EnCompass: Enhancing Agent Programming with Search Over Program Execution Paths

Zhening Li*
Asari AI, MIT CSAIL
zhening.li@asari.ai

Armando Solar-Lezama
Asari AI, MIT CSAIL
asolar@csail.mit.edu

Yisong Yue
Asari AI, Caltech CMS
yisong@asari.ai

Stephan Zheng
Asari AI
stephan@asari.ai

Abstract

We introduce a new approach to *agent programming*, the development of LLM-based agents. Current approaches to agent programming often entangle two aspects of agent design: the core workflow logic and the inference-time strategy (e.g., tree search). We introduce *probabilistic angelic nondeterminism* (PAN), a programming model that disentangles these two concerns, allowing the programmer to describe the agent workflow and independently experiment with different inference-time strategies by simply changing a few inputs. We provide an implementation of PAN in Python as the EnCompass framework, which uses a Python decorator to compile agent workflow programs into a search space. We present three case studies that demonstrate how the framework lets the programmer quickly improve the reliability of an agent and easily switch between different inference-time strategies, all with little additional coding.

1 Introduction

Recent work has shown the power of scaling inference-time compute for LLMs [1, 2], where popular strategies include best-of-N sampling [3, 4, 5], refinement [6, 7], and tree search [8, 9]. In LLM-based agents — systems that define how LLMs and other components interact to solve a task — these same strategies have become common ways of improving performance and reliability. Furthermore, several works have demonstrated the utility of applying sophisticated search and backtracking strategies in AI agents to improve performance in various tasks [8, 10, 11, 12, 13, 14].

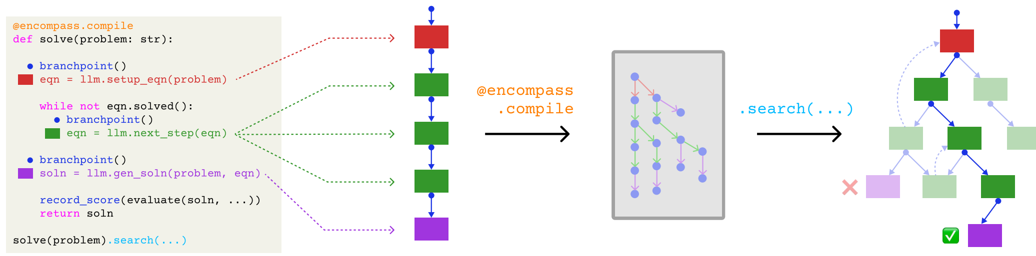
While various frameworks have been developed to simplify the low-level interaction between the program and the LLM [15, 16, 17, 18], a framework for agent inference-time strategies has been absent. Our goal is to develop an *inference-time strategy framework*: a framework that makes it easy to experiment with different inference-time strategies independently of the design and implementation of the underlying agent workflow. Such a framework is intended not to replace, but to be used in conjunction with LLM prompting and tool use frameworks, such as LangChain [15] or DSPy [16].

We target "program-in-control" style agents, where one defines the workflow in code and uses the LLM to accomplish specific subtasks [14, 19, 20, 21, 22].[2] In these agents, inference-time strategies have traditionally been limited to sampling and refinement loops [6, 7, 19, 20], whereas more sophisticated strategies such as beam search and tree search have been rarely explored [14].

*Work performed as a consultant for Asari AI

[2]This "program-in-control" style contrasts with the "LLM-in-control" style where the LLM decides the full sequence of operations (tool calls) in the workflow [8, 9, 10, 11, 12, 13].

39th Conference on Neural Information Processing Systems (NeurIPS 2025).



(a) EnCompass Python code **(b)** Underlying agent workflow **(c)** Compiled search space **(d)** Overlaying search

Figure 1: An EnCompass program specifies an agent workflow, which is compiled into a search space object, and inference-time scaling is accomplished through search over the nondeterministic execution paths of the agent workflow.

We identify the key bottleneck to be the entanglement of the inference-time scaling strategy with the core workflow logic when programming the agent. Programmers typically bake the inference-time strategy into the agent workflow [14, 19, 20, 22], which is inflexible, reduces readability, and limits the kinds of inference-time strategies that can be easily implemented. Therefore, we aim to design a framework that cleanly separates the representation of the core workflow logic from the inference-time scaling strategy. The programmer could then make minimal modifications to their agent to flexibly experiment with different inference-time strategies. Also, different agents would no longer require custom implementations of the same inference-time strategy, but can instead reuse a common implementation.

Our key insight is that inference-time strategies can be viewed as instances of *search over different execution paths of a nondeterministic program*. We developed the EnCompass Python programming framework ("**en**hancing agents with **comp**iled **a**gent **s**earch"), depicted in Figure 1. Figure 1a and Figure 1b show an agent program and its corresponding workflow, respectively. The user specifies the "locations of unreliability" in their agent source code using `branchpoint()` statements. A location of unreliability is an operation such as an LLM call where repeated invocations produce outputs of varying quality. Since these different outputs give rise to multiple possible futures of the program's execution, the program has a tree of possible execution paths. EnCompass compiles the program into a search space object (Figure 1c) so that search can be conducted over this tree of execution paths to find the path with the highest score (Figure 1d). We call this programming model *probabilistic angelic nondeterminism* (PAN). As a form of angelic nondeterminism [23], PAN lets the programmer write their program pretending the unreliable operations always produce good outputs, and the runtime searches the space of possible execution paths for one where the operations indeed produced good outputs.

Our work makes the following concrete contributions:

- We introduce the PAN programming model (Section 2.1), which uses angelic nondeterminism to separate inference-time algorithms (search policy) from the underlying logic of the agent (specification of the search space).
- We present EnCompass, a Python library that implements PAN (Section 2.2), providing 1. primitives like `branchpoint()` that the programmer can use inside their EnCompass function, 2. a Python function decorator that compiles an EnCompass function into a search space object at run-time, and 3. common search algorithms, as well as an interface for implementing custom search algorithms.
- We illustrate how EnCompass provides a unifying framework for common inference-time strategies and agentic patterns, which are special cases of search over nondeterministic execution paths of EnCompass programs (Section 3). EnCompass also provides a natural generalization of these inference-time strategies.
- We present case studies showing how EnCompass enables easy experimentation of various inference-time search strategies over an underlying agent workflow, allowing one to quickly identify the best-performing strategy (Section 4). EnCompass opens up new possibilities for inference-time scaling of program-in-control style agents, where inference-time strategies that were previously considered too cumbersome to implement are now made possible by EnCompass.

2 EnCompass, a Python framework for PAN

In this section, we introduce the PAN programming model (Section 2.1) and describe its Python implementation in the EnCompass framework (Section 2.2). For simplicity, we will ignore the feature of memory sharing across different program execution paths (see Section 3.2). The documentation for EnCompass is in Appendix B and the EnCompass compiler is described in Appendix C.

2.1 Probabilistic angelic nondeterminism (PAN)

The core idea of PAN is to search over the tree of possible execution paths of a probabilistic program — where some operations (e.g., LLM calls) have randomness — to find the path that optimizes a user-specified objective. Given a probabilistic program with branchpoints at certain locations in the program, we model its computation as a Markov chain over the space of possible program states. The Markov chain consists of the following components:

- Branchpoints and the end of the program constitute a set of *marked locations* in the program. In Figure 1, branchpoints are denoted by blue dots •.
- A *program state* is a pair consisting of a marked location of the program and a *memory state*, which is a mapping from variables to values.
- The code that executes from one marked location to the next defines a *probabilistic transition function* that maps from a program state to the next program state. Program states at the end of the program are *final*, i.e., they have no next states. In Figure 1, transitions are denoted by colored boxes ■■■.
- The *initial state* is the program state resulting from executing the program from the start until hitting the first branchpoint.

Normally, executing the probabilistic program results in one sampled trajectory of program states (Figure 1b). In PAN, however, we *search* over the space of possible trajectories (Figure 1d). Our search tree initially has just one node: the initial program state. At every step, the search policy chooses a node in the current search tree, makes a copy of the program state stored at that node, samples a next program state according to the probabilistic transition function, and adds it to the search tree as a child of that node. The goal is to reach a final program state that optimizes a user-specified objective.

Note that search here is formulated differently from the usual graph search formulation because we don't have access to all the children of any given node — we can only stochastically *sample* children of a parent node. However, existing graph search algorithms can be converted to algorithms in PAN by specifying each node's *branching factor*, i.e., the number of children to sample. For example, depth-first search (DFS) with branching factor 3 involves sampling 3 next states from the current state and recursing on each child.

This way of adapting graph search algorithms is currently the dominant approach in LLM-based agents that have tree search with an unenumerable action space of LLM outputs [8, 13, 24, 25]. However, we believe it is worth exploring search strategies beyond fixing the branching factor in an existing graph search algorithm, and Case Study 3 (Appendix A.2) explores this direction by showing that a simple strategy — repeatedly choosing the highest-scoring program state and sampling one next state — can work quite well.

2.2 EnCompass

The EnCompass framework provides an instantiation of the PAN programming model in Python. It is implemented as the `@encompass.compile` function decorator, which makes several new primitive keywords available in the body of the decorated function; the full list is given in Appendix B.1. The decorator compiles the function body into a search space object, which provides an interface for implementing search algorithms (Appendices B.2 and B.3). The compiler is described in Appendix C.

Core primitives The two most important primitives that are available in the body of an EnCompass-decorated function are `branchpoint()` and `record_score()`.

`branchpoint(**branchpoint_params)`

This statement marks a PAN branchpoint (Section 2.1), a location in the program where the program state is added as a new node in the search tree and the program's execution may branch into multiple execution paths.

Branchpoint parameters provide information to the external search algorithm about the branchpoint. For example, `branchpoint(name="foo")` gives the branchpoint a name that can be used to refer to the branchpoint in the search algorithm.

`record_score(score)`

This records the numerical "score" used to guide the search process in many search algorithms (e.g., the heuristic in best-first search and value function in MCTS). Furthermore, the final score (the last score recorded before returning) usually specifies the final evaluation score to be maximized by the search algorithm.

Inference-time search Having defined an EnCompass-decorated function `func`, the programmer can now apply search over its nondeterministic execution paths by calling

`func(...).search(algo, **search_config)`

where `algo` is a string such as `"dfs"` or `"beam"` specifying the search algorithm. This returns the function's return value on the best execution path that search algorithm `algo` could find. Appendix B.4 lists all algorithms that EnCompass provides out-of-the-box.

Custom search algorithms The user can also define and register their custom search algorithm so that it can be invoked through the same search interface. The `Checkpoint` class wraps the program state and provides an interface for implementing custom search algorithms. Its `step()` method samples a next program state: it resumes execution of the program from the current state until hitting the next branchpoint or a return statement, returning the new program state (cf. the probabilistic transition function from Section 2.1). The `Checkpoint` object's `score` attribute contains the score of the program state as recorded through `record_score()`. See Appendix B.3 for more details.

3 Agent inference-time strategies in EnCompass

While EnCompass appears most suitable for implementing tree search in agents, other common inference-time strategies can also be cleanly implemented as search in EnCompass. Furthermore, natural generalizations of these strategies that are otherwise difficult to implement are also easily represented in EnCompass.

3.1 Best-of-N sampling and beam search

Given an agent `agent_forward(...)` and an evaluator `evaluate(...)` that evaluates the output of the agent, *best-of-N* (BoN) samples N times and chooses the output with the highest evaluation score. In EnCompass, this is done by adding a branchpoint at the beginning of the function and recording the evaluation score at the end:

```
@encompass.compile
def agent_forward(...):
    branchpoint()
    ...  # Original body of agent
    record_score(evaluate(result))
    return result

result = agent_forward(...).search(...)
```

This defines a search tree with depth 1, where almost any search algorithm would sample several children from the root node and return the best child, thus reproducing best-of-N sampling.

We call the above *global best-of-N* (GBoN) to contrast it with *local best-of-N* (LBoN), where an agent with multiple verifiable steps has best-of-N sampling applied to each of them. In EnCompass, this is implemented by adding `branchpoint()` before each step and applying beam search with beam width 1:

```python
@encompass.compile
def agent_forward(...):
    branchpoint()
    ...  # Step 1
    record_score(evaluate_step1(...))
    branchpoint()
    ...  # Step 2
    record_score(evaluate_step2(...))
    ...
    branchpoint()
    ...  # Step k
    record_score(evaluate_stepk(...))
    return stepk_result

N = ...  # the "N" in best-of-N
result = agent_forward(...).search("beam", beam_width=1, default_branching=N)
```

Note that the two types of best-of-N sampling described in this section—*global* and *local* sampling—are the two limiting cases of beam search. Global best-of-N sampling is beam search with beam width N and branching factor 1,[3] whereas local best-of-N sampling is beam search with beam width 1 and branching factor N. General beam search can thus be viewed as interpolating between global and local resampling. This has the benefit of effectively constraining the search space with local verification while also not losing global variety. Increasing the branching factor makes sure each step is completed correctly to help prevent compounding errors, while increasing the beam width can help increase variety and thus improve reliability to mitigate potential errors made in earlier steps. In Case Study 1 (Section 4.1), we empirically demonstrate that beam search indeed scales better than global best-of-N or local best-of-N alone in complex agent workflows.

The EnCompass implementation of beam search over an agent workflow also benefits from flexibility in modifying the step granularity. Increasing the granularity (dividing steps up into smaller substeps) or decreasing the granularity (merging multiple steps into one) is as simple as adding or removing branchpoints in EnCompass, whereas a plain Python implementation would require structural changes to the code.

3.2 Refinement and backtracking with memory

Refinement can be viewed as sampling but with additional feedback from past sampling attempts. In EnCompass, this is accomplished by adding a branchpoint to generate multiple samples and a memory of past attempts shared across the different sampled execution paths:

```python
@encompass.compile
def agent_forward(...):
    ... # stuff that comes before
    # Step with refinement
    feedbacks: NoCopy = []
    branchpoint()
    result = do_step(..., feedbacks)
    score, feedback = get_score_and_feedback(result)
    feedbacks.append(feedback)
    record_score(score)
    ... # stuff that comes after

result = agent_forward(...).search("beam", beam_width=1, default_branching=n_refine + 1)
```

Here, the `NoCopy` type annotation tells the EnCompass compiler that the different execution paths should share the same reference to the `feedbacks` variable, so that appending `feedback` is seen across all branches.[4]

By adding another `branchpoint()` right before "`feedbacks: NoCopy = []`", we create multiple parallel refinement loops, thus interpolating between fresh sampling and refinement and maintaining variety that may otherwise be lost from an agent that focuses too heavily on the past feedback. This is not unlike how beam search interpolates between global best-of-N and local best-of-N (Section 3.1). In Case Study 3 (Appendix A.2), we demonstrate how a different approach to interpolating between refinement and sampling — by adding branchpoints to a refinement loop written in plain Python — can result in better scaling than refinement alone.

[3] Except that the root node has branching factor N.

[4] This effect is lost if `feedbacks.append(feedback)` is replaced with `feedbacks = feedbacks + [feedback]`, since that creates a new list instead of modifying the original one.

Note that refinement is the simplest case of *backtracking with memory*: backtracking to a previous step while remembering what happened in previous attempts. In EnCompass, the general pattern for backtracking with memory is to create a shared mutable data structure right before a branchpoint, which serves as a memory shared across all execution paths that follow.

3.3 Self-consistency and group evaluation

Given an agent program `agent_forward(input)`, *self-consistency* samples N times and chooses the output that appeared the most times (the majority vote) [26]. This can be implemented as best-of-N sampling with an evaluation function that evaluates a group of results at once. The EnCompass `record_score()` supports this:

```python
def majority_vote(results):
    counts = defaultdict(int)
    for result in results:
        counts[result] += 1
    return [
        counts[result] for result in results
    ]

```

```python
@encompass.compile
def agent_forward(...):
    branchpoint()
    result = ...
    record_score(majority_vote, result, label=None)
    return result

result = agent_forward(...).search(...)
```

In general, allowing the evaluation function to evaluate a group of results at once is helpful when it is difficult to evaluate one result on its own. Another example of this is CodeT [27], which evaluates a group of LLM-generated code samples against multiple LLM-generated unit test cases by considering both the number of unit test pass rate and agreement among code samples on which test cases they pass.

Inference-time strategies like self-consistency and CodeT are examples of the more general *search with evaluation of a group of execution paths in tandem*. When one writes

`record_score(`*`group_evaluator`*`, `*`evaluation_target`*`, label=`*`group_label`*`)`

the scores of all program states where `record_score()` was called with label *`group_label`* are computed as *`group_evaluator`*`(`*`evaluation_targets`*`)`, where *`evaluation_targets`* is the list of the *`evaluation_target`* variables across all the program states.

4 Case studies

We implemented and extended 3 program-in-control style agents from the literature in EnCompass. These case studies aim to answer the following research questions:

- Does EnCompass make it easier to implement inference-time strategies and search in program-in-control style agents, and if so, how?
- Does EnCompass simplify experimenting with different inference-time strategies and search in program-in-control style agents, and if so, how?

Our case studies suggest that EnCompass enables the exploration of inference-time strategies that are otherwise left unexplored due to their complexity of implementation — potentially unlocking better scaling laws.

Case Study 1 is our main case study and is presented in the main text here. Case Studies 2 and 3 are smaller and more didactic in purpose, and are presented in Appendix A.

In **Case Study 1** (Section 4.1), we implement a Java-to-Python code repository translation agent with a high-level architecture based on that of Syzygy [28]. We then add branchpoints before LLM calls and, by toggling a few parameters, we experiment with a variety of search strategies including local/global best-of-N sampling and beam search at the file level and individual method level. We demonstrate these experiments on Java repositories from the MIT OCW Software Construction class. We find that beam search outperforms simpler sampling strategies, thus demonstrating how one can use EnCompass to discover better inference-time scaling laws. Furthermore, we show how the equivalent plain Python implementation of the EnCompass agent involves defining the search graph as a state machine, where the agent workflow is significantly obscured and modularity is compromised, whereas EnCompass solves these issues.

Table 1: Code modifications to implement search in our case studies, without EnCompass vs. with EnCompass. Metrics include the number lines/words added, changed[a], and removed, the number of new function definitions, and the number of lines of the original code where the indentation level was changed. For context, we also give the number of lines of code used to implement the core logic[b] of the original base agent. All code is found in Appendix D with the modifications annotated.
[a] This excludes changes to the indentation level of existing code. [b] The "core logic" is defined as the functions that require modification when implementing search, hence excluding unmodified code like helper/utility functions and prompt templates.

Case Study		**Added** lines (words)	**Changed** lines (words)	**Removed** lines (words)	**New f'ns**	**Indent changed**
1. Code Repo Translation	−EnCompass	+423 (+2735)	24 (-62/+186)	-9 (-28)	+20	189
LoC = 597	+EnCompass	**+75 (+514)**	**8 (-0/+40)**	**-0 (-0)**	**+1**	**0**
2. Hypothesis Search	−EnCompass	+21 (+120)	3 (-1/+13)	-0 (-0)	+2	10
LoC = 11	+EnCompass	**+8 (+27)**	**1 (-0/+9)**	-0 (-0)	**+0**	**0**
3. Reflexion	−EnCompass	+27 (+181)	6 (-13/+31)	-0 (-0)	+2	8
LoC = 20	+EnCompass	**+9 (+32)**	**3 (-4/+13)**	-0 (-0)	**+0**	**0**

In **Case Study 2** (Appendix A.1), we implement a simplified Hypothesis Search agent [19]. We start with a simple agent with two LLM calls. By adding a branchpoint before each LLM call and applying multithreaded BFS out of the box, we reproduce a parallelized version of Hypothesis Search. We demonstrate how to use EnCompass to experiment with different search strategies (BFS vs. global best-of-N), and find that they perform equally well on a subset of the ARC benchmark [29], the benchmark that Hypothesis Search used. We show how, despite the simplicity of the original agent, the equivalent program in plain Python already noticeably obscures the underlying agent workflow.

In **Case Study 3** (Appendix A.2), we start with Reflexion [7], a simple agent with a refinement loop. We add a branchpoint at the beginning of the agent and at the beginning of the body of the refinement loop, and apply both global best-of-N and a variant of best-first search. Following the original Reflexion paper, we evaluate on LeetCodeHard. We find that increasing N in best-of-N or the number of search steps in best-first search scales better than increasing the number of refinement iterations in vanilla Reflexion. We also show how the equivalent program in plain Python obscures the control flow and data flow of the underlying agent.

Table 1 and Appendix D compare the code modifications required to implement search with EnCompass vs. without EnCompass. On average, EnCompass saves 3–6x of coding in terms of the number of lines/words that are added or changed.

Note that since EnCompass targets program-in-control style agents, our case studies do not include benchmarks of LLM-in-control style agents such as SWEBench [30] or WebArena [31].

4.1 Case Study 1: Code Repository Translation Agent

In this case study, we demonstrate how to use EnCompass to add branchpoints and implement search in a Java-to-Python code repository translation agent based on the Syzygy agent architecture [28]. By comparing with the equivalent plain Python implementation, we identify several concrete benefits of the separation of concerns offered by EnCompass. We also demonstrate experimenting with different search strategies on one repository to find the best-performing strategy ("fine-grained" beam search), and we apply this strategy to other repositories to obtain strong performance compared to simpler strategies (global/local best-of-N).

Base agent We built an agent that translates a Java repository into Python (Listing 18). The agent translates the repository file-by-file in dependency order. For each file, the agent calls the LLM to write the skeleton of the Python file, and for each Java method the agent calls the LLM to translate it into Python. Every translation is followed by validation of the translation by 1) asking the LLM to write a script that generates random test case inputs; 2) asking the LLM to write Java code to run the Java method on those inputs; 3) asking the LLM to write Python code to run the translated Python method on those inputs; and 4) comparing the Python and Java outputs to see if they match.

The EnCompass agent In EnCompass, we modify the base agent by adding a branchpoint before each of the 5 LLM calls present in the program (Listing 19). To prevent different branches of the search from overwriting the same folder, we use Git to manage the repository, and write a wrapper `branchpoint_git_commit()` around the built-in `branchpoint()` (Listing 19, L5–15). We consider search at two different levels of the translation workflow: the file level ("coarse"), and the method level ("fine"). By adjusting the search parameters, we experimented with different search strategies at each level as well as different parameters to the search strategies. We applied 6 combinations of search strategies: "global best-of-N", "local best-of-N (coarse)", "local best-of-N (fine)", "beam (coarse)", "local best-of-N (coarse) + beam (fine)", and "beam (coarse) + beam (fine)".

This was as simple as changing a couple of parameters: the file-level search strategy is specified at line 278 of Listing 19 and the method-level search strategy is specified at line 264. Section 3.1 explains the search algorithm and parameters passed to the `.search_multiple(...)` method to implement global BoN, local BoN, and beam search.

Comparison with equivalent plain Python We demonstrated how EnCompass lets an agent programmer easily switch between different search algorithms. To replicate this flexibility in plain Python, we need to explicitly define the search graph that the EnCompass function defines. The search graph takes the form of a state machine where the states correspond to the branchpoints and the transitions follow the control flow of the program. We maintain a dictionary `frame` with all the local variables of the program as we go through the transitions of the state machine. The result is Listing 20, which is long and difficult to read, so we illustrate this with a simplified version of our code repository translation agent that iterates through functions in a source file, translating each of them one-by-one:

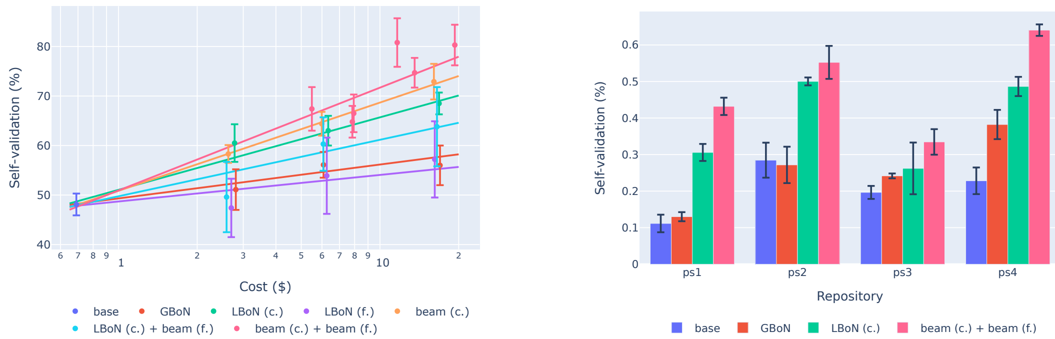
```python
@encompass.compile
def translate_functions(source):
    for source_fn in source:
        branchpoint()
        target_fn = translate(source_fn)
        compile_success = compile_(target_fn)
        record_score(compile_success)

        branchpoint()
        unit_test_score = run_unit_test(target_func)
        record_score(unit_test_score)
```

The equivalent state machine in plain Python is given here with minor simplifications.

```python
class State(Enum):
    TRANSLATE = auto()
    UNIT_TEST = auto()

def step(state: State, frame: dict[str, Any]):
    frame = frame.copy()

    if state == State.TRANSLATE:
        frame["target_fn"] = translate(frame["source_fn"])
        compile_success = compile_(frame["target_fn"])
        return State.UNIT_TEST, frame, compile_success

    if state == State.UNIT_TEST:
        unit_test_score = run_unit_test(frame["target_fn"])
        frame["source_fn"] = next(frame["source"])
        return State.TRANSLATE, frame, unit_test_score
```

Notice that the high-level control flow of "repeatedly translate and unit-test the translation" is no longer obvious from the code; it is difficult to know whether any given variable access `frame[...]` might throw a `KeyError`; and linters and static type checkers can't be applied because variables are accessed through the `frame` dictionary. Furthermore, simple changes to the EnCompass function such as moving or removing a branchpoint would require significant structural changes to the state machine code that further create an opportunity for bugs. All these issues are exacerbated as we increase the complexity of the agent program, so the state machine approach to defining agent search graphs is not scalable. This can be seen in Listing 20 (Appendix D.1), which applies the state machine approach to the original code repository translation agent.

Evaluation setup To make it affordable to run comprehensive experiments comparing the scaling behaviors of various inference-time strategies, we first validated on a small repository consisting of



(a) (b)

Figure 2: Results of using EnCompass to apply different inference-time scaling methods to the code repository translation agent. All error bars show standard errors of the mean over 5 runs. (a) A comprehensive hyperparameter search for `ps0`; (b) For `ps1` to `ps4`, we applying global best-of-N ("GBoN"), file-level local best-of-N ("LBoN (c.)"), and beam search at the file and method level ("beam (c.) + beam (f.)") while controlling for cost.

622 lines of Java code. The repository contains solutions to the first homework (`ps0`) from the Spring 2016 version of the MIT Software Construction class available on MIT OpenCourseWare [32, 33].

Because of the scarcity of test cases in the original repository, we use *self-validation (%)* as the evaluation metric, which is calculated as the percentage match of the Python and Java outputs on the automatically generated test inputs, averaged across all translated non-test methods. If any step of the validation process failed (e.g., test input generation), then the match percentage is considered to be 0.

After identifying the inference-time strategy that scales best on `ps0`, we evaluated it on the other 4 repositories from the class (`ps1` to `ps4`). Each of them contains between 1100 and 1900 lines of code, and all 4 repositories combined contain 5756 lines of code.

For all experiments, we set the LLM temperature to 0.0 for the base agent (no inference-time strategies), and 0.5 for the EnCompass agent (with inference-time strategies).

Evaluation results Figure 2a shows a log-linear plot of the scaling of various inference-time strategies on `ps0`. Consistent with prior work on inference-time scaling [4, 5], we find that performance scales linearly with the logarithm of the cost (all χ^2 p-values > 0.3). The best scaling is achieved with beam search applied at both the file level and the individual method level ("beam (coarse), beam (fine)"), outperforming the second best strategy "beam (coarse)" with a p-value of 0.2 and all other strategies with statistical significance ($p < 0.03$).

Notably, the best-performing strategy ("beam (coarse), beam (fine)") also happens to be the most difficult one to implement in plain Python. It requires the programmer to break up the entire workflow into all the individual LLM-calling steps where each step explicitly stores and retrieves variables from a `frame` dictionary. This finding further demonstrates the merits of having a framework like EnCompass where experimenting with different search strategies can be done via simply changing a few parameters. Combinations of agent and inference-time strategy that have better scaling but that programmers would otherwise choose not to implement due to their complexity of implementation, are now made possible by EnCompass.

We then evaluated the best-performing strategy "beam (coarse), beam (fine)" on `ps1` through `ps4` and compared it with two simpler baselines ("global best-of-N", "local best-of-N") while controlling for cost. For beam search, we used a file-level beam width of 2 and a method-level beam width of 3, whereas we used $N = 16$ for both global and local best-of-N. The average cost of a run was \$20–\$20.5 for `ps1`, \$27–\$30 for `ps2`, \$36–\$39 for `ps3`, and \$13.5–\$14 for `ps4`. The results are shown in Figure 2b. Overall, "beam (coarse), beam (fine)" continues to outperform the other two simpler strategies.

To conclude, we have demonstrated the advantages of the separation of concerns offered by EnCompass. Implementing an inference-time strategy in EnCompass mainly involves adding branchpoints before LLM calls, whereas without EnCompass, significant source code modification that obscures the underlying workflow is often necessary. Furthermore, experimenting with different inference-time strategies in EnCompass is often as simple as changing a few search parameters.

5 Related work

Inference-time strategies for LLMs and agents [2] provides a comprehensive review of algorithms used during LLM inference to improve its reliability and performance. Examples include best-of-N sampling [3, 4, 5], refinement [6, 7], self-consistency [26], and tree search [8, 9, 13], which are also commonly used in LLM-based agents [10, 11, 12, 14]. Section 3 demonstrates how EnCompass unifies and generalizes these inference-time scaling strategies for agents.

AI agent frameworks Several LLM-based agent frameworks have been developed to abstract away boilerplate code and other low-level concerns, and provide abstractions for common agentic patterns and components. AutoGen [18] simplifies multi-agent conversation workflows with tool use, LangChain [15] simplifies linear workflows with RAG and tool use, LangGraph [34] simplifies the creation of agent workflows as state machines, and DSPy [16] automates prompt engineering. Complementary to these efforts, our framework, EnCompass, simplifies applying inference-time scaling strategies to agents. Since EnCompass involves adding statements such as `branchpoint()` to an existing agent written in Python, it can be flexibly incorporated into agents built with an existing Python agent framework.

Angelic nondeterminism Previous implementations of angelic nondeterminism include John McCarthy's `amb` operator in Common Lisp [35] and the list monad in Haskell [36]. The main conceptual difference is that EnCompass implements a *probabilistic* form of angelic nondeterminism, which samples from a probability distribution such as an LLM instead of choosing from a given set of choices.

Probabilistic programming Our work is also inspired by *probabilistic programming*, a programming paradigm that separates the two main concerns of probabilistic inference: specifying the probabilistic model and implementing the inference algorithm. (See, e.g., [37] for a review.) This allows the programmer to efficiently specify a probabilistic model in code while independently experiment with different probabilistic inference algorithms. Similarly, EnCompass aims to separate the two main concerns of agent programming: specifying the core agent workflow and implementing the inference-time search strategy.

6 Limitations

EnCompass targets program-in-control style agents, where implementations without EnCompass typically force the programmer to entangle the underlying agent and the overlaying search strategy. EnCompass is not meant for LLM-in-control style agents, where the two aspects are already decoupled. Nevertheless, there has been increased interest in "LLM+program-in-control" hybrid style agents which involve an LLM writing a program-in-control style agent [38, 39, 40]. It would be interesting to explore using EnCompass to make it easier for the LLM to implement inference-time strategies in LLM-calling programs that it writes.

Although EnCompass simplifies the source code modifications needed to apply inference-time strategies to an existing agent, modifications are still needed. There remains the engineering challenge of choosing the correct places to add branchpoints, adding sufficient and good-quality intermediate reward/verification signal, and designing a good search algorithm. EnCompass could be improved to eliminate the need for source code modifications entirely, where it solves the the majority of these remaining challenges by potentially using a flexible LLM-based search strategy.

7 Conclusion

This work introduced the EnCompass programming framework, which decouples the two fundamental aspects of agent programming: defining the core agent workflow and designing the inference-time scaling strategy. By enabling the integration of sophisticated search strategies into complex agent workflows, EnCompass opens up new possibilities for inference-time scaling of AI agents. Looking ahead, we anticipate that the ability to seamlessly combine agent workflows with powerful search techniques — enabled by EnCompass — will unlock new scaling laws and drive the development of reliable LLM-augmented systems for solving complex real-world tasks.

References

[1] Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025.

[2] Sean Welleck, Amanda Bertsch, Matthew Finlayson, Hailey Schoelkopf, Alex Xie, Graham Neubig, Ilia Kulikov, and Zaid Harchaoui. From decoding to meta-generation: Inference-time algorithms for large language models. *Transactions on Machine Learning Research*, November 2024.

[3] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[4] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

[5] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

[6] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-Refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

[7] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2024.

[8] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of Thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[9] Yuxi Xie, Kenji Kawaguchi, Yiran Zhao, James Xu Zhao, Min-Yen Kan, Junxian He, and Michael Xie. Self-evaluation guided beam search for reasoning. *Advances in Neural Information Processing Systems*, 36, 2024.

[10] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language Agent Tree Search unifies reasoning, acting, and planning in language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 62138–62160. PMLR, 21–27 Jul 2024.

[11] Jing Yu Koh, Stephen McAleer, Daniel Fried, and Ruslan Salakhutdinov. Tree search for language model agents. *arXiv preprint arXiv:2407.01476*, 2024.

[12] Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Yang Wang. SWE-Search: Enhancing software agents with monte carlo tree search and iterative refinement. In *The Thirteenth International Conference on Learning Representations*, 2025.

[13] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[14] Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob Foerster, Jeff Clune, and David Ha. The AI Scientist-v2: Workshop-level automated scientific discovery via agentic tree search. *arXiv preprint arXiv:2504.08066*, 2025.

[15] Harrison Chase, Bagatur Askaryan, and Erick Friis. LangChain 0.3.16, 2025.

[16] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan A, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*, 2024.

[17] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1946–1969, 2023.

[18] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024.

[19] Ruocheng Wang, Eric Zelikman, Gabriel Poesia, Yewen Pu, Nick Haber, and Noah Goodman. Hypothesis search: Inductive reasoning with language models. In *The Twelfth International Conference on Learning Representations*, 2023.

[20] Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. AlphaTrans: A neuro-symbolic compositional approach for repository-level code translation and validation. *Proceedings of the ACM on Software Engineering*, 2(FSE):2454–2476, 2025.

[21] Pei Zhou, Jay Pujara, Xiang Ren, Xinyun Chen, Heng-Tze Cheng, Quoc V Le, Ed Chi, Denny Zhou, Swaroop Mishra, and Huaixiu Steven Zheng. Self-Discover: Large language models self-compose reasoning structures. *Advances in Neural Information Processing Systems*, 37:126032–126058, 2024.

[22] Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The AI Scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.

[23] Robert W Floyd. Nondeterministic algorithms. *Journal of the ACM (JACM)*, 14(4):636–644, 1967.

[24] Di Zhang, Xiaoshui Huang, Dongzhan Zhou, Yuqiang Li, and Wanli Ouyang. Accessing GPT-4 level mathematical Olympiad solutions via Monte Carlo Tree Self-Refine with Llama-3 8B. *arXiv preprint arXiv:2406.07394*, 2024.

[25] Ziyu Wan, Xidong Feng, Muning Wen, Stephen Marcus Mcaleer, Ying Wen, Weinan Zhang, and Jun Wang. AlphaZero-like tree-search can guide large language model decoding and training. In *International Conference on Machine Learning*, pages 49890–49920. PMLR, 2024.

[26] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023.

[27] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*, 2023.

[28] Manish Shetty, Naman Jain, Adwait Godbole, Sanjit A Seshia, and Koushik Sen. Syzygy: Dual code-test C to (safe) Rust translation using LLMs and dynamic analysis. *arXiv preprint arXiv:2412.14234*, 2024.

[29] François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

[30] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

[31] Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. WebArena: A realistic web environment for building autonomous agents. In *The Twelfth International Conference on Learning Representations*, 2024.

[32] Denis Savenkov. `github.com/FizzyBubblech/MIT-6.005`, 2017.

[33] Robert Miller and Max Goldman. 6.005 — Software Construction on MIT OpenCourseWare: Problem Set 0: Turtle Graphics, 2016.

[34] Nuno Campos, Barda Vadym, and William F Hinthorn. LangGraph 0.2.68, 2025.

[35] John McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*. North-Holland, 1963.

[36] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24–30, 1995 Tutorial Text 1*, pages 24–52. Springer, 1995.

[37] Noah D Goodman. The principles and practice of probabilistic programming. *ACM SIGPLAN Notices*, 48(1):399–402, 2013.

[38] Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. Chain of code: Reasoning with a language model-augmented code emulator. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 28259–28277. PMLR, 21–27 Jul 2024.

[39] Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations*, 2025.

[40] Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. Gödel agent: A self-referential agent framework for recursively self-improvement. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 27890–27913, Vienna, Austria, July 2025. Association for Computational Linguistics.

[41] Cormac Flanagan, Amr Sabry, Bruce F Duba, and Matthias Felleisen. The essence of compiling with continuations. *ACM Sigplan Notices*, 28(6):237–247, 1993.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract summarizes the problem and the proposed solution; the introduction defines the scope, identifies the problem, and summarizes the proposed solution and contributions.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: See Limitations section.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [NA]

Justification: The paper proposes a programming framework and makes no theoretical claims.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Compiler details are in Appendix C. Case studies describe experimental setup. Agent programs of case studies are in Appendix D.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [No]

Justification: Company code

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: See case study sections.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: See case study sections, including tables and figures.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)

- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Case study sections describe LLM costs and CPU used.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: No aspect of the research violated the Code of Ethics.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: Foundational research with standard expectations in regards to societal impact

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: No release of data/models with a high risk for misuse

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: Existing agents and benchmarks cited and license terms followed

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.

- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [NA]

    Justification: No new assets released.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: No human subjects involved.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: No human subjects involved.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

Answer: [NA]

Justification: LLMs were not used to help produce this research other than as a coding and writing tool.

Guidelines:

- The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.

A Additional case studies

This appendix presents Case Studies 2 and 3. In these case studies, we study agents much simpler than the code translation agent in our main case study (Case Study 1) so that we can more explicitly compare code written in EnCompass vs. plain Python. Our objective is to illustrate and understand how the modularity that EnCompass provides lets programmers more easily implement and experiment with different inference-time scaling strategies for their agent.

Experiments for all case studies were conducted on a Macbook Pro with an M3 chip and 18 GB of RAM. All LLM calls were made through the OpenAI API.

A.1 Case Study 2: Hypothesis Search Agent

In this case study, we use a simple two-step agent for ARC-AGI [29] to illustrate how EnCompass enables the programmer to quickly implement inference-time search.

Base agent A task in ARC-AGI shows the agent around 3 validation examples of input-output grid pairs, and the objective is to find the rule that transforms input grids into output grids and apply the rule to a test input grid. A simple agent for solving ARC-AGI tasks is as follows (Listing 1): 1. ask the LLM for a natural language hypothesis of the transformation rule; 2. ask the LLM to implement the hypothesis in code.

```python
def two_step_agent(task_info):
    # Step 1: Get natural language hypothesis
    ...
    hypothesis = hypothesis_agent([task_info], hypothesis_instruction)

    # Step 2: Implement the hypothesis in code
    ...
    code = solver_agent([task_info, hypothesis], solver_instruction)
    return get_test_output(code)
```

Listing 1: Simple 2-step agent for ARC

The EnCompass agent To convert this agent into a EnCompass program, we identify the points of unreliability: the two LLM calls. Before each LLM call, we can add a branchpoint to allow the external search algorithm to search over different samples from the LLM. Finally, we add a final verification step that evaluates the generated code on the validation grid pairs, so that the search algorithm knows which execution paths did better. Here is the resulting EnCompass agent (Listing 2):

```python
@encompass.compile
def two_step_agent(task_info):
    # 1st branchpoint results in multiple samples of the natural language hypothesis
    branchpoint()
    # Step 1: Get natural language hypothesis
    ...
    hypothesis = hypothesis_agent([task_info], hypothesis_instruction)

    # 2nd branchpoint results in multiple code samples for each hypothesis
    branchpoint()
    # Step 2: Implement the hypothesis in code
    ...
    code = solver_agent([task_info, hypothesis], solver_instruction)

    # Evaluate
    percent_correct, feedback = run_validation(code)
    record_score(n_correct)
    if percent_correct == 1.0:
        early_stop_search()

    return get_test_output(code)

```

```python
two_step_agent(task_info).search("parallel_bfs", default_branching=8)
```

Listing 2: Two-step agent with BFS in ENCOMPASS reproduces Hypothesis Search

Here, we've chosen 8 samples of subsequent execution from each branchpoint and apply parallelized breadth-first search (parallel BFS) over all program execution paths, In particular, BFS samples 8 natural language hypotheses following the first branchpoint, and for each hypothesis samples 8 code implementations from the second branchpoint. It then chooses the result from the 64 implementations with the highest evaluation score (recorded by `record_score`). This replicates a version of Hypothesis Search [19] without the hypothesis summarization step and execution feedback loop.

We also consider an agent with only the first of the two branchpoints. This gives rise to global best-of-N sampling, i.e., running the base agent N times in parallel and keeping the best run.

Comparison with equivalent plain Python In implementing the ENCOMPASS agent, because the changes made to the original agent are minimal, the underlying logic of the agent is clearly portrayed by the code, with the external search logic (sampling) indicated by a few branchpoint statements.

We now compare this with the equivalent agent in plain Python. For the one-branchpoint hypothesis search agent (best-of-N), it is still relatively straightforward to implement it in plain Python by running N copies of the agent in N parallel threads until we find a solution that passes validation.

However, to further add the second branchpoint — which is just an additional line of code in EnCompass — the equivalent implementation in plain Python of parallel BFS requires significant structural changes. In defining the tasks to be executed in a multithreaded fashion, the underlying agent workflow has been broken up and the program flow obscured, even though the agent only contains two steps (Listing 3).

```python
from concurrent.futures import ThreadPoolExecutor, as_completed


def two_step_agent(task_info, branching):
    results = []
    full_solved = False

    with ThreadPoolExecutor() as executor:

        def run_one_forward_pass():
            if full_solved:
                return
            # Step 1: Get natural language hypothesis
            ...
            hypothesis = hypothesis_agent([task_info], hypothesis_instruction)

            def implement_in_code():
                nonlocal full_solved

                if full_solved:
                    return

                # Step 2: Implement the hypothesis in code
                ...
                code = solver_agent([task_info, hypothesis], solver_instruction)

                # Evaluate
                percent_correct = run_validation(code)
                if percent_correct == 1:
                    full_solved = True
                results.append((get_test_output(code), percent_correct))

            futures = [executor.submit(implement_in_code) for _ in range(branching)]
            for future in as_completed(futures):
                future.result()

        futures = [executor.submit(run_one_forward_pass) for _ in range(branching)]
        for future in as_completed(futures):
            future.result()

    return max(results, key=lambda x: x[1])[0]


two_step_agent(task_info, branching=8)
```

Listing 3: Parallelized BFS in plain Python, obscuring the underlying two-step agent workflow

Table 2: Percentage accuracy of a simple two-step agent on a subset of ARC with progressively more EnCompass branchpoints: no branchpoints, 1 branchpoint at the top, and 2 branchpoints before the 2 LLM calls. Accuracy improves quickly as more branchpoints are added. We also compare with the best agent discovered through meta-agent search (ADAS [39]).

Base model	GPT-3.5		GPT-4o	
	Acc. (%)	Total cost	Acc. (%)	Total cost
Two-step agent	4.3 ± 0.9	\$0.41	24.0 ± 1.5	\$2.85
+ global best-of-N, $N = 8$ (ours)	11.7 ± 0.8	\$3.29	36.3 ± 1.1	\$22.76
+ global best-of-N, $N = 36$ (ours)	$\mathbf{16.0} \pm 1.0$	\$14.81	$\mathbf{38.7} \pm 1.1$	\$95.98
+ BFS, branching $= 8$ (ours)	$\mathbf{15.0} \pm 0.9$	\$15.81	$\mathbf{38.3} \pm 1.2$	\$88.69
ADAS best agent (reported)†	13.7 ± 2.0	—	30.0 ± 2.6	—
ADAS best agent (reproduced)	10.7 ± 0.8	\$2.11	32.7 ± 1.1	\$27.85

†The reported results use a different checkpoint of GPT-4o and the errors are estimated differently, using a bootstrapping confidence interval.

Evaluation The purpose of this evaluation section is to complete our demonstration of using EnCompass to implement and compare different inference-time scaling strategies.

We use a subset of the ARC-AGI benchmark corresponding to the 60 tasks sampled from the "Public Training Set (Easy)" that ADAS [39] used. We report the mean evaluation score as well as its standard error over 5 seeds.

We evaluate the following agents on this ARC-AGI subset:

- The two-step agent (base agent)
- Global best-of-N applied to the two-step agent (one branchpoint), where $N = 8, 36$
- The Hypothesis Search agent [19], i.e., parallelized BFS applied to the two-step agent (two branchpoints), with branching factor 8.

The LLM temperature was set to 0.8 for all experiments.

The evaluation results are shown in Table 2. The results show how scaling inference-time compute by adding `branchpoint()` statements and adjusting search parameters quickly increases the evaluation accuracy to results better than the best agent discovered by costly meta-agent search (ADAS) [39]. Comparing the two scaling strategies, we find that best-of-N and BFS are comparable.

A.2 Case Study 3: Reflexion Agent

In this case study, we show how applying EnCompass to an existing agentic pattern provides a new dimension for the cost-efficient scaling of inference-time compute.

Base agent As our baseline, we use Reflexion [7] as a coding agent (Listing 24), which uses an LLM to iteratively reflect on past attempts and their feedback to improve the response. Feedback includes both LLM-generated self-reflection and results from running LLM-generated unit tests.

The EnCompass agent In EnCompass, we modify Reflexion by adding two branchpoints (Listing 25): one before the initial code generation, and one at the top of the body of the for loop (i.e., before each iteration of self-reflection plus generation). Pass rate on the LLM-generated unit tests feedback is used as the verification score in `record_score()` in the EnCompass agent. We apply two search strategies: one is global best-of-N, and the other one is "reexpand best-first search", our variant of best-first search (BeFS) where the strategy is to simply always choose the node with the highest verification score to step.

Comparison with equivalent plain Python Implementing best-of-N sampling in plain Python is straightforward — simply wrap the agent in a for loop. However, to implement reexpand best-first search in the Reflexion agent, a plain Python implementation requires structural changes to the code when EnCompass only requires adding two branchpoints. In particular, the initial sampling step and the self-reflection step are put into separate functions, corresponding to the 2 actions that the agent is allowed to take (Listing 26 in Appendix D.3). The agent maintains a search tree and iteratively chooses the best node to expand: if the chosen node is the root node, then a new code sample is drawn from the LLM, whereas if the node is not the root, then a self-reflection step is applied to it.

Furthermore, separating the two actions into separate functions loses the natural logical ordering between them (the initial sampling step should occur before the self-reflection step). For more complex agent workflows like the code repository translation agent in Section 4.1, the original underlying agent workflow becomes heavily obscured.

Evaluation The purpose of this evaluation section is to complete our demonstration of using EnCompass to implement and compare different inference-time scaling strategies.

LeetCode is a website with programming exercises to help prepare for software engineer interviews, and the LeetCodeHard benchmark is a collection of 40 hard LeetCode problems [7]. A problem typically has a few dozen test cases (occasionally a few hundred or over a thousand test cases). While the LLM agent does not see these test cases, it can use LLM-generated test cases. We calculate the evaluation score as the average pass rate over all 40 problems, where the pass rate for any given problem is the fraction of test cases passed.

For both the base agent and the EnCompass BeFS agent, we consider 3 different cost settings (low, medium, high) where the number of code generations $n = 5, 8, 13$. In the base agent, we vary the number of feedback loops to be $4, 7, 12$, whereas for the BeFS agent, the number of feedback loops is fixed at 4 but the total number of code generations is controlled by the external search algorithm algorithm. In the best-of-N agent, we have 2 cost settings (low, high) by adjusting $N = 1, 2$. The temperature of the LLM is set to 0.0 in the base agent and 0.5 ($n = 5, 8$) or 1.0 ($n = 13$) in the EnCompass agent.

As shown in Table 3, controlling inference-time scaling through the external search algorithm in EnCompass scales in a more cost-efficient manner than scaling the number of feedback loops in Reflexion: the same performance is achieved at a lower cost. Comparing the two scaling strategies, we find that BeFS and best-of-N are comparable.

Table 3: Increasing the number of search steps in the EnCompass Reflexion agent scales better than scaling the number of refinement loops in the vanilla Reflexion agent: the same performance is achieved at a lower cost. All errors are standard errors of the mean over 5 runs.

Cost setting	Low		Medium		High	
	Acc. (%)	Cost/task ($)	Acc. (%)	Cost/task ($)	Acc. (%)	Cost/task ($)
Reflexion	35.5 ± 1.0	0.279 ± 0.005	35.9 ± 1.3	0.449 ± 0.005	38.2 ± 1.2	0.736 ± 0.010
+best-of-N	35.5 ± 1.0	0.279 ± 0.005	—	—	37.6 ± 1.7	$\mathbf{0.508} \pm 0.013$
+BeFS	36.1 ± 2.1	$\mathbf{0.168} \pm 0.004$	36.1 ± 1.1	$\mathbf{0.289} \pm 0.007$	38.1 ± 1.3	$\mathbf{0.512} \pm 0.006$

B Documentation of EnCompass

EnCompass is an instantiation of the PAN programming framework in Python. It is implemented as the `@encompass.compile` function decorator, which makes several new keywords primitives available in the body of the decorated function. Appendix C describes how the decorator compiles the function body into an object that provides an interface for search.

This appendix is organized as follows:

- Appendix B.1 lists all EnCompass keyword primitives that are made available inside a function with the `@encompass.compile` decorator.
- Appendix B.2 describes the interface of the compiled search space object created by the `@encompass.compile` decorator.
- Appendix B.3 describes the interface of the `Checkpoint` object that represents the program state at a branchpoint or return statement.
- Appendix B.4 describes the search algorithms that EnCompass provides out-of-the-box, as well as the abstract `Search` class that the user can subclass to define their own custom search algorithms.

B.1 EnCompass primitives

The following is the complete list of the 12 EnCompass keyword primitives in alphabetical order. They are available in any function or async function with the `@encompass.compile` decorator.

`branchpoint(**branchpoint_params)`

This statement marks a *branchpoint*. When combined with proper verification signal from `record_score` statements (see below), this creates the illusion that the stochastic operations that follow are now biased to more desirable outputs, and unreliable operations (e.g., LLM calls) have become more reliable.

This illusion (*angelic nondeterminism*) is accomplished through search over the different nondeterministic branches of the program's execution. More specifically, when the program's execution reaches a branchpoint, the program will branch into multiple copies of itself and an external search algorithm implemented using the `Checkpoint` interface searches over the multiple branches of the program.

`branchpoint_params` can include the following keyword arguments (all are optional):

- `name: Any`: A name to label the branchpoint
- `max_protection: int | None`: The maximum number of times stepping to the next branchpoint is allowed to raise an exception that gets protected (see documentation for `protect()`).
- `message_to_agent: Any`: A message to send to the agent (see below for messaging).

Other available keyword arguments depend on the specific search algorithm being used. For example, algorithms derived from graph search algorithms by fixing the branching factor allow the programmer to provide a branchpoint-specific branching factor `branching` and maximum amount of parallelization `max_workers` when sampling the next state.

Example usage: The simplest use case is to add one `branchpoint()` statement at the top of the function body (Listing 4), which amounts to best-of-N sampling (Section 3.1):

```
@encompass.compile
def branchpoint_example(...):
    branchpoint()
    ...  # Do something
    record_score(...)

# Sample 10 times and output the result with the highest score
branchpoint_example(...).search("sampling", num_rollouts=10)

```

Listing 4: `branchpoint` example: Best-of-N sampling `branchpoint()` also supports messaging with the controller (user of the `Checkpoint` interface) with a syntax similar to that of Python `yield`. This lets the programmer implement highly customized search algorithms optimized for their particular agent workflow — decisions on node selection and backtracking can now depend on the details of the execution state of the agent that are sent to the search process via this messaging interface.

Example usage: Listing 5 illustrates how messaging can be used to let the controller decide whether to backtrack based on the execution state of the underlying agent.

```python
@encompass.compile
def branchpoint_messaging(task):
    branchpoint()
    solution = ...
    feedback = ...
    # Python equivalent: response = yield (...)
    response = branchpoint(message_to_controller=(task, solution, feedback))
    print(response)

# Python equivalent: generator = branchpoint_messaging(); next(generator)
checkpoint0 = branchpoint_messaging().start()
# Python equivalent: task, solution, feedback = next(generator)
checkpoint1 = checkpoint0.step()
task, solution, feedback = checkpoint1.message_from_agent
# Decide whether to backtrack
should_backtrack = decide_backtrack(task, solution, feedback)
if should_backtrack:
    # Backtrack and retry last step - no Python equivalent
    checkpoint1 = checkpoint0.step()
# Python equivalent: generator.send(f"backtracked: {should_backtrack}")
checkpoint2 = checkpoint1.step(
    message_to_agent=f"backtracked: {should_backtrack}"
)
```

Listing 5: Example of `branchpoint` with agent-controller messaging

`branchpoint_choose(`*`choices`*`: Iterable, **`*`branchpoint_params`*`):`

This is a variant of `branchpoint` where the resulting branches have the `branchpoint_choose(choices)` expression evaluate to the elements in the iterable `choices`. In other words, this implements regular angelic nondeterminism.

Example usage: The following function (Listing 6) guesses a path from a start node to a goal in a graph. Conducting search over the nondeterministic execution branches becomes equivalent to actual search over the graph.

```python
@encompass.compile
def graph_search(graph, start_node, goal):
    """
    Guess a path from `start_node` to `goal` in a `graph` represented as an adjacency list.
    """
    cur_node = start_node
    path = [cur_node]
    cost_so_far = 0
    while cur_node != goal:
        next_node = branchpoint_choose(graph[cur_node], identity=cur_node)
        path = path + [cur_node]
        cost_so_far += get_edge_cost(cur_node, next_node)
        total_estimated_cost = cost_so_far + estimate_cost_to_go(next_node, goal)
        record_score(-total_estimated_cost)
        cur_node = next_node
    return path

# Conduct best-first search -> shortest path with A* search
```

```python
graph_search(my_graph, my_start_node, my_goal).search("best_first", top_k_popped
    =1, default_branching=None)
```

Listing 6: Graph search example with `branchpoint_choose`

`early_stop_search()`

This early-stops the external search process because, e.g., a correct answer has been found.

Example usage: (also see Case Studies 2 and 3)

```python
@encompass.compile
def early_stop_search_example(...):
    ...  # Do something before
    branchpoint()
    # Ask LLM to generate answer
    answer = llm.generate(...)
    # Check answer
    success = check_answer(answer)
    if success:
        early_stop_search()
    return answer
```

Listing 7: `early_stop_search` example

`kill_branch(`*err*`=None)`

This kills the current branch of program execution. For example, if the LLM generated something irreparably bad, instead of recording a large negative score (i.e., `record_score(-1000)`), one can simply kill the current branch.

Example usage:

```python
@encompass.compile
def kill_branch_example(...):
    ...  # Do something before
    branchpoint()
    # Ask LLM to do something
    response = llm.generate(...)
    sanity_check_passed = sanity_check_llm_response(response)
    if not sanity_check_passed:
        kill_branch()
    ...  # Do something after
```

Listing 8: Example usage of `kill_branch`

var`: NeedsCopy`   *var*`: NeedsCopy = `*expr*

This tells the EnCompass compiler that the variable named *var* needs to be copied upon branching. In other words, this type annotation declares a variable that is independent across all future execution paths of the program, assuming no "*var*`: NoCopy`" declaration ever occurs in the future.

By default, all local variables need copying, so `NeedsCopy` is typically only used to undo an earlier `NoCopy` declaration.

Global variables are never copied. In fact, using "*var*`: NeedsCopy`" in a Python function will actually declare a *local* variable named *var* that needs copying.

Note that variable assignment without a `NeedsCopy` or `NoCopy` declaration will not change whether it is `NeedsCopy` or `NoCopy`.

Example usage: In this example, the programmer wishes to reuse the name of a `NoCopy` variable for something that needs copying (Listing 9):

```python
@encompass.compile
def needs_copy_example(task):
    # Step 1: Iterative refinement using NoCopy
    feedbacks: NoCopy = []
    branchpoint()
    ...
    score, feedback = get_score_and_feedback(...)
    feedbacks.append(feedback)
    record_score(score)

    # Step 2: Summarize every feedback in `feedbacks`
    feedbacks: NeedsCopy  # Different summary attempts mutate differently --- so we want copies of `feedbacks` on different search branches
    branchpoint()  # Sample multiple summary attempts
    for i, feedback in enumerate(feedbacks):
        feedbacks[i] = summarize_feedback(feedback)
    ...

result = agent_forward(task).search("dfs", default_braching=5)
```

Listing 9: `NeedsCopy` example

`var: NoCopy`   `var: NoCopy = expr`

This tells the EnCompass compiler that the variable named *var* need not be copied upon branching. In other words, this type annotation declares a variable that is shared across all future execution paths of the program, assuming no "*var*: `NeedsCopy`" declaration ever occurs in the future.

By default, all local variables need copying, so `NoCopy` is needed to declare a variable to be shared across future execution paths.

Global variables are never copied, so there is no need to use "*var*: `NoCopy`" to specify a global variable that doesn't need copying. In fact, this declaration would actually declare a *local* variable named *var* that doesn't need copying.

Note that variable assignment without a `NeedsCopy` or `NoCopy` declaration will not change whether it is `NeedsCopy` or `NoCopy`.

Example usage: The simplest use case is to modify the best-of-N (one branchpoint at the top) by initializing a shared memory of feedback from past attempts. This gives rise to iterative refinement (Section 3.2).

```python
@encompass.compile
def no_copy_example(task):
    feedbacks: NoCopy = []
    branchpoint()
    result = perform_task(task, feedbacks)
    score, feedback = get_score_and_feedback(result)
    feedbacks.append(feedback)
    record_score(score)

# Sample 10 times and output the result with the highest score
result = agent_forward(task).search("sampling", num_rollouts=10)
```

Listing 10: Iterative refinement

`optional_return(`*`return_value`*`)`

This signals to the external search process that, although the program execution hasn't finished, an output *return_value* has already been produced and should be treated as a possible return value of the program.

Example usage: (also see Listing 25 in Case Study 2)

```python
@encompass.compile
```

```python
def optional_return_example(...):
    answer = llm.generate_answer(...)
    optional_return(answer)
    refined_answer = llm.refine_answer(answer, ...)
    return refined_answer
```

Listing 11: `optional_return` answer

`protect(expr, exception, max_retries=None)`

If evaluating an expression *expr* may raise exception *exception*, then wrapping it in `protect(...)` creates the illusion that it no longer raises the exception. The illusion is created by resampling from the most recent branchpoint until evaluating the expression no longer raises the exception. *max_retries*, if not `None`, sets an upper limit on the number of retries.

Example usage: One example use case is parsing output from an LLM. The following example extracts the Python code block from an LLM and parses it. Both steps could error out because of the unreliability of the LLM, so we can wrap them in `protect`.

```python
@encompass.compile
def parse_llm_output_example(...):
    ...  # Do something before
    branchpoint()
    # Ask LLM to generate Python code
    response = llm.generate(...)
    # Extract Python code
    python_code = protect(response.split("```python\n", 1)[1]
                              .split("```", 1)[0], IndexError)
    # Parse Python code
    python_ast = protect(ast.parse(python_code), SyntaxError)
    ...  # Do something after
```

Listing 12: `protect` example: Safely parsing output from an LLM

`record_costs(**costs)`

This lets the user track various kinds of cost, e.g., LLM usage. The costs are aggregated and accessed through the dictionary `func.aggreagte_costs` where *func* is the compiled function.

Example usage:

```python
@encompass.compile
def record_costs_example(...):
    response, cost = llm.generate(...)
    record_costs(llm_cost=cost, llm_num_calls=1)
    return response
```

Listing 13: `record_costs` example

`record_score(score)`

This is the main means for providing reward/verification signal to the external search algorithm by recording a score. The exact semantics of this score will depend on the search algorithm used (e.g., heuristic for best-first search, value function for MCTS).

Example usage: The simplest example is best-of-N sampling, which samples the agent workflow multiple times and selects the result with the highest score recorded by `record_score`.

```python
@encompass.compile
def branchpoint_example(...):
    branchpoint()
    ...  # Do something
    record_score(...)
```

```python

# Sample 10 times and output the result with the highest score
branchpoint_example(...).search("dfs", default_branching=10)

```

Listing 14: `record_score` example: Best-of-N sampling

`record_score(group_evaluator, eval_target, label=eval_label)`

This overloading of `record_score` enables evaluation that compares across multiple program execution branches. The simplest use case for this is self-consistency majority voting, where evaluating a result must be done relative to all results (Section 3.3).

`searchover(func(...))`

This is the syntax for calling an EnCompass function *func* inside another EnCompass function. This is similar to the `await func(...)` syntax for calling an async function inside another async function, where instead of `await` we use `searchover`.

Example usage: (also see Listing 19 in Case Study 1)

```python
@encompass.compile
def helper_function(...):
    ...

@encompass.compile
def searchover_example(...):
    ...  # Do something before
    helper_result = searchover(helper_function(...))
    ...  # Do something after

```

Listing 15: `searchover` example

`searchover_await(async_func(...))`

This is the asynchronous counterpart to `searchover()`. In other words, it is used to call an asynchronous EnCompass function *async_func* from within another asynchronous EnCompass function.

Example usage:

```python
@encompass.compile
async def async_helper_function(...):
    ...

@encompass.compile
async def searchover_await_example(...):
    ...  # Do something before
    helper_result = searchover_await(async_helper_function(...))
    ...  # Do something after

```

Listing 16: `searchover_await` example

B.2 Compiled search space interface

The interface of the compiled search space allows the user to either *step* through the program or *search* over its nondeterministic execution paths.

In what follows, *func* represents a function compiled with the `@encompass.compile` decorator, and `func(...)` represents the search space object created from calling the compiled function on some arguments.

`func(...).start() -> Checkpoint`

This begins execution of the function with the given arguments until the first branchpoint, i.e., a `branchpoint()` or `branchpoint_choose()`, which could be inside a nested `searchover()` function call. The program state at that point is wrapped into a `Checkpoint` object, which can be used to step through the function, creating checkpoints at branchpoints. A partial interface of `Checkpoint` is given in Appendix B.3.

`async_func(...).async_start() -> AsyncCheckpoint`

(async method) Async equivalent of `func(...).start()` for async EnCompass functions.

`func(...).search(search_algo: str, **search_params) -> Any`

This conducts search over the compiled search space using the given search algorithm and returns the final result, which is usually the return value (from either `return return_value` or `optional_return(return_value)`) from the branch with the highest latest recorded score. Search algorithms available in EnCompass are detailed in Appendix B.4.

`async_func(...).async_search(search_algo: str, **search_params) -> Any`

(async method) Async equivalent of `func(...).search()` for async EnCompass functions.

`func(...).search_multiple(search_algo: str, **search_params) -> list[tuple]`

This is the same as `search()`, except it returns all results and not just the best one. Results are returned as a list of pairs `(rv, score)` where `rv` is the return value of a branch and `score` is its score.

`async_func(...).async_search_multiple(search_algo, **search_params) -> list[tuple]`

(Async method) Async equivalent of `func(...).search_multiple()` for async EnCompass functions.

`func.aggregate_costs: dict[str, float|int]`

This is a dictionary containing the aggregate costs from all `record_cost` statements. Key `"<cost_name>"` is mapped to the sum of all costs recorded with that name via `record_cost(<cost_name>=...)`.

`func.branchpoint_step_counts: dict[Any, int]`

This is a dictionary that maps the name of a branchpoint to the number of times `step()` has been called on a checkpoint of that branchpoint, over all calls to `func` since the last time `zero_branchpoint_counts()` was called (see below). The dictionary will only contain step counts for named branchpoints, i.e., branchpoints with a `name` parameter (i.e., `branchpoint(name=...)` or `branchpoint_choose(choices, name=...)`).

`func.zero_branchpoint_counts() -> None`

This zeros out the recorded total step counts of each named branchpoint.

B.3 Checkpoint object interface

A `Checkpoint` holds the program state at a branchpoint or return statement of an EnCompass program's execution.

`class Checkpoint`

`step(max_protection=None, score_db_flush_queue=True) -> Checkpoint`

This continues execution of the program starting from the stored program state until the next time a branchpoint is hit, returning a new `Checkpoint` object.

Any expressions protected by a `protect(expr, exception)` will trigger resampling whenever the exception occurs, up to a maximum of `max_protection` resamplings if it is not `None`.

If `score_db_flush_queue` is `False`, then pending evaluations recorded through the group-evaluation version of `record_score` will not be processed.

Multiple `step()` calls on the same `Checkpoint` are mostly independent: while variable assignments are independent, references to variables declared as `NoCopy` are shared, so that mutations to a `NoCopy` object created before the current checkpoint are seen by all execution branches descended from this checkpoint.

If the branchpoint is a `branchpoint_choose(choices: Iterable)` instead of a regualar `branchpoint()` statement, then multiple `step()` calls iterate through `choices`, and the resultant branches see the `branchpoint_choose(choices)` call evaluate to the elements in `choices`.

`step_sampler(max_samples=None, max_protection=None, score_db_flush_queue =True) -> Generator[Checkpoint, None, None]`

This calls `step()` repeatedly and yields the resultant `Checkpoint` objects. This is done at most `max_samples` is not `None`; otherwise it samples forever, or until the list of choices have been exhausted in `branchpoint_choose`.

`max_protection` specifies the *total* number of resamplings allowed for protected expression evaluations.

See `Checkpoint.step()` above for `score_db_flush_queue`.

`parallel_step_sampler(max_samples=None, chunk_size=None, max_protection =None, max_workers=None, score_db_flush_queue=True) -> Generator[ Checkpoint, None, None]`

Multithreaded version of `Checkpoint.step_sampler()`, where `max_workers` specifies the maximum number of threads to use and `chunk_size`, if given, does parallel samplings in batches of that size.

`status: Status`

The status of the checkpoint object. One of `Status.RUNNING`, `Status.DONE_STEPPING`, `Status.RETURNED`, and `Status.KILLED`. The `Status.DONE_STEPPING` status is only possible at a `branchpoint_choose` with a finite set of choices.

`has_return_value: bool`

Whether there's a return value from `return return_value` (if the checkpoint is at a return statement) or `optional_return(return_value)` (if the checkpoint is at a branchpoint).

`return_value: Any`

The return value of the function if it exists (i.e., if the checkpoint is at a return statement, or it is at a branchpoint following an `optional_return` statement without an intervening branchpoint).

`early_stopped_search: bool`

Whether an `early_stopped_search()` statement has been called on *any* branch of the program's execution.

`score: float|int`

The most recent score recorded through `record_score()`.

`branchpoint_params: dict`

> This is a dictionary containing the parameters of the branchpoint as specified through `branchpoint(**branchpoint_params)` or `branchpoint_choose(choices, **branchpoint_params)`.

For async ENCOMPASS functions, there's a corresponding `AsyncCheckpoint` with the same interface, except that certain methods are now async, and `step_sampler()` and `parallel_step_sampler()` have been merged into one `async_step_sampler()`.

`class AsyncCheckpoint`

`async_step(max_protection=None, score_db_flush_queue=True) -> Checkpoint`

> *(async method)* Async equivalent of `Checkpoint.step()`.

`async_step_sampler(max_samples=None, chunk_size=None, max_protection=None, max_workers=None, score_db_flush_queue=True) -> AsyncGenerator[Checkpoint, None, None]`

> *(async method)* Async equivalent of `Checkpoint.step_sampler()` and `Checkpoint.parallel_step_sampler()`.

`status: Status`

> See `Checkpoint.status`.

`has_return_value: bool`

> See `Checkpoint.has_return_value`.

`return_value: Any`

> See `Checkpoint.return_value`.

`early_stopped_search: bool`

> See `Checkpoint.early_stopped_search`.

`score: float|int`

> See `Checkpoint.score`.

`branchpoint_params: dict`

> See `Checkpoint.branchpoint_params`.

B.4 Search interface and search algorithms

Search algorithms are implemented over the `Checkpoint` interface. Parameters to a search algorithm can be specified both in the arguments to `search()` when invoking a compiled search space object as well as in branchpoint parameters specified as arguments to `branchpoint()` and `branchpoint_choose()` within the ENCOMPASS function.

ENCOMPASS provides several common search algorithms out-of-the-box. The async implementations take advantage of the I/O-bound nature of LLM applications, whereas the non-async implementations use multithreaded parallelism, which the user can disable if they wish (e.g., to prevent race conditions when there are `NoCopy` variables). Here is the complete list of search algorithms in the current version of ENCOMPASS:

- Depth-first search (DFS)

- Breadth-first search (BFS)
- Best-first search (BeFS)
- Beam search
- Monte-Carlo tree search (MCTS), with a given value function
- Reexpand best-first search, a variant of BeFS where an expanded node can be expanded again. This was used in Case Study 3 (Appendix A.2).
- Explorative reexpand best-first search, a variant of reexpand BeFS where a UCB-like exploration bonus is added to the score.

The user can also implement and register their custom search algorithm by subclassing the abstract `Search` class. Here, we provide a template for defining and registering a custom search algorithm:

```python
@register_search_algo(is_async=False)  # or `is_async=True` if subclassing `AsyncSearch`
class MySearch(Search):  # or `MySearch(AsyncSearch)`
    name = "my_search"
    param_names = ["param1", "param2"]  # names of branchpoint parameters that I will use

    def __init__(self, *, config1, config2, default_param1, default_param2):
        self.config1 = config1
        self.config2 = config2
        self.default_param1 = default_param1
        self.default_param2 = default_param2

    def search_generator(
        self,
        init_program_state: Checkpoint
    ) -> Generator[tuple[Any, ScoreWithCallback], None, None]:
        # or `async def async_search_generator(self, init_program_state: AsyncCheckpoint)`
        # if subclassing `AsyncSearch`
        """
        Yields pairs (return_value: Any, score_with_callback: ScoreWithCallback)
        as they are found.

        ScoreWithCallback is a wrapper around a program state's score
        - it is needed for group evaluation to work properly.
        """
        # REPLACE CODE BELOW WITH YOUR CUSTOM SEARCH ALGORITHM
        next_program_states = init_program_state.parallel_step_sampler(...)
        for next_program_state in next_program_states:
            param1 = next_program_state.get_branchpoint_param("param1", self.default_param1)
            if next_program_state.has_return_value:
                yield next_program_state.return_value, next_program_state._score_with_callback
            ...
        ...
```

C The EnCompass compiler

The EnCompass compiler syntactically transforms an EnCompass function into an equivalent regular Python program by conversion to continuation-passing style (CPS) and applying tail-call optimization.

For simplicity, we only describe how we compile EnCompass functions that are not async. The compiler transformations for async EnCompass functions are nearly identical.

C.1 CPS for branchpoints

In this subsection, we describe how to convert a piece of code containing branchpoints (but not any of the other EnCompass keyword primitives) into CPS.

In its simplest form, transforming a piece of code into CPS results in a function

```
cps_function(frame: Frame, rest: Frame -> None) -> None
```

which runs the piece of code on the variable mapping `frame` to get a new variable mapping, followed by calling the callback `rest` on that new variable mapping. Here, the callback `rest`, called the *continuation*, represents the rest of the program.

For a piece of code that doesn't contain any branchpoints, it suffices to transform variable accesses and assignments to explicitly use `frame`. For example,

```
x = 1
y = x + 1
```

is compiled into

```
frame['x'] = 1
frame['y'] = frame['x'] + 1
rest(frame)
```

Note that we omit the `def cps_function(frame, rest):` in the compiled code, so technically we're compiling to the body of the CPS function. We will call this the *CPS body* to distinguish it from the *CPS function*. We defer the job of wrapping the CPS body into a function to whoever asked for the compilation. This simplifies the issue of naming CPS functions and referring to them with the correct name.

Since the compiled CPS function explicitly runs the continuation `rest(frame)`, adding a branchpoint immediately after the piece of code amounts to modifying the continuation to incorporate the search process. So we replace `rest(frame)` with `branchpoint_callback(frame, rest)`, which defines the rest of the program when we hit a branchpoint, where `rest` here now represents the rest of the program when we resume from the branchpoint. Taking the example above and adding a branchpoint at the end,

```
x = 1
y = x + 1
branchpoint()
```

gets compiled into the following CPS body:

```
frame['x'] = 1
frame['y'] = frame['x'] + 1
branchpoint_callback(frame, rest)
```

Here, `branchpoint_callback(frame, rest)` first stores the current program state `(frame, rest)` as a node in the search tree, then uses the search algorithm to decide a node `(frame1, rest1)` in the search tree to expand, and call `rest1(frame1.clone())` to run the rest of the program resuming from the branchpoint that saved the state `(frame1, rest1)`. Cloning `frame1` is needed because otherwise multiple calls to `rest1(frame1)` would modify the same `frame1` object.

So far we've only defined how to transform programs with no branchpoints and programs with one branchpoint at the end. The transformation of a general program with branchpoints in arbitrary locations can be defined recursively with these two base cases. For example, a program with a branchpoint in the middle,

```
x = 1
branchpoint()
y = x + 1
```

Listing 17: Program with a branchpoint in the middle

is a concatenation of two programs:

```
A:
x = 1
branchpoint()
```

and

```
B:
y = x + 1
```

where we can apply the recursive transformation rule for concatenation,

```
def rest(frame):
    CPS(B)
CPS(A)
```

to obtain the CPS body

```
def rest(frame):
    frame['y'] = frame['x'] + 1
    finish_callback(frame)
frame['x'] = 1
branchpoint_callback(frame, rest)
```

Note that we have replaced `rest(frame)` with `finish_callback(frame)` in the compilation of `B` to avoid name collision with the `def rest(frame)`. As a result, the compiled CPS function of the complete top-level program (AST root node) also has to reflect this name change in its signature: `top_level_cps_function(frame, finish_callback)` instead of `top_level_cps_function(frame, rest)`. So, if Listing 17 is our entire program, then its CPS function is

```
def top_level_cps_function(frame, finish_callback):
    def rest(frame):
        frame['y'] = frame['x'] + 1
        finish_callback(frame)
    frame['x'] = 1
    branchpoint_callback(frame, rest)
```

As a more complicated example, consider the following code:

```
i = 0             # A
branchpoint()     # A
j = 0             # B
while i < 10:     # B
    j -= 1        # B - X
    branchpoint() # B - X
    i += 1        # B - Y
print(j)          # C
```

We've chunked up the statements at the top level into 3 pieces: A, B and C. Each chunk consists of zero or more branchpoint-free statements followed by a statement containing a branchpoint, except for the last chunk C which is branchpoint-free. We apply the concatenation rule to A and $(B;C)$, which recursively applies the concatenation rule to B and C. This then recursively compiles the last statement of B — the while loop. Compiling the while loop using the while loop rule recursively compiles the body of the while loop using the concatenation rule on the chunks X and Y.

The concatenation rule as applied to chunks X and Y gives

```python
def rest(frame):
    # CPS body of Y
    frame['i'] += 1
    continue_callback(frame)
# CPS body of X
frame['j'] -= 1
branchpoint_callback(frame, rest)
```

where to avoid name collision we replaced `rest(frame)` with `continue_callback(frame)`.

Applying the while loop rule gives

```python
def body_cps_function(frame, continue_callback, break_callback):
    # CPS body of (X; Y) (from above)
    def rest(frame):
        # CPS body of Y
        frame['i'] += 1
        continue_callback(frame)
    # CPS body of X
    frame['j'] -= 1
    branchpoint_callback(frame, rest)
def while_cps_function(frame, rest):
    if frame['i'] < 10:
        body_cps_function(frame, lambda frame: while_cps_function(frame, rest))
    else:
        rest(frame)
while_cps_function(frame, rest)
```

Finally, applying the concatenation rule twice in $(A; (B; C))$ gives the CPS body of the entire program:

```python
# CPS body of (A; B; C)
def rest(frame):
    # CPS body of (B; C)
    def rest(frame):
        # CPS body of C
        print(frame['j'])
        finish_callback(frame)
    # CPS body of B
    frame['j'] = 0
    ...  # CPS body of the while loop (from above)
# CPS body of A
frame['i'] = 0
branchpoint_callback(frame, rest)
```

And, as usual, to get the CPS function of this program, we simply wrap the above CPS body into a `def top_level_cps_function(frame, finish_callback)` function.

Note that the general solution to dealing with name collision is to add the correct version of `rest(frame)` to the end of each "body" in the AST during preprocessing, so that we don't have to deal with it during conversion to CPS:

- At the end of the top-level program, add `finish_callback(frame)` during preprocessing. During conversion to CPS, the signature of the CPS function of a top-level program will be `top_level_cps_function(frame, finish_callback)` instead of `top_level_cps_function(frame, rest)`.
- At the end of the body of a for/while loop, add `continue_callback(frame)` during preprocessing. During conversion to CPS, the signature of the CPS function of the body of a for/while loop will be `body_cps_function(frame, continue_callback, break_callback)` instead of `body_cps_function(frame, rest)`. Note that this also specifies the names of the callbacks that `continue` and `break` statements in the body get converted to during conversion to CPS — two birds with one stone.
- At the end of the body of an `if` or an `else`, add `if_else_callback(frame)`. During conversion to CPS, the signature of the CPS function of the body

of an `if` will be `if_body_cps_function(frame, if_else_callback)` instead of `if_body_cps_function(frame, rest)`, and similarly for `else`.

- At the end of the body of a function, add `return_callback(frame.caller_frame)` . During conversion to CPS, the signature of the CPS function of the body of a function will be `function_body_cps_function(frame, return_callback)` instead of `function_body_cps_function(frame, rest)`.

We are now ready to formally write down the full set of transformations for EnCompass programs with the simplest version of EnCompass that only has branchpoints. For simplicity, we only describe the transformations done for synchronous code (no async/await) where only loops, if/else statements and function definitions have branchpoints (`with`, try-except, and `match` statements are all branchpoint-free).

Preprocessing The preprocessing stage consists of the following steps:

1. Convert all names var to `frame['`var`']`.
2. Add `finish_callback(frame)` to the end of the program.
3. Add `continue_callback(frame)` to the end of the body of every for/while loop.
4. Add `if_else_callback(frame)` to the end of the body of every branch of every if-else statement.
5. Add `return_callback(frame.caller_frame)` to the end of the body of every function that doesn't already end in a return statement.
6. Make the following replacements:
   - `continue` → `continue_callback(frame)`
   - `break` → `break_callback(frame)`
   - `return rv` → `return_callback(frame.caller_frame, rv)`

Conversion to CPS Here are the general transformation rules for compiling a top-level program or the body of a function, after the preprocessing steps described above have been completed.

1. *Base case — branchpoint-free:* If A has no branchpoints, make no changes. In other words, $CPS(A) = A$.
2. *Base case — branchpoint:* A branchpoint

```
branchpoint()
```

   becomes

```
branchpoint_callback(frame, rest)
```

3. *Concatenation:* For $(A; B)$ where $A = (A'; a)$ with A' branchpoint-free and a a single statement containing one or more branchpoints (a branchpoint or a for/while/if/else statement containing a branchpoint, but not e.g. a function definition containing a branchpoint),

```
A'  # zero or more branchpoint-free statements
a  # single statement containing one or more branchpoints
B  # zero or more statements
```

   the compiled CPS body is

```
def rest(frame):
    CPS(B)
A'
CPS(a)
```

4. *While loops:* For a while loop containing one or more branchpoints,

```
while e:
    A  # contains one or more branchpoints
```

   the compiled CPS body is

```
def body_cps_function(frame, continue_callback, break_callback):
    CPS(A)
def while_cps_function(frame, rest):
    if e:
        body_cps_function(
            frame,
            lambda frame: while_cps_function(frame, rest),
            rest
        )
    else:
        rest(frame)
while_cps_function(frame, rest)
```

5. *For loops:* For a for loop containing one or more branchpoints,

```
for i in e:
    A  # contains one or more branchpoints
```

the compiled CPS body is

```
def body_cps_function(frame, continue_callback, break_callback):
    CPS(A)
def for_cps_function(frame, rest):
    try:
        i = next(frame.iterables[-1])
    except StopIteration:
        frame.iterables.pop()
        rest(frame)
        return
    def break_callback(frame):
        frame.iterables.pop()
        rest(frame)
    body_cps_function(
        frame,
        lambda frame: for_cps_function(frame, rest),
        break_callback
    )
frame.iterables.append(iter(e))
for_cps_function(frame, rest)
```

6. *If-else statements:* For an if-else statement containing one or more branchpoints,

```
if e:
    A
else:
    B
```

the compiled CPS body is

```
def if_body_cps_function(frame, if_else_callback):
    CPS(A)
def else_body_cps_function(frame, if_else_callback):
    CPS(B)
if e:
    if_body_cps_function(frame, rest)
else:
    else_body_cps_function(frame, rest)
```

C.2 Tail-call optimization

There are two issues with the compiled CPS representation. One issue is performance — the extra function calls cause overhead, and long for/while loops become deep recursive calls that can exceed Python's recursion depth limit. The second issue is that defining the search algorithm by defining `branchpoint_callback(frame, rest)` is unnatural and difficult. Typically, a search algorithm is implemented assuming access to a `step` method that returns a child of a node, `new_state = step(state)`.

We solve both issues via *tail-call optimization*. More specifically, every `branchpoint_callback(frame, rest)` is replaced with `return frame, rest`, and `rest(frame)` no longer resumes from a branchpoint to execute the rest of the program, but only executes until the next branchpoint is hit, at which point the `frame, rest` at that branchpoint is returned. In other words, `new_frame, new_rest = rest(frame.clone())` is exactly the `new_state = step(state)` that we need, where we identify `state` with `(frame, rest)`.

With this modification, reproducing the execution of the program when all branchpoints are ignored now involves a while loop that keeps stepping until the program finishes:

```
frame = {}
rest = lambda frame: top_level_cps_function(frame, lambda frame: (frame, None))
while rest is not None:
    frame, rest = rest(frame)
```

And a simple DFS looks like this:

```
frame = {}
rest = lambda frame: top_level_cps_function(frame, lambda frame: (frame, None))
stack = [(frame, rest)]
results = []
while stack:
    frame, rest = stack.pop()
    for _ in range(branching_factor):
        new_frame, new_rest = rest(frame.clone())
        if new_rest is None:
            results.append(frame)
        else:
            stack.append((new_frame, new_rest))
```

We can wrap the state `(frame, rest)` into a `Checkpoint` object that provides a `step` method wrapping `new_frame, new_rest = rest(frame.clone())`, and any search algorithm can now be implemented using the `Checkpoint` interface.

We also need to modify the CPS transformation rules to return the next state instead of running the entire continuation to completion. The details of the modifications are as follows:

1. *Base case — branchpoint-free:* No change.
2. *Base case — branchpoint:* A branchpoint

   ```
   branchpoint()
   ```

   is now compiled to

   ```
   return frame, rest
   ```

3. *Concatenation:* No change.
4. *While loops:* Prepend `return` to these 3 lines:

   ```
   ...
   def while_cps_function(frame, rest):
       if e:
           return body_cps_function(...)   # <-
       else:
           return rest(frame)              # <-
   return while_cps_function(frame, rest)  # <-
   ```

5. *For loops:* Prepend `return` to these 3 lines:

   ```
   ...
   def for_cps_function(frame, rest):
       ...
           ...
           return rest(frame)             # <-
       ...
       return body_cps_function(...)      # <-
   ```

```
...
return for_cps_function(frame, rest)  # <-
```

6. *If-else statements:* Prepend `return` to these 2 lines:

```
...
if e:
    return if_body_cps_function(frame, rest)    # <-
else:
    return else_body_cps_function(frame, rest)  # <-
```

C.3 Other keywords

Most other EnCompass primitives provide auxiliary information, which we store in a dictionary `info`. We modify our transformation rules so that `info` always occurs alongside `frame`, so the `Checkpoint` object is now a wrapper around the 3-tuple `(frame, info, rest)`. The `Checkpoint` class implements the intended semantics of these additional EnCompass keywords using the information stored inside `info` — details which we will omit.

Note that `info` is copied upon `Checkpoint.step()` similar to how `frame` gets cloned. In other words, stepping is now implemented as `new_frame, new_info, new_rest = rest(frame.clone(), info.copy())`.

For keywords that are used as standalone statements, preprocessing is done to convert these keywords into statements that modify `info`. The only exception is `kill_branch()`, which is transformed into a `finish_callback()` call. Here, we list the preprocessing transformations for all keywords that are used as standalone statements:

- `early_stop_search()` → `info["early_stop_search"] = True`
- `kill_branch(`*e*`)` → `finish_callback(frame,` *e*`, info, killed=True)`
- *v*`: NeedsCopy` →
  `var =` *v*`; if var in info["nocopy"]: info["nocopy"].remove(var)`
  An annotated assignment is broken into two statements — the annotation and the assignment — before this transformation on the annotation occurs.
- *v*`: NoCopy` → `info["nocopy"].add(`*v*`)`
  An annotated assignment is broken into two statements — the annotation and the assignment — before this transformation on the annotation occurs.
- `optional_return(`*e*`)` → `info["optional_rv"] =` *e*
- `record_costs(`*keywords*`)` → `info["costs"] = dict(`*keywords*`)`
- `record_score(`*args*`)` →
  `info["score"] = info["score_db"].submit_score(`*args*`)`
  Here `info["score_db"]` is a `ScoreDB` object whose `submit_score()` method returns a thunk that represents the eventual value of the score. This extra complexity is needed to implement group evaluation (Section 3.3).
  Without group evaluation,
  `record_score(`*e*`)` → `info["score"] =` *e*
  would suffice.

Now, the remaining keyword primitives — `branchpoint()`, `branchpoint_choose()`, `searchover()` and `protect()` — are all used as expressions that can be part of a larger expression or a statement.[5] A statement that contains one or more of these keyword primitives needs to be partially converted to A-normal form [41], where the return value from a keyword primitive is first assigned to a temporary variable, and its occurrence in the statement is replaced with that temporary variable. This is done recursively for keywords nested within keywords. For example, the statement

[5] While Appendix C.1 treated `branchpoint()` as a statement, in fact it can be used to communicate with the controller (user of the `Checkpoint` interface), where messages from the controller appear as the return value of `branchpoint()`.

```
answer = get_answer(
    branchpoint_choose([searchover(agent1(task)), protect(agent2(task), ValueError])
)
```

when converted to A-normal form will become

```
frame.tmp_vars[0] = searchover(agent1(task))
frame.tmp_vars[1] = protect(agent2(task), ValueError)
frame.tmp_vars[2] = branchpoint_choose([frame.tmp_vars[0], frame.tmp_vars[1]])
answer = get_answer(frame.tmp_vars[2])
```

After conversion to A-normal form, each statement that assigns the output of a keyword primitive to a temporary variable is further transformed as follows:

- `frame.tmp_vars[`N`] = branchpoint(`$kwargs$`)` $\rightarrow$ [no change]
- `frame.tmp_vars[`N`] = branchpoint_choose(`e`, `$kwargs$`)` $\rightarrow$

```
iterable = e
iterator, iterator_copy = tee(iterable)
try:
    next(iterator_copy)
except StopIteration:
    info["done_stepping"] = True
frame.tmp_vars["iterator_list"] = [iterator]
frame.tmp_vars[None] = branchpoint(kwargs)  # discard message from controller - not yet supported by branchpoint_choose()
try:
    frame.tmp_vars[N] = next(frame.tmp_vars["iterator_list"][0])
except StopIteration as e:
    raise FinishedSteppingError from e
frame.tmp_vars["iterator_list"][0], iterator_copy = tee(
    frame.tmp_vars["iterator_list"][0])
try:
    next(iterator_copy)
except StopIteration:
    info["last_branchpoint_done_stepping"] = True
```

- `frame.tmp_vars[`N`] = searchover(`e`)` $\rightarrow$ [no change]
- `frame.tmp_vars[`N`] = protect(`$expr$`, `err`)` $\rightarrow$

```
try:
    frame.tmp_vars[N] = expr
except err:
    finish_callback(frame, err, info, killed=True, protected=True)
```

Finally, note that we have to modify the CPS transformation rule for branchpoints from Appendix C.2, as well as adding a CPS transformation rule for `searchover`. The new rules are:

- `frame.tmp_vars[`N`] = branchpoint(`$kwargs$`)`
  $\rightarrow$

```
def branchpoint_rest(frame, info, message_to_agent):
    frame.tmp_vars[N] = message_to_agent
    rest(frame, info)
return frame, info, branchpoint_rest, dict(kwargs)
```

  So now, sampling a next program state is now implemented as `frame, info, branchpoint_rest, branchpoint_params = branchpoint_rest(frame.clone(), info.copy(), message_to_agent)`.

- `frame.tmp_vars[`N`] = searchover(`e`)`
  $\rightarrow$

```
def return_rest(frame, rv, info):
    frame.tmp_vars[N] = rv
    rest(frame, info)
func_call = e
if not isinstance(func_call, SearchSpaceWithArgs):
    raise SearchoverTypeError(f"searchover(...) expects a 'SearchSpaceWithArgs' object, instead got {type(func_call)}")
func_call.compiled_cps_function(
    Frame(
        locals=func_call._args_dict,
        caller_frame=frame,
        enclosing_frame=Frame.from_closurevars(
            getclosurevars(func_call._search_space._wrapped_fn)
        )
    ),
    info,
    return_rest,
)
```

D Code comparisons for case studies: base agent vs. EnCompass agent vs. equivalent plain Python implementation

In this appendix, for each case study, we show the code for the underlying base agent, the agent augmented with search in EnCompass, and the equivalent agent implemented in plain Python. We annotate the changes made relative to the base agent:

- `# +n`: This line was added and it has n words (a *word* is as defined in Vim).
- `# x (-m+n)`: This line was changed; and m words were removed and n words were added.
- `# -m`: This line was removed and it contained m words.
- `# <-k`: This line (or group of omitted lines) was indent to the left by k indentation levels.
- `# ->k`: This line (or group of omitted lines) was indent to the right by k indentation levels.

We do not count lines added that don't contain any code (i.e., that are blank or only contain a comment).

We see that, while changes made to the base agent to support search in EnCompass are minimal, significant changes are needed to support search in the plain Python implemenation, thus demonstrating the representational advantage of EnCompass.

We will omit code that remains unchanged between the base agent, the EnCompass agent, and the EnCompass agent's plain Python implementation. Code segments that have been omitted are indicated by ellipses "...".

D.1 Case Study 1: Code Repository Translation Agent

Base agent:

```python
def run_code_and_compare(method, target_code, source_code, translation_unit):
    ...  # Logging; define some variables

    if method.type == "main":

        test_inputs = None
        if "System.in" in source_code:
            # STEP 1: Write test input generation script and generate test inputs

            ...  # Prompt LLM

            ...  # Get test input format specification from LLM response
            if fatal_error:
                return 0.0

            ...  # Get test input generation script from LLM response
            if fatal_error:
                return 0.0

            ...  # Generate test inputs
            if fatal_error:
                return 0.0

        # STEP 2: Directly run codes and compare them if tested component is main function
        ...
        match = ...
        return float(match)

    # Otherwise, we have to write a main function to test the component

    ...  # Define some variables

    # STEP 1: Write test input generation script and generate test inputs

    ...  # Prompt LLM

    ...  # Get test input format specification from LLM response
    if fatal_error:
        return 0.0

    ...  # Get test input generation script from LLM response
    if fatal_error:
        return 0.0

    ...  # Generate test inputs
    if fatal_error:
        return 0.0

    # STEP 2: Run the target code with the test inputs

    ...  # Prompt LLM

    ...  # Get output format specification from LLM response
    if fatal_error:
        return 0.0

    ...  # Get target main function code from LLM response
    if fatal_error:
        return 0.0

    ...  # Parse target main function code
    if fatal_error:
        return 0.0

    ...  # Extract target main function AST node
    if fatal_error:
        return 0.0

    ...  # Add target main function to target code
    ...  # Run target code with main function on test inputs

    # STEP 3: Generate source code main function and run it

    ...  # Prompt LLM

    ...  # Get source main function code from LLM response
    if fatal_error:
        return 0.0

    ...  # Parse and extract source main function AST node
    if fatal_error:
        return 0.0

    ...  # Add target main function to target code
    ...  # Run target code with main function on test inputs

    matches = ...
    match_fraction = sum(matches) / len(matches)
```

```python

    return match_fraction


def translate_class(translation_unit):
    ...  # Some setup (e.g., read and parse code files)
    methods_to_translate = ...
    num_methods_to_translate = len(methods_to_translate)
    translate_success_count = 0
    pass_tests_count = 0
    for method in methods_to_translate:
        target_code, translate_success = translate_method(method, target_code, source_code, translation_unit)
        if translate_success:
            translate_success_count += 1

            ...  # save target code

            if translation_unit.is_test:
                pass_tests_count += run_test_module(target_code, translation_unit)
            else:
                pass_tests_count += run_code_and_compare(
                    method,
                    target_code,
                    source_code,
                    translation_unit,
                )

    # Separately test main function (Python `if __name__ == "__main__"` block) if it's present
    if not translation_unit.is_test and ...:
        num_methods_to_translate += 1
        translate_success_count += 1
        pass_tests_count += run_code_and_compare(
            "main",
            target_code,
            source_code,
            translation_unit,
        )

    ...  # logging and saving progress

    return pass_tests_count, translate_success_count, num_methods_to_translate, new_branch


def setup_antlr4(source_code_root, target_code_root, temperature):
    source_subdir = 'src/main/antlr4'
    num_successful_translations = 0
    num_successful_parses = 0
    for root, dirs, files in os.walk(source_code_root / source_subdir):
        for file in files:
            ...  # Read antlr4 grammar file

            ...  # LLM modification if needed

            ...  # Write to target directory

            ...  # Run antlr4 to generate target Python classes

            ...  # Check if the generated files can be parsed

    return num_successful_translations + num_successful_parses


def code_translation_agent(source_code_root, target_code_root, args):
    ...  # Set up logging and git repo for saving progress

    # 0.1. Copy resource files (src/main/resources and src/test/resources)
    copy_resource_files(source_code_root, target_code_root)

    # 0.2. Set up antlr4 if applicable (src/main/antlr4)
    setup_antlr4(source_code_root, target_code_root, args.temperature)

    # 1. Get class names in topological order
    translation_units = get_translation_order_and_dependencies(source_code_root, target_code_root)

    for translation_unit in translation_units:
        # 2. Generate stubs for the class
        generate_stubs_success = generate_stubs(translation_unit)

        # 3. Translate each class
        pass_tests_count, translate_success_count, num_methods_to_translate, new_branch = translate_class(translation_unit)

        ...  # Log results

    ...  # Final logging and saving

    return final_commit


code_translation_agent(...)
```

Listing 18: Code repository translation agent

With ENCOMPASS*:*

```python
import uuid  # +2
import encompass  # +2


@encompass.compile  # +4
def branchpoint_git_commit(target_code_root, log_str="Branchpoint reached", new_branch_name="branch", **branchpoint_params):
        # +17
    repo = Repo(target_code_root)  # +6
    with open(target_code_root / "commit.log", "a") as f:  # +16
        f.write(log_str + '\n')  # +9
    repo.git.add(".")  # +6
    repo.git.commit("-m", log_str)  # +10
    cur_commit = str(repo.head.commit)  # +10
    branchpoint(**branchpoint_params)  # +4
    repo.git.checkout(cur_commit)  # +8
    repo.git.switch("-c", f"{new_branch_name}-{uuid.uuid4()}")  # +16


@encompass.compile  # +4
def run_code_and_compare(method, target_code, source_code, translation_unit, base_score):  # x (-0+2)
    ...  # Logging; define some variables

    if method.type == "main":

        test_inputs = None
        if "System.in" in source_code:
            searchover(branchpoint_git_commit(  # +4
                translation_unit.target_code_root,  # +4
                f"Generate test inputs for and test {translation_unit.target_module_path}:{component_name}",  # +15
                f"bp-gen_inputs_test_main-{translation_unit.target_module_path}-{component_name}",  # +12
            ))  # +1

            # STEP 1: Write test input generation script and generate test inputs

            ...  # Prompt LLM

            ...  # Get test input format specification from LLM response
            if fatal_error:
                return 0.0

            ...  # Get test input generation script from LLM response
            if fatal_error:
                return 0.0

            ...  # Generate test inputs
            if fatal_error:
                return 0.0

        # STEP 2: Directly run codes and compare them if tested component is main function
        ...
        match = ...
        return float(match)

    # Otherwise, we have to write a main function to test the component

    ...  # Define some variables

    searchover(branchpoint_git_commit(  # +4
        translation_unit.target_code_root,  # +4
        f"Generate test inputs for {translation_unit.target_module_path}:{method}",  # +13
        f"bp-gen_inputs-{translation_unit.target_module_path}-{method}",  # +12
    ))  # +1

    # STEP 1: Write test input generation script and generate test inputs

    ...  # Prompt LLM

    ...  # Get test input format specification from LLM response
    if fatal_error:
        # pad branchpoints
        searchover(branchpoint_git_commit(translation_unit.target_code_root))  # +8
        searchover(branchpoint_git_commit(translation_unit.target_code_root))  # +8
        return 0.0

    ...  # Get test input generation script from LLM response
    if fatal_error:
        # pad branchpoints
        searchover(branchpoint_git_commit(translation_unit.target_code_root))  # +8
        searchover(branchpoint_git_commit(translation_unit.target_code_root))  # +8
        return 0.0

    ...  # Generate test inputs
    if fatal_error:
        # pad branchpoints
        searchover(branchpoint_git_commit(translation_unit.target_code_root))  # +8
        searchover(branchpoint_git_commit(translation_unit.target_code_root))  # +8
        return 0.0

    record_score(base_score + 0.01)  # +8

```

```python
    searchover(branchpoint_git_commit(  # +4
        translation_unit.target_code_root,  # +4
        f"Running target code for {translation_unit.target_module_path}:{method}",  # +13
        f"bp-run_target-{translation_unit.target_module_path}-{method}",  # +12
    ))  # +1

    # STEP 2: Run the target code with the test inputs

    ...  # Prompt LLM

    ...  # Get output format specification from LLM response
    if fatal_error:
        # pad branchpoints
        searchover(branchpoint_git_commit(translation_unit.target_code_root))  # +8
        return 0.0

    ...  # Get target main function code from LLM response
    if fatal_error:
        # pad branchpoints
        searchover(branchpoint_git_commit(translation_unit.target_code_root))  # +8
        return 0.0

    ...  # Parse target main function code
    if fatal_error:
        # pad branchpoints
        searchover(branchpoint_git_commit(translation_unit.target_code_root))  # +8
        return 0.0

    ...  # Extract target main function AST node
    if fatal_error:
        # pad branchpoints
        searchover(branchpoint_git_commit(translation_unit.target_code_root))  # +8
        return 0.0

    ...  # Add target main function to target code
    ...  # Run target code with main function on test inputs

    record_score(base_score + 0.02)  # +8

    searchover(branchpoint_git_commit(  # +4
        translation_unit.target_code_root,  # +4
        f"Running source code for {translation_unit.target_module_path}:{method}",  # +13
        f"bp-run_source-{translation_unit.target_module_path}-{method}",  # +12
    ))  # +1

    # STEP 3: Generate source code main function and run it

    ...  # Prompt LLM

    ...  # Get source main function code from LLM response
    if fatal_error:
        return 0.0

    ...  # Parse and extract source main function AST node
    if fatal_error:
        return 0.0

    ...  # Add target main function to target code
    ...  # Run target code with main function on test inputs

    matches = ...
    match_fraction = sum(matches) / len(matches)

    return match_fraction


@encompass.compile  # +4
def translate_class(translation_unit):
    ...  # Some setup (e.g., read and parse code files)
    methods_to_translate = ...
    num_methods_to_translate = len(methods_to_translate)
    translate_success_count = 0
    pass_tests_count = 0
    for method in methods_to_translate:
        searchover(branchpoint_git_commit(  # +4
            translation_unit.target_code_root,  # +4
            f"Begin {translation_unit.source_class_path} translation of {method}.",  # +13
            f"bp-translate-{translation_unit.source_class_path}-{method}",  # +12
        ))  # +1

        target_code, translate_success = translate_method(method, target_code, source_code, translation_unit)
        if translate_success:
            translate_success_count += 1
            record_score(translate_success_count + pass_tests_count)  # +6

            ...  # save target code

            if translation_unit.is_test:
                pass_tests = run_test_module(target_code, translation_unit)
                pass_tests_count += pass_tests
            else:
                pass_tests_count += searchover(run_code_and_compare(  # x (-0+2)
```

```python
                    method,
                    target_code,
                    source_code,
                    translation_unit,
                    base_score = translate_success_count + pass_tests_count  # +5
                ))  # x (-0+1)
            record_score(translate_success_count + pass_tests_count)  # +6

    # Separately test main function (Python `if __name__ == "__main__"` block) if it's present
    if not translation_unit.is_test and ...:
        num_components_to_translate += 1
        translate_success_count += 1
        pass_tests_count += searchover(run_code_and_compare(  # x (-0+2)
            "main",
            target_code,
            source_code,
            translation_unit,
            base_score = translate_success_count + pass_tests_count  # +5
        ))  # x (-0+1)
        record_score(translate_success_count + pass_tests_count)  # +6

    ...  # logging and saving progress

    return pass_tests_count, translate_success_count, len(methods_to_translate), new_branch


@encompass.compile  # +4
def setup_antlr4(source_code_root, target_code_root, temperature):
    source_subdir = 'src/main/antlr4'
    num_successful_translations = 0
    num_successful_parses = 0
    for root, dirs, files in os.walk(source_code_root / source_subdir):
        for file in files:
            ...  # Read antlr4 grammar file

            searchover(branchpoint_git_commit(  # +4
                target_code_root,  # +2
                f"Translate antlr4 grammar {source_file_path.stem}",  # +10
                f"bp-translate_antlr4_grammar-{source_file_path.stem}",  # +10
            ))  # +1

            ...  # LLM modification if needed

            ...  # Write to target directory

            ...  # Run antlr4 to generate target Python classes

            ...  # Check if the generated files can be parsed

            record_score(num_successful_translations + num_successful_parses)  # +6

    return num_successful_translations + num_successful_parses


@encompass.compile  # +4
def code_translation_agent(source_code_root, target_code_root, args):
    ...  # Set up logging and git repo for saving progress

    # 0.1. Copy resource files (src/main/resources and src/test/resources)
    copy_resource_files(source_code_root, target_code_root)

    # 0.2. Set up antlr4 if applicable (src/main/antlr4)
    total_score = searchover(setup_antlr4(source_code_root, target_code_root, args.temperature))  # x (-0+5)

    # 1. Get class names in topological order
    translation_units = get_translation_order_and_dependencies(source_code_root, target_code_root)

    for translation_unit in translation_units:
        searchover(branchpoint_git_commit(  # +4
            target_code_root,  # +2
            f"Begin {translation_unit.source_class_path} translation.",  # +10
            f"bp-translate-{translation_unit.source_class_path}",  # +10
        ))  # +1

        # 2. Generate stubs for the class
        generate_stubs_success = generate_stubs(translation_unit)

        total_score += generate_stubs_success  # +3
        record_score(total_score)  # +4

        # 3. Translate each class
        searchover(branchpoint_git_commit(translation_unit.target_code_root, branching=1))  # +12
        translate_class_results = translate_class(translation_unit).search_multiple("beam", beam_width=2, default_branching=2)  # x (-0+14)
        (pass_tests_count, translate_success_count, num_methods_to_translate, new_branch), _ = branchpoint_choose(translate_class_results, branching=len(translate_class_results))  # x (see above)

        # "+1" to prevent agent from "cheating" (have very few e.g. zero stubs to implement)
        total_score += pass_tests_count / (num_methods_to_translate + 1)  # +9
        record_score(total_score)  # +4

        ...  # Log results
```

```python

    ...  # Final logging and saving

    return final_commit


code_translation_agent(...).search("beam", beam_width=3, default_branching=3)  # x (-0+13)
```

Listing 19: Beam search in ENCOMPASS, 5 branchpoints excluding padding

Without ENCOMPASS*:* Explicitly defining a state machine to support general search not only significantly obscures the underlying agent logic, but is also prone to bugs such as `KeyError` when accessing the dictionary `cur_state` that stores all the variables. A lot of newly added code is for bookkeeping to maintain a persistent state, which is implemented as a dictionary that stores the variables of the base agent.

```python
import uuid  # +2
import numpy as np  # +4


def git_commit(target_code_root, log_str="Branchpoint reached"):  # +10
    repo = Repo(target_code_root)  # +6
    with open(target_code_root / "commit.log", "a") as f:  # +16
        f.write(log_str + '\n')  # +9
    repo.git.add(".")  # +6
    repo.git.commit("-m", log_str)  # +10
    cur_commit = str(repo.head.commit)  # +10
    return cur_commit  # +2


def checkout_new_branch(target_code_root, cur_commit, new_branch_name="branch"):  # +11
    repo = Repo(target_code_root)  # +6
    repo.git.checkout(cur_commit)  # +8
    repo.git.switch("-c", f"{new_branch_name}-{uuid.uuid4()}")  # +16


def run_code_and_compare_prelude(cur_state, cur_commit, cur_score):  # x (-8+6)
    # Get used variables from `cur_state`
    method = cur_state["method"]  # +6
    target_code = cur_state["target_code"]  # +6
    source_code = cur_state["source_code"]  # +6
    translation_unit = cur_state["translation_unit"]  # +6

    ...  # Logging; define some variables

    if method.type == "main":

        test_inputs = None

        # Store new variables to `new_state`
        new_state = cur_state.copy()  # +6
        new_state["method"] = method  # +6
        new_state["run_code_log_path"] = run_code_log_path  # +6
        new_state["test_inputs"] = test_inputs  # +6
        new_state["dependency_files_str"] = dependency_files_str  # +6
        new_state["fatal_error"] = fatal_error  # +6

        if "System.in" in source_code:
            # git commit
            commit = git_commit(  # +4
                translation_unit.target_code_root,  # +4
                f"Generate test inputs for and test {translation_unit.target_module_path}:{method}",  # +15
            )  # +1

            return new_state, run_code_and_compare_gen_test_inputs_existing_main, cur_score, commit  # +8

        return run_code_and_compare_run_target_source_codes_existing_main(new_state, cur_score, cur_commit)  # +9

    ...  # Define some variables

    # Store newly defined variables to `new_state`
    new_state = cur_state.copy()  # +6
    new_state["method"] = method  # +6
    new_state["run_code_log_path"] = run_code_log_path  # +6
    new_state["num_test_inputs"] = num_test_inputs  # +6
    new_state["target_code_without_main"] = target_code_without_main  # +6
    new_state["target_code_with_dummy_main"] = target_code_with_dummy_main  # +6
    new_state["dependency_files_str"] = dependency_files_str  # +6
    new_state["fatal_error"] = fatal_error  # +6

    # git commit
    commit = git_commit(  # +4
        translation_unit.target_code_root,  # +4
        f"Generate test inputs for {translation_unit.target_module_path}:{method}",  # +13
    )  # +1
    return new_state, run_code_and_compare_gen_test_inputs, cur_score, commit  # +8


def run_code_and_compare_gen_test_inputs_existing_main(cur_state, cur_commit, cur_score):  # +9
    # Get used variables from `cur_state`
    method = cur_state["method"]  # +6
    translation_unit = cur_state["translation_unit"]  # +6
    run_code_log_path = cur_state["run_code_log_path"]  # +6
    dependency_files_str = cur_state["dependency_files_str"]  # +6
    fatal_error = cur_state["fatal_error"]  # +6

    checkout_new_branch(  # +2
        cur_commit,  # +2
        f"bp-gen_inputs_test_main-{translation_unit.target_module_path}-{method}",  # +12
    )  # +1
```

```python

    # Prepare for next step
    new_state = cur_state.copy()  # +6

    # Write test input generation script and generate test inputs

    ...  # Prompt LLM  # <-2

    ...  # Get test input format specification from LLM response  # <-2
    if fatal_error:  # <-2
        return new_state, translate_class_postlude_2, cur_score, None  # <-2  # x (-3+7)

    ...  # Get test input generation script from LLM response  # <-2
    if fatal_error:  # <-2
        return new_state, translate_class_postlude_2, cur_score, None  # <-2  # x (-3+7)

    ...  # Generate test inputs  # <-2
    if fatal_error:  # <-2
        return new_state, translate_class_postlude_2, cur_score, None  # <-2  # x (-3+7)

    # Store newly defined variables to `new_state`
    new_state["stdin_format"] = stdin_format  # +6
    new_state["gen_inputs_code"] = gen_inputs_code  # +6
    new_state["test_inputs"] = test_inputs  # +6
    new_state["fatal_error"] = fatal_error  # +6

    return run_code_and_compare_run_target_source_codes_existing_main(new_state, cur_score, cur_commit)  # +9


def run_code_and_compare_run_target_source_codes_existing_main(cur_state, cur_commit, cur_score):  # +9
    # Get used variables from `cur_state`
    method = cur_state["method"]  # +6
    translation_unit = cur_state["translation_unit"]  # +6
    run_code_log_path = cur_state["run_code_log_path"]  # +6
    dependency_files_str = cur_state["dependency_files_str"]  # +6
    stdin_format = cur_state["stdin_format"]  # +6
    test_inputs = cur_state["test_inputs"]  # +6
    fatal_error = cur_state["fatal_error"]  # +6

    # Directly run codes and compare them if tested method is main function
    ...  # <-1
    match = ...  # <-1

    # Store new variables to `new_state`
    new_state = cur_state.copy()  # +6
    new_state["pass_tests_count"] += float(match)  # +9

    # Compute new score; decide next step
    score = new_state["translate_success_count"] + new_state["pass_tests_count"]  # +11
    return new_state, translate_class_postlude_2, score, None  # <-2  # x (-3+7)


def run_code_and_compare_gen_test_inputs(cur_state, cur_commit, cur_score):  # +9
    # Get used variables from `cur_state`
    method = cur_state["method"]  # +6
    translation_unit = cur_state["translation_unit"]  # +6
    run_code_log_path = cur_state["run_code_log_path"]  # +6
    num_test_inputs = cur_state["num_test_inputs"]  # +6
    target_code_without_main = cur_state["target_code_without_main"]  # +6
    dependency_files_str = cur_state["dependency_files_str"]  # +6
    fatal_error = cur_state["fatal_error"]  # +6

    checkout_new_branch(  # +2
        cur_commit,  # +2
        f"bp-gen_inputs-{translation_unit.target_module_path}-{method}",  # +12
    )  # +1

    # STEP 1: Write test input generation script and generate test inputs

    ...  # Prompt LLM

    ...  # Get test input format specification from LLM response
    if fatal_error:
        commit = git_commit(translation_unit.target_code_root)  # +8
        return cur_state, run_code_and_compare_idle_1, cur_score, commit  # x (-3+7)

    ...  # Get test input generation script from LLM response
    if fatal_error:
        commit = git_commit(translation_unit.target_code_root)  # +8
        return cur_state, run_code_and_compare_idle_1, cur_score, commit  # x (-3+7)

    ...  # Generate test inputs
    if fatal_error:
        commit = git_commit(translation_unit.target_code_root)  # +8
        return cur_state, run_code_and_compare_idle_1, cur_score, commit  # x (-3+7)

    # Store newly defined variables to `new_state`
    new_state = cur_state.copy()  # +6
    new_state["stdin_format"] = stdin_format  # +6
    new_state["gen_inputs_code"] = gen_inputs_code  # +6
    new_state["test_inputs_list"] = test_inputs_list  # +6
    new_state["fatal_error"] = fatal_error  # +6
```

```python

    # Compute score and git commit
    score = new_state["base_score"] + 0.01  # +10
    commit = git_commit(  # +4
        translation_unit.target_code_root,  # +4
        f"Running target code for {translation_unit.target_module_path}:{method}",  # +13
    )  # +1
    return new_state, run_code_and_compare_run_target_code, score, commit  # +8


def run_code_and_compare_idle_1(cur_state, cur_commit, cur_score):  # +9
    translation_unit = cur_state["translation_unit"]  # +6
    checkout_new_branch(cur_commit)  # +4
    commit = git_commit(translation_unit.target_code_root)  # +8
    return cur_state, run_code_and_compare_idle_2, cur_score, commit  # +8


def run_code_and_compare_run_target_code(cur_state, cur_commit, cur_score):  # +9
    # Get used variables from `cur_state`
    method = cur_state["method"]  # +6
    translation_unit = cur_state["translation_unit"]  # +6
    run_code_log_path = cur_state["run_code_log_path"]  # +6
    target_code_with_dummy_main = cur_state["target_code_with_dummy_main"]  # +6
    dependency_files_str = cur_state["dependency_files_str"]  # +6
    stdin_format = cur_state["stdin_format"]  # +6
    gen_inputs_code = cur_state["gen_inputs_code"]  # +6
    test_inputs_list = cur_state["test_inputs_list"]  # +6
    fatal_error = cur_state["fatal_error"]  # +6

    checkout_new_branch(  # +2
        cur_commit,  # +2
        f"bp-run_target-{translation_unit.target_module_path}-{method}",  # +12
    )  # +1

    # Run the target code with the test inputs

    ...  # Prompt LLM

    ...  # Get output format specification from LLM response
    if fatal_error:
        commit = git_commit(translation_unit.target_code_root)  # +8
        return cur_state, run_code_and_compare_idle_2, cur_score, commit  # x (-3+7)

    ...  # Get target main function code from LLM response
    if fatal_error:
        commit = git_commit(translation_unit.target_code_root)  # +8
        return cur_state, run_code_and_compare_idle_2, cur_score, commit  # x (-3+7)

    ...  # Parse target main function code
    if fatal_error:
        commit = git_commit(translation_unit.target_code_root)  # +8
        return cur_state, run_code_and_compare_idle_2, cur_score, commit  # x (-3+7)

    ...  # Extract target main function AST node
    if fatal_error:
        commit = git_commit(translation_unit.target_code_root)  # +8
        return cur_state, run_code_and_compare_idle_2, cur_score, commit  # x (-3+7)

    ...  # Add target main function to target code
    ...  # Run target code with main function on test inputs

    # Store newly defined variables to `new_state`
    new_state = cur_state.copy()  # +6
    new_state["stdout_format"] = stdout_format  # +6
    new_state["target_code_with_main"] = target_code_with_main  # +6
    new_state["run_target_results"] = run_target_results  # +6
    new_state["fatal_error"] = fatal_error  # +6

    # Compute score and git commit
    score = new_state["base_score"] + 0.02  # +10
    commit = git_commit(  # +4
        translation_unit.target_code_root,  # +4
        f"Running source code for {translation_unit.target_module_path}:{method}",  # +13
    )  # +1
    return new_state, run_code_and_compare_run_source_code, score, commit  # +8


def run_code_and_compare_idle_2(cur_state, cur_commit, cur_score):  # +9
    translation_unit = cur_state["translation_unit"]  # +6
    checkout_new_branch(cur_commit)  # +4

    # Store new variables to `new_state`
    new_state = cur_state.copy()  # +6
    new_state["method_idx"] += 1  # +6

    # git commit; decide next step
    if new_state["method_idx"] == len(new_state["methods_to_translate"]):  # +12
        commit = None  # +3
        next_step = translate_class_postlude_1  # +3
    else:  # +2
        new_state["method"] = new_state["methods_to_translate"][new_state["method_idx"]]  # +13
        commit = git_commit(  # +4
```

```python
            translation_unit.target_code_root,  # +4
            f"Begin {translation_unit.source_class_path} translation of {new_state["method"]}.",  # +15
        )  # +1
        next_step = translate_method_and_save  # +3
    return new_state, next_step, cur_score, commit  # +8


def run_code_and_compare_run_source_code(cur_state, cur_commit, cur_score):  # +9
    # Get used variables from `cur_state`
    method = cur_state["method"]  # +6
    translation_unit = cur_state["translation_unit"]  # +6
    run_code_log_path = cur_state["run_code_log_path"]  # +6
    source_code = cur_state["source_code"]  # +6
    target_code_with_main = cur_state["target_code_with_main"]  # +6
    stdin_format = cur_state["stdin_format"]  # +6
    stdout_format = cur_state["stdout_format"]  # +6
    test_inputs_list = cur_state["test_inputs_list"]  # +6
    fatal_error = cur_state["fatal_error"]  # +6

    checkout_new_branch(  # +2
        cur_commit,  # +2
        f"bp-run_source-{translation_unit.target_module_path}-{method}",  # +12
    )  # +1

    # Prepare for next step
    new_state = cur_state.copy()  # +6
    new_state["method_idx"] += 1  # +6

    # Generate source code main function and run it

    ...  # Prompt LLM

    ...  # Get source main function code from LLM response
    if fatal_error:
        # git commit; decide next step
        if new_state["method_idx"] == len(new_state["methods_to_translate"]):  # +12
            commit = None  # +3
            next_step = translate_class_postlude_1  # +3
        else:  # +2
            new_state["method"] = new_state["methods_to_translate"][new_state["method_idx"]]  # +13
            commit = git_commit(  # +4
                translation_unit.target_code_root,  # +4
                f"Begin {translation_unit.source_class_path} translation of {new_state["method"]}.",  # +15
            )  # +1
            next_step = translate_method_and_save  # +3
        return new_state, next_step, cur_score, commit  # x (-3+7)

    ...  # Parse and extract source main function AST node
    if fatal_error:
        # git commit; decide next step
        if new_state["method_idx"] == len(new_state["methods_to_translate"]):  # +12
            commit = None  # +3
            next_step = translate_class_postlude_1  # +3
        else:  # +2
            new_state["method"] = new_state["methods_to_translate"][new_state["method_idx"]]  # +13
            commit = git_commit(  # +4
                translation_unit.target_code_root,  # +4
                f"Begin {translation_unit.source_class_path} translation of {new_state["method"]}.",  # +15
            )  # +1
            next_step = translate_method_and_save  # +3
        return new_state, next_step, cur_score, commit  # x (-3+7)

    ...  # Add target main function to target code
    ...  # Run target code with main function on test inputs

    matches = ...
    match_fraction = sum(matches) / len(matches)

    # Store new variables to `new_state`
    new_state = cur_state.copy()  # +6
    new_state["pass_tests_count"] += match_fraction  # +6

    # Compute new score; git commit; decide next step
    score = new_state["translate_success_count"] + new_state["pass_tests_count"]  # +11
    if new_state["method_idx"] == len(new_state["methods_to_translate"]):  # +12
        commit = None  # +3
        next_step = translate_class_postlude_1  # +3
    else:  # +2
        new_state["method"] = new_state["methods_to_translate"][new_state["method_idx"]]  # +13
        commit = git_commit(  # +4
            translation_unit.target_code_root,  # +4
            f"Begin {translation_unit.source_class_path} translation of {new_state["method"]}.",  # +15
        )  # +1
        next_step = translate_method_and_save  # +3
    return new_state, next_step, score, commit  # x (-1+7)


def translate_class_prelude(cur_state, cur_commit, cur_score):  # +9
    # Get used variables from `cur_state`
    translation_unit = cur_state["translation_unit"]  # +6

    ...  # Some setup (e.g., read and parse code files)
```

```python
    methods_to_translate = ...
    num_methods_to_translate = len(methods_to_translate)
    translate_success_count = 0
    pass_tests_count = 0
    # for method in methods_to_translate:  # -5

    # Store newly defined variables to `new_state`
    new_state = cur_state.copy()  # +6
    new_state["methods_to_translate"] = methods_to_translate  # +6
    new_state["num_methods_to_translate"] = num_methods_to_translate  # +6
    new_state["translate_success_count"] = translate_success_count  # +6
    new_state["pass_tests_count"] = pass_tests_count  # +6
    new_state["method_idx"] = 0  # +6
    new_state["method"] = methods_to_translate[0]  # +9

    # git commit
    commit = git_commit(  # +4
        translation_unit.target_code_root,  # +4
        f"Begin {translation_unit.source_class_path} translation of {new_state["method"]}.",  # +15
    )  # +1
    return new_state, translate_method_and_save, cur_score, commit  # +8


def translate_method_and_save(cur_state, cur_commit, cur_score):  # +9
    # Get used variables from `cur_state`
    method = cur_state["method"]  # +6
    translation_unit = cur_state["translation_unit"]  # +6
    source_code = cur_state["source_code"]  # +6
    target_code = cur_state["target_code"]  # +6
    translate_success_count = cur_state["translate_success_count"]  # +6
    pass_tests_count = cur_state["pass_tests_count"]  # +6

    checkout_new_branch(  # +2
        cur_commit,  # +2
        f"bp-translate-{translation_unit.source_class_path}-{method}",  # +12
    )  # +1

    new_state = cur_state.copy()  # +6

    target_code, translate_success = translate_method(method, target_code, source_code, translation_unit)  # <-1
    if translate_success:  # <-1
        translate_success_count += 1  # <-1
        score = translate_success_count + pass_tests_count  # +5

        ...  # save target code  # <-1

        if translation_unit.is_test:  # <-1
            pass_tests_count += run_test_module(target_code, translation_unit)  # <-1
        else:  # <-1
            # pass_tests_count += run_code_and_compare(  # -4
            #     method,  # -2
            #     target_code,  # -2
            #     source_code,  # -2
            #     translation_unit,  # -2
            # )  # -1

            # Store newly defined variables to `new_state`
            new_state["base_score"] = translate_success_count + pass_tests_count  # +8

            return run_code_and_compare_prelude(new_state, cur_score, cur_commit)  # +9

    # Store new variables to `new_state`
    new_state["method_idx"] += 1  # +6

    # git commit; decide next step
    if new_state["method_idx"] == len(new_state["methods_to_translate"]):  # +12
        commit = None  # +3
        next_step = translate_class_postlude_1  # +3
    else:  # +2
        new_state["method"] = new_state["methods_to_translate"][new_state["method_idx"]]  # +13
        commit = git_commit(  # +4
            translation_unit.target_code_root,  # +4
            f"Begin {translation_unit.source_class_path} translation of {new_state["method"]}.",  # +15
        )  # +1
        next_step = translate_method_and_save  # +3
    return new_state, next_step, cur_score, commit  # +8


def translate_class_postlude_1(cur_state, cur_commit, cur_score):  # +9
    # Get used variables from `cur_state`
    translation_unit = cur_state["translation_unit"]  # +6
    target_code = cur_state["target_code"]  # +6
    translate_success_count = cur_state["translate_success_count"]  # +6
    pass_tests_count = cur_state["pass_tests_count"]  # +6
    num_methods_to_translate = cur_state["num_methods_to_translate"]  # +6

    new_state = state.copy()  # +6

    # Separately test main function (Python `if __name__ == "__main__"` block) if it's present
    if not translation_unit.is_test and ...:
        num_methods_to_translate += 1
        translate_success_count += 1
```

```python

        # Store newly defined variables to `new_state`
        new_state["num_methods_to_translate"] = num_methods_to_translate  # +6
        new_state["translate_success_count"] = translate_success_count  # +6
        new_state["method"] = "main"  # +8
        new_state["base_score"] = translate_success_count + pass_tests_count  # +8

        return run_code_and_compare_prelude(new_state, cur_score, cur_commit)  # +9

    return translate_class_postlude_2(new_state, cur_commit, cur_score)  # +9


def translate_class_postlude_2(cur_state, cur_commit, cur_score):  # +9
    # Get used variables from `cur_state`
    translation_unit = cur_state["translation_unit"]  # +6
    translate_success_count = cur_state["translate_success_count"]  # +6
    pass_tests_count = cur_state["pass_tests_count"]  # +6
    num_methods_to_translate = cur_state["num_methods_to_translate"]  # +6

    ...  # logging and saving progress

    return_value = (pass_tests_count, translate_success_count, num_methods_to_translate, new_branch)  # x (see below)
    return return_value, None, cur_score, None  # x (-0+11)


def translate_class(translation_unit, beam_width, branching):  # x (-0+4)
    # Use beam search to translate a class method-by-method

    init_state = {"translation_unit": translation_unit}  # +7
    init_step = translate_class_prelude  # +3
    init_commit = None  # +3
    init_score = 0.0  # +5

    beam = [init_step(init_state, init_commit, init_score)]  # +11
    results = []  # +3
    while len(beam) > 0:  # +8
        new_program_states_list = []  # +3
        for state, step, score, commit in beam:  # +11
            new_program_states = [step(state, commit, score) for _ in range(branching)]  # +18
            new_program_states.sort(key=lambda x: x[2], reverse=True)  # +17
            new_program_states_list.append(new_program_states)  # +6
        not_done_new_program_states = []  # +3
        for i in range(len(new_program_states_list[0])):  # +11
            # random permutation of indices to break ties
            for j in np.random.permutation(len(new_program_states_list)):  # +13
                new_program_state = new_program_states_list[j][i]  # +8
                new_state, new_step, new_score, new_commit = new_program_state  # +9
                if new_step is None:  # +5
                    results.append((new_state, new_score))  # +8
                else:  # +2
                    not_done_new_program_states.append(new_program_state)  # +6
        not_done_new_program_states.sort(  # +4
            key=lambda program_state: program_state.score, reverse=True  # +12
        )  # +1
        beam = not_done_new_program_states[:beam_width]  # +6
    return results  # +2


def setup_antlr4_prelude(cur_state, cur_commit, cur_score):  # x (-6+6)
    # Get used variables from `cur_state`
    source_code_root = cur_state["source_code_root"]  # +6

    new_state = cur_state.copy()  # +6

    source_subdir = 'src/main/antlr4'
    new_state["num_successful_translations"] = 0  # +3
    new_state["num_successful_parses"] = 0  # +3
    new_state["root_dirs_files_list"] = list(os.walk(source_code_root / source_subdir))  # x (-8+8)
        # for file in files:  # -5

    new_state["root_dirs_files_idx"] = 0  # +6
    new_state["file_idx"] = 0  # +6

    root, dirs, files = new_state["root_dirs_files_list"][new_state["root_dirs_files_idx"]]  # +14
    file = files[new_state["file_idx"]]  # +8

    ...  # Read antlr4 grammar file  # <-2

    # Store newly defined variables to `new_state`
    new_state["source_file_path"] = source_file_path  # +6
    new_state["grammar_content"] = grammar_content  # +6

    # git commit
    commit = git_commit(  # +4
        target_code_root,  # +2
        f"Translate antlr4 grammar {source_file_path.stem}",  # +10
    )  # +1

    return new_state, setup_antlr4_body, cur_score, commit  # +8


def setup_antlr4_body(cur_state, cur_commit, cur_score):  # +9
```

```python
    # Get used variables from `cur_state`
    source_code_root = cur_state["source_code_root"]  # +6
    target_code_root = cur_state["target_code_root"]  # +6
    temperature = cur_state["temperature"]  # +6
    num_successful_translations = cur_state["num_successful_translations"]  # +6
    num_successful_parses = cur_state["num_successful_parses"]  # +6
    root, dirs, files = cur_state["root_dirs_files_list"][cur_state["root_dirs_files_idx"]]  # +14
    file = files[cur_state["file_idx"]]  # +8
    source_file_path = cur_state["source_file_path"]  # +6
    grammar_content = cur_state["grammar_content"]  # +6

    checkout_new_branch(  # +2
        cur_commit,  # +2
        f"bp-translate_antlr4_grammar-{source_file_path.stem}",  # +10
    )  # +1

    ...  # LLM modification if needed  # <-2

    ...  # Write to target directory  # <-2

    ...  # Run antlr4 to generate target Python classes  # <-2

    ...  # Check if the generated files can be parsed  # <-2

    cur_score = num_successful_translations + num_successful_parses  # +5

    new_state = cur_state.copy()  # +6

    # Increment to next loop iteration
    new_state["root_dirs_files_idx"] += 1  # +6
    new_state["file_idx"] += 1  # +6
    if new_state["file_idx"] == len(files):  # +10
        # Inner for loop completed -- increment outer for loop index
        new_state["root_dirs_files_idx"] += 1  # +6
        new_state["file_idx"] = 0  # +6
        if new_state["root_dirs_files_idx"] == len(new_state["root_dirs_files_list"]):  # +12
            # Outer for loop completed -- return to code repo translation agent
            new_state["total_score"] = num_successful_translations + num_successful_parses  # x (see below)
            return code_translation_agent_prelude_2(new_state, cur_commit, cur_score)  # x (-0+13)

    root, dirs, files = new_state["root_dirs_files_list"][new_state["root_dirs_files_idx"]]  # +14
    file = files[new_state["file_idx"]]  # +8

    ...  # Read antlr4 grammar file  # <-2

    # Store newly defined variables to `new_state`
    new_state["source_file_path"] = source_file_path  # +6
    new_state["grammar_content"] = grammar_content  # +6

    # git commit
    commit = git_commit(  # +4
        target_code_root,  # +2
        f"Translate antlr4 grammar {source_file_path.stem}",  # +10
    )  # +1

    return new_state, setup_antlr4_body, cur_score, commit  # +8


def code_translation_agent_prelude_1(cur_state, cur_commit, cur_score):  # +9
    # Get used variables from `cur_state`
    source_code_root = cur_state["source_code_root"]  # +6
    target_code_root = cur_state["target_code_root"]  # +6

    ...  # Set up logging and git repo for saving progress

    # 0.1. Copy resource files (src/main/resources and src/test/resources)
    copy_resource_files(source_code_root, target_code_root)

    # Store newly defined variables to `new_state`
    new_state = cur_state.copy()  # +6
    new_state["repo"] = repo  # +6
    new_state["results"] = results  # +6
    new_state["temperature"] = cur_state["args"].temperature  # +10

    return setup_antlr4_prelude(new_state, cur_commit, cur_score)


def code_translation_agent_prelude_2(cur_state, cur_commit, cur_score):  # +9
    # Get used variables from `cur_state`
    source_code_root = cur_state["source_code_root"]  # +6
    target_code_root = cur_state["target_code_root"]  # +6

    # Get class names in topological order
    translation_units = get_translation_order_and_dependencies(source_code_root, target_code_root)

    # Store newly defined variables to `new_state`
    new_state = cur_state.copy()  # +6
    new_state["translation_units"] = translation_units  # +6
    new_state["translation_unit_idx"] = 0  # +6

    # git commit
    translation_unit = new_state["translation_units"][new_state["translation_unit_idx"]]  # +10
```

```python
    commit = git_commit(  # +4
        translation_unit.target_code_root,  # +4
        f"Begin {translation_unit.source_class_path} translation.",  # +10
    )  # +1
    return new_state, code_translation_agent_generate_stubs, cur_score, commit  # +8


def code_translation_agent_generate_stubs(cur_state, cur_commit, cur_score):  # +9
    # Get used variables from `cur_state`
    translation_unit = cur_state["translation_units"][cur_state["translation_unit_idx"]]  # +10
    total_score = cur_state["total_score"]  # +6

    checkout_new_branch(  # +2
        cur_commit,  # +2
        f"bp-translate-{translation_unit.source_class_path}",  # +10
    )  # +1

    # Generate stubs for the class
    generate_stubs_success = generate_stubs(translation_unit)  # <-1

    total_score += generate_stubs_success  # +3

    # Store newly defined variables to `new_state`
    new_state = cur_state.copy()  # +6
    new_state["generate_stubs_success"] = generate_stubs_success  # +6
    new_state["total_score"] = total_score  # +6

    # git commit
    commit = git_commit(translation_unit.target_code_root)  # +8
    return new_state, CodeTranslationAgentTranslateClass(2, 2), total_score, commit  # +12


class CodeTranslationAgentTranslateClass:  # +3
    def __init__(self, beam_width, branching):  # +9
        self.beam_width = beam_width  # +5
        self.branching = branching  # +5

        self.called = False  # +5

    def __call__(self, cur_state, cur_commit, cur_score):  # +11
        # Get used variables from `cur_state`
        target_code_root = cur_state["target_code_root"]  # +6
        translation_unit = cur_state["translation_unit"]  # +6
        total_score = cur_state["total_score"]  # +6
        generate_stubs_success = cur_state["generate_stubs_success"]  # +6

        if not self.called:  # +6
            checkout_new_branch(cur_commit)  # +4

            self.translate_class_results = translate_class(translation_unit, beam_width=self.beam_width, default_branching=
      self.branching)  # x (-0+20)
            self.output_idx = 0  # +5
            self.called = True  # +5

        (pass_tests_count, translate_success_count, num_methods_to_translate, new_branch), _ = self.translate_class_results[
      self.output_idx]  # x (see above)

        # "+1" to prevent agent from "cheating" (have very few e.g. zero stubs to implement)
        total_score += pass_tests_count / (num_methods_to_translate + 1)  # +9

        ...  # Log results

        # Increment result_idx
        self.output_idx += 1  # +5

        # Store new variables to `new_state`
        new_state = cur_state.copy()  # +6
        new_state["total_score"] = total_score  # +6
        new_state["results"] = results  # +6
        # for translation_unit in translation_units:  # -5
        new_state["translation_unit_idx"] += 1  # +6

        # git commit; decide next step
        if new_state["translation_unit_idx"] == len(new_state["translation_units"]):  # +12
            commit = None  # +3
            next_step = code_translation_agent_postlude  # +3
        else:  # +2
            new_translation_unit = new_state["translation_units"][new_state["translation_unit_idx"]]  # +10
            commit = git_commit(  # +4
                translation_unit.target_code_root,  # +4
                f"Begin {new_translation_unit.source_class_path} translation.",  # +10
            )  # +1
            next_step = code_translation_agent_generate_stubs  # +3
        return new_state, next_step, total_score, commit  # +8


def code_translation_agent_postlude(cur_state, cur_commit, cur_score):  # +9
    # Use beam search to translate a repository
    target_code_root = cur_state["target_code_root"]  # +6
    repo = cur_state["repo"]  # +6

    ...  # Final logging and saving
```

```
    return_value = final_commit  # x (see below)
    return return_value, None, cur_score, None  # x (-0+8)


def code_translation_agent(source_code_root, target_code_root, args, beam_width, default_branching):  # x (-0+4)
    # Use beam search to translate a class method-by-method

    init_state = {  # +3
        "source_code_root": source_code_root,  # +5
        "target_code_root": target_code_root,  # +5
        "args": args,  # +5
    }  # +1
    init_step = code_translation_agent_prelude_1  # +3
    init_commit = None  # +3
    init_score = 0.0  # +5

    beam = [init_step(init_state, init_commit, init_score)]  # +11
    results = []  # +3
    while len(beam) > 0:  # +8
        new_program_states_list = []  # +3
        for state, step, score, commit in beam:  # +11
            branching = default_branching if not isinstance(step, CodeTranslationAgentTranslateClass) else step.branching *
      step.beam_width  # +19
            new_program_states = [step(state, commit, score) for _ in range(branching)]  # +18
            new_program_states.sort(key=lambda x: x[2], reverse=True)  # +17
            new_program_states_list.append(new_program_states)  # +6
        not_done_new_program_states = []  # +3
        for i in range(len(new_program_states_list[0])):  # +11
            # random permutation of indices to break ties
            for j in np.random.permutation(len(new_program_states_list)):  # +13
                new_program_state = new_program_states_list[j][i]  # +8
                new_state, new_step, new_score, new_commit = new_program_state  # +9
                if new_step is None:  # +5
                    results.append((new_state, new_score))  # +8
                else:  # +2
                    not_done_new_program_states.append(new_program_state)  # +6
        not_done_new_program_states.sort(  # +4
            key=lambda program_state: program_state.score, reverse=True  # +12
        )  # +1
        beam = not_done_new_program_states[:beam_width]  # +6
    return max(results, key=lambda x: x[2])[0]  # +16


code_translation_agent(..., beam_width=3, default_branching=3)  # x (-0+8)
```

Listing 20: Beam search implemented in plain Python

D.2 Case Study 2: Hypothesis Search Agent

Base agent:

```python
def two_step_agent(task_info):
    # Step 1: Get natural language hypothesis
    ...
    hypothesis = hypothesis_agent([task_info], hypothesis_instruction)

    # Step 2: Implement the hypothesis in code
    ...
    code = solver_agent([task_info, hypothesis], solver_instruction)
    return get_test_output(code)


two_step_agent(task_info)
```

Listing 21: Simple 2-step agent for ARC (base)

With EnCompass*:*

```python
import encompass  # +2


@encompass.compile  # +4
def two_step_agent(task_info):
    branchpoint()  # +2
    # Step 1: Get natural language hypothesis
    ...
    hypothesis = hypothesis_agent([task_info], hypothesis_instruction)

    branchpoint()  # +2
    # Step 2: Implement the hypothesis in code
    ...
    code = solver_agent([task_info, hypothesis], solver_instruction)

    # Evaluate
    percent_correct = run_validation(code)  # +6
    record_score(percent_correct)  # +4
    if percent_correct == 1:  # +5
        early_stop_search()  # +2

    return get_test_output(code)


two_step_agent(task_info).search("parallel_bfs", default_branching=8)  # x (-0+9)
```

Listing 22: Parallelized BFS in EnCompass, 2 branchpoints

Without EnCompass*:* The code devoted to parallelization obscures the underlying agent logic.

```python
from concurrent.futures import ThreadPoolExecutor, as_completed  # +8


def two_step_agent(task_info, branching):  # x (-0+2)
    results = []  # +3
    full_solved = False  # +3

    with ThreadPoolExecutor() as executor:  # +6

        def run_one_forward_pass():  # +3
            if full_solved:  # +3
                return  # +1
            # Step 1: Get natural language hypothesis
            ...  # ->2
            hypothesis = hypothesis_agent([task_info], hypothesis_instruction)  # ->2

            def implement_in_code():  # +3
                nonlocal full_solved  # +2

                if full_solved:  # +3
                    return  # +1

                # Step 2: Implement the hypothesis in code
                ...  # ->3
                code = solver_agent([task_info, hypothesis], solver_instruction)  # ->3

                # Evaluate
                percent_correct = run_validation(code)  # +6
                if percent_correct == 1:  # +5
                    full_solved = True  # +3
                results.append((get_test_output(code), percent_correct))  # x (-1+7)

            futures = [executor.submit(implement_in_code) for _ in range(branching)]  # +16
            for future in as_completed(futures):  # +7
                future.result()  # +4

        futures = [executor.submit(run_one_forward_pass) for _ in range(branching)]  # +16
        for future in as_completed(futures):  # +7
            future.result()  # +4

    return max(results, key=lambda x: x[1])[0]  # +16


two_step_agent(task_info, branching=8)  # x (-0+4)
```

Listing 23: Parallelized BFS implemented in plain Python

D.3 Case Study 3: Reflexion Agent

Base agent:

```python
def reflexion_agent(task_info, internal_tests, max_iters):
    # first attempt
    code = solver_agent(task_info)
    percent_correct, feedback = run_validation(code, internal_tests)

    # if solved, exit early
    if percent_correct == 1.0:
        return code

    for cur_iter in range(1, max_iters):
        # self-reflect and apply to next attempt
        reflection = self_reflection_agent(code, feedback)
        code = solver_agent(task_info, code, feedback, reflection)
        percent_correct, feedback = run_validation(code, internal_tests)

        # if solved, exit early
        if percent_correct == 1.0:
            return code

    return code


reflexion_agent(...)
```

Listing 24: Reflexion agent (base)

With ENCOMPASS*:*

```python
import encompass  # +2


@encompass.compile  # +4
def reflexion_agent(task_info, internal_tests, max_iters):
    record_score(0.2)  # +6
    branchpoint()  # +2
    # first attempt
    code = solver_agent(task_info)
    percent_correct, feedback = run_validation(code, internal_tests)
    record_score(percent_correct)  # +4
    optional_return(code)  # +4

    # if solved, exit early
    if percent_correct == 1.0:
        early_stop_search()  # x (-2+2)

    for cur_iter in range(1, max_iters):
        branchpoint()  # +2
        # self-reflect and apply to next attempt
        reflection = self_reflection_agent(code, feedback)
        code = solver_agent(task_info, code, feedback, reflection)
        percent_correct, feedback = run_validation(code, internal_tests)
        record_score(percent_correct)  # +4
        optional_return(code)  # +4

        # if solved, exit early
        if percent_correct == 1.0:
            early_stop_search()  # x (-2+2)

    return code


reflexion_agent(...).search("reexpand_best_first", max_num_results=5)  # x (-0+9)
```

Listing 25: Reexpand best-first search in ENCOMPASS, 2 branchpoints

Without ENCOMPASS*:* Defining separate actions for search obscures the ordering of actions.

```python
from queue import PriorityQueue  # +4


def get_initial_attempt(task_info, internal_tests, max_iters):  # +9
    # first attempt
    code = solver_agent(task_info)
    percent_correct, feedback = run_validation(code, internal_tests)

    # if solved, exit early
    if percent_correct == 1.0:
        early_stop = True  # x (-2+3)

    next_step = do_one_reflexion  # +3
    return next_step, early_stop, percent_correct, code, feedback, 1  # +12


def do_one_reflexion(task_info, internal_tests, max_iters, code, feedback, cur_idx):  # +15
    # self-reflect and apply to next attempt
    reflection = self_reflection_agent(code, feedback)  # <-1
    code = solver_agent(task_info, code, feedback, reflection)  # <-1
    percent_correct, feedback = run_validation(code, internal_tests)  # <-1

    # if solved, exit early
    if percent_correct == 1.0:  # <-1
        early_stop = True  # <-1  # x (-2+3)

    next_idx = cur_idx + 1  # x (-8+4)
    next_step = None if next_idx == max_iters else do_one_reflexion  # +9
    return next_step, early_stop, percent_correct, code, feedback, next_idx  # +12


# Apply best-first search choosing the highest-scoring state
# to apply an action
def reflexion_agent(task_info, internal_tests, max_iters, max_num_results):  # x (-0+2)
    init_program_state = ()  # +3
    init_step = get_initial_attempt  # +3
    program_states_to_expand = PriorityQueue()  # +4
    program_states_to_expand.put((init_step, init_program_state))  # +8
    percent_correct = None  # +3
    finished = False  # +3
    num_results = 0  # +3
    results = []  # +3
    while not program_states_to_expand.empty() and not finished:  # +10
        step, program_state = program_states_to_expand.pop()  # +8
        program_states_to_expand.put(program_state)  # put it back  # +6
        next_step, early_stop, percent_correct, code, feedback, next_idx = step(task_info, internal_tests, max_iters, *program_state)  # +23
        results.append((code, percent_correct))  # +8
        if early_stop:  # +3
            break  # +1
        if next_step is not None:  # +6
            program_states_to_expand.put((next_step, (code, feedback, next_idx)))  # +13
        num_results += 1  # +3
        if num_results >= max_num_results:  # +5
            break  # +1
    return max(results, key=lambda x: x[1])[0]  # x (-1+15)


reflexion_agent(..., max_num_results=5)  # x (-0+4)
```

Listing 26: Reexpand best-first search implemented in plain Python