

A. Acknowledgements

We thank Tanmaya Dabral, Jialin Wu, Radu Soricut, Aditya Avinash, Howard Zhou, Jiao Sun, Deqing Fu, Cyrus Rashtchian, Andrew Tomkins, Tom Duerig, as well as the Agile Modeling team for helpful project discussions and technical support in this project.

B. Examples outputs of our data-synthesis pipeline

We first present step-by-step examples of our data-synthesis pipeline. As discussed in §3, the pipeline contains four steps: program generation, program execution, program filtering, and converting execution traces into chains-of-thoughts. Here we show three filtered examples from three datasets we use. As shown, LLMs like PaLM-2 [3] are able to generate complex programs to solve compositional vision tasks. The synthesized CoTs are used to fine-tune our VLMs.

(a)



Is the bookshelf to the right or to the left of the chair that is to the left of the vase?

PaLM-2 Generated Program

```
1 def execute_command(image):
2     image_patch = ImagePatch(image)
3
4     # Find the chair that is to the left of the vase
5     chair_patches = image_patch.find("chair")
6     vase_patches = image_patch.find("vase")
7
8     chair_patch_on_the_left_of_the_vase = None
9
10    for vase_patch in vase_patches:
11
12        # find all the chairs on the left of the vase.
13        chair_patches_on_the_left = []
14        for chair_patch in chair_patches:
15            if chair_patch.horizontal_center < vase_patch.horizontal_center:
16                chair_patch_on_the_left = chair_patch
17                chair_patches_on_the_left.append(chair_patch_on_the_left)
18
19        # the chair on the left of the vase should be the one that is closest on the vase.
20        if len(chair_patches_on_the_left) > 0:
21            chair_patch_on_the_left_of_the_vase = sorted(chair_patches_on_the_left,
22                                                         key=lambda patch: distance(vase_patch, chair_patch))[0]
23            break
24
25    print(f"the chair at {chair_patch_on_the_left} is to the left of the vase at {vase_patch}.")
26
27    # Find the bookshelf
28    bookshelf_patch = image_patch.find("bookshelf")[0]
29
30    # Check if the bookshelf is to the right or to the left of the chair
31    if bookshelf_patch.horizontal_center > chair_patch_on_the_left.horizontal_center:
32        print(f"the bookshelf at {bookshelf_patch} is to the right of the chair at {chair_patch}.")
33        return "right"
34    elif bookshelf_patch.horizontal_center < chair_patch_on_the_left.horizontal_center:
35        print(f"the bookshelf at {bookshelf_patch} is to the left of the chair at {chair_patch}.")
36        return "left"
```

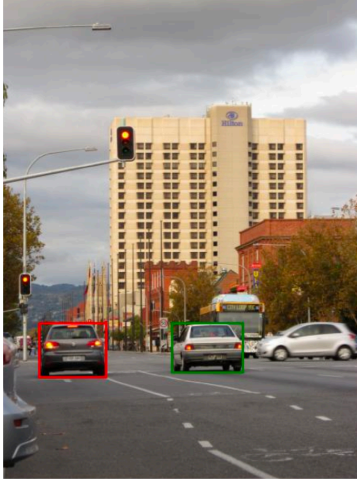
Execution Trace

```
1 Calling find function . Detect chair
2 Detection result : 599 64 655 107 chair and 624 143 836 245 chair and 586 321 782 395 chair and 603 467 771 549 chair
3 Calling find function . Detect vase
4 Detection result : 761 0 889 70 vase and 676 615 756 653 vase
5 the chair at 603 467 771 549 is to the left of the vase at 676 615 756 653.
6 Calling find function . Detect bookshelf
7 Detection result : 505 244 714 359 bookshelf
8 the bookshelf at 505 244 714 359 is to the left of the chair at 603 467 771 549.
9 Program output: left
```

Conversion to CoT

The vase is at **676 615 756 653**. To its left, the chair is at **603 467 771 549**. The bookshelf is at **505 244 714 359**. Therefore, the bookshelf is to the left of the chair.

(b)



How many cars have the brake lights on?

PaLM-2 Generated Program

```
1 def execute_command(image):
2     image_patch = ImagePatch(image)
3
4     # Find all cars. Notice that find() only support single word queries.
5     car_patches = image_patch.find("car")
6
7     # Count the number of cars that have the brake lights on.
8     num_cars_with_brake_lights_on = 0
9     for car_patch in car_patches:
10        if "yes" in car_patch.visual_question_answering("Are the brake lights on?"):
11            print(f"the car at {car_patch} has the brake lights on.")
12            num_cars_with_brake_lights_on += 1
13        else:
14            print(f"the car at {car_patch} does not have the brake lights on.")
15
16    return formatting_answer(str(num_cars_with_brake_lights_on))
```

Execution Trace

```
1 Calling find function. Detect car
2 Detection result: 669 103 779 286 car and 669 468 769 664 car and 668 705 747 991 car
3 Calling visual_question_answering function.
4 Question: Are the brake lights on?
5 Answer: yes
6 the car at 669 103 779 286 has the brake lights on.
7 Calling visual_question_answering function.
8 Question: Are the brake lights on?
9 Answer: yes
10 the car at 669 468 769 664 has the brake lights on.
11 Calling visual_question_answering function.
12 Question: Are the brake lights on?
13 Answer: no
14 the car at 668 705 747 991 does not have the brake lights on.
15 Program output: 2
```

Conversion to COT

The cars at 669 103 779 286 and 669 468 769 664 have the brake lights on. Thus, there are 2 cars with the brake lights on.

(c)



What is usually found in the same room as the word on the sign spelled backwards?

PaLM-2 Generated Program

```
1 def execute_command(image):
2     image_patch = ImagePatch(image)
3
4     # Find the word on the sign
5     word_on_the_sign = image_patch.visual_question_answering("What is the word on the sign?")
6
7     # Spell the word on the sign backwards
8     word_on_the_sign_backwards = word_on_the_sign[::-1]
9     print(f"The word on the sign backward is {word_on_the_sign_backwards}.")
10
11    # Find what is usually found in the same room as the word on the sign spelled backwards
12    object_in_the_same_room = language_question_answering(f"What is usually found in the same room as {word_on_the_sign_backwards}?")
13
14    return formatting_answer(object_in_the_same_room)
```

Execution Trace

```
1 Image Patch 165 0 832 1000 Calling visual_question_answering function.
2 Question: What is the word on the sign?
3 Answer: stop
4 The word on the sign backward is pots.
5 Calling language_question_answering function.
6 Question: What is usually found in the same room as pots?
7 Answer: pans
8 Program output: pans
```

Conversion to COT

The word on the sign is "stop". "Stop" spelled backwards is "pots". Pans are usually found in the same room as pots.

Figure 6. Examples of our data-synthesis pipeline. (a) is from GQA [26]; (b) is from TallyQA [1]; (c) is from A-OKVQA [52].

C. Experimental details

C.1. Model architecture details

For both PaLI models that we considered in our experiments, the architectures are similar: images are encoded into visual tokens individually via a visual encoder. Then, the vision tokens along with the textual inputs are passed through an encoder-decoder UL2 Transformer [58]. The PaLI models were then pre-trained with image-text pairs to perform multimodal tasks. Specifically, PaLI-3 [11] uses a pre-trained 2B SigLIP [77] as visual encoder, and a 3B UL2. The image resolution is 812×812 . PaLI-X [10] uses a pre-trained ViT-22B [15] as visual encoder, and a 32B UL2. The image resolution is 756×756 . Please refer to the PaLI-3 [11] and PaLI-X [10] papers about more architecture details.

C.2. Datasets

The details of the data mixture of academic task-oriented VQA datasets used in VPD training are shown in Table 4. We only use a subset of each dataset’s training set. # labels refers to the total number of examples (containing image, query, and answer) we use. # CoTs refers to the number of examples that we have synthesized CoTs using our programs. In total, there are 89.6K CoTs used during training.

Dataset	Description	# labels	# CoTs
VQAv2 [19]	General	100.0K	
OCR-VQA [48]	OCR	50.0K	
GQA [26]	Compositional	86.0K	38.0K
OK-VQA [45]	Knowledge	9.0K	6.7K
A-OKVQA [52]	Knowledge	17.1K	11.2K
TallyQA [1]	Counting	48.4K	33.7K
Total		310.5K	89.6K

Table 4. Data mixture of academic task-oriented VQA datasets used in VPD training.

Details of each evaluation benchmark we use are in Table 5. For free-form question answering, we run inference with the prompt “Answer with a single word or phrase.”, using greedy decoding without any constraint on the model’s output space. For multiple-choice questions, we run inference with the prompt “Answer with the option letter from the given choices directly.” and generate the option letter.

Dataset	Description	# split	# Metrics
VQAv2 [19]	General VQA. General questions about entities, colors, materials, etc.	test-dev	VQA Score
GQA [26]	Compositional VQA. Built on the scene-graphs in Visual Genome [32]. More compositional questions and spatial relation questions.	test-dev	EM
OK-VQA [45]	Knowledge-based VQA. Questions that need external knowledge to be answered.	val	VQA Score
A-OKVQA [52]	An advanced version of OK-VQA that is more challenging. – Multiple Choice (MC): choose 1 of the 4 options. – Direct Answer (DA): compare with 10 free-form human answers	val, test val, test	EM VQA Score
TallyQA [1]	Counting questions. – Simple: synthesized simple counting questions – Complex: human-written complex counting questions	test-simple test-complex	EM EM
TextVQA [55]	VQA on images that contain text	val	VQA Score
POPE [36]	Benchmark on VLM hallucination. Binary questions of whether an object exists in the image.	dev	EM
MMBench [40]	Comprehensive benchmark on VLMs with multiple-choice questions. Covering 20 ability dimensions across 3 levels (e.g., coarse perception, fine-grained perception, attribute reasoning, relation reasoning, logic reasoning, etc.)	dev	EM

Table 5. Summary of evaluation benchmarks.

C.3. Training details

We use LoRA [22] to fine-tune both PaLI-3 [11] and PaLI-X [10]. For generalist training, we add LoRA weights on each linear layer in the attention blocks and multilayer perceptron (MLP) blocks for both the encoder and decoder in the UL2 transformer. For both models, we use $rank = 8$. We use a cosine learning rate schedule, with warm-up ratio 1% and peak learning rate $1e - 4$. For all models and all settings, we use a batch size of 128 and fine-tune the pre-trained model for 8,000 steps. In terms of training time, we train PaLI-X-VPD with 128 TPU-v3 [27] and it takes about 2 days to finish training. For PaLI-3-VPD, we use 32 TPU-v4 and training takes about 20 hours. We still observe a steady loss drop when we terminate training, which indicates that more computation may lead to even better performance. For per-task fine-tuning, to avoid overfitting, we reduce the number of training parameters. For both models, we only add LoRA weights to encoder layers. We use LoRA $rank = 4$ for PaLI-X-VPD and $rank = 8$ for PaLI-3-VPD. The peak learning rate is $1e - 4$ and we use a cosine learning schedule, with warm-up ratio 1%. For all per-task fine-tuning experiments, we use a batch size of 64. We train for 1 epoch on GQA, and 3 epochs on all other datasets. We use the AdamW [30] optimizer with $\beta_1 = 0.9$ and $\beta_2 = 0.98$, and bfloat16 for all experiments.

C.4. Inference costs

We sampled 300 questions and measured the computation cost. Using 128 TPU v5, the code generation on average takes 4.7s, and program execution takes 4.2s. With the same resource, PaLI-X-VPD takes 0.8s per question. **The cost gap (0.8s vs 8.9s) is also large** with immense consequences for practical applications.

D. Human evaluation

We asked our human annotators to first evaluate each model’s answer, using the criteria described in §4.3. After rating each model answer separately, we also asked them to choose a preferred answer between the two. However, we observed that there are cases where one or both models have similar answers, or both answers are incorrect, in which case it would be difficult for the annotators to choose a favorite, so we also provided the annotators with the options “Both” or “Neither”, giving them the following instruction: *“Please try to choose “Answer 1 is better” or “Answer 2 is better” whenever possible. We also give you the option to choose “Both are equally good.” or “Both are too bad to make a choice.” for the cases when it is hard to make a choice either because both answers are correct and similar, or because both answers are wrong so it makes no sense to choose a favorite.”*

We show some examples from our human evaluation in Table 6. The table contains the images and corresponding text queries (column 2), the answers provided by the two models we compared—PaLI-X Instruct (column 3) and PaLI-X-VPD (column 4)—along with the corresponding annotations given by the human annotators. The human annotations are aggregated across 3 raters per sample. Finally, column 5 shows which of the two answers was preferred by the human raters. When a model’s answer includes a bounding box, we annotate it on the image for convenience. Examples are as follow:

- Example #1 shows a common situation where PaLI-X-VPD succeeds where PaLI-X Instruct fails. By being trained with programs that include calls to an object detection tool, PaLI-X-VPD has learned to produce answers that localize the object in question in the image, which prods the model to correctly perform tasks such as counting.
- Example #2 shows a type of question where neither model produces an explanation, where one is arguably not necessary. However, in spite the lack of explanation, PaLI-X-VPD’s answer is more accurate.
- Example #3 shows an example where from Tally QA where PaLI-X-VPD is able to correctly localize and count the number of tennis balls in the image, as asked by the question, when PaLI-X Instruct counts incorrectly and also does not provide a rationale.
- Example #4 shows a situation where PaLI-X-VPD produces an answer that is deemed both factual and consistent by the human raters, and yet the answer is incorrect. Indeed we can notice that everything that PaLI-X-VPD is accurate with respect to the context image, and yet it does not address the question as expected by the human raters.
- Example #5 shows a case where both models are incorrect, but the annotators prefer PaLI-X-VPD’s answer, despite the fact that both models essentially give the same incorrect final count of 0 yellow busses. However, PaLI-X-VPD provides a consistent rationale, from which we can tell the reason for its mistake: it correctly identifies 2 out of the 3 buses in the image, missing the partially occluded yellow bus in the back. It seems the annotators prefer this kind of mistake over one without an explanation.



Index	Image & Query	PaLI-Instruct Answer & Human Eval	PaLI-VPD Answer & Human Eval	Annotator Preference
1.	 <p>On which side is the knife?</p>	<p>The knife is on the left side of the table.</p> <ul style="list-style-type: none"> • Correct: ✗ • Has Explanation: ✗ • Explanation Factuality: N/A • Explanation Consistency: N/A 	<p>The knife is at 55 693 412 884, which is on the right side of the image.</p> <ul style="list-style-type: none"> • Correct: ✓ • Has Explanation: ✓ • Explanation Factuality: ✓ • Explanation Consistency: ✓ 	PaLI-VPD
2.	 <p>What are this fruits called?</p>	<p>The fruits in the image are apples and broccoli.</p> <ul style="list-style-type: none"> • Correct: ✗ • Has Explanation: ✗ • Explanation Factuality: N/A • Explanation Consistency: N/A 	<p>The fruits are called apples.</p> <ul style="list-style-type: none"> • Correct: ✓ • Has Explanation: ✗ • Explanation Factuality: N/A • Explanation Consistency: N/A 	PaLI-VPD
3.	 <p>How many tennis balls are in the photo?</p>	<p>There are two tennis balls in the photo.</p> <ul style="list-style-type: none"> • Correct: ✗ • Has Explanation: ✗ • Explanation Factuality: N/A • Explanation Consistency: N/A 	<p>There is a tennis ball at 826 665 869 721. Thus, there is 1 tennis ball.</p> <ul style="list-style-type: none"> • Correct: ✓ • Has Explanation: ✓ • Explanation Factuality: ✓ • Explanation Consistency: ✓ 	PaLI-VPD
4.	 <p>Why is one player wearing gloves?</p>	<p>The player is wearing gloves because he is a batter. Batters wear gloves to protect their hands from the ball.</p> <ul style="list-style-type: none"> • Correct: ✓ • Has Explanation: ✗ • Explanation Factuality: ✓ • Explanation Consistency: ✓ 	<p>The player is wearing gloves to catch the ball.</p> <ul style="list-style-type: none"> • Correct: ✗ • Has Explanation: ✓ • Explanation Factuality: ✓ • Explanation Consistency: ✓ 	PaLI-Instruct
5.	 <p>How many yellow buses are in the picture?</p>	<p>There are no yellow buses in the picture.</p> <ul style="list-style-type: none"> • Correct: ✗ • Has Explanation: ✗ • Explanation Factuality: N/A • Explanation Consistency: N/A 	<p>There are 2 buses in the picture. Neither of them is yellow. Thus, there are 0 yellow buses in the picture.</p> <ul style="list-style-type: none"> • Correct: ✗ • Has Explanation: ✓ • Explanation Factuality: ✗ • Explanation Consistency: ✓ 	PaLI-VPD

Table 6. Human Annotation Examples. We show the answers provided by the two models, PaLI-X Instruct and PaLI-X-VPD, along with the aggregated evaluation scores by human raters along the criteria introduced in §4.3.

E. Comparison with visual programs

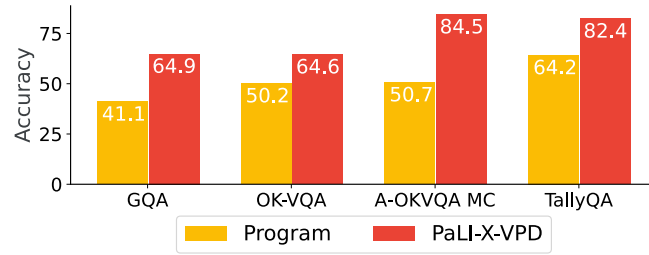


Figure 7. Accuracy of visual programs and PaLI-X-VPD on validation sets.

In Figure 7 also do a side-by-side comparison of the accuracy of visual programs and that of PaLI-X VPD on GQA, OK-VQA, A-OKVQA (multiple choice), and TallyQA (simple and complex combined). We report results on the validation sets, so PaLI-X VPD was not distilled with these exact visual programs, but with visual programs generated in a similar manner on the training set. The results indicate that PaLI-X VPD has much higher accuracy than visual programs on all tasks. This raises an interesting question: why is the student model more accurate than its teacher? One explanation is that our pipeline allows us to leverage labeled data to improve the quality of the visual programs. When ground truth labels are available, we can choose a correct program among 5 candidates, rather than only relying on a single candidate. As supported by the results in Figure 5, this greatly improves the accuracy of our visual programs as teachers, thus making them more helpful for distilling our VLMs.

F. Qualitative Examples on Content Moderation

We present qualitative examples of our methods on the Hateful Memes [29] datasets in Figure 8. We include three unsupervised/zero-shot methods and our state-of-the-art supervised model in this comparison.

Programs is much more accurate than zero-shot PaLI-X-VPD. As exemplified in (a), (c), and (d), despite that our PaLI-X-VPD outperforms all prior zero-shot methods on Hateful Memes, it is still much less accurate than our programs, and is relatively insensitive to hateful content detection.

VPD teaches the generalist VLM to reason like programs on this task, even when no labels are available. As shown in (a) and (b), our PaLI-X-VPD (specialist with zero-shot CoT) is able to reason like the program, and is much more accurate on hateful content detection compared with the generalist model.

Supervised learning further improves the performance of our VPD models. The quantitative results in Table 3 show that the accuracy and AUC-ROC of PaLI-X-VPD (supervised specialist) is much higher than that of the visual programs after training with supervised labels. As shown in (c), PaLI-X-VPD (supervised specialist) is able to capture the nuances expressed by the meme.

Failure case of VLMs. We still observe some failure cases such as the one in example (d), where even our supervised model fails while the program succeeds on hateful content detection.



Figure 8. Example outputs of different methods on Hateful Memes [29] dev set. The unsupervised methods include zero-shot PaLI-X-VPD (generalist), our generated program, PaLI-X-VPD (specialist with zero-shot CoTs). We also include our supervised method, i.e., PaLI-X-VPD (specialist). We also mark whether their outputs matches the gold answer.

G. Prompts

In this section we present the prompts used in our data synthesis pipeline. Refer to §B for step-by-step examples of the programs, execution traces, and the converted CoTs.

G.1. Prompt for code generation

For each image and query, we put the query and a model-generated image caption in the prompt. An LLM takes this prompt and generate the program to answer the query. We modify the original ViperGPT [16] prompt to adapt to the vision tools we use in this paper.

```
1 class ImagePatch:
2     # A Python class containing a crop of an image centered around a particular object, as well as relevant information.
3     # Attributes
4     # -----
5     # cropped_image : array_like
6     #     An array-like of the cropped image taken from the original image.
7     # left, lower, right, upper : int
8     #     An int describing the position of the (left/lower/right/upper) border of the crop's bounding box in the original image.
9     # Methods
10    # -----
11    # find(object_name: str) -> List[ImagePatch]
12    #     Returns a list of new ImagePatch objects containing crops of the image centered around any objects found in the
13    #     image matching the object_name.
14    # visual_question_answering(question: str=None) -> str
15    #     Returns the answer to a basic question asked about the image. If no question is provided, returns the answer to "What is this?".
16    # image_caption() -> str
17    #     Returns a short description of the image crop.
18    # expand_patch_with_surrounding() -> ImagePatch
19    #     Returns a new ImagePatch object that contains the current ImagePatch and its surroundings.
20    # overlaps(patch: ImagePatch) -> Bool
21    #     Returns True if the current ImagePatch overlaps with another patch and False otherwise
22    # compute_depth() -> float
23    #     Returns the median depth of the image patch. The bigger the depth, the further the patch is from the camera.
24
25    def __init__(self, image, left: int = None, lower: int = None, right: int = None, upper: int = None):
26        # Initializes an ImagePatch object by cropping the image at the given coordinates and stores the coordinates as
27        # attributes. If no coordinates are provided, the image is left unmodified, and the coordinates are set to the
28        # dimensions of the image.
29        # Parameters
30        # -----
31        # image: PIL.Image
32        #     An array-like of the original image.
33        # left, lower, right, upper : int
34        #     An int describing the position of the (left/lower/right/upper) border of the crop's bounding box in the original image.
35        #     The coordinates (y1,x1,y2,x2) are with respect to the upper left corner the original image.
36        #     To be closer with human perception, left, lower, right, upper are with respect to the lower left corner of the squared image.
37        #     Use left, lower, right, upper for downstream tasks.
38
39        self.original_image = image
40        size_x, size_y = image.size
41
42        if left is None and right is None and upper is None and lower is None:
43            self.x1 = 0
44            self.y1 = 0
45            self.x2 = 999
46            self.y2 = 999
47        else:
48            self.x1 = left
49            self.y1 = 999 - upper
50            self.x2 = right
51            self.y2 = 999 - lower
52
53        self.cropped_image = image.crop((int(self.x1/1000*self.sz), int(self.y1/1000*self.sz),
54                                         int(self.x2/1000*self.sz), int(self.y2/1000*self.sz)))
55
56        self.width = self.x2 - self.x1
57        self.height = self.y2 - self.y1
58
59        # all coordinates use the upper left corner as the origin (0,0).
60        # However, human perception uses the lower left corner as the origin.
61        # So, need to revert upper/lower for language model
62        self.left = self.x1
63        self.right = self.x2
64        self.upper = 999 - self.y1
65        self.lower = 999 - self.y2
66
67        self.horizontal_center = (self.left + self.right) / 2
68        self.vertical_center = (self.lower + self.upper) / 2
69
70        self.patch_description_string = f"[self.y1] [self.x1] [self.y2] [self.x2]"
71
72    def __str__(self):
73        return self.patch_description_string
74
75    def compute_depth(self):
76        # compute the depth map on the full image. Returns a np.array with size 192*192
77        # Parameters
78        # -----
79        # Returns
80        # -----
81        # float
82        #     the median depth of the image crop
83
84        # Examples
85        # -----
86        # >>> return the image patch of the bar furthest away
87        # >>> def execute_command(image) -> ImagePatch:
88        # >>>     image_patch = ImagePatch(image)
89        # >>>     bar_patches = image_patch.find("bar")
```



```

90 # >>> bar_patches.sort(key=lambda bar: bar.compute_depth())
91 # >>> return bar_patches[-1]
92
93 return depth(self.cropped_image)
94
95 def find(self, object_name: str):
96 # Returns a list of ImagePatch objects matching object_name contained in the crop if any are found.
97 # The object_name should be as simple as example, including only nouns
98 # Otherwise, returns an empty list.
99 # Note that the returned patches are not ordered
100 # Parameters
101 # -----
102 # object_name : str
103 #     the name of the object to be found
104
105 # Returns
106 # -----
107 # List[ImagePatch]
108 #     a list of ImagePatch objects matching object_name contained in the crop
109
110 # Examples
111 # -----
112 # >>> # find all the kids in the images
113 # >>> def execute_command(image) -> List[ImagePatch]:
114 # >>>     image_patch = ImagePatch(image)
115 # >>>     kid_patches = image_patch.find("kid")
116 # >>>     return kid_patches
117
118 print(f"Calling find function. Detect {object_name}.")
119 det_patches = detect(self.cropped_image, object_name)
120 print(f"Detection result: {' and '.join([str(d) + ',' + object_name for d in det_patches])}")
121
122 return det_patches
123
124
125 def expand_patch_with_surrounding(self):
126 # Expand the image patch to include the surroundings. Now done by keeping the center of the patch
127 # and returns a patch with double width and height
128
129 # Examples
130 # -----
131 # >>> # How many kids are not sitting under an umbrella?
132 # >>> def execute_command(image):
133 # >>>     image_patch = ImagePatch(image)
134 # >>>     kid_patches = image_patch.find("kid")
135
136 # >>> # Find the kids that are under the umbrella.
137 # >>>     kids_not_under_umbrella = []
138
139 # >>> for kid_patch in kid_patches:
140 # >>>     kid_with_surrounding = kid_patch.expand_patch_with_surrounding()
141 # >>>     if "yes" in kid_with_surrounding.visual_question_answering("Is the kid under the umbrella?"):
142 # >>>         print(f"the kid at {kid_patch} is sitting under an umbrella.")
143 # >>>     else:
144 # >>>         print(f"the kid at {kid_patch} is not sitting under an umbrella.")
145 # >>>         kids_not_under_umbrella.append(kid_patch)
146
147 # >>> # Count the number of kids under the umbrella.
148 # >>>     num_kids_not_under_umbrella = len(kids_not_under_umbrella)
149
150 # >>>     return formatting_answer(str(num_kids_not_under_umbrella))
151
152 new_left = max(self.left - self.width / 2, 0)
153 new_right = min(self.right + self.width / 2, 999)
154 new_lower = max(self.lower - self.height / 2, 0)
155 new_upper = min(self.upper + self.height / 2, 999)
156
157 return ImagePatch(self.original_image, new_left, new_lower, new_right, new_upper)
158
159
160 def visual_question_answering(self, question: str = None) -> str:
161 # Returns the answer to a basic question asked about the image.
162 # The questions are about basic perception, and are not meant to be used for complex reasoning
163 # or external knowledge.
164
165 # Parameters
166 # -----
167 # question : str
168 #     A string describing the question to be asked.
169
170 # Examples
171 # -----
172
173 # >>> # What is the name of the player in this picture?
174 # >>> def execute_command(image) -> str:
175 # >>>     image_patch = ImagePatch(image)
176 # >>>     return formatting_answer(image_patch.visual_question_answering("What is the name of the player?"))
177
178 # >>> # What color is the foo?
179 # >>> def execute_command(image) -> str:
180 # >>>     image_patch = ImagePatch(image)
181 # >>>     return formatting_answer(image_patch.visual_question_answering("What color is the foo?"))
182
183 # >>> # What country serves this kind of food the most?
184 # >>> def execute_command(image) -> str:
185 # >>>     image_patch = ImagePatch(image)
186 # >>>     food_name = image_patch.visual_question_answering("What kind of food is served?")
187 # >>>     country = language_question_answering(f"What country serves {food_name} most?", long_answer=False)
188 # >>>     return formatting_answer(country)
189
190 # >>> # Is the second bar from the left quaxy?
191 # >>> def execute_command(image) -> str:
192 # >>>     image_patch = ImagePatch(image)
193 # >>>     bar_patches = image_patch.find("bar")
194 # >>>     bar_patches.sort(key=lambda x: x.horizontal_center)
195 # >>>     bar_patch = bar_patches[1]
196 # >>>     return formatting_answer(bar_patch.visual_question_answering("Is the bar quaxy?"))

```

```

197 answer = vqa(self.cropped_image, question)
198
199
200 print(f"Calling visual_question_answering function.")
201 print(f"Question: {question}")
202 print(f"Answer: {answer}")
203
204 return answer
205
206 def image_caption(self) -> str:
207     # Returns a short description of the image.
208     return image_caption(self.cropped_image)
209
210 def overlaps(self, patch) -> bool:
211     # check if another image patch overlaps with this image patch
212     # if patch overlaps with current patch, return True. Otherwise return False
213
214     if patch.right < self.left or self.right < patch.left:
215         return False
216     if patch.lower > self.upper or self.lower > patch.upper:
217         return False
218     return True
219
220
221 def language_question_answering(question: str, long_answer: bool = False) -> str:
222     # Answers a text question using a language model like PaLM and GPT-3. The input question is always a formatted string with a variable in it.
223     # Default is short-form answers, can be made long-form responses with the long_answer flag.
224
225     # Parameters
226     # -----
227     # question: str
228     # the text question to ask. Language model cannot understand the image. Must not contain any reference to 'the image' or 'the photo', etc.
229     # long_answer: bool
230     # whether to return a short answer or a long answer. Short answers are one or at most two words, very concise.
231     # Long answers are longer, and may be paragraphs and explanations. Default is False.
232
233     # Examples
234     # -----
235     # >>> # What is the city this building is in?
236     # >>> def execute_command(image) -> str:
237     # >>>     image_patch = ImagePatch(image)
238     # >>>     building_name = building_patch.visual_question_answering("What is the name of the building?")
239     # >>>     return formatting_answer(language_question_answering(f"What city is {building_name} in?", long_answer=False))
240
241     # >>> # Who invented this object?
242     # >>> def execute_command(image) -> str:
243     # >>>     image_patch = ImagePatch(image)
244     # >>>     object_name = object_patch.visual_question_answering("What is this object?")
245     # >>>     return formatting_answer(language_question_answering(f"Who invented {object_name}?", long_answer=False))
246
247     # >>> # Explain the history behind this object.
248     # >>> def execute_command(image) -> str:
249     # >>>     image_patch = ImagePatch(image)
250     # >>>     object_name = object_patch.visual_question_answering("What is the object?")
251     # >>>     return formatting_answer(language_question_answering(f"What is the history behind {object_name}?", long_answer=True))
252     print(f"Calling language_question_answering")
253     print(f"Question: {question}")
254
255     answer = language_model_qa(question, long_answer).lower().strip()
256     print(f"Answer: {answer}")
257     return answer
258
259
260 def distance(patch_a: Union[ImagePatch, float], patch_b: Union[ImagePatch, float]) -> float:
261     # Returns the distance between the edges of two ImagePatches, or between two floats.
262     # If the patches overlap, it returns a negative distance corresponding to the negative intersection over union.
263
264     # Parameters
265     # -----
266     # patch_a : ImagePatch
267     # patch_b : ImagePatch
268
269     # Examples
270     # -----
271     # Return the qux that is closest to the foo
272     # >>> def execute_command(image):
273     # >>>     image_patch = ImagePatch(image)
274     # >>>     qux_patches = image_patch.find('qux')
275     # >>>     foo_patches = image_patch.find('foo')
276     # >>>     foo_patch = foo_patches[0]
277     # >>>     qux_patches.sort(key=lambda x: distance(x, foo_patch))
278     # >>>     return qux_patches[0]
279
280     return dist(patch_a, patch_b)
281
282
283 def formatting_answer(answer) -> str:
284     # Formatting the answer into a string that follows the task's requirement
285     # For example, it changes bool value to "yes" or "no", and clean up long answer into short ones.
286     # This function should be used at the end of each program
287
288     final_answer = ""
289     if isinstance(answer, str):
290         final_answer = answer.strip()
291
292     elif isinstance(answer, bool):
293         final_answer = "yes" if answer else "no"
294
295     elif isinstance(answer, list):
296         final_answer = ", ".join([str(x) for x in answer])
297
298     elif isinstance(answer, ImagePatch):
299         final_answer = answer.image_caption()
300
301     else:
302         final_answer = str(answer)
303

```

```

304     print(f"Program output: {final_answer}")
305     return final_answer
306
307
308 Given an image and a query, write the function execute_command using Python and the ImagePatch class (above), and the other functions above that could be executed to provide an answer to the query.
309 For reference, a model generated image description is also provided, so that the function can be customized for the given image. The image description is model-generated and may not be reliable, so do not trust it.
310
311 Consider the following guidelines:
312 — Use base Python (comparison, sorting) for basic logical operations, left / right / up/down, math, etc.
313 — Use the language_question_answering function to access external information and answer informational questions NOT concerning the image.
314 — The program should print out the intermediate traces as it runs. So add print function in the program if needed.
315
316 For usual cases, follow the guidelines below:
317 — For simple visual queries, directly call visual_question_answering to get the answer.
318 — For queries that need world knowledge, commonsense knowledge, and language reasoning, use visual_question_answering, language_question_answering, and sometimes image_caption to get the answer.
319 — For queries that require counting and spatical relations, in addition to the above functions, use find function to help getting the answer.
320 — For queries involve "behind" and "front", consider using compute_depth function.
321
322
323 Some examples:
324 Image description: a woman is walking several dogs
325 Query: how many dogs are to left of the person?
326 Function:
327 def execute_command(image):
328     image_patch = ImagePatch(image)
329     person_patch = image_patch.find("person")[0]
330     dog_patches = image_patch.find("dog")
331
332     # Count the number of dogs whose leftmost x-coordinate is less than the person.
333     num_dogs_left = 0
334     for dog_patch in dog_patches:
335         if dog_patch.left < person_patch.horizontal_center:
336             print(f"dog at {dog_patch} is on the left of human.")
337             num_dogs_left += 1
338
339     return formatting_answer(num_dogs_left)
340
341 # [other in-context examples]
342
343 Image description: INSERT_IMAGE_CAPTION_HERE
344 Query: INSERT_QUERY_HERE
345 Function:

```

G.2. Prompt for result verification

After running each program, we get an output. As discussed in §3.1, we adopt the method of [28] and use an LLM to determine if the program output matches human answers. The LLM takes the program output and reference answers as input. The prompt is as follows:

```

1 Given a visual question, several human annotator answers, and a candidate answer, determine if the candidate is correct.
2 The candidate is considered correct if it is allowed to have formatting differences compared with the human answers.
3 If the candidate is correct, return the gold answer it matches. Otherwise, return None.
4
5 Question: INSERT_QUESTION_HERE
6 Answers: INSERT_ANSWERS_HERE
7 Candidate: INSERT_CANDIDATE_HERE
8 Is the candidate correct?

```

G.3. Prompt for CoT conversion

Finally, once a program is filtered, we convert its execution trace into chain-of-thought using an LLM. The LLM takes the query, program, execution trace, program output as input, and summarizes the execution trace into a chain-of-thought rationale. The prompt we use as as follows:

```

1 Given an image and a question, I wrote the function execute_command using Python and the ImagePatch class (above), and the other functions above that could be executed to provide an answer to the query.
2 As shown in the code, the code will print execution traces.
3 I need you to rewrite the execution trace into a natural language rationale that leads to the answer.
4
5 Consider the following guidelines:
6 — Use the bounding box information in the rationale.
7 — Referencing the execution trace, write a reasoning chain that leads to the most common human answer. Notice that the output should be the same as the human answer, not necessarily the program output.
8
9
10 Some examples:
11 Question: How many wheels does the plane have?
12 Program:
13 def execute_command(image):
14     image_patch = ImagePatch(image)
15
16     # Find the plane in the image
17     plane_patch = image_patch.find("plane")[0]
18
19     # Count the number of wheels on the plane
20     num_wheels = 0
21     for wheel in plane_patch.find("wheel"):
22         num_wheels += 1
23
24     return formatting_answer(str(num_wheels))
25 Execution trace:
26 Calling find function. Detect plane
27 Detected plane at 153 25 647 972
28 Calling find function. Detect wheel
29 Detected wheel at 603 471 649 515
30 Detected wheel at 621 85 646 113
31 Detected wheel at 615 383 645 428
32 Program output: 3

```

```
33 Rationale: The plane at 153 25 647 972 has wheels at 603 471 649 515, 621 85 646 113, and 615 383 645 428.Thus, it has 3 wheels.
34
35 [Other demonstration examples]
36
37 Question: INSERT_QUESTION_HERE
38 INSERT_PROGRAM_HERE
39 Execution trace :
40 INSERT_EXECUTION_TRACE_HERE
41 Rationale:
```

H. Limitations and future directions

Among directions for improvement, we have noticed that the stronger the programs are, the bigger the gain that VPD brings. We find that there are some problems that our visual program framework (ViperGPT [16]) cannot solve. Example failures are in Appendix B. We list the limitations below, along with future work that may address these challenges.

Adding fine-grained and dense labeling tools. We find that programs tend to fail when there are multiple overlapping bounding boxes. For example, when there is one person standing behind another, their bounding boxes overlap. This makes programs fail to accurately determine what the person behind is wearing, since the bounding box will be dominated by the person in the front. Adding dense-labeling tools like Segment Anything [31] can address this challenge. For example, recently LISA [33] have proposed combining segmentation with LLMs. Future work can make dense labeling tools available in VPD in a similar way, which will further boost VLM performance.

Agents, rather than static programs. There exist complex visual-language tasks that cannot be easily solved with one program. However, recent work [67, 73] have explored the idea of LLM reinforcement learning agents, where LLMs interact with an environment and do planning interactively. We may be able to leverage this idea in our scenario, and have an LLM update the generated code given the new information gathered from vision tools.

Better ways to filter multimodal chain-of-thought data. As shown in §5, VPD can be effective even when there are no labels to filter the programs. Future work may discover more efficient ways to filter multimodal CoT data, and apply VPD on large-scale image datasets to produce large amount of complex instruction-tuning data for VLMs.