Domain-Specific Language for the Abstraction and Reasoning Corpus

Michael Hodel
mihodel@ethz.ch

March 2023

Contents

Chapter 1

Introduction

1.1 The Abstraction and Reasoning Corpus

The Abstraction and Reasoning Corpus (ARC) [1] is a dataset, intended to serve as a general artificial intelligence benchmark, published alongside the paper On the Measure of Intelligence [2]. ARC consists of 1000 tasks, 800 of which are public (400 intended as a training set and 400 intended as an evaluation set) and 200 of which are private and used as a hidden test set. Each task consists of around a handful of examples, where an example consists of an input grid and an output grid. For each example, the output grid is the result of applying the same task-specific transformation to the input grid. The goal for a test-taker is to infer the transformation from the few examples and successfully apply it on the test examples where the output is missing. Despite this domain being restricted (i.e. only one of 10 distinct colors for each pixel and at most a 30 x 30 resolution are allowed), the tasks that it allows are extremely broad. The ARC tasks cover a wide spectrum of concepts from objectness, arithmetic, geometry, etc. [2] and range from simpler transformations such as mirroring and rotation of grids to more complex ones involving object detection and conditionally applying operations such as object movement, modification or removal.

Most of the ARC tasks are easily solved by adults, but so far no current computer programs have shown much promise: In the 2020 Kaggle competition [3] with almost 1000 participating teams, merely a dozen managed to reach an accuracy score of over 10% of tasks on the leaderboard. Only the winner managed cross the 20% mark. Especially current standard machine learning techniques did not do well. Many of the current top attempts follow a program synthesis approach close to what François Chollet suggested, i.e. creating a Domain-Specific Language (DSL) capable of expressing solution programs for any ARC task and then constructing candidate programs by searching combinations of DSL components. This discrepancy in performance is largely due to the focus of ARC on requiring broad generalization (i.e. ARC exhibiting great task diversity) and few shot learning (i.e. only very few examples per task), thus ARC is not inherently suitable for machine learning techniques.

1.2 Motivation and Overview

The main goal was to

> *build a strong foundation for future work on ARC by constructing a concise domain specific language that enables short programs solving ARC tasks.*

The motivation for wanting to do so was the belief that such a DSL could lay important groundwork for downstream work. In my opinion, ARC can be viewed as a search problem that can be mapped to a significantly easier search problem by working within the right domain such as a good DSL.

1.2.1 Creating a DSL

The domain specific language to be created shall satisfy two main high-level goals:

- It ought to be sufficiently expressive in the sense that for most imaginable ARC tasks there are programs expressible in and only in the DSL.
- It ought to be abstract and generic in the sense that the number of primitives it defines is small and that they are each useful for many ARC tasks.

The reason for the first goal is it being a necessary condition for a language that should have potential at tackling ARC as a whole. The reason for the second goal is to limit overfitting on the training tasks as well as allowing for concise task solution programs, which is again a necessary condition for any program synthesis approach.

The DSL will follow an entirely functional approach that relies on simple types, as opposed to defining custom classes. The components of the DSL (also referred to as primitives or functions) can roughly be categorized into property functions, transformation functions and helper functions.

1.2.2 Constructing Solvers

The reasons for creating task-solving programs are manyfold, the most important ones being: It strongly aids the development of the DSL, can to some degree serve as a test for the DSL, in the sense of a proof-of-concept, allows for an overall better understanding of ARC and is hence indirectly helpful for designing a program synthesis approach, gives statistics which may directly be leveraged for constraining or guiding search and may serve as a foundation for future work, e.g. data augmentation or generating new ARC tasks.

The goal was to create solvers (i.e. task-solving programs) for the ARC training tasks using only DSL functions and only few function calls each. Hence, no other constructs such as for loops or if statements shall be allowed that occur outside the DSL, and the number of DSL function occurrences in each solver should be small, that is, around a dozen or ideally even just a handful.

1.2.3 Improving DSL and Solvers

Since it is infeasible to create solvers for a fixed DSL (which has not yet been used) or refactor a DSL with a fixed set of solvers, the work on the DSL as well as implementing solvers will inevitably be an iterative hand-in-hand process. The goal was to first implement a set of DSL components that are deemed reasonable and generally useful, without considering specific tasks. Next, without modifying this first version of the DSL in the process, solvers for a small subset of the training tasks were constructed. Subsequently, there were alternation between improving the DSL and solvers by using heuristics such as

- **removing DSL components** whenever their usage was too infrequent or even nonexistent, their functionality was deemed too specific, their signatures too complex or their functionality easily expressible as a short combination of other DSL components and
- **adding DSL components** whenever they corresponded to compositions of existing functions very frequently used in the solvers, were required to get rid of using non-DSL constructs, or allowed writing (significantly) shorter solvers.

Chapter 2

Domain-Specific Language

2.1 Workflow

I constructed a first draft of the DSL after I obtained a good grasp of the spirit of ARC tasks, primarily by looking at them and thinking about how one could go about writing code that solves them. Some shallow investigation of existing DSLs was done up front - done in the first place to get a rough idea of how other people went about it and done only in a shallow fashion to avoid already being steered too strongly in a certain possibly suboptimal direction. This first iteration of the DSL was created without looking at individual tasks, with the aim of avoiding introducing DSL primitives that may be overly specific to a single task. Almost any function that was deemed potentially useful for ARC was implemented.

In a next step, the first couple dozen ARC training tasks were tackled: Where somewhat straight-forward, solvers were implemented, while trying to primarily use the DSL primitives at hand and as little native constructs like loops or branches as possible. A solver denotes a task-specific program that can correctly transform the input grid into the output grid for each example of the given task. Quite some tasks were skipped due to the infancy of the DSL and only few were implemented using solely DSL functions. Occasionally, new DSL primitives were added if their usefulness across multiple tasks became obvious (and hence their functionality general) enough. Notes were kept on observations about lacking or redundant functionalities.

In a third step, the DSL was thoroughly revised. Many primitives were either simplified, made more general or removed when too redundant or too rarely used. E.g. for many predicates, there were redundant primitives that denoted the opposite, such as "filter each element in a container by a condition" and "filter each element in a container by the negation of the condition", which is not in accordance of keeping the DSL simple. Thus those counterpart primitives were removed and instead a helper primitive was added that simply negates and hence allows for the same functionality via function composition. Quite some primitives were added since they were deemed generic and useful, i.e. allow tackling previously challenging tasks or lessening or even avoiding the usage of non-DSL constructs in solvers. As progression continued, the need to add DSL primitives became significantly rarer and - besides having a better DSL also due to familiarity with the DSL and the style of ARC tasks - writing solvers purely within the DSL became much easier.

Eventually, I ended up writing solvers purely within my DSL for the entire 400 training

tasks. As with many other undertakings, a minority caused the majority of the work. For many tasks, writing a solver program was very straight-forward and would only take a minute or so, but for some it was a very challenging and long process, mainly not due to lacking a good conception of how to write a program that solves the task in the first place, but due to having to write a program that only uses DSL primitives, thus with limited flexibility. Amongst the ones mentioned in the reasons for constructing such solvers described in the respective section of the approach outline section, this (i.e. solving the entirety of the training tasks) was done to make a stronger case for the adequateness of a DSL and the ability of a somewhat small and generic DSL to be able to write largely concise solver programs.

2.2 Design Principles

Here I intend to give a rough overview of design principles that were followed when constructing the DSL. The main design principles that I tried to follow were:

- to adhere to a functional and type-based design, i.e. there are no custom classes such as e.g. an "object" class that stores a bunch of additional properties of the object, or factors in contextual information about the object's surroundings: A grid is simply a vector of vectors of integers, an object is simply a set of cells (pixels) that are part of the object, where each cell again is represented as a tuple where the first entry corresponds to the color and the second entry corresponds to the location of the cell;

- to write abstract functions, i.e. abstract away details: E.g. the notions of "shape of" or "set of colors occurring in" are valid for both a grid and an object on the grid, thus those primitives should be able to take either of those types as an argument;

- to write generic functions, i.e. try to avoid as much as possible DSL primitives that are used only very rarely for writing solvers. (Note that some exceptions to this were made, in cases where the functions were deemed generic enough and just out of chance or late introduction were used only rarely despite this);

- enforcing simple function signatures, i.e. the number of arguments that the functions take should be small, and they should always only return a single entity.

- Constrain to simple types, i.e. there shall be only a small number of different types, such as "grid" or "object" or "integer" and

- avoiding redundancy, i.e. having DSL primitives that can very concisely be expressed as a combination other DSL primitives. (Note that some exceptions to this were made, primarily in cases where the usage of a specific combination of some DSL primitives was very frequent).

The development of the DSL was somewhat test-driven, in that each primitive has some corresponding unit tests that serve as sanity checks and make development more robust. In fact, having tests proved really useful for refactoring.

2.3 Result

2.3.1 Overview

The DSL [4] can be put into three roughly equally sized categories of primitives:

- **transformations**: functions that transform a grid or an object, e.g. resize, mirror, rotate, recolor, move, delete, etc.,
- **properties**: functions that extract some features of an entity, e.g. leftmost occupied cell, center of mass, shape, size, whether an object is a line, a square, etc. and
- **utils**: functions that implement set operations (e.g. difference, union, intersection, insertion, removal), arithmetic operations (e.g. addition, subtraction, multiplication, floor division) or provide various helper functionality (e.g. filtering, function composition, parameter binding, branching, merging of containers).

Note that this categorization is neither intentional (this is just an observation to give oversight and less so a segmentation by design) nor clear (some few primitives may not fit well into any of those categories and quite some could be considered as belonging to more than one).

2.3.2 Types

The primitives work on largely simple and often custom types. The goal of fixing the allowed types and keeping them constraint was manyfold, e.g. to allow annotating the DSL primitives (very useful, i.e. almost necessary, in the context of searching through combinations of DSL primitives for solvers), restricting the space of possible values that can occur, better structure and overview of the code, gaining a better understanding of useful levels of abstractions, etc. The types are constructed using the Python typing module. The motivation for only using hashable types (e.g. FrozenSet instead of Set or Tuple instead of List) is that this allows having nested containers in an unordered container (e.g. a set of objects), something that enables more flexibility, as well as e.g. much faster lookups in dictionaries during a search. Note that some are irreducible base types such as integer or boolean, and some are simply containers of a simpler type (e.g. objects as a container of items of type object). Hence, there are many natural child-parent correspondences, (i.e. "type of element in container" - "type of container"), such as integer-integerset, object-objects, etc. Having those correspondences was also something that would prove useful for program synthesis, since it e.g. allows inferring the specific return type of a function (annotated with returning a generic container) based

on the type of the provided input value(s) (e.g. think of argmax-ing). The reasoning for opting to avoid the dictionary type was mainly simplicity and the fact that it was not needed, i.e. there are easy workarounds, on top of hashable dictionaries not being easily supported. Similarly, floating point numbers are also not supported, mainly due to simplicity, and also since they would only really be useful in rare occasions (given the discrete nature of ARC) and even in those occations workarounds can be found. The reason for having both tuples and sets is that order sometimes matters (e.g. (x, y) coordinates) and set operations are handy built-in operations, an early design choice. However in hindsight the upside from having fewer and more consistent types resulting from opting for only ordered containers (tuples) would have probably been worth the trade-off with occasionally more concise or faster code. A similar trade-off was made when opting for having related types for Cell (IntegerTuple) Object (Indices) and Objects (IndicesSet): When the color does not matter or the notion of color is nonsensical (e.g. an (x, y) vector representing a direction of movement or shape of a grid or object), one does not need it and can thus simplify the primitives by not having to deal with them. However this introduces some redundancy; if I were to (or once I will) refactor the DSL, odds are great I would get rid of the Indices and IndicesSet as types (however keep the IntegerTuple type).

In total, there are 56 different input signatures (i.e. tuples of the form ("type of first argument", "type of second argument", etc.). The most common input signatures take only a patch, grid, piece or numerical, or two patches, or a container and a callable, etc. In total, there are 160 primitives (see tables 2, 3, 4, 5, 6), of which 79 take a single argument, 67 take two arguments, 13 take three arguments and only one (namely the object detection function) takes four arguments, and no primitive takes more than four arguments. Of the 20 defined types, 18 appear at least once as a return value type; most common are grid, integer, indices and boolean.

2.3.3 Objectness

While not every ARC task relies on working on the level of objects (e.g. some are better tackled at the pixel level, some at the grid level), the notion of objects is central. My DSL has one main primitive for extracting a set of objects from a grid, where the arguments of the objects primitive are as follows:

- grid: The grid from which to extract objects from.
- univalued: Whether cells that are part of the same object are allowed only a single color (or whether they may be of different colors).
- diagonal: Whether a cell needs to be directly adjacent to at least one cell in an object in order to belong to that object (or whether it is allowed to be merely diagonally adjacent).
- background: Whether cells of the background color (defined as the most common color) are part of objects as well (or whether they are ignored).

There are two further primitives that extract objects and consider each pair of cells with the same color as part of the same object (one that considers and one that ignores the background color). This allows for $2^3 + 2$ different notions of objectness. The objects primitive is the 6th-most commonly used primitive, used 248 times and in over 50% of the tasks. Even though they are fairly primitive and limited, i.e. somewhat restricted, they proved sufficient for most cases, and for the few cases where it was not sufficient, there were workarounds (e.g. there are cases where the notion of distinguishing between diagonally and directly adjacent cells is not sufficient, as in one would be better off specifying a maximum allowed distance between two cells of the same object). One could imagine an in quite some regards more elegant generic object detection function, e.g. a primitive that takes a grid alongside a boolean function that takes two cells (i.e. two tuples of color and location) which returns a boolean indicating whether the two cells shall belong to the same object. But of course even such a primitive would not allow detecting arbitrary objects, as there are some even more advanced cases where boundaries between objects are very hard to automatically detect, and not necessarily solely depending on colors or distances between cells.

The code for extracting the objects works as follows: It iterates over all the pixels in the grid. Whenever a pixel is either already occupied by another object or the color of that pixel is the background value (and the background color is not an allowed object color), the pixel is ignored. Otherwise, an object is initialized, at first containing only the pixel at hand. The object is then grown by repeatedly inserting all neighboring pixels which satisfy the criteria (and are not already marked as belonging to the current or a previous object) until there are no more pixels in the neighborhood.

2.3.4 Example

The following is an example of a solver intended to give some idea how simple combinations of simple DSL primitives allow expressing almost arbitrary programs.

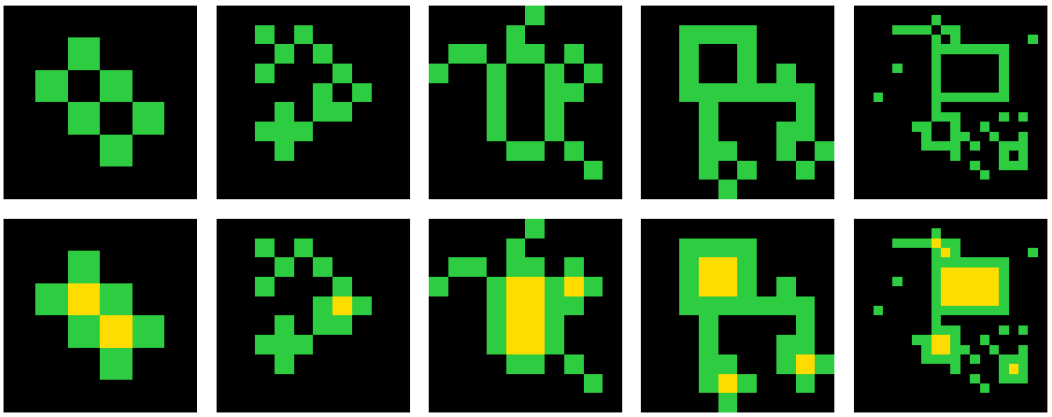

Figure 2.1: ARC task 00d62c1b

```
def solve_00d62c1b(I):
    x1 = objects(I, T, F, F)
    x2 = colorfilter(x1, ZERO)
    x3 = rbind(bordering, I)
    x4 = compose(flip, x3)
    x5 = mfilter(x2, x4)
    O = fill(I, FOUR, x5)
    return O
```

Figure 2.2: Solver program for task 00d62c1b written in the DSL

The function solve_00d62c1b takes an input grid I and returns the correct output grid O. An explanation of what the variables store and how their values were computed:

x_1: the set of objects extracted from the input grid I that are single-colored only, where individual objects may only have cells that are connected directly, and cells may be of the background color (black); the result of calling the objects primitive on I with univalued=True, diagonal=False and without_background=True

x_2: the subset of the objects x_1 which are black; the result of filtering objects by their color, i.e. calling colorfilter with objects=x1 and color=ZERO (black)

x_3: a function with signature $f : \mathit{object} \rightarrow \mathit{bool}$ that returns True iff an object is at the border of the grid; the result of fixing the right argument of the bordering primitive to I by calling the function *rbind* on function=bordering and fixed=I

x_4: a function that returns the inverse of the previous function, i.e. a function that returns True if and only if an object does not border the grid border; the result of composing the flip primitive (which simply negates flips a boolean) and $x3$

x_5: a single object defined as the union of objects x_2 for which function x_4 returns True, i.e. the black objects which do not border the grid border (corresponding to the "holes" in the green objects); the result of calling mfilter (which combines merging and filtering) with container=x2 and condition=x4

x_5: the output grid, created by coloring all pixels of the object x_5 yellow; the result of calling the fill primitive on I with color=FOUR (yellow) and patch=x5

2.3.5 Proof of Concept

As a proof of concept, the DSL was used to solve the 400 training tasks [5]. The rules I set for creating solvers were as follows: There is a solver for each task, where a solver is a function specific to the task that takes the input grid of any example of that task as a single argument and returns the correct corresponding output grid. The only allowed operations are storing the result of a function call in a variable, where all arguments must

either be the input grid, some constants such as integers or common vectors indicating directions, or a variable previously computed within the same solver, and each function that is being called must either be a DSL primitive or a variable previously constructed within the same solver. This also means that each line of code is enforced to be a single function call. This style of programming is rather unusual, but allows for easier analysis (e.g. for strictly formatted code, constructing control flow graphs or analyzing the usage frequencies of primitives is considerably easier).
The goal was to minimize the lengths of the solvers, i.e. the lines of code in the functions (however, not globally, since there is a trade-off with the complexity of the DSL, i.e. one could cheat a minimal program length for each task by simply constructing a corresponding DSL primitive and thereby entirely defeating the purpose). This was achieved by iterating, as described earlier, between making progress on implementing and improving existing solvers as well as modifying the DSL. Some techniques used to write more concise solvers were for example pulling up very frequent within-solver segments of code into the DSL or semi-automatically trying to detect potential for refactoring.
In that regard, the principle of "shorter is better" was followed, and the DSL was designed accordingly, under the knowledge that finding shorter programs is more likely, as well as that shorter programs are less likely to overfit on a task: Imagine a task where each input grid has a single object that is always shifted to the right by a constant k, which coincidentally happens to also be the same as the width of the object for all examples. In that case, both shifting by the constant as well as by the width would give valid programs, but maybe the input grid of the test example has an object of a different width, in which case it would fail. Of course it could also be the opposite, i.e. the task was intended to be "shift by the width of the object", but the former is arguably easier and hence more likely to be the "correct one". This touches on the important topic of having "overdetermined" tasks, i.e. tasks where a program is found that successfully solves all the training examples but fails on (some of) the test example(s). Given the nature of ARC (i.e. rules are described by natural language, which is fuzzy), it is impossible to prove that a program that works on the training examples is the "correct one", because this notion of correctness does not exist. It however would exist if the ARC-creator were to have provided a capable DSL alongside a metric for program complexity as well as the claim that "the least complex program explaining all training examples is by definition the correct one". Luckily, despite this being something I feared may become an issue when searching for programs, it turned out to almost be a non-issue in the sense that in the vast majority of cases where programs that work on the training examples were found, they also worked on the test examples - speaking either for the quality of François' work (or, also, my DSL).
A target I set for myself was to use at most ten primitives (i.e. lines of code) per solver for as many tasks as possible. In the end, this was achieved for 235 out of the 400 tasks (of which 79 were solved in at most 5 function calls). Further, 78 programs are longer than 20 lines and only three are longer than 50 lines.

2.4 Limitations

There are surely some primitives that one could think of that would be fairly generic and useful and that are currently not part of the DSL. For example, it is believed that things like operator-accumulation or recursion (where the recursion depth is not identifiable in advance) that very rarely occur can currently not be expressed neatly in the DSL and instead are implementable only in a cumbersome fashion via e.g. vast branching to make case distinctions. Further, there are quite some operations that would ideally be generalized: currently, the operations such as rotation, trimming or splitting, implemented for grids, are not compatible with patches, even though conceptually they may apply on them just as well and would also be potentially useful. Also, the design principle for operations that depend on integer tuples (e.g. indicating an offset, direction or axis) is not uniform: For some operations such as mirroring, shifting, drawing a line, etc., the axes or directions are indicated by providing an integer tuple as an argument, whereas in other cases that could be covered by similar function signatures, there are multiple primitives, one for each direction or axis, such as for retrieving the corner of an object. This difference was mainly a result of which of the two scenarios was deemed to allow for shorter solver programs. However, one could imagine a more generic design where such functionalities would always be a single primitive, taking an additional argument. This would not only make the DSL more concise and less redundant, but also allow for some cases where the program control flow would be more natural, e.g. a direction vector directly inferrable and being passable to a primitive instead of having to use branching to select the respective direction- or axis-specific primitive. There were many more such trade-offs, where the better design choice was non-trivial.
Further, there is some redundancy in the types: Unordered containers such as sets are not actually needed, they were just an early design choice made because set operations are neatly already existent in the programming language. Concepts like intersection and only allowing unique elements can obviously also easily be supported with ordered containers such as tuples.
Also, there were two design principles used in the solvers: One that largely applies functions on ordered containers for most of the computations and one that largely constructs functions and then applies the constructed functions only once. This could also be unified. Moving away from constructing functions would have the advantage of having simpler program control flows, moving away from vectorized operations would have the advantage of allowing greater direct insight into the subprograms used by the solvers, e.g. one could more easily investigate questions like "what is the distribution of functions constructed within the solvers?"
It should also be noted that the predefined types were not followed strictly, i.e. some sub types are used in the solvers that are not specified in the type set: For example, sometimes a set of functions was used, something that code allows since some primitives take generic containers as arguments, however a "function container" is not explicitly listed among the defined types. Ideally, one would constrain the allowed types more strictly, i.e. enforce each of the arguments to primitives to be of a predefined type.

Bibliography

1. Chollet, F. *The Abstraction and Reasoning Corpus* (2019).
2. Chollet, F. *On the Measure of Intelligence* (2019).
3. Kaggle. *Abstraction and Reasoning Challenge* (2020).
4. Hodel, M. *DSL for ARC* (2023).
5. Hodel, M. *Solver Programs for the ARC Training Tasks* (2023).

Chapter 2

Tables

Name	Definition	Description
Boolean	bool	true or false
Integer	int	whole number
IntegerTuple	Tuple[Integer, Integer]	(x, y) tuple
Numerical	Union[Integer, IntegerTuple]	number or vector
IntegerSet	FrozenSet[Integer]	set of integers
Grid	Tuple[Tuple[Integer]]	2d colored grid
Cell	Tuple[Integer, IntegerTuple]	(color, (x, y)) tuple
Object	Frozenset[Cell]	set of cells
Objects	Frozenset[Object]	set of objects
Indices	Frozenset[IntegerTuple]	set of (x, y) tuples
IndicesSet	Frozenset[Indices]	set of location sets
Patch	Union[Object, Indices]	either object or indices
Element	Union[Object, Grid]	either grid or object
Piece	Union[Grid, Patch]	either grid or patch
Tuple	Tuple	ordered container
FrozenSet	FrozenSet	unordered container
TupleTuple	Tuple[Tuple]	tuple of tuples
ContainerContainer	Container[Container]	container of containers
Callable	Callable	function
Any	Any	anything

Table 1: DSL Types

Primitive Name	Argument Types	Return Type
identity	(Any)	Any
add	(Numerical, Numerical)	Numerical
subtract	(Numerical, Numerical)	Numerical
multiply	(Numerical, Numerical)	Numerical
divide	(Numerical, Numerical)	Numerical
invert	(Numerical)	Numerical
even	(Integer)	Boolean
double	(Numerical)	Numerical
halve	(Numerical)	Numerical
flip	(Boolean)	Boolean
equality	(Any, Any)	Boolean
contained	(Any, Container)	Boolean
combine	(Container, Container)	Container
intersection	(FrozenSet, FrozenSet)	FrozenSet
difference	(Container, Container)	Container
dedupe	(Tuple)	Tuple
order	(Container, Callable)	Tuple
repeat	(Any, Integer)	Tuple
greater	(Integer, Integer)	Boolean
size	(Container)	Integer
merge	(ContainerContainer)	Container
maximum	(IntegerSet)	Integer
minimum	(IntegerSet)	Integer
valmax	(Container, Callable)	Integer
valmin	(Container, Callable)	Integer
argmax	(Container, Callable)	Any
argmin	(Container, Callable)	Any
mostcommon	(Container)	Any
leastcommon	(Container)	Any
initset	(Any)	FrozenSet
both	(Boolean, Boolean)	Boolean
either	(Boolean, Boolean)	Boolean
increment	(Numerical)	Numerical
decrement	(Numerical)	Numerical
crement	(Numerical)	Numerical
sign	(Numerical)	Numerical
positive	(Integer)	Boolean

Table 2: DSL Primitives (Arithmetic, Set Theory)

Primitive Name	Argument Types	Return Type
toivec	(Integer)	IntegerTuple
tojvec	(Integer)	IntegerTuple
sfilter	(Container, Callable)	Container
mfilter	(Container, Callable)	FrozenSet
extract	(Container, Callable)	Any
totuple	(FrozenSet)	Tuple
first	(Container)	Any
last	(Container)	Any
insert	(Any, FrozenSet)	FrozenSet
remove	(Any, Container)	Container
other	(Container, Any)	Any
interval	(Integer, Integer, Integer)	Tuple
astuple	(Integer, Integer)	IntegerTuple
product	(Container, Container)	FrozenSet
pair	(Tuple, Tuple)	TupleTuple
branch	(Boolean, Any, Any)	Any
compose	(Callable, Callable)	Callable
chain	(Callable, Callable, Callable)	Callable
matcher	(Callable, Any)	Callable
rbind	(Callable, Any)	Callable
lbind	(Callable, Any)	Callable
power	(Callable, Integer)	Callable
fork	(Callable, Callable, Callable)	Callable
apply	(Callable, Container)	Container
rapply	(Container, Any)	Container
mapply	(Callable, ContainerContainer)	FrozenSet
papply	(Callable, Tuple, Tuple)	Tuple
mpapply	(Callable, Tuple, Tuple)	Tuple
prapply	(Callable, Container, Container)	FrozenSet

Table 3: DSL Primitives (Utils)

Primitive Name	**Argument Types**	**Return Type**
mostcolor	(Element)	Integer
leastcolor	(Element)	Integer
height	(Piece)	Integer
width	(Piece)	Integer
shape	(Piece)	IntegerTuple
portrait	(Piece)	Boolean
colorcount	(Element, Integer)	Integer
colorfilter	(Objects, Integer)	Objects
sizefilter	(Container, Integer)	FrozenSet
asindices	(Grid)	Indices
ofcolor	(Grid, Integer)	Indices
ulcorner	(Patch)	IntegerTuple
urcorner	(Patch)	IntegerTuple
llcorner	(Patch)	IntegerTuple
lrcorner	(Patch)	IntegerTuple
crop	(Grid, IntegerTuple, IntegerTuple)	Grid
toindices	(Patch)	Indices
recolor	(Integer, Patch)	Object
shift	(Patch, IntegerTuple)	Patch
normalize	(Patch)	Patch
dneighbors	(IntegerTuple)	Indices
ineighbors	(IntegerTuple)	Indices
neighbors	(IntegerTuple)	Indices
objects	(Grid, Boolean, Boolean, Boolean)	Objects
partition	(Grid)	Objects
fgpartition	(Grid)	Objects
uppermost	(Patch)	Integer
lowermost	(Patch)	Integer
leftmost	(Patch)	Integer
rightmost	(Patch)	Integer
square	(Piece)	Boolean
vline	(Patch)	Boolean
hline	(Patch)	Boolean
hmatching	(Patch, Patch)	Boolean
vmatching	(Patch, Patch)	Boolean

Table 4: DSL Primitives (Core, I)

Primitive Name	Argument Types	Return Type
manhattan	(Patch, Patch)	Integer
adjacent	(Patch, Patch)	Boolean
bordering	(Patch, Grid)	Boolean
centerofmass	(Patch)	IntegerTuple
palette	(Element)	IntegerSet
numcolors	(Element)	IntegerSet
color	(Object)	Integer
toobject	(Patch, Grid)	Object
asobject	(Grid)	Object
rot90	(Grid)	Grid
rot180	(Grid)	Grid
rot270	(Grid)	Grid
hmirror	(Piece)	Piece
vmirror	(Piece)	Piece
dmirror	(Piece)	Piece
cmirror	(Piece)	Piece
fill	(Grid, Integer, Patch)	Grid
paint	(Grid, Object)	Grid
underfill	(Grid, Integer, Patch)	Grid
underpaint	(Grid, Object)	Grid
hupscale	(Grid, Integer)	Grid
vupscale	(Grid, Integer)	Grid
upscale	(Element, Integer)	Element
downscale	(Grid, Integer)	Grid
hconcat	(Grid, Grid)	Grid
vconcat	(Grid, Grid)	Grid
subgrid	(Patch, Grid)	Grid
hsplit	(Grid, Integer)	Tuple
vsplit	(Grid, Integer)	Tuple
cellwise	(Grid, Grid, Integer)	Grid
replace	(Grid, Integer, Integer)	Grid
switch	(Grid, Integer, Integer)	Grid
center	(Patch)	IntegerTuple
position	(Patch, Patch)	IntegerTuple
index	(Grid, IntegerTuple)	Integer
canvas	(Integer, IntegerTuple)	Grid

Table 5: DSL Primitives (Core, II)

Primitive Name	**Argument Types**	**Return Type**
corners	(Patch)	Indices
connect	(IntegerTuple, IntegerTuple)	Indices
cover	(Grid, Patch)	Grid
trim	(Grid)	Grid
move	(Grid, Object, IntegerTuple)	Grid
tophalf	(Grid)	Grid
bottomhalf	(Grid)	Grid
lefthalf	(Grid)	Grid
righthalf	(Grid)	Grid
vfrontier	(IntegerTuple)	Indices
hfrontier	(IntegerTuple)	Indices
backdrop	(Patch)	Indices
delta	(Patch)	Indices
gravitate	(Patch, Patch)	IntegerTuple
inbox	(Patch)	Indices
outbox	(Patch)	Indices
box	(Patch)	Indices
shoot	(IntegerTuple, IntegerTuple)	Indices
occurrences	(Grid, Object)	Indices
frontiers	(Grid)	Objects
compress	(Grid)	Grid
hperiod	(Object)	Integer
vperiod	(Object)	Integer

Table 6: DSL Primitives (Core, III)