∇ giza: A Related Set of Bash Expansions of Relevance to AI

Arash Abadpour - arash@abadpour.com

November 19, 2023

Abstract

wip

In this paper, we discuss the Mathematics of building Machine Vision AI systems in Linux. We review the general challenge of translating the description of an AI operation in human language into a human-readable, machine-executable script. We select multiple Machine Vision AI challenges that we first describe in human language. Then, in each case, we build the language to convert the description in human language into one or more scripts we execute on machines. We use AWS SageMaker [1] for development and training and AWS Batch [2] for inference and discuss API calls. The main contribution of this paper is a mathematical framework for building an AI language for a practical use-case in Machine Vision. We hope that researchers in other fields of AI use and extend this framework in their disciplines. We present a reference implementation [3] of this framework and multiple use-cases [4] - *revision-1.110.1*

Contents

[1] `https://aws.amazon.com/sagemaker/`
[2] `https://aws.amazon.com/batch/`
[3] `https://github.com/kamangir/awesome-bash-cli`, *awesome-bash-cli*, *abcli*.
[4] bird watching in downtown Vancouver with AI, `https://github.com/kamangir/Vancouver-Watching`, *Vancouver-Watching*, *vanwatch*.

1 Introduction

Bash is a "Unix shell and command language first released in 1989 that has been used as the default login shell for most Linux distributions" [5]. A shell is a "macro processor that executes commands" [6], where "macro processor means functionality where text and symbols are expanded to create larger expressions" (same reference). There are seven kinds of expansions in Bash [7].

Brace Expansion [8] is the first and the quickest to explain,

```
 > bash$ echo a{d,c,b}e
ade ace abe
```

Tilde Expansion [9] relates to words that begin with an unquoted tilde character (˜). *Parameter and Variable Expansion* [10] enable the use of variables, as `${variable}`, as well as more elaborate pattern matching forms such as `${parameter/#pattern/string}`. *Command Substitution* "allows the output of a command to replace the command itself" [11]. *Arithmetic expansion* [12] enables arithmetic operations using the form `$(( expression ))` and *Word Splitting*[13] governs the splitting of the command to words. Finally, *Filename Expansion* [14] enables the familiar wildcard reference to filenames using '*' and '?'. In Section 2 we propose a set of relevant expansions to AI operations that are implemented using Python [15].

This work, first, proposes several novel Bash expansions based on command substitution 2.1. Then, using typed positional arguments, we propose *options* 2.5 and *objects* 2.6, including *object metadata* **??**. Then, we discuss argument injection as a suffix 3.2 and *Prefixing* 2.7 to transform `<command>` to `<prefix> <command>`, as used by `@docker` **??**, for example. The next two expansions, `<command> help` **??** and `@init <args>` 3.4 are suggestions to reduce the cognitive load of the user.

Then, we discuss the `<core>` **??**, which is the callable that is `source`'d in a startup file [16]. The `plugins` 3.5 enable the addition of branches to the syntax, and `scripts` **??** implement the last mile. We then discuss the `@seed` 2.8; the notion that code generates code that is transferred into another machine through the clipboard, a key, or a `scp` [17]–style protocol to terraform the machine and start a process A.3. `@start` 3.6 is a necessity; the first intelligent command to start the day with. `@start` behaves according to the machine it runs on and other aspects of the state A.1. On a MacBook, `@start` logs in and starts an `ssh` session to the default machine. On that machine, `@start` starts the docker container. Note that the specific action of `@start` is decided by its immediate user [18]. We then discuss practical expansions for `@conda` **??**, `@git` 2.9, and wip. Finally, we discuss the `@` **??**.

This work also contributes a set of conventions 3 that enable more effective use of the proposed expansions.

[5] https://en.wikipedia.org/wiki/Bash_(Unix_shell)
[6] https://www.gnu.org/software/bash/manual/html_node/What-is-a-shell_003f.html
[7] https://www.gnu.org/software/bash/manual/html_node/Shell-Expansions.html
[8] https://www.gnu.org/software/bash/manual/html_node/Brace-Expansion.html
[9] https://www.gnu.org/software/bash/manual/html_node/Tilde-Expansion.html
[10] https://www.gnu.org/software/bash/manual/html_node/Shell-Parameter-Expansion.html
[11] https://www.gnu.org/software/bash/manual/html_node/Command-Substitution.html
[12] https://www.gnu.org/software/bash/manual/html_node/Arithmetic-Expansion.html
[13] https://www.gnu.org/software/bash/manual/html_node/Word-Splitting.html
[14] https://www.gnu.org/software/bash/manual/html_node/Filename-Expansion.html
[15] https://github.com/kamangir/awesome-bash-cli
[16] https://www.gnu.org/software/bash/manual/html_node/Bash-Startup-Files.html
[17] https://linux.die.net/man/1/scp
[18] the tool is adapted to the tool user.

2 Expansions

2.1 Command Substitution

During the execution of the command `<part-1>$(<sub-command>)<part-2>`, `<sub-command>` is executed and its outcome, `<outcome>`, is used to generate the updated command as `<part-1><outcome><part-2>`, which is then executed [19]. Here is an example from `roofAI` 4.3,

```
roofAI semseg predict \
    profile=FULL,upload \
    $(@ref roofAI_semseg_model_AIRS_full_v2) \
    $(@ref roofAI_ingest_AIRS_v2) \
    $(@timestamp)
```

Here, `@ref <keyword>` reads the value of `<keyword>` from the `cache` 2.2 and `@timestamp` generates a unique timestamp for use as an `<object-name>`. Collectively, this command runs the "Pytorch Segmentation Model" [20] that is cached as `roofAI_semseg_model_AIRS_full_v2` on the dataset that is cached as `roofAI_ingest_AIRS_v2` and uploads the results in a uniquely named object 2.6. Tags 2.3 and relations 2.4 are other object metadata relevant to this expansion.

Command substitution is useful for generating the command components through Python or Bash. For example, in the AWS Open Data Registry [21] there is the notion of datasets, such as `hst` [22] for Hubble Space Telescope and the metadata the dataset is maintained in `yaml` files a git repository [23].

For example, here is the command to access `ibrma2f2q_drc.jpg` in object `public/ibrm/ibrma2f2q` in the dataset `hst`,

```
aws s3 cp $auth $s3_uri$filename $path
```

Here, `$auth` and `$s3_uri` are generated as,

```
auth=$(abcli_hubble_get auth $dataset_name)
s3_uri=$(abcli_hubble_get s3_uri $dataset_name $hubble_object_name)
```

Here, `abcli_hubble_get` is a Bash wrapper around a Python call.

```
function abcli_hubble_get() {
    python3 -m hubble get \
        --what "$1" \
        --dataset_name "$2" \
        --object_name "$3" \
        "${@:4}"
}
```

2.2 `@cache`

The cache is a keyword-value dictionary available on every machine A.1 for reading, writing, and searching, that is enabled either through a SQL database [24] or a tool such as `mlflow` [25].

```
value=$($@cache read <keyword>)

@cache write <keyword> <value>

value=@cache search <options>
```

When used for objects 2.6, `@cache` provides a tagging 2.3 mechanism,

[19] https://www.gnu.org/software/bash/manual/html_node/Command-Substitution.html
[20] https://github.com/qubvel/segmentation_models.pytorch
[21] https://registry.opendata.aws/
[22] https://registry.opendata.aws/hst/
[23] https://github.com/awslabs/open-data-registry/blob/main/datasets/hst.yaml
[24] https://github.com/kamangir/awesome-bash-cli/blob/2023-06-aws-batch-a/abcli/plugins/tags/functions.py
[25] https://mlflow.org/

```
@cache read <object-name>.<keyword>

@cache write <object-name>.<keyword> <value>

@cache clone <object-1> <object-2>
```

2.3 `@tag`

An object 2.6 2.6 can have many tags. A tag is a boolean or valued property of the object and is `set` and `get`, and can be `search`ed,

```
@tags set <object-name> <options>

@tags get <object-name>
@tags get <object-name> <keyword>

@tags search <options>
```

2.4 `@relations`

Two objects 2.6 can be related in several ways, each defined as a pair, to enable directional relations [26]. Here is an example,

```
{
    "added-to": "contains",
    "cloned": "cloned-by",
    ...
    "trained": "trained-on"
}
```

`relations` can be `set`, `get`, and `search`ed,

```
@relations set <object-name-1> <object-name-2> <relation>

@relations get <object-name-1> <object-name-2>
@relations get <object-name-1> <object-name-2> <relation>
@relations get <object-name-1>
@relations get <object-name-1> <relation>

@relation search <object-name> [--relation <relation>]
```

2.5 `options`

An `options` is a string representation of a dictionary, such as,

```
<keyword-1>=<value-1>,<keyword-2>=<value-2>,...,<keyword-3>,-<keyword-4>},...
```

`options` is implemented using basic Python [27] and, therefore, the *options expansion* is available to Bash commands through command substitution 2.1. In practice, a second `@option::bool` expansion is defined to cover boolean variables [28],

```
value=$(@option "$options" <keyword> <default>)

value=$(@option::bool "$options" <keyword> 0|1)
```

[26] https://github.com/kamangir/awesome-bash-cli/blob/2023-06-aws-batch-a/abcli/plugins/relations/relations.json

[27] https://github.com/kamangir/awesome-bash-cli/blob/2023-06-aws-batch-a/abcli/options

[28] It may be possible to combine `@option::bool` into `@option`, which remains an interest of the author.

Another useful expansion is the *options choice expansion* [29],

```
choice=$(@option::choice "$options" <comma,separated,list> <default>)
```

The following three operations are also useful on `options`.

default `default <options-1>` to the corresponding values in `<options-2>`. The keyword set of the output is the concatenation of the keyword sets of the two inputs, wherein the values from `<options-1>` take priority. For example, defaulting `x=1,y=2` to `x=3,z=4` yields `x=1,y=2,z=4`. This is the *default option expansion*, which is achieved through `<options-2>,<options-1>`.

subset return the `<options-1> subset` of `<options-2>`. The keyword set of the output is the same as the keyword set of `<options-1>`, wherein the values from `<options-2>` take priority. For example, the `x=1,y=2` subset of `x=3,z=4` yields `x=3,y=2`. This is the *option subset expansion*,

```
options=$(@option::subset <options-1> <options-2>)
```

update `update <options-1>` to `<options-2>`. The keyword set of the output is the concatenation of the keyword sets of the two inputs, wherein the values from `<options-1>` take priority. For example, updating `x=1,y=2` to `x=3,z=4` yields `x=3,y=2,z=4`. This is the *option update expansion*, which is achieved through `<options-1>,<options-2>`.

[30]

2.6 Objects

Commands A.3 consume and generate objects. Objects are accessible on any machine A.1 by `<object-name>`, and an object may be *selected*,

```
@select <object-name>
```

```
@select <type> <typed-object-name>
```

When an object is selected, `.` expands to `<object-name>`. Similarly, `..`, `...`, and so on, as deep as needed, expand to the name of the previously selected object and the one before that. Commands default the objects they consume and modify to `.`, `..`, and so on. Because the commands in a script use the same objects, selecting the objects enables their names to be omitted in a script.

[29] As an example, this expansion allows `abcli list cloud|local <object-name>` using `where=$(option::choice "$options" cloud,local cloud)`.

[30] An `options` can be

- read from,
  - a single keyword,
    - that is boolean: `@option::bool`
    - that is not boolean: `option`.
  - a group of keywords: `option::choice`.
- written to,
  - `keywords=<options-1>.keywords`,
    - priority: `<options-1>.values`: `<option-1>`
    - priority: `<options-2>.values`: `@option::subset <options-1> <options-2>`
  - `keywords=<options-2>.keywords`,
    - priority: `<options-1>.values`: `@option::subset <options-2> <options-1>`
    - priority: `<options-2>.values`: `<option-2>`
  - `keywords=<options-1>.keywords + <options-2>.keywords`,
    - priority: `<options-1>.values`: `<option-2>+<option-1>`
    - priority: `<options-2>.values`: `<option-1>+<option-2>`

The list of expansions and the mathematical properties of a dictionary are related. The author wishes to understand this relationship better.

```
@select <object-1>
@select <object-2>
@download
<command-1> # ., .. omitted for convenience
<command-2>
<command-3>
@upload
```

An object may have a *type*, such as model or dataset. Commands that consume objects specify a type for the argument. This enables the user to simultaneously select different types of objects and run commands on them. Here is an example from *hubble* [31], wherein the user selects an object, then selects a hubble dataset, then selects an object in that dataset and downloads it.

```
@select
hubble select dataset hst
hubble select object public/u4ge/u4ge0106r
hubble download -dryrun
```

If `<object-name>` is not provided or is given as `-`, then an object with a unique name is created and used. An object points to an S3 bucket [32] [33]. *Metadata* is information about Objects 2.6, such as their *tags* 2.3 and relations 2.4, and the information cached 2.2 about them. Objects can be downloaded, uploaded, and listed,

```
@download [.|<object-name>] [filename=<filename-1>+<filename-2>]

@upload [.|<object-name>] [filename=<filename-1>+<filename-2>]

@list cloud|local <object_name>
```

It is recommended that additional `download` and `list` expansions are defined for typed objects. See hubble 4.1 for examples.

2.7 Prefixing

wip

2.8 @seed

2.9 @git

git pull

[31] https://github.com/kamangir/hubble

[32] https://aws.amazon.com/s3/

[33] Special variables, such as `$abcli_object_name`, `$abcli_object_path`, `$abcli_hubble_dataset_object_name`, `$abcli_object_name_prev2` carry the name of the selected object, its path, the name of current selected hubble dataset and the second previous selected object, respectively.

3 Conventions

Conventions augment and enable expansions 2 or are found to be helpful.

- The *core* is a callable A.3.1 that loads the plugins and terraforms the machine A.1.
- When the command ends with help, a help message prints.
- @ref is an alias for @cache read that enables $(@ref <keyword>) and, thus, object alias [34].

3.1 $<core>_is_<what>

During initialization, a series of variables are set by the *core* and are used to harmonize paths and other machine A.1–specific parameters to enable the same Python code to run on different machines simultaneously. An alias of env lists these variables [35],

```
@env [<keyword>]
```

3.2 --<keyword> <value>

argparse [36], click [37], fire [38], and many other command line parsers support the --<keyword> <value> convention. By ending Python calls with "$@:<number>" the rest of the arguments are captured. Here is an example,

```
abcli message submit \
    [--data <data>] \
    [--filename <filename>] \
    [--recipient <host_1,host_2>] \
    [--subject <subject>]
```

This is achieved through [39],

```
python3 -m abcli.plugins.message \
    submit_object \
    --object_name "$object_name" \
    --recipient "$recipient" \
    "${@:5}"
```

3.3 --delim space

Command substitution 2.1 is useful in for loops and other usages where a delimited list of keywords is consumed.

```
    local object_name
    for object_name in $(<command-1> --delim space); do
        <command-2> $object_name <args>
    done
```

Here is one way to compose the .py cli,

[34] Refer to https://github.com/kamangir/roofAI/blob/main/roofAI/semseg/README.md for an example.

[35] On a MacBook, at the time of writing, abcli_is_64bit=true, abcli_is_amazon_linux=false, abcli_is_docker=false, abcli_is_ec2=false, abcli_is_headless=false, abcli_is_in_notebook=false, abcli_is_jetson=false, abcli_is_mac=true, abcli_is_rpi=false, abcli_is_sagemaker=false, abcli_is_sagemaker_system=false, abcli_is_ubuntu=false, abcli_is_vnc=false.

[36] https://docs.python.org/3/library/argparse.html

[37] https://palletsprojects.com/p/click/

[38] https://google.github.io/python-fire/

[39] https://github.com/kamangir/awesome-bash-cli/blob/2023-06-aws-batch-a/bash/plugins/message.sh

```
...
parser.add_argument("--count", type=int, default=-1)
parser.add_argument("--delim", type=str, default=",")
...
delim = " " if args.delim == "space" else args.delim
...
elif args.task == "foo":
    output = foo()
    if count != -1:
        output = output[:count]
    print(delim.join(count))
...
```

3.4 @init

wip

3.5 Plugins

A plugin generally defines one or more callables.

3.6 @start

4 Examples

4.1 hubble

`hubble` [40] is a callable A.3.1 that `selects`, `lists`, and `downloads` data from AWS Open Data Registry [41].

```
hubble select [dataset] <hubble-dataset-name>
hubble select [object] <hubble-object-name>

hubble list [dataset] [.|<hubble-dataset-name>]
hubble list [object] [.|<hubble-object-name>]

hubble download \
    [~dryrun,filename=<filename>|all,~ingest,upload] \
    [.|<hubble-object-name>] \
    [.|<object-name>]
```

4.2 OpenAI code generation

Experiments w/ the OpenAI API [42]. wip

4.3 roofAI

`roofAI` [43] is a callable that,

1. terraforms the machine and the shell **??**.
2. ingests and reviews datasets 4.3.1.
3. train semantic segmentation [44] models 4.3.2.
4. runs semantic segmentation predictions 4.3.3.

4.3.1 roofAI dataset

`roofAI dataset ingest source=<source> <object-name>` ingests a dataset from `source` into `<object-name>` and tags it for future discovery. `roofAI dataset review - <object-name>` reviews the dataset in `<object-name>`.

```
roofAI dataset ingest \
    [source=AIRS|CamVid,register,suffix=<v1>] \
    <object-name> \
    [<args>]

roofAI dataset review \
    [-] \
    <object-name> \
    [<args>]
```

[40] https://github.com/kamangir/hubble
[41] https://registry.opendata.aws/
[42] https://github.com/kamangir/openai
[43] https://github.com/kamangir/roofAI
[44] semseg

4.3.2 roofAI semseg train

`semseg train - <dataset-object-name> <model-object-name>` trains a model on `<dataset-object-name>` that it stores in `<model-object-name>` and tags for future discovery.

```
semseg train \
    [device=cpu|cuda,register,suffix=<v1>] \
    <dataset-object-name> \
    <model-object-name> \
    [<args>]
```

4.3.3 roofAI semseg predict

`semseg predict - <model-object-name> <dataset-object-name>` runs a prediction on the dataset `<dataset-object-name>` using the model `<model-object-name>`.

```
semseg predict \
    [device=cpu|cuda] \
    <model-object-name> \
    <dataset-object-name> \
    <prediction-object-name>
```

4.4 Vancouver-Watching (vanwatch)

`vanwatch` [45] is a callable that,

1. terraforms the machine and the shell **??**.
2. discovers the cameras in an area 4.4.1.
3. ingests images from the cameras discovered in an area 4.4.2.
4. detects the objects in the images ingested from an area and produces summary statistics 4.4.3.

4.4.1 vanwatch discover

Cameras are represented in different formats in different areas. `vanwatch discover area=<area>` discovers the cameras in `<area>` and stores them in `<area>.geojson` in the object `<object-name>` 2.6 tagged 2.3 for use by `ingest` 4.4.2.

```
vanwatch discover \
    [area=<area>] \
    [-|<object-name>] \
    [<args>]
```

`object-name` is tagged for retrieval by `vanwatch list discovery`.

4.4.2 vanwatch ingest

`vanwatch ingest area=<area>,count=<count> <object-name>` finds the latest set of cameras discovered 4.4.1 in `<area>` through tag search 2.3 and ingests `count` images into `<object-name>` and the runs `vanwatch process` 4.4.3 unless `-process`.

```
vanwatch ingest \
    [area=<area>,count=<-1>,model=<model-id>,~process,publish] \
    [-|<object-name>] \
    [<args>]
```

`object-name` is tagged for retrieval by `vanwatch list ingest`.

[45] https://github.com/kamangir/Vancouver-Watching

4.4.3 vanwatch process

`vanwatch process - <object-name>` runs object detection [46] on the images ingested into `<object-name>` and updates `<area>.geojson`. `vanwatch process` reuses the inference in the object and completes the missing pieces.

```
vanwatch process \
    [model=<model-id>,publish] \
    [.|<object-name>] \
    [<args>]
```

`+publish` tags `object-name` for retrieval by `vanwatch update_QGIS`.

[46] https://hub.ultralytics.com/models/R6nMlK6kQjSsQ76MPqQM?tab=preview

A Concepts

A.1 Machines and Shells

A *machine* is a state machine that is connected to many other machines and shares some of its state with them for read and write. A *shell* is a stateful access mechanism to a machine that an *operator* A.2 uses to run *commands* A.3. Running a command in a shell can modify the state of the shell, the machine on which the shell is running, and potentially the states of all other machines.

Machines and shells may be restarted by an operator or by code. After a restart, machines and shells maintain some of their state.

Two examples of machines are a Raspberry Pi that runs Linux and is connected to AWS [47] and a docker container running in AWS Batch. GNU Bash [48] is an example of a shell.

A.2 Operators

The *operator* generates commands and runs them on different shells on different machines. The operator attempts to maximize an objective function that depends on the state of multiple machines.

A.3 Commands

A *command* is any Bash command [49] and can be represented in a Python string of characters [50]. Here is an example command,

```
vanwatch ingest \
    vancouver \
    dryrun \
    . \
    --count 12
```

The above command and the one below are *identical*.

```
vanwatch ingest vancouver dryrun . --count 12
```

Two commands are identical if running them on two machines in identical states yields the same states. In theory, the state of any machine depends on the state of any other machine, and it is almost impossible to run two commands in identical states, including the time of execution. Therefore, when we refer to two identical commands, we either use a derivation-based proof of identity or consider a validation in a limited "relevant" subset of the state representation.

For any shell on any machine, at known states, there is a mapping between the set of all commands and $\{True, False\}$ that we address as "whether the command is found". In Bash, for example, the following message is printed when a command "is not found".

```
-bash: void: command not found
```

Note that writing to the standard streams *stdin* and *stdout* are examples of state changes in the shell and the machine.

Terraforming is the process of running commands that modify the state of the shell and the machine in ways that make additional commands found. Terraforming is also intended to modify the state change caused by a set of commands favourable to the interests of an operator. For convenience, we address a command "that is found" as a *valid* command and *invalid* otherwise. Terraforming may also ensure the states of the machine and the shell after a restart. Terraforming generally includes a modification of *bashrc*, *bash_profile*, *screenrc*, and *desktop* files.

Commands know the machine and the shell they are running in and adjust their operation accordingly. For example, a script that submits jobs inside a docker container may download the artifacts generated through previous submissions on a user-facing machine, such as a Macbook.

[47] https://aws.amazon.com/
[48] https://www.gnu.org/software/bash/
[49] https://www.gnu.org/software/bash/manual/bash.html#Shell-Syntax
[50] https://docs.python.org/3/library/string.html

The first word [51] in a command is generally the callable A.3.1. The rest of the command is expected to follow the conventions of the callable. In this paper we propose guidelines that we later demonstrate lead to useful expansions 2.

A.3.1 Callables

A *callable* is a valid command with no space and control operators [52]. The list of callables depends on the machine's state and is generally extended through terraforming. Some of the well-known callables are *git* [53], *docker* [54], *pushd* [55], *nano* [56].

Theorem 1 *For any callable ⟨callable⟩, and any string ⟨suffix⟩, "⟨callable⟩ ⟨suffix⟩" is a valid command.*

Theorem 2 *For any valid command ⟨command⟩, and any string ⟨suffix⟩, "⟨command⟩ ⟨suffix⟩" is a valid command.*

The `core` **??** is a callable. Many plugins 3.5 define their callable.

A.3.2 Command Templates

Commands can be similar when considered as strings of characters. Here is a command that is similar to the above,

```
vanwatch ingest toronto upload . --count 3
```

A *command template* is a representation that yields similar commands, given the following rules. First, ⟨description⟩ can be replaced with any string of characters that can be described as "description". See *objects* 2.6, *options* 2.5, and *arguments* **??**, for the next rules. Here is a command template for the two above,

```
vanwatch ingest \
    <area> \
    [dryrun,˜upload] \
    [<object-name>] \
    [--count <-1>]
```

A.4 Computational Model

A group of operators create and modify a set of scripts **??** maintained in a code repository, such as *git* 2.9. The core **??** and the plugins 3.5 are also maintained in one or more repositories. Each operator can access a set of machines and create shells on them. Each operator can also access a set of repositories and clone them on the machines where they create shells and receive updates. The operator can modify any of the repositories that they have access to following a collective peer-reviewed *pull-request* [57] process.

The operators act asynchronously while communicating with each other. Multiple operators may simultaneously use the same machine, and the same operator may simultaneously use multiple machines. Only one operator uses a shell at one time.

Some machines are exogenous to this model, yet the operators can access their states in read or write mode. Cloud storage and compute resources are examples of these machines.

[51] https://www.gnu.org/software/bash/manual/html_node/Shell-Syntax.html
[52] https://www.gnu.org/software/bash/manual/bash.html#index-control-operator
[53] https://git-scm.com/docs/git
[54] https://docs.docker.com/engine/reference/commandline/cli/
[55] https://www.gnu.org/software/bash/manual/bash.html#Directory-Stack-Builtins
[56] https://www.nano-editor.org/
[57] https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-reque
about-pull-requests

B Background and Context

Almost five years ago, on Thursday, November 8, 2018, I acquired a Raspberry Pi [58] on Amazon. Since then, my personal and professional lives have focused on Linux. Professionally, I do AI and, more recently, geospatial AI. In my personal life, I mix AI, cloud, and Mathematics into minimal forms that seek survival [59]. Over the years, I have built a set of mechanisms for building AI systems that I will document in this paper. Therefore, this is an attempt to produce formal mathematical definitions for the AI mechanisms that I will collectively refer to as *giza*. I seek to understand these mechanisms through this effort better to use them more optimally and along new dimensions.

This paper discusses concepts at the intersection of mathematics, software science, and computer science, and lacks scientific rigour in many places. I intend to push the practical development of this theory to fruition and hope to receive guidance along the way from experts in the field and solidify the theoretical foundations.

[58] https://www.raspberrypi.org/
[59] https://github.com/kamangir