IML – An Image Manipulation Language

Conference Paper · September 2019

DOI: 10.1145/3355378.3355382

CITATIONS
0

READS
188

2 authors, including:

Carlos Vieira
Universidade Federal do Rio Grande do Norte
4 PUBLICATIONS **29** CITATIONS

SEE PROFILE

IML - An Image Manipulation Language

Carlos Vieira
vieiramecarlos@gmail.com
Federal University of Rio Grande do Norte
Natal, Brazil

Sérgio Queiroz de Medeiros
sergiomedeiros@ect.ufrn.br
Federal University of Rio Grande do Norte
Natal, Brazil

ABSTRACT

Several image manipulation tools support the use of at least one general scripting language (e.g., Python, JavaScript), for task automation. But, users of such tools usually do not have much experience or skill with these (or often any) programming languages, which represents a barrier for the use of such languages when automating a task. With this in mind, we present IML, a work-in-progress, domain-specific language designed for easy and clear image manipulation. Besides describing the basic constructs and operations of this language, we compared a simple IML program with equivalent implementations in the languages currently supported by the popular image manipulation tool GIMP. This illustrates how IML might make the image editing automation process simpler, easier to learn, and more straightforward.

CCS CONCEPTS

• **Computing methodologies** → **Image manipulation**; • **Software and its engineering** → **Domain specific languages**.

KEYWORDS

domain specific language, image processing, programming languages

ACM Reference Format:
Carlos Vieira and Sérgio Queiroz de Medeiros. 2019. IML - An Image Manipulation Language. In *XXIII Brazilian Symposium on Programming Languages 2019*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3355378.3355382

1 INTRODUCTION

Several image editing tools support some form of scripting in order to automate repetitive or complex tasks: Adobe Photoshop allows scripting with JavaScript (also AppleScript and VBScript); Corel PaintShop Pro has a scripting engine based on Python; and GIMP can be scripted with Script-Fu (which is based on the Scheme language) or Python.

However, the scripting facilities offered by these tools usually rely on considerable knowledge on some general programming language, even when the users of such tools usually do not have much previous programming experience. Additionally, these scripting options are heavily coupled to the functionalities of the editing software, and knowledge of one does not usually transfer in any considerable way to another.

Given this context, we present IML (Image Manipulation Language), a domain-specific language for image editing designed to address these two points, being simpler than a general purpose language, and thus presumably easier to learn for non-programmers; and independent of any specific editing tool.

An additional benefit of the use of a language tailored for this purpose is to achieve generally clearer and more legible code. A program written in IML for a given task should be more easily comprehensible for a domain expert than an equivalent one written in a general programming language. We could also consider the creation of an image manipulation library for a more general programming language, but that would be constrained to the syntax of the language it's made for, so we could not achieve the same levels of readability and simplicity as we might with a DSL.

There has been some effort towards enabling end-user programming for image editing, particularly with natural language-based approaches [1, 3], but with limitations on accuracy and supported editing operations. Another kind of related work is that of domain-specific languages for image processing [2, 4, 5], but these do not target end-users for image editing, but programmers who work with lower-level image processing tasks.

Now, in Section 2, we give an overview of IML and of its most prevalent features. Following that, in Section 3, we present a sample automation task, and compare its implementation in IML to implementations in other scripting languages currently supported by popular image manipulation tools. Finally, we conclude in Section 4, and discuss future work.

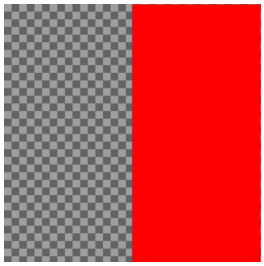
2 LANGUAGE FEATURES

The design of IML revolves around a basic set of operations for manipulation of 2D images. To support this, IML has six basic data types: (i) image; usual numeric types (ii) integer and (iii) float; (iv) path, an extension of the usual string type, used for representing a path to a directory or file; (v) dimensions, an image's dimensions in pixels as a (width, height) tuple; and (vi) section, a rectangular section of an image, as a tuple with leftmost, uppermost, rightmost, and lowermost limits, in pixels. Note that when using sections, an image's origin is taken to be its upper left corner.

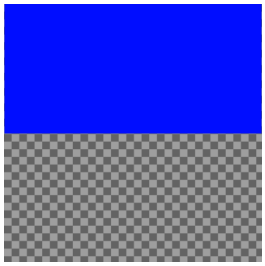
In Listing 1 we can see in line 1 the attribution of an image to a variable called `red`, loaded from a file named 'red_column.png' (a path), as illustrated in Figure 1. In line 8 we define a section to be removed from an image proportionate to its size (see Section 2.2 for more details on the operation on line 9), and print it, as shown in Figure 3b. What follows is a discussion with more details about operations on these types: first those operations exclusive to images (Section 2.1), then the more general ones (Section 2.2).

SBLP '19, September 2019, Salvador, Bahia, Brazil

ACM ISBN 978-1-4503-7638-9/19/09…$15.00
https://doi.org/10.1145/3355378.3355382

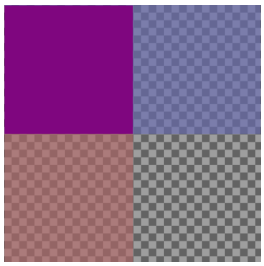


(a) `red`, as loaded from "red_column.png"

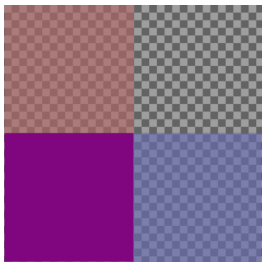


(b) `blue`, as loaded from "blue_bar.png"

Figure 1: Values of variables `red` and `blue`, after the first two lines of Listing 1



(a) result of the expression `red * blue`



(b) value of `purple` variable after line 5

Figure 2: Images produced during execution of line 5 in Listing 1

Listing 1: Example of IML code

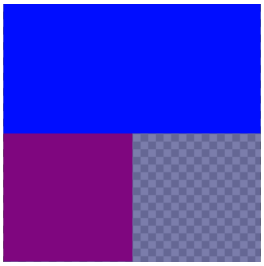
```
1  red = image in "red_column.png"
2  blue = image in "blue_bar.png"
3
4  flip red horizontally
5  purple = flip red * blue vertically
6  purple = purple + blue
7
8  center = (0.4, 0.4, 0.6, 0.6)
9  center = center * (purple dimensions)
10
11 print purple - center
```

2.1 Image Operations

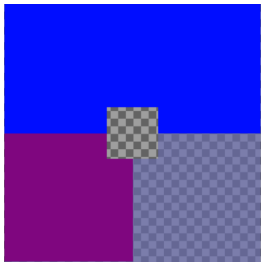
Image operations in IML are statements that produce one image from another, and that can be used both as expressions, representing the newly produced image, as in line 5 of Listing 1; and as commands, altering the original image, as in line 4.

There are five of these operations:

- **rotate**, rotates an image anticlockwise by the given number of degrees (clockwise if negative);
- **flip**, simply flips an image vertically or horizontally;
- **resize**, resizes an image to an absolute dimensions value, or relatively by a given multiplicative factor;
- **crop**, crops a section from an image, producing a new image from that image section;



(a) value of `purple` after line 6



(b) image shown by the print command in line 11

Figure 3: Results of further operations in Listing 1

Table 1: Supported binary operations by operand types.

	Image	Numeric	Dims	Section
Image	+, -, *, /	+, *, /		-
Numeric	+, *	+, -, *, /	+, *	+, *
Dims		+, *, /	+, -, *, /	*
Section	-	+, *, /	*	+, -, *, /

- **modify**, modifies a color attribute of the image by a given factor.

Regarding the modify operation, there are four possible color attributes which can be modified: sharpness, contrast, brightness, and color.

Images can also be modified in IML through unary and binary operators, as we show in Section 2.2. These two ways of modifying images can be composed together, using image operations as expressions, as in line 5 of Listing 1.

2.2 Unary and Binary Operations

Besides what we have described as image operations, there are two unary operators, and four binary ones. We describe here the valid operations in IML for each of these operators.

Regarding the unary operations, there is the unary minus, valid only for numeric types (integer and float); the `dimensions` operator, for obtaining an image's dimensions; and the channel access operator, valid only for images, which is of the form (R), (G), or (B), producing an image containing only the channel of the specified operator (red, green, or blue, respectively).

Now, for the binary operators, we summarize in Table 1 all valid IML operations, for a left operand with type corresponding to each line, and a right operand to each column. Operators are colored when their corresponding operation is non-commutative. Operands of type integer or float are represented as having 'numeric' type, and operands with path type are not depicted, as paths only operate with other paths, but these operations are discussed below as well.

When both operands are images, we have the following operations: with the `+` operator, the right hand side image is simply pasted over the left hand side one (as in Figure 3a); for `-`, the resulting image is given by the pixel-by-pixel difference between the two operands; for `*`, a composite image is produced from the two operands, being an even blend of both (as in Figure 2a); for `/`, we

have the same behaviour of *, but between the first operand, and the color inverse of the second.

When both operands are of numeric type, we have the usual behaviours for each operation. When both are paths, the + operator appends the filename on the second path to the first; and / extends the first path with the second (ignoring filename).

When both operands are of the same type, and are dimensions or sections, the operation is done numerically, element-wise. We have a similar behaviour (element-wise numerical operation) when one of the operands is a dimenions or section, and the other is numerical. When one of the operands is a dimension and the other is a section, the section is considered to be given as proportions, and * produces a section appropriately scaled to the given dimensions.

Finally, when one of the operands is an image and the other is a section, we have the - operator, which removes a section from an image, or removes the image except for the section given, depending on the order of operands. This is exemplified in Figure 3b, which would show only the part remove from Figure 3a, if the operands were reversed in line 11 of Listing 1.

3 SAMPLE COMPARISON

Now, for the sake of illustrating our design objectives for IML, we will take a sample image manipulation task suitable for automation. It consists of placing or overlapping an "overlay" (a partially transparent image, such as a watermark or frame) over each image in a certain directory.

We can see an implementation of this task in IML in Listing 2, and equivalent implementations in Python and Script-Fu for GIMP in Listings 3 and 4, respectively. As can be seen in these examples, the code in IML is generally simpler and makes each step of the task being implemented more evident and clear.

Listing 2: Example of IML code

```
overlay = image in "frame.png"

dir = "my/pics/"
for pic in dir {
  img = image in pic
  resized = resize overlay to (img dimensions)
  img = img + resized
  save img as pic / "new"
}
```

4 CONCLUSION

In summary, we have introduced IML, a work-in-progess DSL for image manipulation. We have showcased some of its features, and compared it to some current alternatives available for automating image manipulation tasks.

The main efforts of future works would be in surveying the intended end users of IML on its usability and readability, and on common image manipulation tasks, which would help guide work on the language. Besides that, we would like to offer better support of operations between images of different size – for which the current version[1] of IML offers little to no support – and, as a related issue, more descriptive runtime errors.

[1]https://github.com/carlosemv/iml/releases/tag/sblp2019

Listing 3: Equivalent example in Python for GIMP

```
dir = "images/"
for file in filter(isfile, (dir+f for f in listdir(dir))):
  img = pdb.gimp_file_load(file, 1)

  overlay = pdb.gimp_file_load_layer(img, "frame.png")
  img.insert_layer(overlay)
  overlay.scale(img.width, img.height)

  path, name = file.rsplit('/', 1)
  new_file = path + "/new/" + name
  layer = img.merge_visible_layers(CLIP_TO_IMAGE)
  pdb.gimp_file_save(img, layer, new_file, new_file)
```

Listing 4: Equivalent example in Script-Fu for GIMP

```
(define (script-fu-batch-overlay)
  (let ((files (cadr (file-glob "images/*.png" 0))))
    (map apply-overlay files)
  )
)

(define (apply-overlay file)
  (let*
    (
      (path (car (strbreakup file "/")))
      (name (cadr (strbreakup file "/")))
      (newfile (string-append path "/new/" name))
      (img (car (gimp-file-load 1 file file)))
      (width (car (gimp-image-width img)))
      (height (car (gimp-image-height img)))
      (front-layer (car (gimp-file-load-layer
                          1 img "frame.png")))
    )
    (gimp-image-insert-layer img front-layer 0 -1)
    (gimp-layer-scale front-layer width height FALSE)
    (let ((drawable
          (car (gimp-image-merge-visible-layers img 0))))
      (gimp-file-save 1 img drawable newfile newfile)
    )
  )
)
```

REFERENCES

[1] Jianbo Chen, Yelong Shen, Jianfeng Gao, Jingjing Liu, and Xiaodong Liu. 2018. Language-based image editing with recurrent attentive models. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 8721–8729.

[2] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. 2012. Diderot: a parallel DSL for image analysis and visualization. In *Acm sigplan notices*, Vol. 47. ACM, 111–120.

[3] Gierad P Laput, Mira Dontcheva, Gregg Wilensky, Walter Chang, Aseem Agarwala, Jason Linder, and Eytan Adar. 2013. Pixeltone: A multimodal interface for image editing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2185–2194.

[4] Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. 2015. Hipa cc: A domain-specific language and compiler for image processing. *IEEE Transactions on Parallel and Distributed Systems* 27, 1 (2015), 210–224.

[5] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.