERG.DE
University of Bamberg, Markusplatz 3, 96047 Bamberg, Germany

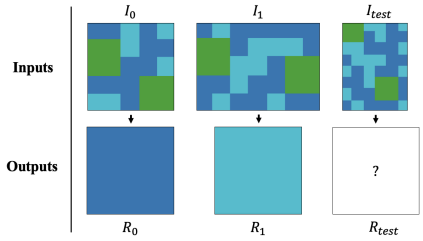
Abstract

A major bottleneck in search-based program synthesis, which learns programs from input/output examples, is the synthesis of large programs. As the size of the target program increases, so does the search depth, which leads to an exponentially growing number of candidate programs. Humans mitigate the combinatorial explosion that arises from deep program search: they build complex programs from smaller parts. We introduce a new strategy for program synthesis called Divide, Align & Conquer (DA&C) that exploits the compositionality of real-world domains to guide the synthesis towards useful subprograms. *Divide* decomposes each example using a segmentation procedure that is synthesized as part of the learning problem. *Align* matches the components in the decomposed input/output examples to steer the search toward combinations that lead to the synthesis of useful subprograms, and *Conquer* then solves a standalone synthesis problem on each pair of aligned input/output components. We show how replacing a deep program search with a linear number of much smaller synthesis tasks leads us to efficiently discover useful subprograms that are then combined into a solution program. Our agent outperforms current Inductive Logic Programming (ILP) methods on string transformation tasks even with minimal knowledge priors. Unlike existing methods, the predictive accuracy of our agent monotonically increases for additional examples. It approximates an average time complexity of $\mathcal{O}(m)$ in the size m of subprograms for highly structured and, hence, decomposable domains such as strings. Finally, we demonstrate the scalability of our technique on high-dimensional abstract visual reasoning tasks from the Abstract Reasoning Corpus (ARC) for which ILP methods were previously infeasible. We are competitive with state-of-the-art agents outside of ILP, despite generating only 0.2% as many candidate programs from a knowledge prior of only 11 generic geometric primitives.

1. Introduction

A key challenge in program synthesis (Gulwani & Jain, 2017), which is concerned with learning programs from examples, is the synthesis of large programs. Program synthesis is

often framed as a search problem over a space of programs, and therefore, the larger the program, the more difficult it is to find (Alur, Singh, Fisman, & Solar-Lezama, 2018).



(a) Abstract visual reasoning task.

Inputs	Outputs
Schedule,08.00 Bring kids to kindergarten,08.30 Stop at bakery	08.00,08.30
Schedule,09.00 Morning team meeting,09.30 Architecture deep dive,10.00 Walk with James,11.00 Telephone conference	09.00,09.30,10.00,11.00
Schedule,12.00 Lunch with customer,13.00 Workshop NAFTA,13.00 Telephone conference,15.00 Meeting with Katie	12.00,13.00,13.00,15.00
Schedule,12.00 Lunch with colleagues,12.30 Strategy meeting,13.00 Workshop NAFTA,14.00 Literature review	?

(b) String transformation task.

Figure 1: Programming by Examples (PBE) tasks from the Abstraction and Reasoning Corpus (ARC) (Chollet, 2019) (Figure 1a) and the real-world string transformations data set (Cropper et al., 2020) (Figure 1b). Agents must search for a program that transforms inputs into outputs. In Figure 1a: "Color the output light blue whenever there is a light blue connecting pathway between the green squares in the input." In Figure 1b: "Extract all times from the meeting schedule and concatenate them using commas."

Divide and conquer (D&C) strategies are a common solution to this problem (Alur, Černý, & Radhakrishna, 2015). Existing D&C strategies break down the synthesis task by dividing the *set of examples* into subsets (Cropper & Dumančić, 2022; Alur, Radhakrishna, & Udupa, 2017; Cropper, 2022). Each subset of examples defines an independent synthesis task to be *conquered*. The solutions to the individual synthesis tasks, defined over subsets of examples, are later combined into a global solution, i.e., the final program. Imagine, for example, a synthesis task in which the input is a pair of numbers $(N1, N2)$, and the desired output is the larger of the two numbers. It is easy to find solutions for individual examples, e.g., return either $N1$ or $N2$. In order to obtain the task solution program, these subprograms are combined with an appropriate branching condition ('if $N1 > N2$ then...'). Given that we discover subprograms independently of the branching condition, we only ever search for expressions of length one even though the global solution program has a depth of two (the 'if branch' and its 'if condition'/ 'if consequent'). This is how standard D&C leads to an exponential decrease in search space through a linear reduction in search depth.

However, standard D&C makes two assumptions that limit the types of problems on which it achieves a significant decrease in search space: (1) It assumes a solution program can be split into individual pieces, which in turn can be identified from non-overlapping subsets of examples. (2) Solutions to subsets of examples are easier to synthesize than a solution covering all examples. Consider the task in Figure 1a in which the color of the output is determined by whether a light blue path exists between the green squares in the input image. Dividing the task into subsets of examples (with and w/o connecting paths) only marginally simplifies the problem: the search for a program that checks the path's existence remains equally difficult. We extend D&C to the level of components within a single example in order to efficiently discover program pieces needed to solve an input/output pair.

```
IF
ContainsString(x, ",", 4)
THEN
Concat( SubStr(x, PosPair( Regex(x, "Number", 1),
                           Regex(x, "Number", 1))),
        Concat( ConstStr(","),
                Concat( SubStr(x, PosPair( Regex(x, "Number", 1),
                                           Regex(x, "Number", 1))),
                        Concat( ConstStr(","),
                                Concat( SubStr(x, PosPair( Regex (x, "Number", 1),
                                                           Regex (x, "Number", 1))),
                                        Concat( ConstStr(","),
                                                SubStr(x, PosPair( Regex (x, "Number", 1),
                                                                   Regex (x, "Number", 1)))
)               )       )       )       )       )
ELSE
ConstStr("08.00,08.30")
```

(a) Program synthesized from the PROSE SDK.

```
FOR component IN Segment(x, ',') DO
    IF
        component.index == -1
    THEN
        SubStr(component, [:6])
    IF
        component.index ! = -1 ∧
        component.containsNumber
    THEN
        SubStr(component, [:6]), ConstStr(",")
```

(b) Program synthesized DA&C (ours).

Figure 2: A divide, align, & conquer (DA&C) strategy yields a compact and well-generalizing program for the task in Figure 1b.

In this work, we explore an alternative *divide*, *align* & *conquer* strategy (DA&C), which divides each example into a set of independent components that are conquered separately. Its atomic operation is to discover a program that solves parts of an example compared to standard D&C, which requires a solution for an example as its atomic operation. Consider the task in Figure 1b: It takes a meeting schedule as input and extracts from it a list of times. A traditional synthesis approach (Gulwani, 2011) generates the solution program in Figure 2a[1]. The program correctly solves the task. However, it is cumbersome and does not generalize to varying lengths of similarly formatted meeting schedules. The task becomes much easier once we acknowledge that examples have inherent compositional structure. A DA&C strategy breaks down the inputs/outputs into substrings using commas as delimiters and from those extracts times as prefixes. It flexibly generalizes to schedules of varying lengths (Figure 2b).

Two major challenges in working with smaller components beyond the level of examples are, first, the question of how to *segment* examples into components and, second, the problem of finding meaningful alignments between components in the inputs/outputs. An alignment between an input and output component is meaningful whenever it leads to a program that reconstructs the component in the output given the component in the input. An example output is solved if all of its components are successfully reconstructed from components in the input. In the example of Figure 1b, synthesis is performed on the aligned components '08.30 Stop at bakery' and '08.30' that lead to the program SubStr(*component*,[:6]). One could explore every possible correspondence between input/output components, but that is likely to diminish the benefit of problem decomposition as the number of synthesis steps (needed to discover that all but one correspondence is meaningless) will be large. The problem is further complicated by the fact that not every input component needs to be present in the output and, thus, does not need to have a correspondence (e.g., 'Schedule' in Figure 1b).

1. The program was produced using the text transformation API of the Microsoft Program Synthesis using Examples SDK PROSE, a fleet of program synthesis APIs that are the basis of commercial tools like FlashFill in Microsoft Excel. The code is available on GitHub (Sumit Gulwani, 2023).

We propose to mitigate this problem by *structurally aligning* the input and output scenes in a process that mimics *analogical reasoning*. A meaningful alignment between two scenes maximizes their shared structure. For instance, in the last training example in Figure 1b, there are two conflicting meetings booked at the same time. Any one of them could produce the '13.00,13.00' substring in the output by repetition. However, only an alignment that respects the ordering of components and aligns '13.00 Workshop NAFTA' with the first '13.00' output substring and '13.00 Telephone conference' with the second '13.00' output substring will produce a minimal solution program that generalizes to arbitrary meeting schedules. In order to arrive at this mapping, we leverage analogy engines such as the structure-mapping engine (SME) (Falkenhainer, Forbus, & Gentner, 1989), a computational model implementing structure-mapping theory (SMT) (Gentner, 1983), a formal account of analogy-making in humans.

We combine the synthesis performed on subsets of components with a hierarchical search (Wang, Cheung, & Bodik, 2017), which allows us to learn parts of a final solution program sequentially, thereby efficiently pruning irrelevant areas of the search space. First, segmentation operations are learned independently of the transformation program. The information available about the segmented output components is used to prune transformation programs that do not need to be explored during synthesis (e.g., during the synthesis of a lowercase output component, all programs with 'capitalize' operators can be pruned away). Second, only after we have discovered a successful program (on a tuple of input/output components) do we search for similar pairs and learn an abstract classifier that tells us when to apply it (on which components in the input).

In addition, our approach demonstrates progress on two key topics in program synthesis: First, we apply DA&C to high-dimensional examples without manually pre-processing them into symbolic encodings. The segmentation of examples is discovered as part of the synthesis in order to optimize reconstruction accuracy in the output. This strategy prepares synthesis on raw real-world inputs. Second, synthesis tools struggle with large knowledge libraries. DA&C performs synthesis on subsets of input/output components, where the information on the output component is used to prune unapplicable background knowledge. This is similar to a relevance mechanism that selects predicates to be used in the current search from a (growing) global library, e.g., the synthesis of a green output object does not need any other color constants besides green.

To summarize, we propose a 'divide, align & conquer' strategy (DA&C), which performs a hierarchical search on partial examples in order to infer generative programs from a minimal number of examples. Specifically, our contributions are as follows:

1. We re-conceptualize the D&C strategy such that synthesis can exploit repeating compositional structure within individual examples. We demonstrate how this allows agents to synthesize more complex programs, even on high-dimensional inputs such as bitmaps.

2. We introduce analogical reasoning as a means to mitigate the combinatorial explosion of correspondences between input/output components. We demonstrate how the information on an aligned output component can be used to prune parts of the program search.

3. We implement DA&C in an algorithmic agent called BEN. Its performance is evaluated on the established setting of string transformation tasks and the challenging abstract visual reasoning data sets from the Abstraction and Reasoning Corpus (ARC). (Chollet, 2019).

2. Related Work And Background

Our work lies in search-based program synthesis, specifically the *Programming by Examples* (PBE) systems, which learn programs that are consistent with a semantic specification implicitly defined by a set of input/output examples (Gulwani & Jain, 2017).

Recent work has focused on two areas: (1) improving the efficiency of search by guiding it towards more promising candidates and (2) rewriting task specifications to simplify the search problem itself. Our work falls into the second category. It also shares common goals with works in the first area.

Improving search efficiency. The first area has recently focused on search optimizations that leverage the intermediate states of programs and evaluate their distance from the target solution (Ellis, Nye, Pu, Sosa, Tenenbaum, & Solar-Lezama, 2019; Cropper et al., 2020; Nye, Pu, Bowers, Andreas, Tenenbaum, & Solar-Lezama, 2021). Often, these approaches are neuro-symbolic hybrids that statistically learn a heuristic function from past synthesis tasks. DA&C, in its current form, does not make use of statistical learning across tasks. Instead, it guides the synthesis search by pruning the search space. It uses the information about an output component (gained from an example segmentation) to prune away irrelevant parts of the search space. Apart from heuristic search, other approaches use re-representation techniques to condense programs and, thus, make it easier to search them: compression-driven rewriting or functional abstraction (e.g., predicate invention) (Henderson & Muggleton, 2014; Cropper & Dumančić, 2020; Dumančić, Guns, & Cropper, 2021). The programs learned through DA&C are similarly compact but are produced by *rewriting* the examples, not the program. The way that DA&C rewrites examples is by searching for a meaningful segmentation into components. There exist search optimizations that already make use of decomposition operators in order to guide the synthesis: Symbolic backpropagation (Gulwani & Jain, 2017) is a top-down deductive search that uses inverse operators to propagate constraints on the overall solution program to sub-expressions of the intermediate program. These sub-specifications are used to filter substrings from the input with the help of regular expressions $pos(x, R1, R2, k)$. This approach is different from DA&C in that it first assumes an intermediate program and then deductively finds the most probable segmentation to support the program. In contrast, we first pick a segmentation and then inductively search for a program. The downside of symbolic backpropagation is that it requires inverse operators of each language primitive to work from the example output backward. Gulwani and Jain (2017) address the combinatorial challenges of inverse synthesis using forwardprop filtering of inverse candidates. However, the more expressive the transformation language, the larger the number of conceivable inverses. For example, a 'replace' operator, which replaces a substring in the input with a constant substring in the output, will generate an intractable number of inverses. In DA&C, 'replace' operators are cheap because synthesis is executed on pairs of input/output components where the

constant substring is available through the output component, and its inverse is simply the input component.

Rewriting the search problem. Previous work in this area has demonstrated D&C (divide and conquer) strategies on the level of examples (Alur et al., 2015). For instance, Cropper (2022) combines D&C with modern constraint-solving using answer sets. Successful intermediate programs are used to search for a more general program that applies to multiple chunks until all positive examples are covered. These approaches only work on subsets of examples (called *chunks*), while DA&C performs synthesis on subsets of components within an example. We use antiunification in the underlying domain-specific language (DSL) to learn branching conditions that combine independent programs on components into a global solution program. Our approach is more similar to that of Alur et al. (2017), which combines intermediate programs using a conditional expression grammar in a multi-label decision tree learning paradigm. However, their approach also only works with subsets of examples. DA&C exploits the innate structure of input/output examples to decompose a semantic specification into sub-specifications that are solved in multiple smaller synthesis tasks. This trades some of the exponential complexity of a deep program search with a linear number of additional synthesis tasks.

Analogical reasoning. In order to mitigate the combinatorial explosion that arises from decomposing examples into components and searching for alignments between components, we make use of *analogical reasoning* (Evans, 1964; Mitchell, 1993). Research in the cognitive sciences has highlighted the importance of analogies for problem-solving (Hofstadter, 2001; Mitchell, 2021). In the past, these models were applied to study psychometric tests of intelligence (Snow, Kyllonen, & Marshalek, 1984), e.g., in number series completion, string transformations, verbal analogies, and Raven's matrices. For instance, Lovett and Forbus (2017) use analogical reasoning to compute 'patterns of variance' (descriptive statements of how objects change) across subsequent scenes within each row of a Raven's matrix. In contrast, we explicitly learn *actionable* transformation programs that are capable of *generating* new outputs. We make use of the structure-mapping-engine (SME) (Falkenhainer et al., 1989; Gentner, 1983), a computational model of analogical reasoning in humans, to determine how an input segmentation is analogous to the segmented output and derive from it pairwise correspondences. SME is a symbolic approach to structure-mapping that purely relies on the syntactic representation of a problem. It is a good fit for our goal because it works across domains and does not require training on large data sets.

Program synthesis on high-dimensional examples (e.g., bitmaps) has seen much less work than the established domains such as bit vector and string manipulations or generation of invariants (Alur et al., 2018). Ellis, Solar-Lezama, and Tenenbaum (2015) synthesized programs to represent visual concepts and perform item classification using probabilities in a generative process. The synthesis itself was not directly performed on bitmaps but on automatically parsed symbolic encodings. Cropper et al. (2020) learned programs to generate bitmaps using an example-dependent loss function instead of logical entailment in order to guide the synthesis on large programs better. We apply DA&C to the Abstraction and Reasoning Corpus (ARC) (Chollet, 2019), a much more diverse collection of bitmap data sets that was introduced to foster research on the efficiency with which agents acquire

new skills. The corpus is a collection of heterogeneous visual reasoning tasks. For each task, the agent synthesizes a program that takes a bitmap as input in order to generate a bitmap as output (e.g., task in Figure 1a). ARC is especially interesting to our work as visual reasoning programs tend to be large (in the current program synthesis context) and, thus, out of reach for existing synthesis techniques. Visual scenes also intuitively demonstrate the idea of being composed of objects (Johnson, Vong, Lake, & Gureckis, 2021; Wagemans, Elder, Kubovy, Palmer, Peterson, Singh, & Heydt, 2012). ARC is a challenging benchmark where the top-ranked agents only solve about 20% of tasks on a hidden test set and perform a brute force 'generate & test' approach using elaborately handcrafted domain-specific languages (DSL) (Wind, 2020; de Miquel, Corominas, & Ariyasu, 2020). Instead, we are investigating a systematic program synthesis approach that scales to high-dimensional examples using the idea of task decomposition.

2.1 Background

In this subsection, we briefly introduce the idea of program synthesis as search and the structure-mapping theory (SMT) (Gentner, 1983), both of which are integral parts of DA&C. Readers already familiar with these concepts are free to skip ahead.

Program synthesis as search. Learning a program is formulated as searching through a language space. In addition to a semantic specification (e.g., a set of input/output examples), the agent is provided syntactic constraints (e.g., grammar $\mathcal{G}$) on the set of program candidates (Alur et al., 2018). Every derivation from G is a candidate program. A grammar $\mathcal{G}$ with a finite number of production rules can produce infinitely many programs of increasing length. One of the central challenges in the field is scaling the synthesis to large programs. The longer a solution program, the larger its search space, which exponentially grows ($b^{|d|}$) in the depth of program d and the average branching factor b of the grammar. DA&C contributes to the goal of scaling search-based program synthesis by trading a deep program search with a linear number of smaller synthesis tasks.

Structure-mapping theory. Analogies help us reason about an unfamiliar target domain (e.g., the structure of an atom: the relationship between electrons and its nucleus) using existing knowledge of a familiar base domain (e.g., the structure of our solar system: planets orbitting around the sun). Structure-mapping theory (SMT) systematizes the process by which humans perform analogical reasoning. Computational models that implement SMT consume propositional scene representations of a base and target and search for an alignment between the two that maximizes their shared relational structure (systematicity principle). For example, electrons orbit around the nucleus of an atom, similar to how planets orbit around the sun. The alignment between the base and target (called mapping) consists of a set of matched entities and predicates, e.g., an electron is to the nucleus what a planet is to the sun because both share the relationship of orbitting around an object with greater mass. Mappings are evaluated systematically through three steps:

1. Generation of *local match hypotheses mh*: Each predicate pair in the base/target is evaluated through a set of match constructor rules. If successful, the predicate pair forms a local match hypothesis. Local match hypotheses will be combined in the

following steps to find isomorphic subgraphs between the base and target. Match constructor rules put syntactic constraints on the types of local match hypotheses that are formed. An example of such a constructor rule is given below: In this case, any two predicates with matching functors that are not attributes form a *mh*.

```
(mhc-rule (:filter ?b ?t :test (and (equal (exp-functor ?b) (exp-functor ?t))
            (not (attribute? (exp-functor ?b))))) (construct-mh ?b ?t))
```

In the example above, the constructor rule leads to a match hypothesis being formed over the predicate pairs *(orbits electron nucleus)* and *(orbits planet sun)*. Let's assume that this step also returns a match between the predicate pair *(greater_mass sun planet)* and *(greater_mass nucleus electron)*.

2. Derivation of *global mappings* (GMAPs): A GMAP is a maximal and structurally consistent set of local match hypotheses in the base/target. A GMAP is structurally consistent if (1) all matched predicates also form match hypotheses between their arguments and (2) all *mhs* in the GMAP yield a consistent set of one-to-one correspondences between entities in the base/target.

   In the example, the predicate pair 'orbits' is structurally consistent if its arguments 'electron'/ 'planet' and 'nucleus'/ 'sun' also form match hypotheses. The set of predicates 'orbits' and 'greater_mass' is then also structurally consistent because the 'greater_mass' predicate enforces the same one-to-one entity mapping as 'orbits': 'electron' is paired with 'planet', 'nucleus' is paired with 'sun'.

3. Ranking of GMAPs: All maximal structurally consistent GMAPs are ranked according to their extent of matched relational structure. Nested relational structures are favored over shallow matches, which is characteristic of meaningful analogies.

   A meaningful analogy in the running example is the combined set of both the 'orbits' and 'greater_mass' predicates, including all derived entity mappings.

We will make use of SMT to answer the following question: How is the input of an example structurally similar to its output? From there, we systematically search for programs that transform a part of an input into its corresponding (SMT-derived) part in the output. The search will first explore those pairwise correspondences that contribute most to a structurally consistent and maximal alignment between the input/output.

3. Problem Definition

More formally, we solve a standard synthesis task represented as a tuple $(\Phi, \mathcal{G})$ of a specification Φ and a grammar $\mathcal{G}$. The specification is given in the form of (positive) examples $\mathcal{Q}$, where each example q_l is an input/output pair $q_l = (I_l, R_l)$. A program p is said to solve the task when $p \in \mathcal{G}$ such that $\forall (I, R) \in \mathcal{Q},\ p(I) = R$.

In this work, we focus on problem domains with separable specifications, which is a commonly studied field of synthesis problems (Neider, Saha, & Madhusudan, 2016). A specification is separable if it only relates an input to its output and gives no further

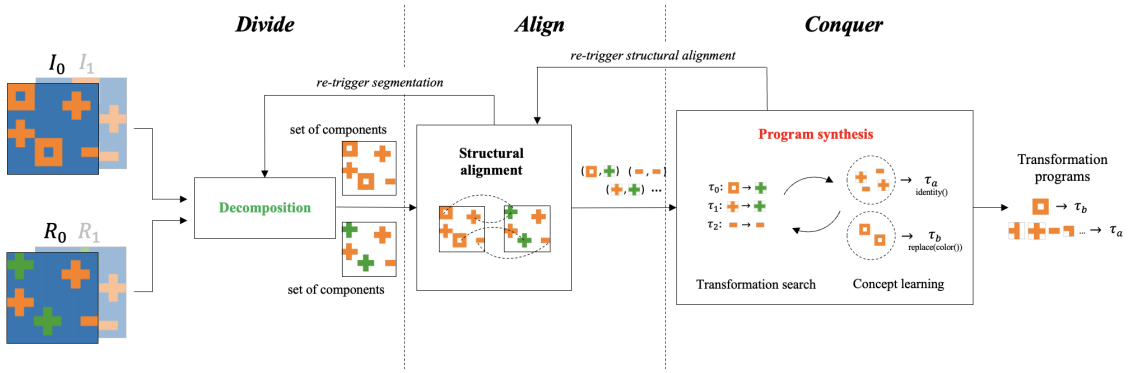


Figure 3: Every example is decomposed into component parts (e.g., visual objects in a scene, words in a sentence). We use the information about components and their relations to produce a structural alignment between the input/output scenes. Each correspondence from this alignment is solved independently using off-the-shelf synthesis techniques. Correspondences are considered in the order in which they contribute to the structural alignment (*analogy*). For each unique partial program, we learn a formula that specifies its corresponding input space (the space of components on which it should be executed). The combination of an input space and its transformation is called a transformation rule. A solution program consists of a set of transformation rules that, if applied to the components in the input, reconstruct all components in the output.

constraints on the relation between outputs of different inputs, $p(I) = R \ \wedge \ \Phi(I, R)$ with program p and formula Φ (Alur et al., 2015). All synthesis tasks that make use of examples fulfill this condition because Φ is implicitly specified over input/output pairs (I, R). We split the problem of synthesizing a program into three subtasks (Figure 3): *divide*, *align*, and *conquer*.

First, we learn to *divide* the input/output examples into sets of components. The purpose of divide is to identify components that repeat across examples. For instance, divide on the task in Figure 3 yields a set of orange and green shapes for each example input/output.

Second, we search for an *alignment* of the components in the input set of an example with the components in its output set. The goal of the align step is to identify pairs of input/output components that can be transformed into each other with a minimal program. In the example, the mapping between orange squares in the input and green crosses in the output is assigned a high probability because they share the same relational placement within the image. The information on pairwise correspondences is used as a heuristic to guide the search for a successful program. This is why it is no concern that orange crosses in the input are also matched with green crosses in the output due to their surface similarity. These matches are assigned a lower score and are never explored because, by that time, the search will have already found a set of transformation programs that fully reconstructs all output examples.

Third, we *conquer* each pair of aligned input/output components separately, which means synthesizing a transformation program that reconstructs the output component us-

S	$\rightarrow$	DECOMPOSE, $\mathcal{T}$
$\mathcal{T}$	$\rightarrow$	τ, $\mathcal{T}$ | ϵ
τ	$\rightarrow$	if CONCEPT then TRANSFORM
TRANSFORM	$\rightarrow$	o_i | primitive01(TRANSFORM, args) | primitive02(TRANSFORM, args) | ...

Figure 4: Divide-align-conquer synthesis grammar $\mathcal{G}$.

ing the aligned component in the input. In order to combine transformation programs into a global solution program, we learn a concept over all those input components (across examples) that made use of the same transformation.

New test examples are evaluated by first decomposing their examples into components and then evaluating each component against the concepts learned for the transformation programs. If the evaluation succeeds, the transformation is applied to the input component; its result is pasted to the output.

If *align* does not find a meaningful alignment for any of the components in the output, it re-triggers divide to produce a new segmentation of the examples into updated sets of components. If *conquer* is unable to synthesize a transformation for any of the output components, it re-triggers align to find a new component pair for the missing output component. We now introduce the problem starting from the language space of programs and then formalize the DA&C paradigm on top of this definition. As is standard in PBE, DA&C uses syntactic priors to restrict the search space of programs. What distinguishes DA&C from other PBE methods is that it requires the syntactic priors to be assigned to one of the three phases of the hierarchical search (divide, align, conquer). Specifically, DA&C expects as input constraints on the segmentation of examples $\mathcal{G}_{decomp}$ ('divide'), an encoding scheme for segmented components $\mathcal{G}_{concept}$ ('align'), and a grammar for the transformation programs $\mathcal{G}_{transform}$ ('conquer'). We later show that the total amount of syntactic priors provided to a DA&C agent across these three inputs is not different from the syntactic priors provided to other synthesis agents within a single synthesis grammar. The process we used to define these inputs for the experimentation domains in this paper (abstract visual reasoning and string transformations) was analogous to how one would define a single grammar: We defined decomposition constraints, component attributes, and transformation primitives, and then tested and extended them in a process of iterative refinement. In order to assess the effectiveness of splitting the synthesis grammar across three inputs and probing the dependency of this architecture on custom-engineered domain-specific primitives, we later describe an experiment in which we purposefully introduce nonsense primitives into the agent's knowledge base. We also present an extensive ablation analysis in the experiments section.

Space of programs. $\mathcal{G}$ is a typed grammar that yields programs $p : \mathcal{B} \rightarrow \mathcal{B}$ which consume and produce a domain-dependent standard type $\mathcal{B}$ (e.g., 2D bitmap, list of chars). Every $p \in \mathcal{G}$ is a composite program with three learnable subroutines (Section 3):

1. A decomposition function $\delta : \mathcal{B} \rightarrow \{\mathcal{B}\}_{i=0}^{N}$ which decomposes an input type $\mathcal{B}$ into a set of components of the same type $\forall q_l : \delta(I_l, R_l) = (\{o_i\}_{i=0}^{N}, \{o_j\}_{j=0}^{M})$. 'Divide' expects a configurable family of decomposition functions (e.g., in the form of a simple set definition or a domain-specific grammar $\mathcal{G}_{decomp}$). It reflects basic syntactic knowledge priors on how components comprise a domain example (e.g., words are separated using whitespaces, a visual scene can be comprised of multiple objects and a partially occluded background). We will provide detailed examples in the following sections. In the task of Figure 3, the learned decomposition function segments example images into objects whose pixels are equally colored and directly neighboring.

   The two remaining subroutines belong to the body $\mathcal{T}$ of the program. They make up the if-statements defined by the synthesis grammar $\mathcal{G}$ in Section 3. Each if-statement is a transformation rule $\tau : \mathcal{B} \rightarrow \{\mathcal{B}\}_{j=0}^{Y}$ that consumes exactly one component from the input and produces a set of components in the output. We explicitly define solution programs using a grammar with top-level branches to make it easy to combine transformations (Section 3).

2. The transformation program $p'(o_i) = o_j$ is the result of a synthesis performed on a component tuple (o_i, o_j) using function primitives (e.g., *drop_first_char()*, *color(c)*) provided by a domain grammar $\mathcal{G}_{transform}$. The task in Figure 3 yields two transformations: One replaces objects of shape ◘ with green objects of shape ✚, and the other copies the remaining objects to the output.

3. The rule condition learns the context in which to apply a transformation. This information is used to combine transformations on specific components into a solution program on the task level. The rule condition is expressed as a formula over component attributes (e.g., length of a substring, color of an object) taken from a domain grammar $\mathcal{G}_{concept}$. It is a binary classifier that returns True iff the component belongs to its concept: $\{\mathcal{B}\}_i \rightarrow [\top, \bot]$. Because examples demonstrate a recurring logic, there will be component tuples (across examples) that are solved by the same transformation program p'; their set is denoted as $CORR_{p'}$. We learn a concept $C_{p'}$ over the input components of tuples in $CORR_{p'}$.

   ```
   if o.shape == ◘ then o.replace_by(✚).color(green)
   if o.shape != ◘ then o.identity()
   ```

   The unite operator $\oplus$ combines tuples consisting of a program and its concept $(p'_0, C_{p'_0})$ $(p'_1, C_{p'_1})$ into a single program.

$$\tau_0 \oplus \tau_1 = (C_{p'_0} \cdot p'_0) \oplus (C_{p'_1} \cdot p'_1) = \text{ if } C_{p'_0} \text{ then } p'_0, \text{ if } C_{p'_1} \text{ then } p'_1 \tag{1}$$

At test time, the learned decomposition function is applied to each test example and yields a set of components. On each input component o_i, we check if it belongs to any of the concepts $C_{p'}$ in the transformation rules and if it does, execute p' to produce a component in the output: $\tau = \text{ iff } o_i \in C_{p'} \text{ then } p'(o_i)$.
To summarize, a DA&C solution assigns to every component in the output $o_j \in \delta(R_l)$

a component o_i in the input $o_i \in \delta(I_l)$, with $corr(o_i, o_j)$ and a transformation program $\tau_k = C_{p'} \cdot p'$ transforming o_i into $o_j : p'(o_i) = o_j$ such that if all transformation programs $\tau_k \in \mathcal{T}$ are applied to the decomposed input $\delta(I_l) = \{o_i\}_{i=0}^{N}$, their resulting components fully reconstruct the correct output, $compose(\bigcup_{i=0}^{n} \bigcup_{\tau_k}^{\mathcal{T}} \tau_k(o_i)) = R_l$.

The divide, align, and conquer subroutines are interdependent. In the next section, we introduce a specific implementation that leverages these interdependencies: We show how to efficiently discover decomposition functions and correspondences between components that minimize the complexity of a solution program.

4. Architectural Overview

We now detail the algorithmic approach to each of the DA&C parts following the running example of Figure 3 already introduced in the last section.

The goal of the decomposition phase is to segment examples into sets of *meaningful* components. A set of input components is meaningful whenever it leads the program search to discover compact transformation programs $\mathcal{T}$ that fully reconstruct the output. This idea is captured in the joint probability $P(\mathcal{Q}, \delta, \mathcal{T})$. We minimize its negative log-likelihood in order to find a decomposition function δ and learn a minimal set of transformation rules $\mathcal{T}$ that fulfill the specification given by inputs/outputs $\mathcal{Q} = (I_l, O_l)$. The joint probability is the result of a generative process that starts from the input examples and applies a decomposition function together with a set of transformation programs to generate output examples.

$$-log(P(\mathcal{Q}, \delta, \mathcal{T})) = -logP(\delta) - \sum_{l=0}^{N} (logP(R_l|\mathcal{T}, \delta(I_l)) + logP(\mathcal{T}|\delta(I_l, R_l)))) \tag{2}$$

The terms in Equation (2) from left to right are a prior probability on the decomposition function, a likelihood expressed as an example-dependent reconstruction accuracy, and a prior on the set of transformation programs. The prior probability of a set of transformation programs is dependent on the components produced by δ on a specific example (I_l, R_l) because their information is used to prune the space of programs, for example: to reconstruct the green cross in the output (Figure 3), any program that makes use of a color terminal other than green is pruned away. On the contrary, the prior on δ is moved outside the sum because it is applied to all examples of a task equally.

The search is a generate & test approach outlined in Algorithm 1. For now, we only consider the high-level control flow without optimizations to illustrate its main ideas. First, we compute an ordered list of decomposition functions Δ (line 1), ranked in the order of estimated usefulness to the downstream synthesis. The subsequent DA&C loop proceeds from the highest ranking decomposition function. The frontier set O keeps track of all segmented components in the example outputs that have not been considered for synthesis. We pick one (line 7) and do greedy best-first search over its ranked list of input components. We learn a transformation program (line 12) on the highest ranked correspondence and add it to the library $\mathcal{T}'$. Every time the library changes, we check if it contains sufficient

Algorithm 1 Overview Divide-Align & Conquer

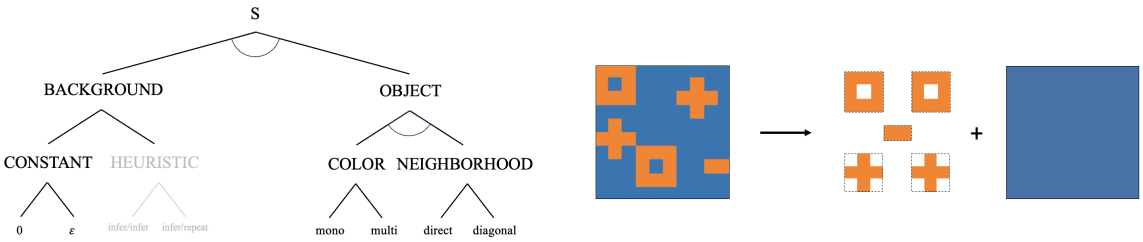
Input: specification $\Phi = \{(I, R)\}_{l=0}^{L}$, program grammar: $\mathcal{G}_{decomp}$, $\mathcal{G}_{concept}$, $\mathcal{G}_{transform}$
Parameters: search depth d
Output: $p \in \mathcal{G}$ s.t. $\forall (I, R) \in \Phi : p(I) = R$

1: $\Delta = ExtractAndRank(\mathcal{G}_{decomp})$
2: **while** $|\Delta| > 0$ **do**
3: $\quad decomposition\_func \leftarrow Pop(\Delta)$
4: $\quad O \leftarrow \bigcup_{l=0}^{L} decomposition\_func(R_l)$ ▷ **Divide each example**
5: $\quad \mathcal{T}' \leftarrow \{\}, \mathcal{C} \leftarrow \{\}$
6: $\quad$ **while** $|O| \geq 1$ **do**
7: $\quad\quad o_{curr}^{l} \leftarrow Choose(O)$
$\quad\quad\quad$ ▷ l is the example index which contains the current output component
8: $\quad\quad$ **if** o_{curr}^{l} not previously explored **then**
9: $\quad\quad\quad C[o_{curr}^{l}] \leftarrow RankCorrespondences(decomposition\_func(I_l), o_{curr}^{l})$ ▷ **Align**
10: $\quad\quad$ **end if**
11: $\quad\quad o_i \leftarrow Pop(C[o_{curr}^{l}])$
12: $\quad\quad \tau = LearnTransformation((o_i, o_{curr}^{l}), \mathcal{G}_{transform}, \mathcal{G}_{concept}, d)$ ▷ **Conquer**
13: $\quad\quad \mathcal{T}' \leftarrow \mathcal{T}' \cup \tau$
14: $\quad\quad$ **if** $\forall o_j \in \{decomposition\_func(R_l)_{l=0}^{L}\}$ covered by $\mathcal{T}'$ **then**
15: $\quad\quad\quad$ **return** $decomposition\_func, MinimalRuleSet(\mathcal{T}', \delta, \Phi)$
16: $\quad\quad$ **end if**
17: $\quad\quad$ **if** $|C[o_{curr}^{l}]| == 0$ **then**
18: $\quad\quad\quad O \leftarrow O \setminus o_{curr}^{l}$
19: $\quad\quad$ **end if**
20: $\quad$ **end while**
21: **end while**

transformation programs to reconstruct the output components in all examples (line 14), and if it does, we return a global solution program that consists of a minimal set of transformation rules. A minimal program set is one that minimizes the optimization function in Equation (2). The current output component o_{curr} is deleted from the frontier set O if all of its correspondences in the input have been explored and the DA&C loop repeats.

4.1 *Divide* - Decomposition

We learn a bottom-up parse that extracts structured components from examples instead of performing inductive synthesis on unstructured high-dimensional inputs at the level of their atoms (e.g., pixels, letters). This is done in the *divide* stage: We synthesize a decomposition function δ and segment the task examples into complete sets of mutually exclusive components. In the following sections, we will continue to use abstract visual reasoning as an illustration example. We refer to our agent implementing DA&C as 'BEN'.



(a) AND/OR graph of $\mathcal{G}_{decomp}$. (b) Application of a decomposition function δ.

Figure 5: Segmentation grammar $\mathcal{G}_{decomp}$ used for abstract visual reasoning tasks in ARC.

4.1.1 Segmentation

The decomposition of a standard type (e.g., string, image) $\delta(\mathcal{B}) \to \{\mathcal{B}\}_n$ yields a set of components of the same type, called segmentation of $\mathcal{B}$. In this paper, we use a symbolic definition of decomposition functions δ over a set of constraints C that determine if two atoms (e.g., pixels, characters) are part of the same component. *Mono-colored* is an example of such a constraint, which applies to all objects in the task of Figure 3. The reason we use a symbolic implementation instead of a non-parametric statistical model (e.g., deep autoencoders, CNNs) is because the domains we focus on only provide a single digit number of training examples, exhibit low noise, and offer intuitive segmentation heuristics that can be readily articulated by human task solvers (e.g., words are separated by whitespaces) (Raza & Gulwani, 2017). ARC images also have at most 10 colors and are bounded in size (30x30 pixels).

Our agent, BEN, works with a context-free grammar $\mathcal{G}_{decomp} = (\Psi_N, \Psi_T, \Psi_S, \mathcal{R})$ to encode these segmentation priors (visualized as an AND/OR graph in Figure 5a for the domain of abstract visual reasoning). Ψ_N is the set of non-terminals (e.g., background, object), Ψ_T is the set of terminals (constraints), Ψ_S is the start symbol, and $\mathcal{R}$ is the set of production rules to derive a unique set of constraints. The language of $\mathcal{G}_{decomp}$ is the set of decomposition functions $\{\delta \in \Psi_T^* | \Psi_S \Rightarrow^*_{\mathcal{G}_{decomp}} \delta\}$ derivable from $\mathcal{G}_{decomp}$ under the transitive closure $\Rightarrow^*_{\mathcal{G}_{decomp}}$. For example, the expression $\mathcal{D}[\![c_1, c_2]\!]$ with constraints $c_1 \triangleq$ 'mono-colored' and $c_2 \triangleq$ 'direct neighbors' is an abstraction over decomposition functions that produce components containing only equally colored, directly neighboring pixels. A decomposition function δ is applied to a bitmap by evaluating its constraints $\{c_i\}_{i=0}^{Z}$ on pairs of pixels, Equation (3). Any two pixels that satisfy all constraints are merged into the same object. The subsequent synthesis is performed on these objects instead of individual pixels.

$$\bigwedge_{i=0}^{Z} [\![c_i]\!](pixel_1, pixel_2) = \top \to (pixel_1, pixel_2) \in object \tag{3}$$

In the running task, the decomposition function learned by BEN segments the first example into four orange objects and a blue background (Figure 5b). In order to deal with natural background/foreground separation, our domain grammar $\mathcal{G}_{decomp}$ treats the

background object as a dense square which is partly occluded by objects in the foreground compared to a sparse square with holes ('Law of Prägnanz' from Gestalt psychology (Wagemans et al., 2012)) (Figure 5a). Its color is either constant or inferred using simple heuristics that exploit the fact that backgrounds often make up most of an image. Abstract visual reasoning tasks frequently make use of occlusion to create the illusion of a depth ordering amongst objects. Accepting occlusion as one fundamental principle of our world model leads to simpler object parses and, later on, simpler solution programs.

In order to work with other domains, we update $\mathcal{G}_{decomp}$ to reflect the compositionality priors of the new domain. For instance, string manipulation tasks have intuitive segmentation boundaries in the form of special characters (e.g., comma, whitespace, colon). The domain of string manipulation tasks is introduced in the experiments section.

4.1.2 Optimization - Generalization Difficulty

We recall that the goal of the *divide* stage is to identify components that lead the program search to discover a set of transformation programs $\mathcal{T}$ that are compact and successfully reconstruct the task outputs. A naive approach could use the domain grammar as described in Section 4.1.1 to linearly search through $\mathcal{L}_{\mathcal{G}_{decomp}}$ until it finds a segmentation which yields a perfect reconstruction of the output given some library of transformations. However, the number of decomposition functions in $\mathcal{L}_{\mathcal{G}_{decomp}}$ is too large to be traversed exhaustively. Even the evaluation of a single δ is costly because it requires us to minimize the negative log-likelihood of $P(Q, \delta, \mathcal{T})$ by doing best-first search over a potentially large number of pairwise correspondences.

The problem is further complicated by the fact that only slight syntactical differences in the segmentation constraints can lead to segmentations with vastly different semantics (usefulness in terms of the downstream synthesis). This means that the evaluation feedback from one δ cannot easily be used to direct the search toward more promising decomposition functions. Instead, we leverage the observation that decomposition functions that produce meaningful segmentations and lead to perfect reconstruction on the first input/output pair (I_1, R_1) are also more likely to yield meaningful segmentations on the remaining pairs q_l. The same argument applies to individual components: decomposition functions that lead to the successful reconstruction of the first component in the first output are also more likely to yield meaningful segmentations for the remaining components and examples. This means that we estimate the usefulness of a decomposition function δ by evaluating its joint probability on a single segmented component in the first output. We apply this estimation to all decomposition functions within $\mathcal{G}_{decomp}$.

$$\hat{P}(Q, \delta, \mathcal{T}) = P((I_1, o_1), \delta, \mathcal{T}) \quad and \quad o_1 \in R_1 \tag{4}$$

4.2 *Align* - Structural Alignment

The synthesis goal is to learn a minimal set of transformation programs $\mathcal{T}$ that act on input components and reconstruct output components. We denote this partial mapping as a set of learned correspondences $corr : o_i \rightarrow o_j$ in the Cartesian product $\delta(I_l) \bigtimes \delta(R_l)$. This step

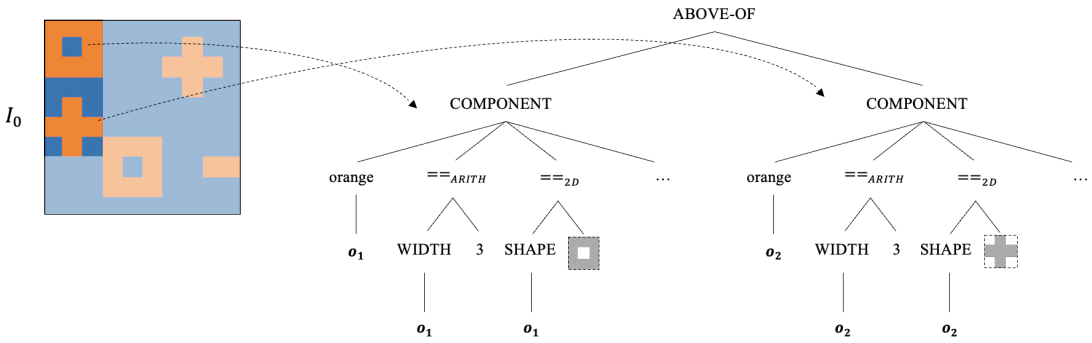


Figure 6: Example of a qualitative spatial propositional encoding of two decomposed objects in an abstract visual reasoning task.

is combinatorially expensive in theory. However, compositional real-world domains provide rich constraints that we use to find likely correspondences fast.

Our approach is inspired by analogical reasoning, which has been investigated as a foundational mechanism humans use to map knowledge of familiar situations onto new domains. The field's most prevalent theory is structure-mapping theory (SMT) (Gentner, 1983), first developed in the cognitive sciences and afterwards implemented as a computational model in the computer sciences, called the structure mapping engine (SME) (Falkenhainer et al., 1989). We leverage SME to search for a structural alignment between input/output examples that maximizes their shared relational structure. From there, we rank pairwise correspondences between components in the input/output relative to their contribution to the structural alignment and explore them iteratively until we have recovered enough transformation programs to solve the entire output.

In the experiments, we evaluate how maximizing the joint relational structure between examples speeds up synthesis and biases the search toward well-generalizing programs. The structural alignment performed by SME is symbolic and, thus, requires a propositional encoding of the decomposed scenes.

4.2.1 Propositional Encoding

We build on a set of basic assumptions that make the symbolic encoding of scenes domain-independent. A propositional encoding is a directed acyclic graph (DAG) that relates expressions about components within a scene. Expressions are either primitive entities (components) or predicates. Every predicate is either a relation (e.g., *above-of*), a function (e.g., *width*), or an attribute (e.g., *orange*). An expression E is a vertex in a graph and forms edges to all of its arguments (Figure 6). These are its descendants. They share E as a common ancestor. An expression with no ancestors is called a root. A propositional scene encoding can contain more than one root. An expression E' is reachable from an expression E if it is part of its transitive closure $\mathcal{R}^*(\mathcal{E})$. The depth of an expression is simply the minimum number of edges needed to reach it, starting at a root node.

The objective of the encoding is to facilitate a fast alignment of the shared structure in the input/output examples, which in turn leads to simpler solution programs. This is

because the more similar a pair of aligned components across input/output, the shorter the program that is needed to transform one into the other. Structure-mapping theory (SMT) argues that humans solve the combinatorial explosion of possible mappings between a base and target through *structural* alignment. This means they favor mappings between the base and target with deep shared relational structure (systematicity principle). Evidence from the cognitive sciences suggests that qualitative spatial relations are especially crucial to this process (Lovett & Forbus, 2017).

The specific relations used are domain-dependent, for example, basic positional (e.g., *left-of*) and topological relations from the Region Connection Calculus (RCC8). The complexity of a program that transforms one component into another is influenced both by the degree of matched relational structure and the extent of feature overlap between the two components. Therefore, we also add object features to the encoding.

Object features - Abstract visual reasoning

ID	Feature	Explanation	Derived from
1	color		
2	num_colors	number of unique colors in the object	color
3	row_origin_bbox	top left corner of the object's bounding box	
4	col_origin_bbox	top left corner of the object's bounding box	
5	shape	2D array of pixels in bounding box	
6	size	number of pixels in the object	shape
7	width		shape
8	height		shape
9	ranked_color	ordinal: decreasing with the number of objects of this color	color
10	ranked_color_rev	ordinal: increasing with the number of objects of this color	color
11	ranked_size	ordinal: decreasing with size	size
12	ranked_size_rev	ordinal: increasing with size	size
13	ranked_shape	ordinal: decreasing with the number of objects of this shape	shape
14	ranked_shape_rev	ordinal: increasing with the number of objects of this shape	shape
15	filled	boolean flag: bbox is filled by the object completely	shape

Table 1: Object features used in BEN on abstract visual reasoning tasks.

Nominal domain features (e.g., a component's color) are encoded as attributes. Any other features, such as a component's width or shape, are encoded as functions. For each functional output type (e.g., scalar, matrix), there is an equivalence relation (e.g., $=_{ARITH}$, $=_{2D}$) that compares specific values with constants of that type (Figure 6). Domain predicates are manually provided. We use 15 features (Table 1) for the abstract visual reasoning data sets.

4.2.2 Structure-Mapping

We provide the propositional encodings to SME and follow its three-step evaluation described in the background to extract correspondences between input/output components. In the running example (Figure 3), a local match hypothesis mh (Section 2.1) is formed

between the ◘-shaped object in the input and the green ✚-shaped object in the output. During the second step, we derive maximal and structurally consistent global mappings between the input/output (isomorphic subgraphs in their DAGs) (line 1 in Algorithm 2). Finally, we adapt how SME scores GMAPs to take into account both matched relational structure as well as feature similarity between matched components (line 5 in Algorithm 2). We do this to guide the downstream synthesis toward compact transformation programs. The updated scoring function Equation (5), therefore, evaluates local match hypotheses based on their depth d in the DAG and a feature similarity metric sim, where more importance is given to the matched relational structure $\omega_0 > \omega_1$. The score of a GMAP is equal to the sum of the scores of its match hypotheses.

$$score(GMAP) = \sum_{f=0}^{|root \in GMAP|} \sum_{g=0}^{|mh \in \mathcal{R}^*(root_f)|} \omega_0 * d(mh_g) + \omega_1 * sim(mh_g) \tag{5}$$

Algorithm 2 RankCorrespondences

Input: base representation $\mathcal{E}_\mathcal{I}$, target representation $\mathcal{E}_\mathcal{R}$
Parameters: match constructor rules MHC
Output: ranked correspondences $CORR$

1: $GMAP = SME(\mathcal{E}_\mathcal{I}, \mathcal{E}_\mathcal{R}, MHC)$
2: $CORR \leftarrow \{\}$
3: **for** $o_j \in \mathcal{E}_\mathcal{R}$ **do**
4: $\quad CORR[o_j] \leftarrow \{\}$
5: $\quad$ **for** $gmap \in$ GMAP **do**
6: $\quad\quad score = Score(gmap)$ ▷ see Equation (5)
7: $\quad\quad CORR[o_j] \leftarrow \{CORR[o_j] \cup (o_i, score) | (o_i, o_j) \in gmap\}$
8: $\quad$ **end for**
9: $\quad$ Sort $CORR[o_j]$ in decreasing order of $score$
10: $\quad$ Backfill $CORR[o_j]$ with missing correspondences from $\{(o_i, o_j) | o_i \in \mathcal{E}_\mathcal{I}\}$
11: **end for**
12: **return** $CORR$

In the running example in Figure 1a, input objects are preferably matched with their position-invariant counterparts in the output. These pairwise correspondences receive high evaluation scores because they map most of the qualitative relational structure in the input to the output. Correspondences between, e.g., the orange ✚-shaped objects in the input and green ✚-shaped objects in the output, are also generated due to feature similarities. However, their scores are much lower as they do not account for any of the relational structure shared between the input/output.

In the following *conquer* stage, we perform synthesis on individual pairs of component correspondences extracted from GMAPs. Those correspondences are explored in the order of their associated GMAP scores (line 6 in Algorithm 2). This means that we greedily perform synthesis on input/output pairs with similar relational structures and features. However, the best-first search will never cause DA&C to miss a transformation program

because $CORR$ contains an exhaustive list of all pairwise correspondences of an output object with the input (line 10 in Algorithm 2). In the worst case, it will iteratively explore the Cartesian product of component correspondences between the input/output.

Line 1 in Algorithm 2 calls SME (Structure Mapping Engine). We refrain from reprinting detailed pseudo code in this paper and refer to (Falkenhainer et al., 1989) for the original implementation of SME.

Complexity analysis. (1) The construction of local match hypotheses has a worst-case performance of $\mathcal{O}(N^2)$ with N being the average number of expressions in $\mathcal{E}(I)$ and $\mathcal{E}(R)$. Match constructor rules are applied to every predicate pair in the base/target. (2) The worst-case performance of finding maximal structurally consistent GMAPs is that of finding maximal sets of isomorphic subgraphs between the base/target, which is given by $\mathcal{O}(N!)$. (3) The computation of a GMAP evaluation score, Equation (5), is a graph walk over the number of roots within each GMAP and each of their transitive closures $\mathcal{R}^*(root)$. Its performance is bounded by the total number of local match hypotheses, which is $\mathcal{O}(N^2)$ in the worst case. The initial generation of the encodings (using unary and binary predicates) has a worst-case performance of $\mathcal{O}(N^2)$, which does not change the worst-case performance of the entire algorithm. In practice, we observe that the structural alignment using SME (Algorithm 2) performs significantly below its worst-case bound and depends heavily on the propositional scene encoding. Structural alignment performs best on deeply nested relational encodings with a variety of different relations (Falkenhainer et al., 1989). In the experiments section, we analyze the impact of analogical reasoning as part of the ablation studies.

4.3 *Conquer* - Synthesis Of Transformation Programs

The structural alignment of two scenes yields a ranked list of component tuples (o_i, o_j) between the input and output scenes. From this, the search for transformation rules τ proceeds in two steps. (1) Rule consequent: We treat each tuple as its own synthesis specification where the goal is to learn a transformation program $p'(o_i) = o_j$ that transforms o_i into o_j. (2) Rule condition: For every transformation program p', we learn a concept $C_{p'}$ over all input components that make use of p' across examples. A transformation rule, therefore, consists of a specific manipulation and the context in which it is applied.

BEN repeatedly picks the highest-ranking component tuple (as evaluated in the *align* stage) and solves a synthesis task, after which it updates the information on (1) known transformations and (2) their respective concepts. At the end of each iteration, if the set of collected transformations up to this point is sufficient to reconstruct all components across all outputs, it provides a solution program that consists of a minimal set of transformations. It can update its solution program as more compact transformation rules are discovered until it eventually terminates once synthesis has been performed on all correspondences in the Cartesian product $\delta(I_l) \bigtimes \delta(R_l)$.

We now consider a single synthesis loop (line 12 in Algorithm 1) on the highest-ranking correspondence (component tuple).

τ	$\rightarrow$	(if CONCEPT then TRANSFORM)

TRANSFORM	$\rightarrow$	oi, border(TRANSFORM, corr, env, #hole),
		inner(TRANSFORM, corr, env),
		color(TRANSFORM, corr, env, oj.color),
		shape(TRANSFORM, corr, env, oj.bbox),
		replace(TRANSFORM, corr, env, select(REFERENCE)),
		cut(TRANSFORM, corr, env, #hole),
		denoise(TRANSFORM, corr, env),
		move(TRASNFORM, corr, env, MD, #hole),
		scale(TRANSFORM, corr, env, #hole),
		rotate(TRANSFORM, corr, env, #hole),
		mirror(TRANSFORM, corr, env, #hole),
		complement(TRANSFORM, corr, env)

MD	$\rightarrow$	by, to, in
REFERENCE	$\rightarrow$	most_colorful, largest, other

Figure 7: Domain grammar for abstract visual reasoning $\mathcal{G}_{ARC}$.

4.3.1 Rule Consequent: Search For A Transformation

The *conquer* stage can use an off-the-shelf synthesis engine. BEN, for example, follows a generate & test approach using top-down enumerative search that explores all programs up to a predefined depth d. Transformation programs in the running example are derived from the domain grammar $\mathcal{G}_{ARC}$ in Section 4.3, which contains primitives for basic geometric operations such as scaling, translating, rotating, and filling of objects.

Optimizations. Here we introduce a crucial adaptation to standard enumerative search in order to exploit the fact that synthesis is executed on component tuples: The domain grammar $\mathcal{G}_{ARC}$ which houses the geometric primitives used to reason about abstract visual scenes does not contain derivations for the arguments of its primitives. Their arguments are either analytically specified by directly referencing the information contained within the output component of the underlying correspondence (e.g., see the 'oj.color' in the color() primitive) or they make use of 'holes' which are dynamically filled once a program candidate is executed. When a primitive is called with a 'hole' as an argument, it derives a parametrization that is correct in the context of the current correspondence or returns an error upon which the program candidate is discarded immediately. For example, the scale primitive analytically computes the scaling factor that is needed in order to change the size of the input component to that of the output component and replaces the 'hole' with this value. The use of 'holes' significantly reduces the size of the search space because fewer programs are enumerated. In the experiments, we show that by exploiting the information that is available within the component tuple, the speed-up in the synthesis can well account for the overhead needed to identify and encode meaningful component tuples.

Depending on the type of arguments and their permissible range of values, 'holes' allow us to work with primitives such as the 'shape' operator which would otherwise be intractable to search through: Consider the transformation program *o.shape(✚).color(green)* which is the result of the synthesis performed on the ◘-shaped component in the input and the

green ✚-shaped component in the output. The 2D shape argument can be deduced from the matched output component and does not need to be searched. The same is true for the color argument.

Performing synthesis on a single component tuple potentially leads to a large number of successful transformation programs, many of which do not generalize to other examples. In order to bias the synthesis towards minimal and well-generalizing solution programs, after each synthesis loop, we only keep track of whatever transformation program reconstructed the most additional output components across examples. This is the reason why, in line 12 in Algorithm 1, the synthesis only returns a single transformation program. The dictionary $\mathcal{T}'$ initialized in line 5 in Algorithm 1 records and updates which components are solved by which transformation program.

4.3.2 Rule Condition: Learn A Concept Definition

In the previous step, we synthesized transformation programs on input/output component tuples. In order to combine these individual transformations into a program that solves the entire task, we now learn Boolean logic formulas that describe the context in which a transformation is to be executed. Afterward, we will use these to combine transformation programs with 'if-then' statements.

The problem is a standard binary concept learning task over components $f(o_i) \rightarrow \{0, 1\}$ where the concept to be learned represents a subset of components. For each transformation program p', we partition the set of input components into three groups:

1. The set of positive components P that successfully reconstruct a component in the output using p', $\forall o_i \in P : p'(o_i) \in \{\delta(R_l)\}_{l=0}^{L}$.

2. The set of negative components N that incorrectly reconstruct parts of the output if p' was applied to a component from this set.

3. A set of neutral objects which do not generate false outputs but also don't reconstruct any component in the output if p' was applied to a component from this set (e.g., partial reconstruction of an output component, out-of-bounds transformations).

A key challenge is the limited number of observations available per task. There are only as many examples as there are components in all inputs, generally between 3 and 30. However, the fact that a task requires perfect reconstruction of all outputs introduces a strong inductive bias: Namely, the concept must cover all positive components P but none of the negative components Z. If the concept did include even a single negative example, at least one of the example outputs would be reconstructed incorrectly. The learner imposes no constraints on neutral components. In addition, we expect a good concept to only use a small number of component features for generalization purposes.

Given that we have already generated symbolic encodings of each component during the *align* phase, we learn a Disjunctive Normal Form (DNF) (Valiant, 1985) over component features. A DNF is a disjunction of conjunctions. It is best understood as a set of rules, where each rule specifies a set of properties. If at least one rule applies, the DNF evaluates to true. It is a function $\{0, 1\}^m \rightarrow \{0, 1\}$. In order to learn a DNF over a variety of domain

Algorithm 3 Constrained DNF learner

Input: X: component representations $\{0,1\}^{(n,m)}$, Y: labels of $\{0,1\}^n$
Parameters: j: max number of conjunctions
Output: DNF

```
1: DNF = []
2: P+ = {o_i | o_i ∈ X ∧ Y[o_i] == 1}
3: N− = {o_i | o_i ∈ X ∧ Y[o_i] == 0}
4: for i in 1..j do
5:     conj ← solve Equation 6-9 on P+ and N−
6:     Add conj to DNF
7:     Remove from P+ all components covered by conj
8:     if P+ is empty then
9:         return DNF                          ▷ Perfect classifier, done
10:    end if
11: end for
12: return DNF
```

features, we first hash any non-numeric features (e.g., color, shape) and then double one-hot encode all attributes, such that there is a Boolean for each attribute-value combination as well as its negation (e.g. 'color == orange' and 'color != orange'; 'shape == ✚' and 'shape != ✚' etc).

Instead of enumerating all conjunctions up to a fixed number of conjuncts and selecting from those, we use an *implicit* generation approach where we formulate the problem of generating a single conjunction through constraint optimization. We make use of constraint programming for item set mining (Guns, Nijssen, & De Raedt, 2013) to formulate the following constrained optimization problem:

$$maximize_S \quad w\sum O^+ - \sum S \tag{6}$$
$$s.t. \quad O^+ = cover(S, P) \tag{7}$$
$$O^- = cover(S, N) \tag{8}$$
$$\sum O^- = 0 \tag{9}$$

where S ($|S| = m$) contains a Boolean decision variable for every Boolean attribute in the double one-hot encoded component representation, P and N are the positive and negative components (their representations), O^+ and O^- contain a Boolean decision variable for every positive/negative component that represents whether the component is covered by the conjunction S or not. The constraints on lines (7) and (8) compute which objects in P and N are covered by S. The constraint on line (9) ensures no negative components are covered, and the objective function on line (6) is a lexicographic optimization that first maximizes the number of covered P and then minimizes the number of Boolean attributes used in the conjunction. The pseudocode of the overall DNF learner is in Algorithm 3.

While the optimization problem has to search over a worst-case exponential number of conjunctions, there are only a few examples, and the constraints provided by negative instances greatly limit the search space. In practice, constraint solvers find solutions to this problem very rapidly. In the running example (Figure 1a), the learned DNFs contain a single conjunction with a single Boolean attribute set to true.

```
if o.shape == ▫ then o.replace_by(➕).color(green)
if o.shape != ▫ then o.identity()
```

To summarize, a transformation rule consists of a condition and a consequent. Conditions are concepts about sets of components. Consequents are transformation programs over domain-specific transformation primitives. Multiple transformation rules, together with a decomposition function, make up a solution program. When a solution program is executed on a test input, the input is first decomposed into components, which are then evaluated against each of the transformation rules of the program. Whenever a rule condition evaluates to true, it triggers the execution of the corresponding transformation program on the current input component completing the DA&C paradigm.

5. Experiments

Following our description of the DA&C paradigm, we claim that the use of segmentation and analogical matching for structured domains enables agents to learn complex programs in less time. To this end, we evaluate BEN, our implementation of DA&C using standard top-down enumerative synthesis, across two domains: abstract visual reasoning used as a running example throughout the paper (Section 5.2) and string transformation tasks (Section 5.1). We select those because they cover the spectrum from an established experimentation domain for program synthesis (string transformations) to a challenging new domain benchmark that has proven difficult for current synthesis strategies. Our experiments seek to investigate three research questions:

Q1 How does the predictive accuracy of BEN behave in comparison to state-of-the-art ILP systems? We pay special attention to how their predictive accuracies are influenced by the size of the example sets.

Q2 How does the use of analogical matching on structured real-world synthesis tasks influence the runtime complexity of BEN which is expected to run with a worst case time complexity of $O(n^2)$ with n being the number of segmented components in the example input/output?

Q3 What is the effect of growing solution programs on solution times of BEN, especially in the context of real-world structured domains in which solution programs are frequently observed to grow linearly in the number of independent subprograms?

All experiments are run on a desktop with a single M1 Max CPU and 64GB of RAM. Our code and an overview of the tasks solved by BEN are available on GitHub.

5.1 Experiment 1: String Transformations

We evaluate our approach using the established domain of string transformation tasks.

Materials. We experiment with a publicly available data set of 130 real-world string transformation tasks from Cropper et al. (2020). The initial subset of tasks was curated by Gulwani (2011) from online Microsoft Excel forums, later expanded by Lin, Dechter, Ellis, Tenenbaum, and Muggleton (2014) with additional handcrafted spreadsheet manipulations and has been repeatedly used as a benchmark in program synthesis.

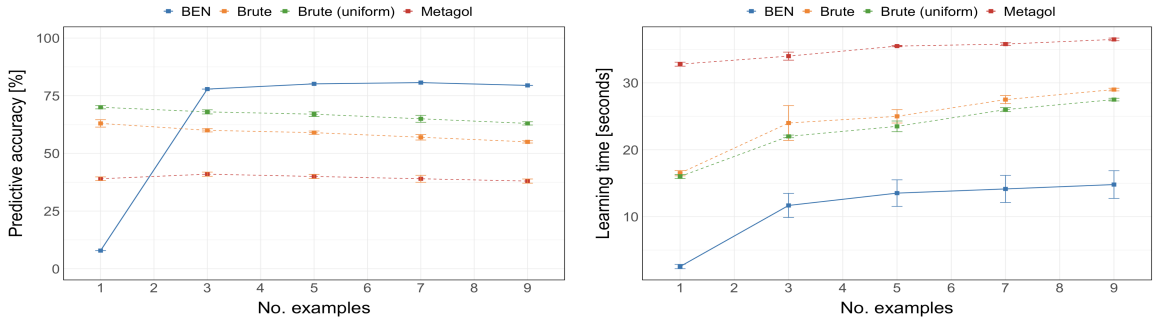
For this experiment, we provide BEN access to string features (Table 2) and primitives used to manipulate character sequences (Table 3). We make sure all agents are evaluated using a set of similar manipulation primitives. We define the decomposition grammar $\mathcal{G}_{decomp}$ over subsets of non-alphanumerical characters (e.g., whitespace, comma, dot). A decomposition function δ is one that splits a string based on the characters defined in this subset. In addition, we consider numeric characters to be possible delimiters and include a total decomposition that splits strings into individual characters. We chose this decomposition grammar because it reflects much of the compositional structure of real-world strings. At the same time, it is already too expressive to be searched by naive enumeration. During decomposition, delimiters are appended to the preceding component. We include *drop_last()* as a primitive operation into BEN's knowledge base, which can be used to access components without the trailing delimiter.

BEN is evaluated against Brute (Cropper et al., 2020) and Metagol (Muggleton, Lin, & Tamaddoni-Nezhad, 2015; Cropper & Muggleton, 2016), two state-of-the-art ILP systems for learning recursive programs. **Brute** performs best-first-search guided by an example-dependent loss function. It was specifically designed for the synthesis of large programs. In

Object features - String transformation tasks

ID	Feature	Explanation	Derived from
1	content	list of characters	
2	index_front	index position of substring from the front	
3	index_back	index position of substring from the back	
4	index_even	boolean: even front index	index_front
5	length	number of characters	content
6	number_of_uppers	number of capitalized characters	content
7	number_of_lowers	number of lowercase characters	content
8	number_of_digits	number of digits	content
9	number_of_alphas	number of alphabetical characters	content
10	number_of_alnums	number of alphanumeric characters	content
11	all_upper	boolean: all capitalized characters	content
12	all_lower	boolean: all lowercase characters	content
13	all_digits	boolean: all digits	content
14	starts_with_upper	starts with capitalized character	content

Table 2: Object features used in BEN on string transformation tasks.



(a) Predictive accuracy.

(b) Mean learning time.

Figure 8: Performance on the real-world string transformations data set. We report first-found solutions for all agents. All agents were executed with a time budget of 60s per task.

addition, we also compare against a version of Brute with a traditional entailment-based loss function. **Metagol** works with user-specified meta-rules that serve as a declarative bias on the type of clauses considered as hypotheses, thus limiting the search space. We supply Metagol with the *identity*, *inverse*, *precon*, *postcon*, *chain* meta-rules as recommended for learning dyadic programs by Cropper and Muggleton (2016).

BEN achieves significantly higher predictive accuracy than Brute and Metagol, even though BEN cannot learn recursive programs and cannot perform predicate invention. Its predictive accuracy is 17 percentage points above that of Brute and double that of Metagol on tasks with nine input examples **Q1**.

BEN solves over 80% of test examples given 2-3 training examples per task. In comparison, Brute achieves a peak predictive accuracy of only 70%. Its peak performance is lower because Brute has to find the entire solution program within a single synthesis search, which

Transformation primitives - String transformation tasks

ID	Transformation	Explanation	Arguments
1	drop_first(n)	drops leading n characters	n $\in \{1...len(o_i)\}$
2	drop_last(n)	drops last n characters	n $\in \{1...len(o_i)\}$
3	take_from_front(n)	selects leading n characters	n $\in \{1...len(o_i)\}$
4	take_from_front(n)	selects last n characters	n $\in \{1...len(o_i)\}$
5	to_uppercase()	capitalizes string	
6	to_lowercase()	lowercase string	
7	capitalize_first()	capitalizes first character	
8	add_space()	appends white space	
9	add_dot()	appends dot	
10	add_comma()	appends comma	
11	replace(s)	replace with string s	s is given by the output string o_j

Table 3: Transformation primitives used in BEN on string transformation tasks.

becomes exponentially more difficult in the size of the program. For example, in order to extract the CPU usage '*95%*' from a command line output '*16,079 inferences, 0.003 CPU in 0.003 seconds (5842660 Lips, 95% CPU)*', Brute learns a logic program that consists of 9 clauses and 31 literals that recursively deletes characters from the front and back of the input string until only the CPU usage remains. BEN, in comparison, learns a program that first segments the input using a whitespace delimiter and extracts from it the second-to-last substring from which it deletes the trailing whitespace.

DA&C leads to a speed-up in synthesis, which more than makes up for the additional time needed to compute a decomposition, find a meaningful structural alignment between substrings in the input/output, and learn concepts for each transformation program. This shows in the average learning times in Figure 8b. BEN consistently performs below the average learning times of both Brute and Metagol. On tasks with nine examples, Brute runs 28 s on average, and Metagol runs 37 s, while it only takes BEN 15 s to process a task on average. Reported learning times for BEN include the time spent searching for a segmentation, encoding the segmented components, aligning the input/output, and learning concepts for individual transformation programs.

BEN, compared to Brute and Metagol, shows a unique trend in predictive accuracy over the number of input examples. Brute reaches its peak performance on tasks with a single training example. Its performance monotonically decreases to 63% as more training examples are added. More training examples make it more difficult to find hypotheses that cover this growing set of examples (Figure 8a). BEN, on the other hand, does not show a degradation in predictive accuracy on larger example sets. Its predictive accuracy on tasks with three training examples is not significantly different from its predictive accuracy on tasks with nine training examples **Q1**. The reason for this is that BEN only performs synthesis on component tuples and not sets of training examples where the complexity of a single synthesis loop is independent of the number of total training examples in a task. The number of examples directly impacts the second part of the *conquer* stage, the concept learning. This is because in order to learn a concept, every component is assigned one of three labels (positive, negative, neutral see Section 4.3.2), which requires the execution of the transformation program on each input component. The impact shows in BEN's average learning times, which follow a logarithmic increase. As the number of training examples increases by a factor of three, mean learning times increase by 50%. Notably, established ILP methods such as Brute and Metagol don't show a saturation in learning times because their synthesis loop requires that each program candidate gets evaluated across all training examples (Figure 8b).

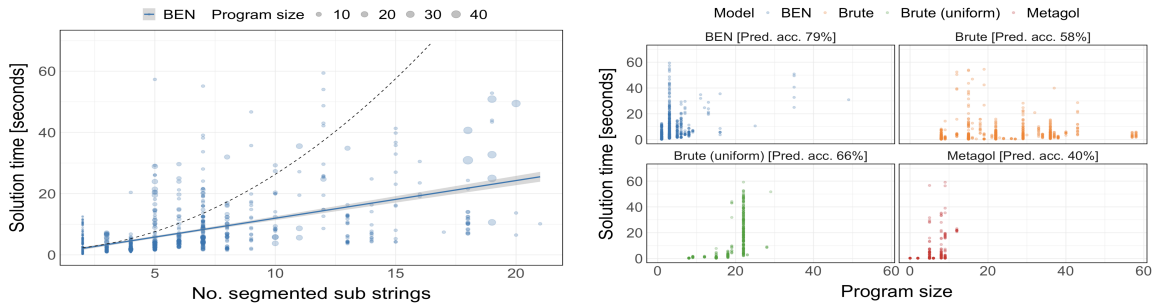
If the training set consists of only a single example, BEN cannot determine how well a transformation program generalizes to other training examples (Figure 8a). As a consequence, it tends to learn overly specific transformations (e.g., extensive use of *shape* operations). In much the same way, the concept learning in BEN requires a minimum amount of 2-3 positive examples in order to learn an informative selector with high generalization power to hold-out test examples (and their components). Brute and Metagol perform significantly better on tasks with only a single example because they have better knowledge priors on what constitutes a generalizable program. Missing segmentation primitives are the primary reason why BEN does not solve more tasks. BEN cannot segment examples

with arbitrary regex expressions; for instance, the letter sequence 'aabb' would be needed in order to extract the first substring from the following input 'a38bz2saabb21u17a' → 'a38bz2s'.

Finally, we compare empirical solution times of BEN against its theoretical worst case bound of $\mathcal{O}(n^2)$ in the number n of substrings in the input and output (dashed graph in Figure 9a). The number of substrings per example significantly influences learning times even after controlling for the size of the solution program ($\mathrm{F}(3, 1137) = 218.8, p < .001$) with an explained variance of 36%. This is to be expected because examples with more components will have a larger set of plausible component tuples to search through. In practice, useful transformations are already recovered after only a few synthesis loops well below the expected theoretical worst-case bound **Q2**. The search space of Brute (uniform with logical entailment) and Metagol exponentially grows in the size of $\mathcal{T}$, whereas the search space in BEN remains constant in size and is traversed a linear m times depending on the number m of transformation rules in a solution program. The total size of the solution program shows no exponential impact on learning times in BEN **Q3** (Figure 9b). When Brute uses an example dependent loss function instead of logical entailment, its solution times also do not show an exponential trend with larger programs. The loss function it uses to guide the synthesis search is the reconstruction accuracy in the output string, which is similar to BEN's evaluation of reconstructed substrings (components) in the output. By working with substrings instead of characters, however, BEN not only solves more tasks but also learns much shorter solution programs on average (Figure 9b). Even though Metagol also learns compact solutions programs fast, its predictive accuracy is only half of that of BEN, which makes a direct comparison difficult.

5.2 Experiment 2: Abstract Visual Reasoning

In order to demonstrate the effectiveness of DA&C on program synthesis in high-dimensional domains, we apply BEN to abstract visual reasoning tasks. In this setting, we compare to



(a) Solution times per segmented substrings.

(b) Solution times per program size.

Figure 9: Time complexity on string transformation tasks. We report first-found solutions on tasks with at least three input examples for all agents. Only tasks solved within the time budget of 60 s are considered.

the state-of-the-art agents outside of ILP (Section 5.2.1) and evaluate different ablated baselines of BEN (Section 5.2.2).

Materials. We use the training part of the Abstraction and Reasoning Corpus (ARC) (Chollet, 2020) (Apache 2.0 license), which consists of 400 data sets that each contain 2-10 examples comprising an input and an output where the outputs are generated by an unspecified program that we wish to synthesize. The language of programs is determined by developers and agents themselves; the benchmark does not specify a language over programs.

5.2.1 Performance Benchmark

We compare BEN against ARGA (Abstract Reasoning with Graph Abstractions) (Xu, Khalil, & Sanner, 2023), an agent specifically developed for ARC. It is equipped with object-centric priors, which it uses first to segment examples into scenes of objects and then perform greedy best-first search over graph representations using a DSL to manipulate the 'scene graph'. We also compare against the winning submission of the ARC Kaggle competition, which makes use of a custom-engineered DSL with handcrafted primitives and a performance-optimized implementation in C++ to perform brute-force search over bitmap manipulations (Wind, 2020). We evaluate BEN with the object features and geometric transformation primitives introduced in the main section of this paper (Table 1 and Section 4.3).

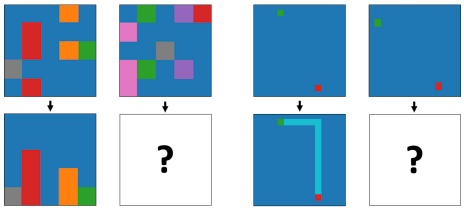
We first report results on the entire ARC dataset and then move on to its subsets for 'movement', 'recoloring', and 'augmentation' tasks defined by Xu et al. (2023). The Kaggle agent solves the most tasks overall, close to 47% within a 2 min time budget. Its performance-optimized implementation in C++ allows it to generate and test over 1.3 Mio programs per task, which it does in 35 s on average. BEN solves about half as many tasks for which it only generates an average of 0.2% as many candidate programs per task. The use of program 'holes' in the hierarchical DA&C search provides a succinct solution space in which arguments are deduced from the components in the examples instead of being searched top-down as done by the Kaggle agent. BEN takes an average of 23 s to solve a task, which is competitive with the Kaggle agent despite its Python implementation. Both BEN and the ARGA agent use a significantly reduced DSL compared to the Kaggle winner. Whereas the Kaggle agent has access to 42 unique transformation primitives, many of which were handcoded to solve specific tasks, BEN and ARGA purely rely on 11 generic geometric primitives. Despite sharing the same number of primitives, BEN solves twice as many tasks and generates three times fewer candidate programs than ARGA. BEN also consistently solves more tasks on the 'augmentation', 'movement', and 'recolor' subsets than ARGA but cannot quite reach the performance of the Kaggle agent for the restrictive setting of a 2 min time budget per task. Notably, BEN consistently generates far fewer candidate programs than any of the other two agents. On the 'movement' and 'recolor' subsets, this even results in a reduction of the search space by one order of magnitude compared to the graph-based representation of ARGA, which also makes heavy use of segmentation priors.

Knowledge Priors. We now take a look at specific task examples and work towards an ablation analysis of BEN's DA&C framework in order to investigate which part of its

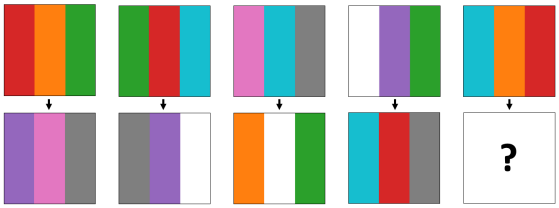
Data set	Model	Number solved	Candidates explored	Avg. solution time
ARGA - augmentation	#1 Kaggle	**24/67** (35.82%)	1932368	35.90s
	ARGA	15/67 (22.39%)	7282	17.51s
	BEN	19/67 (28.36%)	6818	29.60s
ARGA - movement	#1 Kaggle	**19/31** (61.29%)	1906875	34.48s
	ARGA	9/31 (29.03%)	12233	14.22s
	BEN	13/31 (41.94%)	1743	17.83s
ARGA - recolor	#1 Kaggle	**23/62** (37.10%)	1571957	30.10s
	ARGA	18/62 (29.03%)	18851	25.63s
	BEN	22/62 (35.48%)	2304	23.42s
All	#1 Kaggle	**186/400** (46.50%)	1293492	34.65s
	ARGA	45/400 (11.50%)	12412	19.75s
	BEN	90/400 (22.50%)	3311	23.20s

Table 4: Results on the training part of the Abstraction and Reasoning Corpus (ARC). First-found solutions within a maximum time budget of 2 min per task reported. The #1 Kaggle agent and BEN were both run at a search depth of four on all tasks. During the actual Kaggle competition, the #1 Kaggle agent ran as an ensemble at different search depths to optimize scheduling. We report more challenging results at a search depth of four. The ARGA agent does not expect a search depth as input; we use a time budget of 2 min.

performance is due to better search versus only a refined domain-specific grammar. Most of the tasks that BEN does not solve are due to either missing transformation primitives or control flow logic in its current grammar $\mathcal{G}_{ARC}$. The two tasks in Figure 10a are examples of that: The first task requires a solution program that gravitates objects to the bottom of the frame. This is a complex motion program that has to enforce that objects closest to the bottom frame are translated first and that objects are only moved if their bottom neighboring pixel is empty. The second task is a version of a shortest path problem where the solution program is expected to find a trajectory that minimizes turns. Both of these tasks are only solved by the Kaggle agent (Wind, 2020) because it has access to handcrafted primitives in its prior transformation library, namely a 'gravity' operation and a 'shortest path' function, which were custom-engineered by its developer. Some of its 42 primitives only get used on one or a few tasks, whereas BEN only has access to 11 generic geometric transformations. In other words, the performance of the current state-of-the-art agent is largely due to hand-crafted, ad-hoc primitives rather than better search. No combination of primitives in BEN's library (independent of the final size of a possible solution program) can reason about the path between two objects unless that path is a straight line. To make the comparison fairer, we conservatively reduce the agent's primitives to the ones that semantically match those in the domain grammar $\mathcal{G}_{ARC}$ of BEN (leaving 30 primitives). The matching was performed manually by studying the implementation of the primitives in the Kaggle agent. In this case, the performance of the Kaggle agent drops from 47% to less than 25%. That is, BEN solves a comparable amount of tasks with only a third of the primitives, which is possible only because BEN decomposes tasks into recurring components that lead to simpler transformations with a more compact search space. BEN's search space is



(a) BEN does not solve either one of these two tasks, unlike the state-of-the-art Kaggle agent, which has handcrafted primitives, one to 'simulate gravity' and another primitive that solves shortest path problems.



(b) BEN executes a total of eight synthesis loops to retrieve the eight binary color swap transformations that make up the solution program, instead of synthesizing eight binary color swap operations in a single deep program search.

Figure 10: Performance on example visual reasoning tasks taken from ARC.

more compact because a portion of the 30 primitives in the Kaggle agent serves as segmentation (e.g., filter by color) and selector functions (e.g., pick largest bitmap). In DA&C, segmentation and selector primitives are part of the divide phase and the concept learning. They don't influence the size of the program search space in the main synthesis loop. In fact, selector primitives are not even searched but are entirely inferred from components by induction. Because the synthesis loop in the Kaggle agent is confounded with segmentation and selector primitives, it has to search through orders of magnitudes more candidate programs to achieve comparable expressiveness to that of the main synthesis loop in BEN augmented with a standalone segmentation and concept learning phase. Another related reason why DA&C leads to a more compact search space is the combinations of segmentation and selector primitives in the DSL of the Kaggle agent. For instance, it includes a 'cutPickMax' primitive that combines the 'cut' and 'pickMax' primitives into a standalone function, presumably to make it available at lower search depths. BEN learns programs with the same semantics but offloads the 'pickMax' to the inductive concept learner, which it can also apply to any other geometric primitive besides 'cut' with no impact on the search space within the main synthesis loop. For the Kaggle agent, the computational challenge of synthesizing complex programs in a high-dimensional domain was overcome by handcrafted subprograms that outsourced the cognitive effort to the developer rather than the program synthesis. Instead, we use ARC to demonstrate progress in the synthesis of complex programs from low-level primitives. BEN uniquely solves 8% of the data set on which the Kaggle agent fails, even with its original transformation library of 42 primitives.

By splitting a deep program search across a linear number of much shallower synthesis searches, BEN is able to solve tasks that were previously out of reach for state-of-the-art agents on ARC. For example, the task in Figure 10b requires an agent to learn a program that consists of eight binary color swap operations. If an agent were to learn all eight operations in a single program search, the required search depth during synthesis is also eight - too deep for most grammars to remain exhaustively searchable. However, the color swap operations can be quickly learned from individual object tuples. In this case, the maximum search depth is reduced to one, and the same search space is traversed eight times. BEN rapidly finds solutions to this and similar tasks.

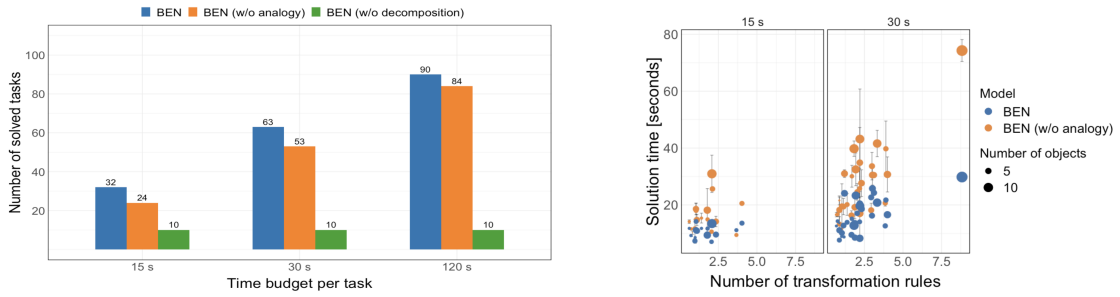
5.2.2 Ablation Analysis

We perform an extensive ablation analysis covering all stages of the DA&C paradigm, and also compare against two ablated versions of BEN: one without analogical reasoning, which performs synthesis on random object tuples and iterates through them until it has discovered enough transformation programs to solve the task. The other, without segmentation at all, performs synthesis on the input/output images directly.

The results in Figure 11a show that analogical reasoning helps BEN (blue bars) solve more tasks in less time. In the restrictive case of a 15 s time budget per task, BEN solves roughly 33% more tasks than its ablated baseline without analogical reasoning. The performance lead decreases to 7% for a 2 min time budget as the random search eventually covers a larger fraction of the total search space. Without segmentation, less than 12% of tasks are being solved within 2 min. We verified that each of the tasks solved by either of the two baselines is also solved by BEN.

BEN shows an average solution runtime of 15 s per task which is significantly lower than that of the random baseline, with a mean of 20 s, evaluated using a Wilcoxon signed-rank test on paired samples Z=160, p<.001 for a time budget of 30 s per task (Figure 11b). The solution times include the time spent on segmentation and the time needed to identify meaningful object correspondences, which shows that the additional computational effort of SME (structural alignment of the input/output examples) is limited and easily compensated for in the overall DA&C algorithm.

The solution times in BEN remain low even for crowded scenes with a large number of decomposed objects (β = 1.25, p<.001), whereas the random baseline without analogical reasoning does not scale to larger scenes (β = 3.81, p<.001) consistent with previous observations on string transformation tasks **Q2**. The program size in Figure 11b refers to the number of transformation programs in the first found solution program, where each of the transformation programs consists of multiple primitives and variables. The number of transformation programs is a lower bound on the number of synthesis searches that BEN needs to perform. Depending on the order in which correspondences are selected, more synthesis searches will be necessary. On average, BEN runs 4.7 synthesis searches on a task



(a) Performance of BEN's ablated baselines.

(b) Solution times per program size.

Figure 11: Performance benchmark on the abstract visual reasoning data sets from ARC. We report first-found solutions for all ablations. (Time budgets in Figure 11b are only enforced for BEN.)

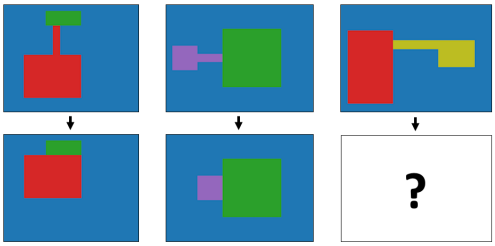
Component	Ablation	30 s	120 s
BEN		63	90
Divide	w/o multicolor objects	-24%	-33%
	w/o diagonal neighbors	-5%	-22%
Conquer: synthesis search	depth ≤ 2	-37%	-50%
	depth ≤ 3	+25%	-3%
	+25% nonsense primitives in $\mathcal{G}_{transform}$	-22%	±0%
Conquer: concept learner	primary features	-34%	-27%
	derived features	-24%	-10%
	+50% nonsense attributes in $\mathcal{G}_{concept}$	±0%	±0%

Table 5: An ablation analysis of BEN's core architecture design and domain-specific knowledge priors shows that the contents of its transformation grammar $\mathcal{G}_{transform}$ and its component attributes $\mathcal{G}_{concept}$ are especially robust to nonsense information.

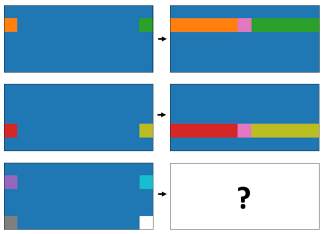
with an average number of 16.4 correspondences, where the first found solution program contains an average of 2.5 transformations. This means BEN explores about twice as many correspondences as would be minimally required to solve a task but only a fourth of all possible correspondences. This explains why program size is not a significant predictor of task solution times **Q3**. Instead, the interaction of analogical program synthesis and the number of objects within a scene is a significant predictor of task solution times ($F(3,176)=24.6$, $p<.001$). This confirms that the guidance from analogical synthesis is especially helpful for tasks that require learning transformations over large scenes with many components.

In order to investigate the impact of the DA&C design on BEN's performance, we systematically manipulate its search as well as its domain grammars and report the ablations in Table 5. Most notably, adding as much as 50% additional nonsense attributes to component encodings $\mathcal{G}_{concept}$ or 25% additional nonsense primitives to the transformation grammar $\mathcal{G}_{transform}$ only has a small effect on BEN's performance. In fact, adding more nonsense features neither caused BEN to miss tasks nor did it increase solution times. This is because we learn concepts bottom-up on successful transformation programs instead of searching for them as logical formulas in a domain grammar, which is what ARGA does. Additional transformation primitives prolong the time of a single synthesis search and, therefore, cause BEN to timeout on 22% of tasks it had previously solved within a challenging 30 s time budget per task. Its performance under a time budget of 120 s remains unaffected. We present this as evidence that the domain-specific knowledge provided to BEN has not been custom-engineered, and its performance is mainly due to intelligent hierarchical search in the DA&C framework. Generally, reducing the depth of the synthesis search leads to decreased performance. On small time budgets, however, a slight reduction in search depth can have the opposite effect when the majority of subprograms (needed for a solution program) can be retrieved from a shallow synthesis search. As for the decomposition language, its expressiveness (e.g., diagonal neighborhood, multicolor segmentation) directly impacts the agent's task performance.

1-1 Correspondences. On some of the tasks, it is counter-intuitive how the reliance on pairwise correspondences is improving rather than restricting search. We find that it is a viable heuristic to guide the program search even in cases where there exist no obvious 1-1



(a) BEN learns a program that moves both objects together, thereby compressing the shape of the elongated object.



(b) BEN also invents new objects in the output and can take into account information from the input, such as the x-coordinate at which an object should be created.

Figure 12: BEN also solves ARC tasks that involve n-1 mappings or require the generation of new components in the output.

mappings between input and output components. For instance, n-1 mappings frequently occur in visual reasoning tasks in which information from multiple input components is pooled (e.g., shape and color from two different components) to reconstruct a component in the output. Another example of n-1 mappings is directed movements towards other objects (Figure 12a). In this example, BEN still finds the correct transformation starting from any of the 1-1 correspondences within the n-1 mapping because its transformation grammar is able to refer to knowledge from other input components (Section 4.3). We have introduced the 'REFERENCE' non-terminal in $\mathcal{G}_{transform}$ to make this explicit. The references in this paper are hard coded ('largest object', 'other object' ...), but this does not need to be the case. Similar to how DA&C uses constraint solving to learn a logic formula over input components that make use of the same transformation program, constraint solving could be used to learn a logic formula over 'referred components' within transformation programs. BEN also solves tasks that require the invention of new objects in the output, such as the example in Figure 12b. The correspondence that BEN leverages to synthesize the pink pixel could be with any of the two pixels in the input. If the alignment produces 1-1 correspondences that don't yield a meaningful transformation program, those will never cause BEN to miss a task that it had otherwise solved because the alignment is only used as a heuristic on the order in which to search through the program space. In the worst case, BEN explores the Cartesian product of all correspondences between the segmented input and segmented output. We emphasize that DA&C is not a universal strategy for all synthesis problems. It is ideally suited for those with compositional structure. If task examples have no inherent compositional structure, BEN degrades to program synthesis search on the unsegmented input/output examples.

Importance of the family of decomposition functions δ. In order to analyze the impact of the prior knowledge of $\mathcal{G}_{decomp}$ on BEN's performance, we conducted an exploratory study on the solution strategies of human task solvers ($N = 4$) (Witt, Dumancic, Guns, & Carbon, 2021): Participants were presented with ARC tasks on a web interface which asked them to reconstruct the missing output for each test case and to highlight individual objects in each image tuple $(I_l, R_l) \in \mathcal{Q}$. We used reconstruction accuracy on test cases to assess

whether participants understood a task. Only the image segmentations from successful task solvers were considered in the subsequent analysis.

We find that the segmentations produced by BEN perfectly match those of human task solvers for roughly 38% of the tasks. Whenever a segmentation deviates from those of human participants, BEN is less likely to solve a task. The introductory ARC task in Figure 1a is an example of the contrary: BEN finds an alternative segmentation in which directly neighboring light blue and green pixels are merged into the same object. The output is colored light blue whenever a multicolored object exists in the input and its width is larger than three. This solution program does not fully capture the semantics of the task but is sufficient to solve all test cases. In order to estimate the percentage of tasks that BEN fails to solve due to insufficient segmentation, we randomly choose 50 failed tasks and directly execute BEN on segmentations produced by human task solvers. BEN then solves 16% of these previously unsolved tasks, which suggests that missing transformation primitives and object features are the main bottlenecks instead of the current segmentation grammar.

6. Discussion and Future Work

Humans effortlessly induce large programs that generalize well to previously unseen test cases, even in a few-shot learning setting, as in ARC or the real-world string transformations data set. Our work suggests that program synthesis can exploit the compositional nature of structured domains to guide the search for well-generalizing programs in vast language spaces and in high-dimensional domains. That is done by decomposing the problem of learning a single nested program into a two-stage process: first, we find a segmentation of examples into meaningful components and then perform multiple program synthesis tasks on component tuples in the input/output. Second, separating the problem of searching for component-specific transformations from the task of learning the contexts in which they apply (the concept definition) leads to better generalizing programs. We implemented the DA&C paradigm in our agent BEN using top-down enumerative search as a synthesis technique. Future work that evaluates the integration with other advanced synthesis techniques appears promising. DA&C is a program synthesis framework that exploits the compositional structure in task examples and otherwise degrades to whichever synthesis technique it uses in its main loop.

Although BEN solves more string transformation tasks than state-of-the-art ILP baselines and a fair share of highly heterogeneous visual reasoning tasks, it does not yet match the performance of an average human task solver. Humans seem to use additional strategies that deviate from DA&C and traditional search-based synthesis in important ways. For example, humans make efficient use of context switching, where the synthesis has access to only those object features and transformation primitives that appear important to the task at hand. This reduces the load of having to process many different encodings all at once and limits the search space. Neurally-guided search could be a computational means to this, which would require an additional learning component or the neural implementation of one of the DA&C subroutines.

It is well established in the cognitive sciences that higher-order functions in human cognition, such as abstract reasoning over programs, influence the grouping of perceptual

units and the encoding of their features. In the future, we will explore how the decomposition phase could be conditioned on the progress of the synthesis to group together what is likely to act as a single functional unit in the solution program.

Finally, the assumption of independent object transformations is a simplification. Humans are readily able to find a suitable sequential order of object transformations, for example, to make use of overpainting. The methods proposed in this paper do not rely on independent object transformations per se, and hence, there is a need to investigate ways of composing programs with complex dependencies between their transformations.

Acknowledgments

The authors wish to thank Stef Rasing for his help in implementing and evaluating BEN on string transformation tasks.

References

Alur, R., Radhakrishna, A., & Udupa, A. (2017). Scaling enumerative program synthesis via divide and conquer. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS-17)*, Vol. 10205 of *Lecture Notes in Computer Science*, pp. 319–336.

Alur, R., Singh, R., Fisman, D., & Solar-Lezama, A. (2018). Search-based program synthesis. *Communications of the ACM*, 61(12), 84–93.

Alur, R., Černý, P., & Radhakrishna, A. (2015). Synthesis through unification. In *Proceedings of Computer Aided Verification (CAV-15)*, Vol. 9207 of *Lecture Notes in Computer Science*, pp. 163–179.

Chollet, F. (2019). On the Measure of Intelligence. ArXiv. https://doi.org/10.48550/arXiv.1911.01547.

Chollet, F. (2020). The Abstraction and Reasoning Corpus (ARC). GitHub. https://github.com/fchollet/ARC.

Cropper, A. (2022). Learning logic programs through divide, constrain, and conquer. In *Proceedings of the Conference on Artificial Intelligence (AAAI-22)*, pp. 6446–6453.

Cropper, A., & Dumančić, S. (2020). Learning large logic programs by going beyond entailment. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-20)*, pp. 2073–2079.

Cropper, A., & Dumančić, S. (2022). Inductive Logic Programming At 30: A New Introduction. *Machine Learning*, 111(1), 147–172.

Cropper, A., Morel, R., & Muggleton, S. H. (2020). Learning higher-order programs through predicate invention. In *Proceedings of the Conference on Artificial Intelligence (AAAI-20)*, pp. 13655–13658.

Cropper, A., & Muggleton, S. H. (2016). Metagol System. GitHub. https://github.com/metagol/metagol.

de Miquel, A., Corominas, R. G., & Ariyasu, Y. (2020). ARC Kaggle competition: 2nd place solution. Kaggle.

Dumančić, S., Guns, T., & Cropper, A. (2021). Knowledge refactoring for inductive program synthesis. In *Proceedings of the Conference on Artificial Intelligence (AAAI-21)*, pp. 7271–7278.

Ellis, K., Nye, M. I., Pu, Y., Sosa, F., Tenenbaum, J., & Solar-Lezama, A. (2019). Write, execute, assess: Program synthesis with a REPL. In *Proceedings of Neural Information Processing Systems (NeurIPS-19)*, pp. 9165–9174.

Ellis, K., Solar-Lezama, A., & Tenenbaum, J. (2015). Unsupervised learning by program synthesis. In *Proceedings of Neural Information Processing Systems (NeurIPS-15)*, pp. 973–981.

Evans, T. G. (1964). A heuristic program to solve geometric-analogy problems. In *Proceedings of the Spring Joint Computer Conference (AFIPS-64)*, pp. 327–338.

Falkenhainer, B., Forbus, K. D., & Gentner, D. (1989). The Structure-Mapping Engine: Algorithm and Examples. *Artificial Intelligence*, 41(1), 1–63.

Gentner, D. (1983). Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, 7(2), 155–170.

Gulwani, S. (2011). Automating string processing in spreadsheets using input-output examples. In *Proceedings of Principles of Programming Languages (POPL-11)*, pp. 317–330.

Gulwani, S., & Jain, P. (2017). Programming by examples: PL meets ML. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS-17)*, Vol. 10695 of *Lecture Notes in Computer Science*, pp. 3–20.

Guns, T., Nijssen, S., & De Raedt, L. (2013). k-Pattern set mining under constraints. *IEEE Transactions on Knowledge and Data Engineering*, 25(2), 402–418.

Henderson, R., & Muggleton, S. (2014). Automatic invention of functional abstractions. In Muggleton, S., & Watanabe, H. (Eds.), *Latest Advances in Inductive Logic Programming*, pp. 217–224. Imperial College Press, London, UK.

Hofstadter, D. (2001). Analogy as the core of cognition. In Gentner, D., Holyoak, K., & Kokinov, B. (Eds.), *The Analogical Mind: Perpectives from Cognitive Science*, pp. 499–538. MIT Press, Cambridge, MA, USA.

Johnson, A., Vong, W. K., Lake, B. M., & Gureckis, T. M. (2021). Fast and flexible: Human program induction in abstract reasoning tasks. ArXiv. https://doi.org/10.48550/arXiv.2103.05823.

Lin, D., Dechter, E., Ellis, K., Tenenbaum, J., & Muggleton, S. (2014). Bias reformulation for one-shot function induction. In *Proceedings of the European Conference on Artificial Intelligence (ECAI-14)*, pp. 525–530.

Lovett, A., & Forbus, K. D. (2017). Modeling visual problem solving as analogical reasoning. *Psychological Review*, 124(1), 60–90.

Mitchell, M. (1993). *Analogy-making as perception: A computer model.* MIT Press, Cambridge, MA, USA.

Mitchell, M. (2021). Abstraction and analogy-making in artificial intelligence. *Annals of the New York Academy of Sciences*, 1505(1), 79–101.

Muggleton, S. H., Lin, D., & Tamaddoni-Nezhad, A. (2015). Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, 100(1), 49–73.

Neider, D., Saha, S., & Madhusudan, P. (2016). Synthesizing piece-wise functions by learning classifiers. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS-16)*, Vol. 9636 of *Lecture Notes in Computer Science*, pp. 186–203.

Nye, M. I., Pu, Y., Bowers, M., Andreas, J., Tenenbaum, J. B., & Solar-Lezama, A. (2021). Representing partial programs with blended abstract semantics. In *International Conference on Learning Representations (ICLR-21)*. https://openreview.net/forum?id=mCtadqIxOJ.

Raza, M., & Gulwani, S. (2017). Automated data extraction using predictive program synthesis. In *Proceedings of the Conference on Artificial Intelligence (AAAI-17)*, pp. 882–890.

Snow, R. E., Kyllonen, P. C., & Marshalek, B. (1984). The topography of ability and learning correlations. In Sternberg, R. J. (Ed.), *Advances in the psychology of human intelligence*, pp. 47–103. Erlbaum, Hillsdale, NJ, USA.

Sumit Gulwani (2023). Microsoft Program Synthesis using Examples (PROSE) SDK. GitHub. https://github.com/microsoft/prose/tree/main.

Valiant, L. G. (1985). Learning disjunction of conjunctions. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-85)*, pp. 560–566.

Wagemans, J., Elder, J., Kubovy, M., Palmer, S., Peterson, M., Singh, M., & Heydt, R. (2012). A century of gestalt psychology in visual perception: I. Perceptual grouping and figure-ground organization. *Psychological Bulletin*, 138(6), 1172–1217.

Wang, C., Cheung, A., & Bodik, R. (2017). Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI-17)*, pp. 452–466.

Wind, J. S. (2020). ARC Kaggle competition: 1st place solution. Kaggle.

Witt, J., Dumancic, S., Guns, T., & Carbon, C.-C. (2021). A grammar-based description of perceptual organization for scene segmentation. *Perception*, 50(1S), 84–84.

Xu, Y., Khalil, E. B., & Sanner, S. (2023). Graphs, Constraints, and Search for the Abstraction and Reasoning Corpus. In *Proceedings of the Conference on Artificial Intelligence (AAAI-23)*, pp. 4115–4122.