LapsPython

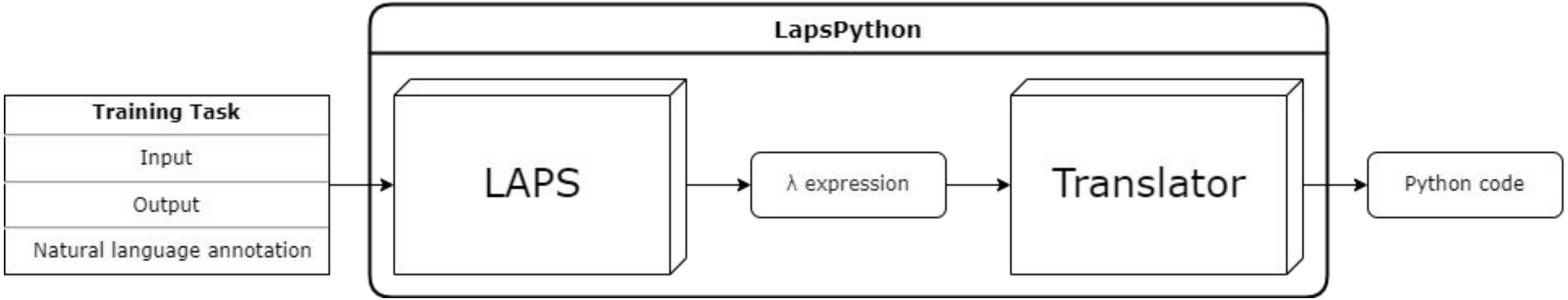
Extend LAPS to synthesize Python/R

Christopher Brückner & Enisa Sabo

13.06.2022

Objective

Extend LAPS to synthesize Python/R code from natural language



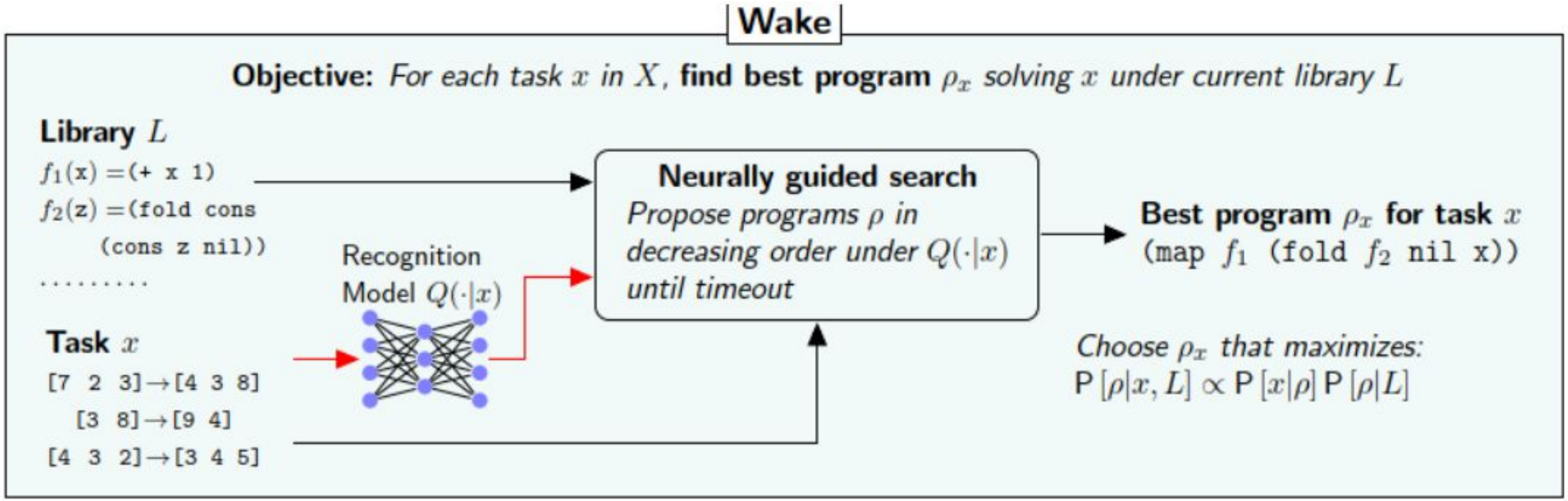
- Create rule-based translator from λ-calculus to Python code
- Define sets of primitives and tasks that target useful domains

LAPS Recap

- LAPS has a library of primitives
  - Simple functions implemented in Python and OCaml

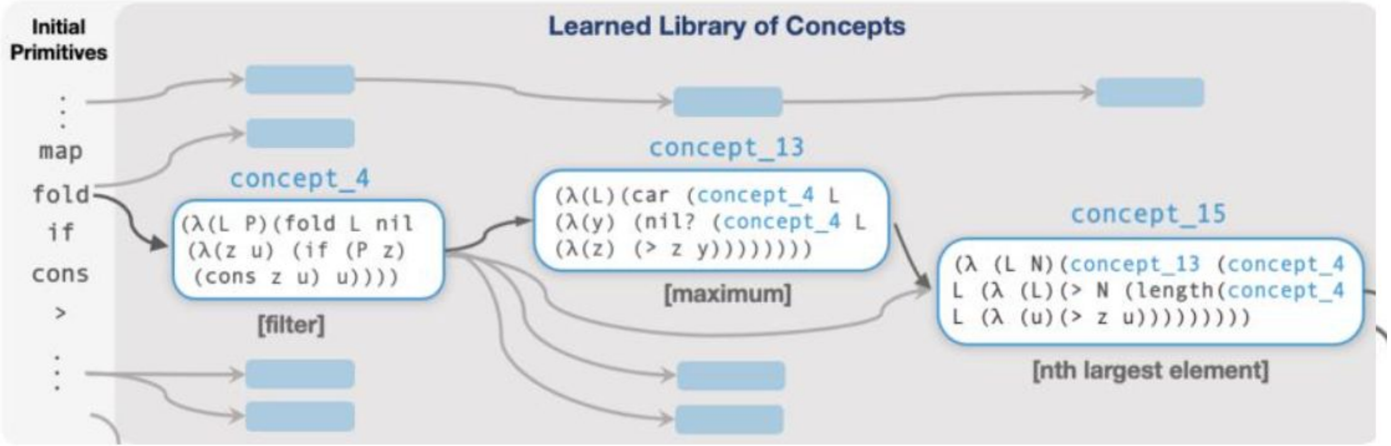
- Each iteration consists of 3 phases
  - Program synthesis
  - Program compression
  - Model training

LAPS Recap: Phase 1



- Tasks are annotated with natural language descriptions
- Search is also guided by natural language model

LAPS Recap: Phase 2



- Find patterns (reused "code") in synthesized programs
- Add patterns to library as "invented primitives"
- Compress programs using the extended library

LapsPython: Current State

- After each iteration, LAPS saves a checkpoint containing:
  - Libraries of each iteration
  - Synthesized programs for all solved tasks

- LapsPython loads these checkpoints and extracts:
  - Source codes of pre-implemented primitives in current library
  - Invented primitives in current library
  - All synthesized programs for each task
    - Alternative: Only the “best” ones

- LapsPython does not yet directly interact with LAPS

Example: Extracted Primitives

```
def _rsplit(s1): return lambda s2: __regex_split(s1, s2)
```

```
(λ (_rsplit s1 s2)) = _rsplit(s1)(s2)
```

⇒ We reformat extracted primitives

```
def _rsplit(s1, s2):
    return __regex_split(s1, s2)
```

Example: Translation

Task: if the word ends with any letter, add w after that

```
(λ (_rflatten (_rappend _w (_rsplit _d $0))))
```

Translation:

```
def __regex_split(s1, s2):
    [...]
def if_the_word_ends_with_any_letter_add_w_after_that(arg1):
    _rsplit_1 = __regex_split('d', arg1)
    _rappend_1 = _rsplit_1 + ['w']
    return "".join(_rappend_1)
```

Project Plan: Sprint 1

Extraction of programs Deadline: 06.06.

- Extract implementations of primitives as strings for translation ✔
- Extract synthesized λ expressions to be translated ✔
- Extract λ expressions from learned library to be translated ✔
- Parse λ expressions to construct Abstract Syntax Tree

LAPS stores its synthesized programs in a tree structure

⇒ We skipped the last issue

Project Plan: Sprint 2

Translation of programs   Deadline: 20.06.

- Implement Python code generation for simple trees (arithmetics, procedures) ✔
- Extend translation to subset of 1 pre-implemented domain (string editing) ✔
- Extend translation to full domain

- Translation works if there are no invented primitives
  - Invented primitives must be translated as well, this is not working yet

- For many tasks, no solutions are found (1 day runtime)
  - Impossible to test the implementation for the full domain
  - Goal: Extend translation to the results we have

Translation Testing

How to verify generated Python code?

⇒ Python's built-in exec() function

```
for example in task:
    input, example_output ∈ example
    exec("python_output = translated_function(input)")
    assert python_output == example_output
```

Fun Fact: Notation

LAPS is not actually using LISP notation like in the paper:

Standard Notation (~LISP)	de Bruijn Notation
λx.λy.x	λ.λ.1
λx.λy.λs.λz.x s (y s z)	λ.λ.λ.λ.3 1 (2 1 0)
(λx.λx.x) (λy.y)	(λ.λ.0) (λ.0)

de Bruijn indices bind exactly 1 variable to each λ

⇒ compact, but harder to parse (fortunately, LAPS provides necessary tools)

⇒ λ in paper bind more than 1 variable (better readability)