StitchArc Technical Report

Dennis Lindberg[1]
University of Amsterdam, ILLC

Supervisor: Dr. Fausto Carcassi

March 27, 2025

1 Introduction

The *Abstract and Reasoning Corpus for Artificial General Intelligence* (ARC-AGI), introduced by Chollet [9], is a dataset designed to evaluate progress towards artificial general intelligence. The tasks in ARC-AGI are structured as small grid-based puzzles requiring reasoning and pattern recognition, where the contestant must infer transformation rules from a few input-output examples. There have been many attempts at solving the challenge, but as of March 19, 2025, the best solutions[2] hover around 55% accuracy on the private evaluation set [8].

As evident from the 2024 Arc prize competition, the highest performance came from deep learning based models. Three out of the top five submissions used some form of deep learning, and the first place winner relied on fine-tuning large language models [3, 7, 10]. All of these methods relied on synthetically generating training data, which was enabled by Michael Hodel's *Re-arc* repository [11]. Hodel encodes the underlying logic of all 400 training tasks in Python, allowing users to create new input-output examples based on the same logic. Programs are written in a Domain Specific Language (DSL) that defines primitives like `bottomhalf`, which extracts the bottom half of the grid. However, Hodel's repository is more than a way to generate training data, it is a collection of programs that solve ARC-AGI puzzles.

Program synthesis often refers to constructing programs from a collection of parts such that the final program satisfies semantic and/or syntactic specification [13]. In the early days, program synthesis was the go-to approach for solving ARC-AGI puzzles, and to this day, many of the most successful attempts make use of it [7]. These methods rely on a carefully crafted DSL and search to combine primitives into programs. Another way to do program synthesis is to search through a collection of programs and find commonalities from which abstractions can be made. This approach is often called *library learning*, and a notable example of one such method is Stitch [4]. Stitch builds abstractions from programs written in a DSL, and matches syntactic structures across programs. To the best of my knowledge, Stitch has never been applied to solve ARC-AGI puzzles. One of the barriers is that Stitch only works on programs written in a lambda-calculus lisp-like syntax.

This project is a first attempt at bringing down the barrier that holds back Stitch from being applied to the Arc challenge. By enabling a reliable translation from Python to Stitch programs, it would open the possibility to apply large-scale library learning on programs tailored to solve ARC puzzles.

1.1 Overview of Translation Pipeline

There are four major parts used to enable Python to Stitch translations.

1. **Flatliner** [12]: flattening multi-line Python functions into a single line of lambda expressions.
2. **Transformations**: refines the single-line lambda expressions into a format supported by the Python-to-Stitch translator.
3. **Python-to-Stitch translator** [5]: translates the refined Python expressions to Stitch.
4. **Stitch** [4]: the library learning algorithm that runs on a collection of Stitch programs.

Here is an example of a Python function being translated to a Stitch expression:

[1]Code is available at: `https://github.com/dt-lindberg/stitchArc`.
[2]There are models that perform better on the public evaluation set [6].

Input Python Function

```
def f(x):
    if x > 0:
        return x % 2
    else:
        return x ** 2
```

Resulting Stitch Expression

```
(lam (ifElse (gt $0 0) (mod $0 2) (pow $0 2)))
```

Stitch uses de Bruijn indices, represented by $0, which can be read as a variable that is bound to the first lambda preceding it [2]. The purpose of this report is to walk through the modules that perform this translation, mainly focusing on the second module *transformations*.

2 Translation Modules

Module Source: `./src/Flatliner/Flatliner.py` — Github Link

The Flatliner repository [12] takes Python files as input and transforms the code into a single lambda expression. It relies on Python's built-in abstract syntax tree (AST) [1]. Running Flatliner on the previous example Python function yields the following:

Flatliner Output

```
lambda x: ((x % 2) if (x > 0) else (x ** 2))
```

The problem with this lambda expression is that it relies on certain syntax specific to Python, making it incompatible with the Python-to-Stitch translator. In this example the problems are the modulo operator (%), power operator (**), greater than operator (>), and Python's if-else syntax. In order to make expressions compatible, parts like these must be replaced by function calls. These functions need not be defined, but they must consistently encode the underlying logic. This is what the *transformations* module does.

Module Source: `./src/Flatliner/transformations.py` — Github Link

The transformations module defines a variety of AST node transformers, which allows it to rewrite Python expressions while keeping the underlying logic intact. The AST objects can identity operators like greater than (>) and substitute it with a function `gt`, seamlessly taking care of > being an infix operator, while `gt` is used as a prefix. Running the transformations on the Flatliner output above yields the following:

Transformations Output

```
lambda x1: ifElse(gt(x1)(0))(mod(x1)(2))(pow(x1)(2))
```

Below is a simplified version of the program that transformed the Flatliner output. It begins by parsing the input into an AST, then it transforms the operators (%, >, **), followed by transforming the if-else statement, and lastly, curries the resulting expression.

```
def transform_lambda(expr_str):
    tree = ast.parse(expr_str, mode='eval')   # ---------- Examples ----------
    tree = OperatorTransformer().visit(tree) # 13+2 -> add(13, 2)
    tree = IfExpTransformer().visit(tree) # if z then x else y -> ifElse(z, x, y)
    tree = CurryTransformer().visit(tree)    # add(x, y) -> add(x)(y)
```

All of the `...Transformer` classes follow a similar structure, here is an example of `OperatorTransformer`:

```
class OperatorTransformer(ast.NodeTransformer):
    OP_MAP = {
        # more operators ...
        ast.Mod: "mod",
        ast.Pow: "pow",
        ast.Gt: "gt",
        # more operators ...
    }

    def visit_BinOp(self, node):
        func_name = self.OP_MAP.get(type(node.op))
        if func_name:
            return ast.Call(
                func=ast.Name(id=func_name, ctx=ast.Load()),
                args=[self.visit(node.left), self.visit(node.right)],
                keywords=[]
            )
        return self.generic_visit(node)
```

I obscured part of the `OP_MAP` for readability, but it defines which operators we want to act on and how to deal with them. `ast.Mod:"mod"` means that the modulo operator (%) will be turned into a function `mod`. The definition of `visit_BinOp` is overloading `ast.NodeTransformer`, and whenever a binary operator is encountered, the program will be executed. It first checks if `func_name`, i.e., verifying that this is a binary operator we know how to deal with. If that is true, then it returns a new AST node with a function named `func_name`, with its arguments being its left and right nodes. If the `func_name` is none, then it returns a generic visit, meaning that it does nothing.

Module Source: `./src/LOTlib3/StitchBindings/python_to_stitch.py` — Github Link

Once expressions has undergone these transformations, they are fed to the Python-to-Stitch translator [5]. It formats the Python expressions into strings of Stitch expressions, for example by replacing variable names with de Bruijn indices. Running the translation on the transformations output yields the final Stitch expression:

Python-to-Stitch Output

```
(lam (ifElse (gt $0 0) (mod $0 2) (pow $0 2)))
```

3 Results

After running this pipeline on all 400 generator functions from the Re-arc repository, 223 (55.75%) make it through to become valid Stitch expressions. 14 (3.5%) functions are not properly processed by the Flatliner and Transformations modules, and an additional 163 (40.75%) functions are not properly transformed by the Python-to-Stitch translator. With the remaining 223 expressions, Stitch runs for 200 iterations with an arity of 4. It takes 2 hours for the process to terminate on hardware with 18 CPU cores and 120 GB of memory, of which at most 15 GB were used at any given time. Note that calling compress with `iterations=200` did not work because it ran out of memory. Therefore, it was necessary to perform one iteration at a time, and wrap the compress call in a loop, feeding the most recent re-written programs as input.

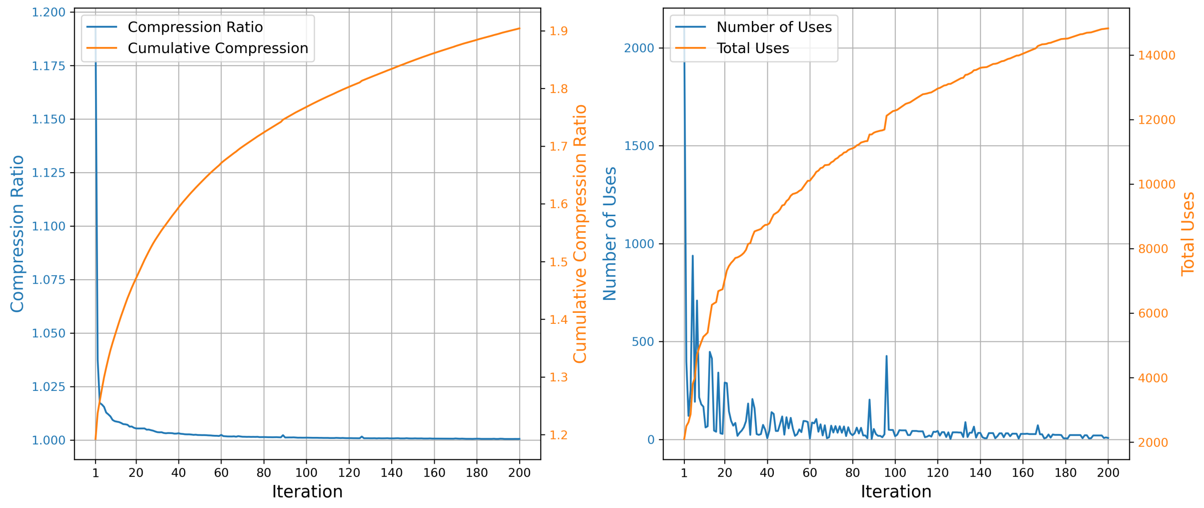
Module Source: `./src/stitchArc/run_stitch.py` — Github Link

```
from stitch_core import compress

latest_programs = stitch_functions # Initialize programs

for i in range(n_iterations):
    # Run the Stitch on the latest programs
    res = compress( latest_programs,
                    iterations=1,
                    max_arity=4,
                    threads=num_threads-1,
                    previous_abstractions=i)

    # Update the latest programs
    latest_programs = res.rewritten
```

`stitch_functions` is a list of strings, each string being a Stitch expression. After each iteration of compress, `latest_programs` is updated to contain the re-written programs.

In figure 1 the abstractions compression ratio and number of uses are displayed. The blue lines represent the statistic for an abstraction found during that iteration, while the orange lines are cumulative metrics. After 200 iterations, the cumulative compression ratio reached 1.9, meaning that re-writing the original programs with the abstractions (approximately) cut the length in half.

Figure 1. *Abstraction Metrics.*



The left plot illustrates the compression ratio (in blue), the first few abstractions had high compression (>2.5%). The cumulative compression ratio (in orange) represents the total reduction in the programs length. The right plot illustrates (in blue) the number of times an abstraction was used across all programs, and (in orange) the total number of segments that have been re-written using abstractions.

From the top three abstractions found by Stitch, two are the result of how Flatliner represents for loops. They are hard to read and their structure not very interesting per say. However, it suggests that *looping* is an integral part of solving ARC-AGI puzzles. The second abstraction is not about how Flatliner represents anything, but highlights the fundamental aspect of an input-output pair. Abstraction 2 is a high-level specification of a dictionary containing an input and an output. It is difficult to read, so I only include its Python translation: (`x0`, `x1` are arguments to the function)

Abstraction 2

```
{("input", x1), ("output", x0)}
```

Two more abstraction that I found interesting are highlighted below, I only include their Python translations.

Abstraction 14

```
canvas(x2, (x1, x0))
```

Abstraction 14 makes use of the primitive `canvas`. `canvas` is designed to construct grids, where `x2` specifies a color and the tuple `(x1, x0)` represents the dimensions of the grid (x-axis, y-axis).

Abstraction 31

```
floordiv(x0, 2)
```

Abstraction 31 displays that floor division is often performed with the denominator 2. In other words, being able to take an entire half of something is useful to solve puzzles.

4 Conclusions & Future Work

There are many things that can be improved with this pipeline. Almost half of the 400 programs were not properly translated to Stitch expressions. My suspicion is that most of these require a very small fix. For example, I noticed cases in which the Transformations module does not catch expressions that are too deeply nested in a structure. Another possible bug is that the Python-to-Stitch translator does not have access to enough variable names. It takes a list of input variables, currently set to 400, but some Re-arc generators might result in expressions with more than 400 variables, thus causing the translator to fail.

Interpreting the abstractions found by Stitch is a major challenge. After only 40 or so iterations, many abstractions are built using prior abstractions, thus becoming challenging to make sense of. Some number of abstractions might have a direct influence on the grid, thereby allowing themselves to be visualized. Another idea is to create an interactive display, where abstractions can be substituted for their original counter-parts and vice versa. This display should also allow one to translate the Stitch expressions to Python, as that will be much easier to read.

Stitch optimizes an internal cost function to choose the next abstraction. As figure 1 (right plot) shows, the number of uses is not the most important thing. Playing around with the internal cost function could yield abstractions better suited for this challenge.

References

[1] ast — Abstract Syntax Trees. URL: https://docs.python.org/3/library/ast.html.

[2] de Bruijn index, March 2025. Page Version ID: 1280858066. URL: https://en.wikipedia.org/w/index.php?title=De_Bruijn_index&oldid=1280858066.

[3] Guillermo Barbadillo. Omni-ARC. URL: https://ironbar.github.io/arc24/05_Solution_Summary/.

[4] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-Down Synthesis for Library Learning. *Proceedings of the ACM on Programming Languages*, 7(POPL):1182–1213, January 2023. arXiv:2211.16605 [cs]. URL: http://arxiv.org/abs/2211.16605, doi:10.1145/3571234.

[5] Fausto Carcassi. thelogicalgrammar/LOTlib3, June 2023. original-date: 2023-05-16T09:11:23Z. URL: https://github.com/thelogicalgrammar/LOTlib3.

[6] Francois Chollet. OpenAI o3 Breakthrough High Score on ARC-AGI-Pub. URL: https://arcprize.org/blog/oai-o3-pub-breakthrough.

[7] Francois Chollet, Mike Knoop, Gregory Kamradt, and Bryan Landers. ARC Prize 2024: Technical Report, January 2025. arXiv:2412.04604 [cs]. URL: http://arxiv.org/abs/2412.04604, doi:10.48550/arXiv.2412.04604.

[8] François Chollet. ARC Prize - Leaderboard. URL: https://arcprize.org/leaderboard.

[9] François Chollet. On the Measure of Intelligence, November 2019. arXiv:1911.01547 [cs]. URL: http://arxiv.org/abs/1911.01547, doi:10.48550/arXiv.1911.01547.

[10] Daniel Franzen, Jan Disselhoff, and David Hartmann. The LLM ARChitect: Solving ARC-AGI Is A Matter of Perspective.

[11] Michael Hodel. Addressing the Abstraction and Reasoning Corpus via Procedural Example Generation, April 2024. arXiv:2404.07353 [cs]. URL: http://arxiv.org/abs/2404.07353, doi:10.48550/arXiv.2404.07353.

[12] Haocheng Hu. hhc97/flatliner-src, February 2025. original-date: 2022-04-22T02:16:10Z. URL: https://github.com/hhc97/flatliner-src.

[13] Armando Solar-Lezama. *Introduction to Program Synthesis*. MIT Press. URL: https://people.csail.mit.edu/asolar/SynthesisCourse/Lecture1.htm.