Oracular Programming: A Modular Foundation for Building LLM-Enabled Software

Jonathan Laurent

CMU-CS-26-108

April 2026

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
André Platzer (Chair)
Marijn Heule
Zico Kolter
Armando Solar-Lezama (MIT)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work was sponsored by the National Science Foundation under award numbers CNS-1657530 and CNS-1739629, the United States Air Force under award numbers FA95501610288 and F875018C0092, the United States Army under award number W911NF1710073, and the Alexander von Humboldt Professorship program. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Large Language Models, Language Design, Nondeterministic Programming, Reinforcement Learning, AlphaZero, Theorem Proving

For Adeline

Abstract

Large Language Models (LLMs) can solve previously intractable tasks given only natural-language instructions and a few examples, but they remain difficult to steer and lack a key capability for building reliable software at scale: the modular composition of computations under enforceable contracts. As a result, they are often embedded in larger software pipelines that use domain knowledge to decompose tasks and improve reliability through validation and search. Yet the complexity of writing and maintaining such pipelines has so far limited their sophistication.

We propose *oracular programming*: a foundational paradigm for integrating traditional, explicit computations with inductive oracles such as LLMs. It rests on two directing principles: the full separation of *core* and *search* logic (allowing the latter to freely evolve without breaking the former), and the treatment of few-shot examples as *grounded* and *evolvable* program components (allowing their consistency with the rest of the program to be enforced through its evolution, with breakages easily identifiable and repairable). Within this paradigm, programmers express high-level problem-solving strategies as programs with unresolved choice points. These choice points are resolved at runtime by LLMs, which generalize from user-provided examples of correct and incorrect decisions. An *oracular program* is composed of three orthogonal components: a *strategy* that consists of a nondeterministic program with choice points that can be reified into a search tree, a *policy* that specifies how to navigate this tree with the help of LLM oracles, and a set of *demonstrations* that describe successful and unsuccessful tree navigation scenarios across diverse problem instances. Each component is expressed in a dedicated language.

We address the key programming language design challenges of modularly composing oracular programs and enforcing consistency between their components as they evolve. We also demonstrate universal self-improvement mechanisms for oracular programs, in which training and tuning data is automatically extracted from successful and unsuccessful runs. Finally, we present *Delphyne*, an open-source framework for oracular programming based on Python, and empirically evaluate our approach through several case studies.

Acknowledgments

I am deeply grateful to my advisor, André Platzer, for his openness, unwavering support, and for being a scientific role model through his remarkable intellectual breadth and ambition. I would also like to thank Jean Yang, my first advisor at Carnegie Mellon University, who taught me a great deal about public speaking and whose kindness and enthusiasm I remember fondly. My thoughts also go to Walter Fontana, who mentored and supported me at the beginning of my PhD and instilled in me an appreciation for slow thinking and for asking big questions.

I would like to express my sincere thanks to my thesis committee members, Armando Solar-Lezama, Zico Kolter, and Marijn Heule, for their thoughtful and stimulating feedback. My work was also nourished by valuable scientific exchanges with collaborators and researchers, in particular Aditi Kabra, Rose Bohrer, Nathan Fulton, Stefan Mitsch, Matt Fredrikson, Frank Pfenning, Alwyn Goodloe, Hector Medina Abarca, Jean Krivine, and Jérôme Feret.

I am grateful to my dear friend Léonard Blier for our frequent and unending conversations on science, life, and politics, and for his advice and support. I am also grateful to Théis Bazin, Michael Coblenz, and all my friends and colleagues from the computer science department at Carnegie Mellon University and the *Logic of Autonomous Dynamical Systems* group at the Karlsruhe Institute of Technology.

My thanks also go to my brother, Mickaël Laurent, a formidable programmer, scientist, and human being, who makes me very proud and with whom I have shared many conversations. I am also indebted to my parents, Muriel and Robert Laurent, for their constant love and support. Finally, my deepest gratitude goes to my wife, Adeline, for her love, her patience, and for standing by me through every step of this journey. To our children, Mélodie, Éden, and Ernest, thank you for filling our lives with joy and light.

Contents

List of Figures

List of Tables

1 Introduction

Large Language Models (LLMs) have introduced a fundamentally new form of inductive computation, leveraging common-sense knowledge and generalization to perform tasks implicitly specified through natural-language instructions and examples. As a consequence, they have significantly widened the range of problems amenable to computational automation and led to breakthroughs across a wide range of areas, including reasoning-intensive domains such as program synthesis [64], mathematical problem solving [62], and formal theorem proving [32, 46].

Yet, despite these successes, LLMs remain difficult to steer precisely and lack a capability fundamental to building reliable software at scale: the modular composition of computations under enforceable contracts. As a result, LLMs are often deployed as components within larger software pipelines that leverage domain-specific knowledge to decompose complex tasks [80], provide intermediate feedback [103, 104], and improve reliability through validation and search [62, 104]. In turn, building such pipelines raises novel engineering challenges, including the tuning of prompts and search parameters and the writing and maintenance of high-quality examples. These challenges have limited the complexity and robustness of existing LLM-enabled software.

This thesis proposes a new programming paradigm for writing software that combines implicit, inductive computations performed by LLM oracles and traditional, explicit computations. In this paradigm, which we name *oracular programming*, high-level problem-solving strategies can be expressed as *nondeterministic programs*, whose choice points are resolved at runtime by oracles that generalize from examples of correct and incorrect decisions. To make this idea concrete, we begin with an illustrative example before presenting the key principles and contributions underlying oracular programming in more abstract terms.

1.1 Illustrative Example

Let us illustrate oracular programming on a simple yet complete example. Our goal is to solve the following toy task: given a mathematical expression $F_n(x)$, parameterized by an integer

```python
import sympy as sp
import delphyne as dp
from delphyne import Branch, Fail, IPDict, Strategy, strategy

@strategy
def find_param_value(expr: str) -> Strategy[Branch|Fail, IPDict, int]:
    """
    Find an integer `n` that makes a given math expression nonnegative
    for all real `x`. Prove that the resulting expression is nonnegative
    by rewriting it into an equivalent form.
    """
    x, n = sp.Symbol("x", real=True), sp.Symbol("n")
    symbs = {"x": x, "n": n}
    try:
        n_val = yield from dp.guess(int, using=[expr])
        expr_sp = sp.parse_expr(expr, symbs).subs({n: n_val})
        equiv = yield from dp.guess(str, using=[str(expr_sp)])
        equiv_sp = sp.parse_expr(equiv, symbs)
        equivalent = (expr_sp - equiv_sp).simplify() == 0
        yield from dp.ensure(equivalent, "not_equivalent")
        yield from dp.ensure(equiv_sp.is_nonnegative, "not_nonneg")
        return n_val
    except Exception as e:
        yield from dp.ensure(False, "sympy_error", message=str(e))
```

Figure 1.1: A Simple Strategy Written in Delphyne. SymPy [68] is used to check expression equivalence (via simplification) and nonnegativity (via a simple interval arithmetic check). The `guess` operator is built on top of a more primitive operator named `branch` (see Section 4.3.3).

n and a real variable x, find a value of n such that $F_n(x) \geq 0$ for all $x \in \mathbb{R}$. For instance, if $F_n(x) = x^2 - 2x + n$, then $n = 1$ is a correct answer because $F_1(x) = (x-1)^2$, whereas $n = 0$ is not because $F_0(1) < 0$. A natural high-level strategy for solving this task is to *(i)* guess a candidate value of n, *(ii)* substitute it into F_n, and *(iii)* rewrite the resulting expression into a form whose nonnegativity can be established easily. Importantly, such a *strategy* can be expressed as a *nondeterministic program*, in which difficult decisions are left unspecified and delegated to runtime oracles.

We show an example of such a program in Figure 1.1, expressed in *Delphyne*, our open-source framework for *oracular programming* (Chapter 4). It can be read as a standard Python program that uses two additional primitives (underlined): `guess`, for nondeterministically generating an object of a given type, and `ensure`, for asserting that a given property holds. The program induces a search tree in which infinitely branching internal nodes correspond to calls to `guess`, failure leaves correspond to failed `ensure` assertions, and success leaves correspond

to successful program termination. Such a tree can be explored in various ways using LLM oracles for guidance, as defined by a *policy*.

An example policy is shown in Figure 1.2, which uses sequential depth-first search (`dfs`), making at most two proof attempts for every parameter guess, sampling branching candidates using OpenAI's GPT-5-mini model, and consuming a total inference budget of at most 10^{-4} dollars. By default, prompts are automatically generated for each LLM request. They include the source code of the strategy shown in Figure 1.1, the identifier of the current choice point, and the values of local expressions provided as context to `guess` via its `using` argument. Other policies are possible, making different trade-offs in terms of latency, parallelism, reliability, and cost. For example, multiple answers can be sampled at once for each choice point (only paying for input tokens once) and the associated continuations explored in parallel, or an ensemble of models can be used for increased diversity. Importantly, Delphyne enables a full separation between *strategies* and *policies*, allowing programmers to effectively optimize the latter without ever breaking the former.

```
def serial_policy():
    model = dp.standard_model("gpt-5-mini")
    budget = dp.BudgetLimit({dp.DOLLAR_PRICE: 1e-4})
    return dp.with_budget(budget) @ dp.dfs() & {
        "n_val": dp.few_shot(model),
        "equiv": dp.take(2) @ dp.few_shot(model)}
```

Figure 1.2: A Policy for the Strategy from Figure 1.1.

LLM oracles often perform best when given concrete examples of correct and incorrect decisions. We introduce a dedicated *demonstration language* for writing and maintaining such examples, which supports an interactive, test-driven development workflow. Figure 1.3 illustrates this workflow: a user interactively solves a specific problem instance, providing examples of correct and incorrect decisions at each step, possibly with explanations. Demonstrations are kept consistent with the underlying strategy via unit tests, so that broken examples can be automatically detected and repaired interactively. Once enough demonstrations are written by humans to bootstrap the LLM oracles, further demonstrations can be automatically extracted from program executions, enabling self-improvement.

Comparison with ReAct agentic frameworks. Oracular programming encourages a style of programming that is dual and complementary to the increasingly pervasive ReAct-style [103] agentic framework exemplified by systems such as Codex or Claude Code, in which a large reasoning model with an append-only context orchestrates traditional computations via tool calls. In contrast, our example showcases a traditional program that orchestrates LLM queries, trading off generality for tighter control over computation, inference cost, parallelism, and security (see Chapter 6 for a detailed discussion). We do not argue for one mode to supplant the other, but rather for the two to be tightly integrated and woven together, as oracular programming explicitly allows.

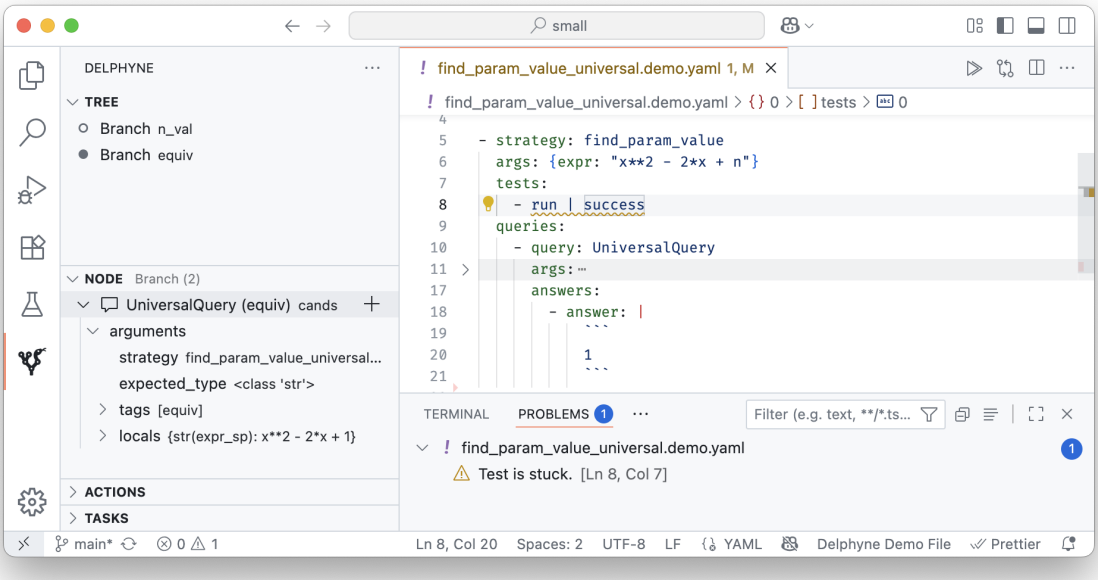


Figure 1.3: Interactively Writing a Demonstration.

1.2 Principles of Oracular Programming

Equipped with a concrete example, we can now motivate and state the core principles of *oracular programming* in more general terms.

As we have argued, despite their impressive successes, LLMs remain difficult to steer precisely and lack a capability for modularly composing computations under enforceable contracts. As a result, they are often deployed as components within larger software pipelines. At first glance, developing such pipelines may seem deceptively simple. After all, LLMs can be prompted through straightforward APIs, and developers have access to the full arsenal of existing programming languages and software-engineering tools. Yet, as we argue, building reliable LLM-enabled software raises challenges that are not properly addressed by existing programming abstractions.

Effectively Tuning Search Logic. LLMs offer a powerful but inherently unreliable programming primitive, making the ubiquitous use of *search* and *validation* essential for dependability. At the same time, prompting LLMs is expensive, so retries and backtracking incur significant costs. Resolving this tension requires careful tuning of this search logic, which must also adapt to user-specific constraints (e.g., inference budget, latency tolerance). Evolving this logic can require costly and frequent refactorings, especially when it is intertwined with the higher-level logic that governs how large problems are decomposed into smaller subproblems and prompts.

Writing and Evolving Relevant Examples. Prompting language models often works best when examples of successfully solving similar task instances are provided (i.e., *few-shot prompting* [10]). Such examples are therefore crucial components of LLM-enabled programs, but writing them can be time-consuming. In addition, these examples need to be kept synchronized with the rest of the program: any change can silently render some

examples obsolete and necessitate new ones. While numerous techniques exist for maintaining correctness and consistency in traditional programs—most notably type systems, contracts, and testing—LLM-enabled programs present a unique and largely unaddressed challenge. Yet, these programs stand to gain disproportionately from *fearless refactoring*, since the inherent opacity and unpredictability of LLMs demand frequent iteration and rapid development cycles.

Together, these challenges have limited the complexity and robustness of existing LLM-enabled software. We tackle them through principled language design, introducing a new paradigm for integrating inductive oracles in traditional programs while enforcing strong modularity, consistency, and evolvability properties. This paradigm, which we call *oracular programming*, rests on several interlocking principles and contributions, outlined below.

Separating Core and Search logic. Existing LLM-enabled programs often conflate two distinct kinds of logic: *(i)* the *core logic*, which describes how complex problems can be recursively decomposed into LLM queries and validation contracts; and *(ii)* the *search logic*, which specifies how those queries are answered and how the resulting search space is explored (e.g., sequentially or in parallel, using depth-first search or Monte Carlo Tree Search). The former arises naturally from the expression of domain-specific knowledge, whereas the latter typically demands heavier iteration and tuning. A foundational principle of oracular programming is the *complete separation* of these two kinds of logic into orthogonal components, which we call *strategies* and *policies* respectively. This separation enables *vastly different* search algorithms and prompting techniques to be explored concurrently, without requiring *any* modification to the program's core logic. Crucially, achieving such separation while supporting a wide range of search algorithms—which no existing framework does—is incompatible with treating LLM queries as ordinary function calls. Instead, we treat them as *nondeterministic assignments*. We define *strategies* as nondeterministic programs that can be reified into search trees (where choice points induce branching nodes and contract violations yield failure leaves) and *policies* as generic functions that manipulate these trees.

A Modular and Extensible Strategy Language. We define a *strategy language* for expressing high-level problem-solving strategies as nondeterministic programs that can be reified into search trees. Its design is guided by two key requirements of *extensibility* and *modularity*, which drive innovations well beyond traditional formulations of nondeterministic programming [33]. The *extensibility* requirement arises from the diversity of existing search algorithms, which often exploit specific structure and annotations within search trees to improve efficiency (e.g., independent subgoals, quantitative value estimates). Our strategy language is therefore equipped with an *extensible effect system* that makes it easy to define new tree types. The *modularity* requirement mandates that *(i)* heterogeneous strategies producing different types of trees can be composed while keeping their corresponding policies independent, and *(ii)* any LLM query can be *locally* and *transparently* refined into a dedicated sub-strategy without impacting its parent strategy or any associated policy. We meet these goals by introducing the concept of *opaque space* that unifies strategies and queries from a policy's perspective, together with a *search stream protocol* that enables arbitrary search algorithms to communicate in a resource-aware manner.

A Layered, Resource-Aware Policy Language. We define a *policy language* for expressing functions that navigate strategy-defined trees in pursuit of solutions. This language is *layered*: while policies are typically assembled by composing standard building blocks (in the form of prompting policies, search algorithms, stream transformers, and tree transformers), new primitives can also be defined easily using a dedicated language of *search-stream combinators*. In both cases, the proper enforcement of *resource limits* (e.g., LLM inference budget) is guaranteed *by construction*: each policy component can be assigned a global budget limit that is automatically inherited by its sub-components.

Few-shot Examples as First-Class Program Components. Few-shot examples are central to LLM-enabled programs, yet are traditionally treated as isolated data, disconnected from the logic that ties prompts together. As a result, they can be difficult to author (even determining *which* questions must be answered to solve a particular problem is often nontrivial in multi-prompt settings), and even harder to maintain as the program evolves. We introduce a *demonstration language* for describing examples while *grounding* them in concrete scenarios of navigating strategy trees. More precisely, a *demonstration* collects a set of query-answer pairs along with a set of unit tests that assemble them into paths within a search tree. The demonstration language enables a tool-assisted, test-driven workflow in which demonstrations can be created interactively and later repaired when strategy changes cause test failures. The universality of this workflow is established by a completeness theorem. Finally, demonstrations are *policy-agnostic*: they can be developed independently of policies and cannot break as a result of policy changes. This independence is achieved by associating each effect definition in the strategy language with a local navigation behavior that extends the semantics of demonstrations.

Self-Improving Oracular Programs. Every oracular program (defined by a strategy, a policy, and demonstrations) is automatically equipped with a natural self-improvement mechanism reminiscent of Expert Iteration [2]. Each time a problem is solved through search, examples of correct decisions can be extracted along the tree path leading to success. These examples can then be used to improve future executions, for instance through few-shot prompting, retrieval [34], or fine-tuning. However, domain-specific knowledge can often be leveraged to refine this coarse mechanism—for example, by integrating negative feedback, assigning credit for the successful solving of independent subproblems, or retroactively revisiting answers in strategies that perform iterative repair. We propose a unified mechanism for collecting feedback from oracular program executions, based on the modular specification of custom *feedback backpropagation passes* within strategies.

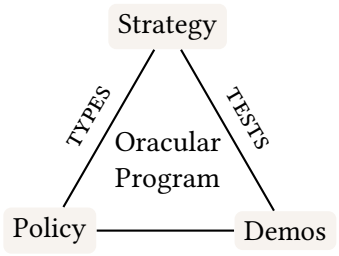
1.3 Thesis Statement

Nondeterministic programming provides a foundation for integrating traditional computations with generative AI, offering maximal flexibility and control. Separating core and search logic enables the concurrent exploration of advanced search algorithms and supports self-improvement, while grounding few-shot examples in coherent demonstrations facilitates their maintenance.

1.4 Thesis Structure

Chapter 2 demonstrates the possibility of automatically refining nondeterministic programs by training neural oracles through self-supervised, AlphaZero-style learning [87]. The described work [59] predates the rise of large language models and in-context learning [10] and achieves the challenging machine-learning feat of training small neural oracles *from scratch*, without *any* prior knowledge or human supervision, at the cost of only allowing a limited, *finite-choice* model of nondeterminism: instead of the `guess` operator from Figure 1.1, only a `choose` operator is available that nondeterministically selects an element from a finite list of values. This work demonstrates the *first* example of full self-supervised learning on a challenging theorem proving task (loop invariant synthesis) and provides conceptual foundations for oracular programming. Although LLMs now offer a clear alternative for building universal oracles, many ideas from this work are likely to become even more relevant as LLM scaling enters a data-bound phase. These include our proposal of *conditional generative strategies*, which produce reinforcement-learning curricula by repeatedly generating problem instances *conditional* on randomly sampled constraints (an idea later proposed independently in the LLM setting [27]).

Chapter 3 introduces the theoretical foundations of the three programming languages that underlie *oracular programming*, as outlined earlier in Section 1.2. It derives principled designs for the *strategy*, *policy*, and *demonstration* languages, guided by strong principles and guarantees (e.g., separation of core and search logic, modularity, extensibility, type-safety, correct resource management, completeness, and evolvability). This chapter is independent from Chapter 2.



Chapter 4 introduces *Delphyne*, an open-source framework for oracular programming embedded in Python, which offers rich tooling support in the form of a VSCode extension. Beyond presenting our framework and the associated engineering challenges, this chapter reflects on the trade-offs involved in embedding oracular programming in different host languages (e.g., Python versus Haskell), on the importance of co-designing languages with their associated tooling, and on navigating tensions between simplicity, generality, and extensibility.

Chapter 5 empirically validates oracular programming through two case studies. The first case study demonstrates the power of combining smaller language models with advanced search algorithms, using the problem of *loop invariant synthesis*. It shows how Delphyne enables advanced LLM pipelines to be built in a matter of hours, resulting in agents that are competitive with standard ReAct [103] baselines at only a fraction of the inference cost. The second case study demonstrates the seamless integration of *horizontal* and *vertical* pipelines (where LLMs orchestrate traditional computations and vice versa), and illustrates *self-improvement* for oracular programs, using *theorem proving in Lean* as a target task.

Finally, **Chapter 6** concludes the thesis by discussing the strengths and limitations of oracular programming, situating it within current trends in agentic design, and proposing avenues for future work.

1.5 Other Contributions

This section summarizes additional contributions made during my PhD that fall outside the scope of this dissertation while still revolving around common themes.

Nondeterministic Programming for Safe Reinforcement Learning. In collaboration with Yao Feng, on a project that I initiated and supervised, I developed a programming language framework for building *adaptive* safety monitors for learning-enabled cyber-physical systems [28]. The goal is to equip such systems with safety shields that override any action outside a provably safe control envelope, which becomes increasingly permissive as knowledge about the environment is gathered at runtime. We achieved this goal by leveraging *nondeterministic programming* (an overarching theme in this thesis) in two different ways. First, experts can define and verify control envelopes as *nondeterministic controllers* expressed in differential dynamic logic [74]. For adaptivity, these controllers can be parametric with respect to conservative bounds on unknown environment parameters, to be estimated and refined at runtime in a statistically sound way. Second, we allow untrusted, learned policies to be used at runtime to build such estimates by aggregating data and selectively applying probabilistic bounds within a fixed global safety budget. To guarantee soundness, such policies are restricted to operate within an envelope implicitly defined by a provably sound nondeterministic inference program, written by experts using a domain-specific language. By leveraging language design and theorem proving, our proposed framework empowers experts to design adaptive monitors with an unprecedented level of modeling flexibility, while providing rigorous, end-to-end probabilistic safety guarantees.

Controller Synthesis for Cyber-Physical Systems. In collaboration with Aditi Kabra, in an effort led by her, I worked on the problem of synthesizing provably safe symbolic controllers for cyber-physical systems [47, 48, 49, 50]. Our latest work in this direction [48, 49], which explores LLM-driven synthesis, leverages the Delphyne framework (Chapter 4).

A High-Profile AlphaZero Library. In the process of conducting my work on refining strategies without supervision (Chapter 2, [59]), I authored a high-profile AlphaZero library [58] written in Julia. At the time of release, it was one to two orders of magnitude faster than comparable, generic implementations (competitive alternatives based on Python/JAX have emerged since). Due to its simple API, high-quality documentation, and comprehensive observation and diagnosis tools, it enabled game designers with no background in computer science or machine-learning to successfully train AI agents for their game. The project garnered more than 1,300 stars on GitHub and attracted several rounds of funding in the context of the *Google Summer of Code* and *NumFOCUS* programs.

Causal Analysis of Complex Biological Systems. I have worked on the topic of analyzing the causal structure of protein-protein interaction networks, in collaboration with my initial thesis supervisor Jean Yang and Walter Fontana from Harvard Medical School.

The problem is, given a formal model that describes a set of local interactions between proteins as stochastic graph-rewriting rules (e.g., a protein of type A can bind a protein of type

B if it is bound to a protein of type C already, with a certain likelihood per time unit), to uncover minimal causal accounts of how compounds of interest can be assembled, in the form of directed acyclic graphs of causally related rule applications. Such graphs formalize the fuzzy biological concept of a *signalling pathway*.

The pre-existing solution to this problem [20] involved running simulations and then extracting minimal subsets of necessary events (i.e., rule applications) leading to outcomes of interest, using a form of SAT-solving. My first major contribution [61] was to argue that such methods were fundamentally incapable of capturing a form of inhibitory influence that is ubiquitous in biology, where an event A causes an event B to happen, not directly but rather by preventing an event C that is incompatible with B (by definition, event C does *not* appear in actual simulations). To capture such influences, I defined a new concept of causality based on *counterfactuals* (i.e., had event A not happened, B wouldn't have happened either) and extended the KaSim simulator [9] to simulate counterfactual scenarios. This research bridged two previously unrelated notions of causality, based on event noncommutativity and counterfactuals respectively [61, Theorem 2]. It served as a key impetus for Joseph Halpern (the originator of the concept of actual causality) and others to generalize the theory of structured equation models [38][72, Abstract and Section 5.2].

Finally, I developed a tool for quantifying the occurrence of causal patterns in simulation traces [60], which played an enabling role in a biology PhD thesis studying the Wnt pathway [67].

2

Refining Strategies Without Supervision

This chapter demonstrates the possibility of automatically refining nondeterministic programs by training neural oracles through self-supervised, AlphaZero-style learning. Although this work ultimately led us to propose oracular programming as a general framework (Chapter 3), it started in 2019 and was published in 2022 [59], and thus precedes the LLM era. It was initially motivated by a particular application, namely the scaling of formal theorem proving with machine-learning guidance despite data scarcity.

The dominant approach at the time for augmenting tactic-based interactive theorem provers with neural guidance used imitation learning on corpora of formalized mathematics [5, 39, 44, 63, 102]. However, despite efforts based on self-supervised pre-training [39] and data augmentation [92], this approach remained limited by the acute scarcity of training data: human-produced formal proofs are rare in general and virtually nonexistent in some domains. An alternative approach inspired by the success of AlphaZero [87] was to use reinforcement learning and let an agent self-train to interact with a theorem prover via trial and error [4, 99]. However, previous attempts to do so had been hampered by two fundamental issues:

- First, tactic-based theorem provers offer infinite action spaces not amenable to random exploration. Also, they are optimized for formalizing the outcome of human insights concisely but often fail to define a tractable search space for deriving those insights in the first place [82]. For example, using tactics effectively often requires providing insights in advance that are more easily found later in the search process (e.g. constructing a term to instantiate an existential quantifier [37]). More generally, most of the human process of discovering proofs, which involves trial and error, sketching, and abductive reasoning, is not captured by standard prover tactics, and so we can hardly expect a reinforcement learning agent to learn this process through sheer interaction with a tactic-based prover.
- Second, although reinforcement learning alleviates the need for human proofs during training, the agent must still be provided learning tasks of suitable relevance, diversity, difficulty, and generalizability in the form of theorem statements to be proved. This is in contrast with applications of AlphaZero in board games, where the symmetric nature of

```
assume x >= 1;
y = 0;
while (y < 1000) {
  x = x + y;
  y = y + 1;
}
assert x >= y;
```

A desirable *invariant* is a formula $I[x, y]$ such that:

$$\forall x, y \begin{cases} x \geq 1 \wedge y = 0 \rightarrow I[x, y] & \text{(holds initially)} \\ y < 1000 \wedge I[x, y] \rightarrow I[x + y, y + 1] & \text{(preserved)} \\ y \geq 1000 \wedge I[x, y] \rightarrow x \geq y & \text{(implies post)} \end{cases}$$

Such an invariant is $I[x, y] := (x \geq y \wedge x \geq 1 \wedge y \geq 0)$

Figure 2.1: An Example Program and an Associated Loop Invariant.

Chess or Go enables leveraging symmetric self-play as an infinite source of training tasks.

This chapter proposes a novel approach to learning automated theorem proving that does not rely on human-provided proofs and theorems. Instead, a teacher agent is trained to generate interesting and relevant tasks while a solver agent is co-trained to solve them. Our core insight is that *both* agents can be implemented by using reinforcement learning to refine high-level search *strategies* [82] written by experts in the form of nondeterministic programs. We also define novel design principles and language constructs for expressing such strategies. These include *conditional generative strategies* as a template for implementing teachers, *abductive reasoning* as a design principle that is made scalable by self-learned guidance, and *strategy events* as an abstraction that enables easier reward engineering and better sample-efficiency. We evaluate the resulting framework in a well-contained yet challenging setting, namely the automatic verification of imperative programs with loops, but frame our contributions in a way to emphasize general applicability.

Background: Verifying Imperative Programs with Loops

Suppose we are given a program such as the one in Figure 2.1 and we want to prove that the final assertion always holds. The way to proceed is to find a predicate called a *loop invariant* with the following properties: *i)* it is true before the loop is entered, *ii)* it is preserved by the loop body when the loop guard holds and *iii)* it implies the final assertion when the loop guard does not hold. By "preserved", we mean that if the invariant is true before executing the loop body then it is also true afterwards. Finding loop invariants is the most crucial aspect of program verification [16, 40] and still resists automation [30].

Despite the difficulty of automatically synthesizing loop invariants, humans do so routinely using well-understood search strategies. For example, in the case of the example in Figure 2.1, one would first try to prove the postcondition $x \geq y$ itself as an invariant and then note that it is not preserved by the loop body as the following implication does not hold: $y < 1000 \wedge x \geq y \not\rightarrow x + y \geq y + 1$. However, simplifying the right-hand side, we see that the proof works if we can show $x \geq 1$ to be an invariant itself. Using a similar form of abductive reasoning, we find that this in turn requires $y \geq 0$ to be an invariant and we end up proposing $x \geq y \wedge x \geq 1 \wedge y \geq 0$ as a loop invariant.

2.1 Approach

We show how high-level search strategies can be expressed as nondeterministic programs (Section 2.1.1) and then refined with oracles learned via reinforcement learning (Section 2.1.2), with training tasks generated by a separate teacher strategy (Section 2.1.3).

2.1.1 Expressing Strategies as Nondeterministic Programs

The search strategy we followed in the previous example can be formalized as a nondeterministic program, which is shown in Figure 2.2. As a nondeterministic program, it features a `choose` operator that takes a list of objects as an input and selects one of them nondeterministically. A nondeterministic program can be refined into a deterministic one by providing an external oracle to implement the `choose` operator. It also defines a search tree that can be explored with or without a guiding heuristic. In this work, we use neural networks to implement such oracles and guiding heuristics.

In addition, a key aspect of this strategy is to infer missing assumptions under which a currently failing proof obligation would hold. This form of reasoning is called *abductive reasoning* and it is fundamental in the way humans search for proofs [69, 76]. It is implemented in a separate `abduct` procedure that takes a formula as an input and returns either `Valid` or a list of abduction candidates. For example, the `abduct` procedure fails to prove the implication $x \geq 0 \rightarrow x + y \geq 1$ but may suggest $x + y \geq 1$, $x < 0$ and $y \geq 1$ as possible missing assumptions. The use of abductive reasoning for theorem proving and loop invariant synthesis specifically has been proposed in the past [24, 25]. However, abduction procedures are hard to implement [26] and typically only available for specific decidable theories. Also, using them in proof search tends to scale poorly in the absence of good heuristics to rank and filter abduction candidates. Our proposed framework makes abductive reasoning practical by addressing both issues: it allows leveraging self-learned guidance to select abduction candidates and it allows implementing abduction procedures as nondeterministic programs that are amenable to learning themselves.

Another notable feature of the strategy in Figure 2.2 is the use of the `reward` operator on line 17 to incentivize finding short invariants (short proofs are desirable in general as they tend to be easier to check, interpret and generalize). The `reward` operator can be used multiple times and at any point of the strategy execution. An implicit reward of 1 or -1 is emitted when the strategy successfully returns or fails respectively. An optimal execution of a nondeterministic strategy is one that maximizes the cumulative amount of collected rewards. In this example, we bound the maximal invariant size penalty in such a way to ensure that finding a proof is always rewarded more than failing at doing so.

As argued by Selsam [82], defining search strategies as nondeterministic programs provides a natural and flexible way for experts to leverage domain-specific knowledge while outsourcing difficult proof decisions to search algorithms and learned heuristics. This paradigm differs from the standard paradigm of tactic-based theorem provers in which an external entity must orchestrate stateless tactics that do not call for any form of interactive guidance internally. In contrast, the nondeterministic programming approach *inverts* control and has strategies call external oracles rather than the other way around, which allows for a much tighter control of the resulting search space.

```
1  def solver(
2        init: Formula, guard: Formula,
3        body: Program, post: Formula) -> Formula:
4
5     def prove_inv(inv: Formula) -> List[Formula]:
6        assert valid(Implies(init, inv))
7        inductive = Implies(And(guard, inv), wlp(body, inv))
8        match abduct(inductive):
9           case Valid:
10             return [inv]
11          case [*suggestions]:
12             aux = choose(suggestions)
13             return [inv] + prove_inv(aux)
14
15    inv_cand = choose(abduct(Implies(Not(guard), post)))
16    inv_conjuncts = prove_inv(inv_cand)
17    reward(max(-1, -0.2 * len(inv_conjuncts)))
18    return And(*inv_conjuncts)
```

Figure 2.2: A Simple Strategy for Finding Loop Invariants. In this strategy, written in Python-like pseudocode, an initial invariant candidate implying the postcondition is selected nondeterministically (line 15). One ensures that this candidate holds initially (line 6), without which the strategy fails immediately. Then, one attempts to prove that it is preserved by the loop body (lines 7 and 8, where `wlp` denotes Hoare's weakest liberal precondition operator [43]). If the candidate is not preserved, the `abduct` function suggests a list of assumptions that would make it so. One then uses the `choose` operator to nondeterministically select one of them, which we then try to prove invariant recursively (line 13). Because the number of abduction candidates to choose from can be large, successfully using such a strategy depends on having an effective oracle to guide search. To provide sufficient context to such an oracle, calls to `choose` must provide some extra information that is omitted here for brevity. In this case, we would pass a special token to indicate the call site, the program being analyzed, and the value of `inv` in the case of line 12. See Section 2.3.3 for more details and for a full listing of the invariant synthesis strategy that we use in our experiments. Chapter 3 introduces rigorous foundations for a general strategy language.

2.1.2 Refining Strategies with Reinforcement Learning

A nondeterministic program induces a (deterministic) Markov Decision Process (MDP) where intermediate states are choice points and final states are either success states (the program returns successfully) or failure states (an assertion is violated or `choose` is called on an empty list of choices). Standard search algorithms can be used to navigate this MDP but doing so in an efficient and scalable way requires strong heuristics for guiding search.

In this work, we propose to learn such heuristics in a self-supervised fashion using the AlphaZero algorithm [19, 87]. In AlphaZero, a neural network is trained to map any state to a probability distribution over available actions along with a value estimate (i.e. an estimate of the expected sum of future rewards). The neural network alone can be used as a policy for navigating the MDP. However, a stronger policy results from combining the network with a tree search algorithm such as Monte-Carlo Tree Search (MCTS) [11]. The key insight underlying AlphaZero is that the network can be trained via an iterative improvement process where it is successively *i)* used as an MCTS heuristic to try and solve problems and then *ii)* updated using gradient descent so as to better predict the outcome of each attempt along with the action selected by MCTS on all states encountered along the way. More details on AlphaZero can be found in the literature [87].

Using AlphaZero, we can refine nondeterministic proof search strategies *without* external proof examples to learn from. However, AlphaZero must still be provided with a set of training problems to be solved. Training problems should be diverse, relevant, and numerous enough to allow proper generalization. They should also vary in difficulty to allow learning to bootstrap.

2.1.3 Generating Training Problems

In theorem proving, high-quality training problems produced by humans are often not available in quantities even remotely matching the needs of reinforcement learning to properly generalize across problem instances. A possible solution is to use procedural generation techniques to assemble large datasets of synthetic problems. This works well in some specific areas [12] and particularly for problems that can be framed as inverse problems such as symbolic integration [56]. In other areas, generating interesting problems is as hard as solving existing ones, possibly harder.

This is the case in particular with the problem of loop invariant synthesis. Here, a natural idea for generating problem instances would be to use a probabilistic grammar for repeatedly sampling triples consisting of a program, a loop invariant and an assertion and then reject all triples in which the properties defining valid invariants do not hold. Unfortunately, not only would doing so naively lead to a very high rejection rate, but the resulting dataset would be heavily biased towards trivial samples that are hardest to reject (e.g. programs where the final assertion is the negation of the loop guard and where the invariant does not even matter). In contrast, many classes of interesting problems from standard human-written benchmarks would only be sampled with an infinitesimal probability.

In this work, we propose to generate problems using the same methodology we use to solve them: by leveraging reinforcement learning to refine nondeterministic search strategies. Specifically, experts can define *teacher strategies* in the form of stochastic and nondeterminis-

Name	Type	Description
`num-inv-main-disjuncts`	`none|1|2`	If sampled as an integer n, then `inv_main` is refined into a disjunction of n atomic formulas.
`has-conditional`	`bool`	Whether `body` must include a conditional statement.
`loop-guard-useful-for-post`	`bool`	Whether assuming the negation of the loop guard is useful in proving the postcondition `post`.
`body-implies-main-inv`	`bool`	If `true`, then `inv_main` always holds after executing `body` regardless of whether or not it holds before.
`eq-only-for-init`	`bool`	Whether or not to use equalities only in `init`.

Table 2.1: Examples of Teacher Constraints. Some descriptions refer to names introduced in Figure 2.3. The full list of constraints we used in our experiments is provided in Section 2.3.4.

tic programs, each run of which either fails or successfully returns with a problem instance. Reinforcement learning can then be used to refine such programs with the two objectives of maximizing the diversity and interestingness of generated problems while avoiding rejection. However, these objectives are naturally in tension since an agent may be locally incentivized to avoid generating particular types of problems that are easier to reject. We introduce a design template for a class of strategies that avoid this obstacle. We call such strategies *conditional generative strategies*.

A conditional generative strategy generates a problem in three steps: *i)* it samples a set of constraints that define desired features for the generated problem, *ii)* it nondeterministically generates a problem that respects as many constraints as possible and gets rewarded for doing so and *iii)* it applies a sequence of random and validity-preserving problem transformations as a way to further increase diversity. We provide a high-level description of such a teacher strategy for invariant synthesis in Figure 2.3. The key advantage of such an architecture is that it decouples the two conflicting objectives of generating valid and diverse problems: diversity is guaranteed by the first and third steps, while learning can focus on the second step. This also allows using the exact same learning algorithm for training both teacher and solver agents (AlphaZero in this work).

2.1.4 Predicting Events Rather Than Rewards

In the standard AlphaZero setting, the network is trained to predict the value of encountered states, that is, the expected sum of future rewards. However, such a number conflates a lot of valuable information. For example, suppose a teacher state is assigned a value of 0. Does 0 mean that the teacher is predicted to either fail or succeed with equal probability, or that it will certainly succeed in generating a problem that violates two constraints? And if so, which two?

Although this information is not relevant to MCTS, there are several reasons for having the network predict it anyway. First, doing so can result in an increased sample efficiency

```
1  def teacher(rng: RandGen) -> Prog:
2     cs = sample_constrs(rng)
3     p = generate_prog(cs)
4     p = transform(p, rng)
5     p = hide_invariants(p)
6     return p
7
8  def generate_prog(cs: Constrs):
9     p = Prog("
10       assume init;
11       while (guard) {
12         invariant inv_lin;
13         invariant inv_aux;
14         invariant inv_main;
15         body;
16       }
17       assert post;")
18    p = refine_guard(p, cs)
19    p = refine_inv(p, cs)
20    p = refine_body(p, cs)
21    assert valid(inv_preserved(p))
22    p = refine_post(p, cs)
23    assert valid(inv_post(p))
24    p = refine_init(p, cs)
25    assert valid(inv_init(p))
26    nv = num_violations(p, cs)
27    reward(max(-1.5, -0.5 * nv))
28    return p
29
30 def transform(p: Prog, rng: RandGen):
31    p = shuffle_formulas(p, rng)
32    p = shuffle_instrs(p, rng)
33    p = add_useless_init(p, rng)
34    p = add_useless_post(p, rng)
35    ...
36    return p
```

Figure 2.3: Simplified Teacher Strategy. Problems are generated by nondeterministically refining the template in lines 10-17 in a way that optimizes randomly sampled constraints (see examples in Table 2.1). In this template, the invariant is expressed as a conjunction of at most three sub-invariants: `inv_lin` is a linear equality or inequality, `inv_main` is a disjunction of atomic formulas (i.e. comparisons), and `inv_aux` is a conjunction of atomic formulas that can be used in proving `inv_main` but not `post`. These three formulas are refined nondeterministically by the call to `refine_inv` on line 19. After an invariant is chosen, a loop body is also generated nondeterministically (line 20) that preserves the invariant (line 21). Finally, the `num_violations` function penalizes the presence of useless or redundant problem components. For example, a penalty is applied if `inv_main` features a disjunct that can be removed without invalidating the problem. Section 2.3.4 provides more details on how the `refine_*` functions can be implemented. The simplest way to do so is to have a grammar and use the `choose` operator to select rules recursively. However, fancier strategies can easily be implemented in the nondeterministic programming framework. In particular, our implementation uses the `abduct` procedure introduced in Figure 2.2 to suggest values for constants and subformulas.

by providing the network with more detailed feedback on its mistakes. Such an effect has been previously demonstrated in the KataGo project [96]. Second, it is interesting from an interpretability point of view and makes it easier to diagnose problems and weaknesses in a given network. We found a third reason that is more subtle, which has to do with getting the combined benefits of issuing intermediate and final rewards.

Indeed, in the teacher strategy shown in Figure 2.3, we only penalize the agent for violating a constraint *at the very end*. Intuitively, there is a lost opportunity here since many violations could be detected much earlier while the problem is still being refined. Emitting an intermediate reward at this point would certainly improve learning efficiency. However, the same violation can potentially be detected at several different points in time, and we must ensure that a reward is only issued the first time. This in turn makes the job of the value-prediction network harder since it needs to keep track of whether or not each violation has already been penalized. Encoding this information alone can be costly, especially for attention-based network architectures such as transformers with a quadratic inference cost. Having the network predict constraint violations separately offers an elegant way out: all rewards are issued at the end, but from the moment a constraint violation is detected, the network's predicted probability for this violation is overridden to 1 whenever a value estimate is computed.

More formally, we introduce the concept of an *event*. Strategies can declare an arbitrary number of events e and each one is associated with a reward r_e along with a maximal number of occurrences m_e ($m_e = 1$ for constraint violation events) that can be counted towards the final reward. The `reward` operator introduced in Figure 2.2 is replaced by an `event` operator. When a strategy terminates, a reward is issued implicitly with a value of -1 in case of a failure and

$$\max\left\{1 + \sum_e r_e \min(n_e, m_e),\ r_{\texttt{min}}\right\}$$

in case of a success. In this expression, n_e denotes the number of calls made to $\texttt{event}(e)$ during the whole episode and setting $r_{\texttt{min}} > -1$ guarantees that successes are always rewarded more than failures. Note that it is important not to issue event penalties after a failure. Otherwise, faced with a likely failure, an agent may be incentivized to simply give up and fail immediately to avoid further penalties in search of success. Also, there are two reasons for bounding the maximal number of occurrences of event e that are counted towards the final reward by a constant quantity m_e. First, this allows having a network with a finite softmax head predict the number of occurrences of e (as we see below). Second, the $m_e = 1$ case is particularly useful to model events such as constraint violations that can be detected multiple times but should only be penalized once.

In this framework, the value head of the network is tasked with predicting the following probabilities: *i)* the probability p_0 of a failure, *ii)* the probability p_1 of succeeding with minimal reward $r_{\texttt{min}}$, *iii)* the probability $p_2 = 1 - p_0 - p_1$ of succeeding with a greater reward and *iv)* the probabilities p_e^i that $\min(n_e, m_e) = i$ for all events e and $0 \leq i \leq m_e$. A value estimate derives from these probabilities:

$$-p_0 + p_1 \cdot r_{\texttt{min}} + p_2 \cdot \sum_e \sum_i \hat{p}_e^i \cdot i \cdot r_e$$

where $\hat{p}_e^i \propto \mathbf{1}\{n_e \leq i\} \cdot p_e^i$ is a corrected probability estimate accounting for the number of times n_e that `event`(e) was raised already. In our invariant synthesis experiments, we use events in both the teacher and solver strategies to represent constraint violations $(m_e = 1)$ and encourage short proofs by penalizing individual proof steps respectively $(m_e > 1)$.

2.2 Implementation and Engineering Contributions

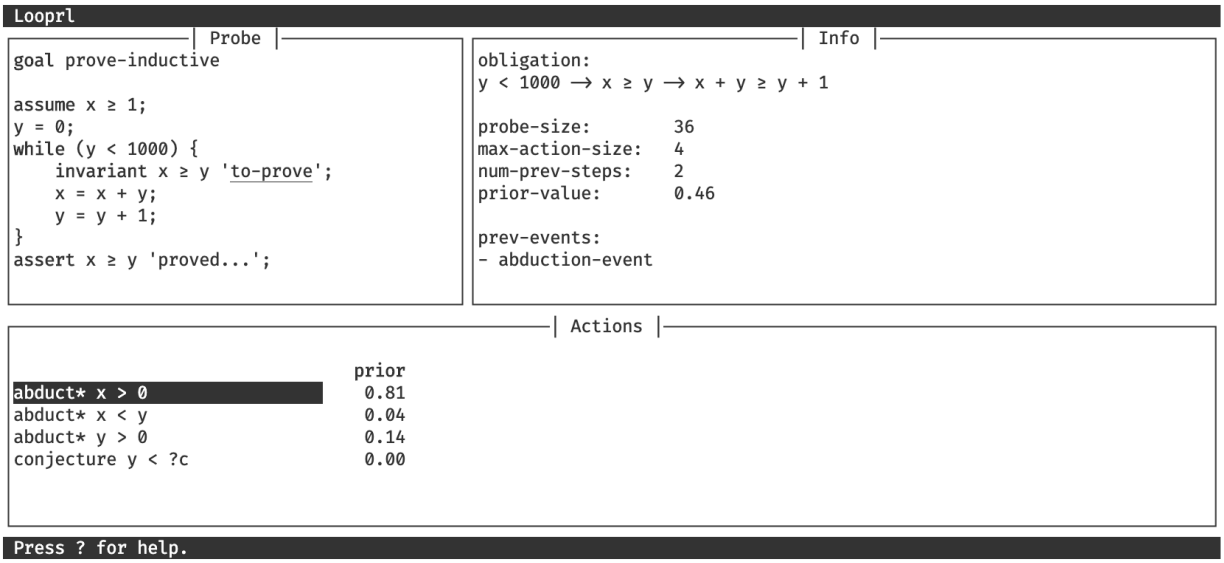
This chapter advocates for a hybrid approach to automated theorem proving where human experts from a variety of areas can formalize their knowledge succinctly in the form of nondeterministic proving and teaching strategies. Reinforcement learning is leveraged to fill in the blanks in all inevitable cases where the human intuition escapes formalization.

However, for this vision to be practical, writing strategies and iterating on them must be as frictionless as possible. New tools and abstractions are needed to automate the process of converting nondeterministic programs into reinforcement learning environments, ease debugging and foster code reuse. In this work, we accomplish a crucial step in this direction by introducing the Looprl theorem prover. Looprl consists of the following collection of features:

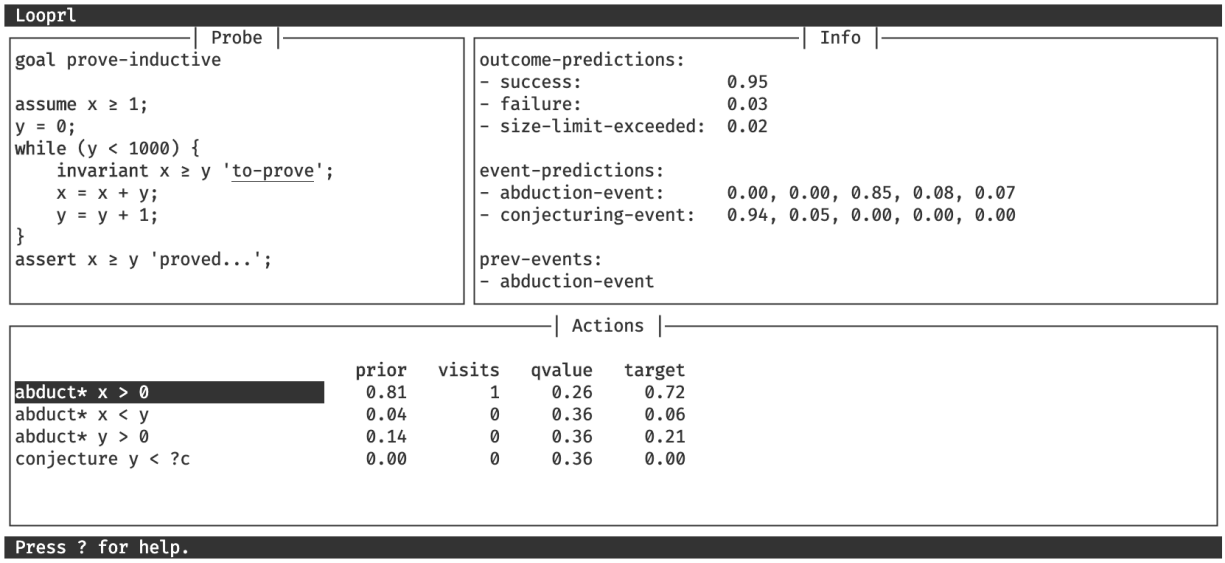
- A domain-specific language for writing strategies embedded in OCaml, with first-class support for neural-guided nondeterministic programming. Strategies are automatically compiled into reinforcement learning environments that can be explored in Python.
- A graphical interface that can be used to interact with strategies manually while inspecting the neural network's predictions and the behavior of MCTS (see Figure 2.4).
- A library of utility functions for writing program verification strategies. This library includes an implementation of the `abduct` function introduced in Figure 2.2 for integer linear arithmetic (see details in Appendix A.1).
- A parallel, high-performance implementation of the (Gumbel) AlphaZero algorithm [19, 87] for refining search strategies. Our implementation uses PyTorch [71] and Ray [70]. It provides support for events (see Section 2.1.4) and unbounded, variable-size action spaces.

The first point on developing a domain-specific language for writing strategies is particularly important. In our experience, having such a language is not just a mere matter of convenience but one of feasibility. In fact, we did try to implement an initial version of a teacher strategy for loop invariant synthesis by writing pseudocode on paper and manually compiling it into an MDP implemented in Python. However, doing so resulted in a complexity explosion where any single-line change to the pseudocode could take days of work to implement.

The reason compiling a nondeterministic program into an MDP manually is so tedious is that the whole program state must be made explicit, which includes the program stack. A tempting alternative would be to write nondeterministic code as normal Python code parametrized by an arbitrary `choose` function. However, this does not allow using tree search on the resulting program since doing so would require cloning the whole execution state of a Python program. Fortunately, there exists a solution to this problem in the field of programming languages using *search monads* [41, 82] and which essentially enables writing arbitrary nondeterministic code and then reifying it into a search tree that can be manipulated explicitly. Chapter 3 formalizes and develops this idea for a more general and extensible strategy language.



(a) Proof obligation associated with the abduction call along with the neural network policy prior.



(b) Event predictions along with MCTS statistics.

Figure 2.4: Visualizing the Solver Strategy with the Looprl UI. In the screenshots above, the UI is used to examine a choice point in the solver strategy where the current invariant candidate $x \geq y$ cannot be proved to be inductive and the user must choose between proving one of several abduction candidates or making a conjecture (this roughly corresponds to the call to `choose` on line 33). Here, one can see the network assigning a high prior probability to the optimal proof action, which is to try and prove $x > 0$ as an invariant. The network also predicts a value of 0.46 for this state, which is close to the truth of 0.4 (proving this problem requires three abduction events with cost 0.2 each). The details of how the value is estimated can be consulted in screenshot 2.4b. Here, we can see that the network predicts a 0.95 probability of success along with a probability of 0.94 for not requiring any conjecture and a probability of 0.85 for needing two more abduction events.

In Looprl, we implement a domain-specific language embedded in OCaml for expressing nondeterministic strategies. OCaml is a natural choice because it is fast (about two orders of magnitude faster than Python for symbolic code such as proof inferences) and it has good support for monads and Python interoperability. Our language provides built-in support for the `choose` and `event` operators. Also, it defines an intermediate graph-based format for representing data to be sent to neural networks in an architecture-agnostic way. Any piece of information that is passed to the `choose` operator must be convertible to this format and feature suitable metadata ensuring a seamless integration with the Looprl user interface and debugging tools.

2.3 Experiments

We tested Looprl on the problem of synthesizing loop invariants for single-loop imperative programs and evaluated it on the standard Code2Inv [85] benchmark. This benchmark contains a set of 132 programs written in C. All programs feature a single loop along with a final assertion to be proven. They feature linear integer arithmetic and sometimes involve conditionals, assumptions and nondeterministic tests (some Code2Inv problems are shown in Table 2.3 from Section 2.3.3). Most existing invariant generation tools can only solve a subset of these problems [86]. To the best of our knowledge, only one existing tool can solve them all [79]. However, it does not use learning and relies on specific optimization techniques that are not generalizable beyond the setting of purely numerical programs.

We used Looprl to implement an invariant generation strategy, which we describe in Section 2.3.3. In a nutshell, it generalizes the simple strategy described in Figure 2.2 by adding features such as the ability to abduct disjunctive invariants or to conjecture invariant templates with placeholder constants to be refined later using abduction. To our surprise, although this generic strategy can be described in only one page of pseudocode, it is sufficient to solve *every* Code2Inv benchmark problem in a few seconds when combined with the vanilla MCTS algorithm [11] (with no learned heuristic).

We therefore evaluate a self-trained solver agent on its ability to solve as many Code2Inv problems as possible *without* resorting to any search. At every decision point, it greedily takes the highest ranked action according to the network's policy head and no backtracking is allowed. This metric is of particular interest and significance. Indeed, finding an invariant for a single loop is of limited interest of its own. Rather, doing so is typically useful as a subtask in a hierarchy of increasingly complex and relevant problems. The next level of this hierarchy may be to prove a piece of code with nested loops, and then to synthesize a function respecting a specification, which is still several levels away from writing full-fledged software systems. Complex problems cannot be solved if backtracking search is needed at every level of this deep hierarchy.

2.3.1 Training Protocol

We train a teacher agent and a solver agent in sequence using the (Gumbel) AlphaZero algorithm. Both agents use a similar 1.6M parameters Dynamic Graph Transformer network [83] (see architecture details in Appendix A.3). The teacher agent is trained first for 20 AlphaZero iterations. During each iteration, it goes through 8000 problem generation episodes, using 64

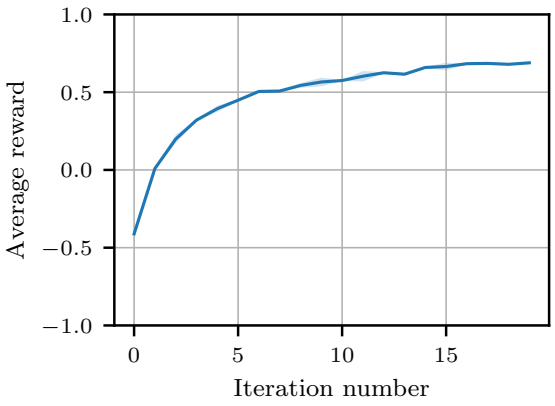


Figure 2.5: Teacher Training Curve.

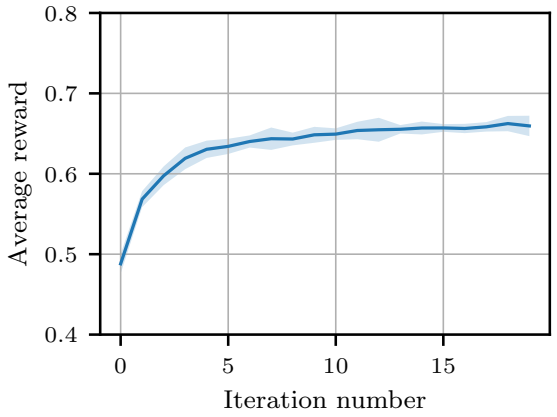


Figure 2.6: Solver Training Curve.

MCTS simulations per step. Then, the network is updated using samples from the k previous iterations with k increasing from 1 to 6 during training. Each iteration simulates 800 additional episodes to generate *validation* samples that are not used to train the network but to control overfitting via early stopping. After the teacher agent is trained, a dataset of 50K problems is gathered for training the solver, which includes a mix of 40K problems generated during the last five training iterations and 10K new ones that are generated without Gumbel exploration noise [19]. The solver agent is also trained for 20 iterations. During each iteration, it attempts to solve 20K randomly selected problems using 32 MCTS simulations per step. In addition, 5K additional problems are solved to generate validation samples. See Appendix A.2 for a complete list and explanation of all training hyperparameters.

A complete training run takes about 16h on a single machine with a 10-core Intel i9-10900KF processor, 64GB of RAM and a NVIDIA GeForce RTX 3080 GPU.

2.3.2 Experimental Results

We show training curves for the solver and teacher agents in Figures 2.5 and 2.6 respectively. These curves record the evolution of the average reward collected by MCTS combined with the latest network during each iteration. Error intervals are estimated based on two random seeds. Note that the maximum theoretical reward obtainable by the teacher agent is <1 since problem generation constraints may be sampled that are mutually incompatible, in which case the agent simply does its best to minimize the number of violated constraints. The same holds for the solver agent since all generated invariants are penalized proportionally to their size. The solver agent goes through a more modest reward gain during training (the y-axis is rescaled in Figure 2.6 to emphasize the trend). This is reflective of the fact that most problems generated by the teacher can be solved by MCTS alone and so most of the training concentrates on learning to solve about 10% of hard problems.

We show the results of our evaluation on the Code2Inv benchmark in Table 2.2. Three different agents are compared on the average ratio of benchmark problems for which they can successfully generate an invariant of minimal size *without* search or backtracking. The first

Policy	% Problems solved
Random	18.4 ± 0.0
Network (untrained teacher)	39.7 ± 1.6
Network (trained teacher)	$\mathbf{61.5 \pm 0.4}$

Table 2.2: Experimental Results on the Code2Inv Benchmark. No backtracking search is used. The score for the Random heuristic is computed as an average across 10K attempts. Standard deviations are computed based on two independent training runs with different random seeds.

agent is a baseline that simply selects proof actions at random. The second agent is a solver network that was trained using problems generated by an untrained teacher (i.e. using MCTS but no network heuristic). Finally, the third agent is a solver network trained using the full protocol of Section 2.3.1. These results demonstrate the critical impact of learning for both the teacher and solver agents. Unsurprisingly, an inferior teacher leads to an inferior solver with decreased generalization capabilities.

We now provide more details on the solver and teacher strategies used in our experiments (Sections 2.3.3 and 2.3.4 respectively).

2.3.3 Details on the Solver Strategy

In Figure 2.7, we provide some detailed pseudocode for the solver strategy used in our invariant synthesis experiment. This strategy is similar to the simple one introduced in Figure 2.2 but it comes with the following extensions:

- **Ability to abduct disjunctive invariants:** Imagine a proof obligation fails and the `abduct` function returns $a_1, \ldots, a_n$ as possible missing assumptions. In some cases, none of those can be proved invariant but the disjunction of a subset of them can. This is why the `suggest_missing` function defined on line 28 is used to build an invariant candidate as a disjunction of at most 3 (atomic) abduction candidates.
- **Ability to strengthen abducted invariants:** An abducted invariant candidate may have to be strengthened before it can be proved invariant. For example, the abduction engine may suggest $x \neq 0$ as a missing assumption but $x \neq 0$ may not be a valid invariant whereas $x > 0$ (or $x < 0$) is. The call to `strengthen` on line 47 is used to optionally and nondeterministically strengthen an invariant candidate. Our current strategy supports two kinds of strengthening: replacing a formula of the form $A \neq B$ by either $A > B$ or $A < B$ or weakening an inequality of the form $A \leq B$ into $A \leq B + c_?$ where $c_?$ is a nonnegative constant whose exact value is to be determined later (see next point).
- **Ability to instantiate constants lazily via abduction:** Invariant candidates can feature metavariables that denote unknown constants. The `constrs` variable defined in line 8 collects global constraints about these metavariables. If a missing assumption is suggested that only features metavariables, then it is added as a constraint rather than as a new

invariant candidate (line 42). After an invariant is proved, the metavariables it contains are instantiated using concrete values in a way to satisfy all global constraints (see call to `abduct_refinement` on line 57). A metavariable that appears in the invariant as an upper bound is instantiated with a value that is as low as possible, whereas a metavariable that appears as a lower bound is instantiated with a value that is as high as possible (this information is contained in the `btype` variable).

- **Ability to conjecture invariant templates:** In some cases, abduction alone is not enough to discover a missing invariant. For example, if two disjunctive invariants I_1 and I_2 are needed for proving the postcondition, one may have to conjecture the first one and then use abduction to find the other. The strategy in Figure 2.7 allows conjecturing invariant templates with metavariables to be instantiated via abduction (see previous point). The `conjectures` function called on line 29 returns three kinds of conjecture candidates: *i)* conjectures of the form $t \odot c_?$ where $\odot \in \{\geq, =\}$ and $t = \sum_i a_i x_i$ is a linear combination of variables that is preserved by the loop body, *ii)* relaxations of the loop guard (e.g. $x \leq 10 + c_?$ with $c_? > 0$ if the loop guard is $x \leq 10$) and *iii)* initial assumptions that only contain variables that are not modified by the loop body.

The solver strategy also emits two types of events (see Section 2.1.4) at lines 38 and 36 respectively. Conjecturing events are associated with a reward of -0.3 and abduction events are associated with a reward of -0.2. Both kinds of events are counted at most four times ($m_e = 4$) and the minimum total reward delivered in case of a success is $r_{\texttt{min}} = 0$.

Every call to `choose` is implicitly passed the program counter along with the value of all global parameters and all variables that are defined in lines 2 to 9. In particular, the `pending` variable summarizes all information from the program stack that is relevant to the neural network.

Hints on how to use our proposed solver strategy to solve Code2Inv benchmark problems are available in Table 2.3.

2.3.4 Details on the Teacher Strategy

The teacher strategy we use in our experiment follows the structure already introduced in Figure 2.3. A screenshot that illustrates it in the Looprl UI is available in Figure 2.8. We provide additional details below:

- **Sampling constraints:** The full list of available constraints is given in Table 2.4. Values are sampled *mostly* independently for each constraint type. In our implementation, we hardcode a small number of correlations (e.g., we are more likely to sample a disjunctive formula for `inv_main` if `inv_lin` is not used). We also reject a number of constraint combinations that are clearly uninteresting or unsatisfiable.
- **Fixed constraints:** In addition to penalizing the violation of sampled constraints, the teacher implements fixed hard constraints that are always enforced. A list of all such constraints is available in Table 2.5. Violating one of these constraints leads to an immediate failure along with a reward of -1.
- **Refining formulas and using abduction:** Different parts of the problem template shown in Figure 2.3 are nondeterministically refined in turn. To refine an atomic formula,

```
def solver(
  init: Formula,
  guard: Formula,
   body: Program,
   post: Formula) -> List[Formula]:

   invs_proved: List[Formula] = []
   constrs: List[Formula] = []
   pending: List[Pending] = []

   def prove_post():
      pending[-1].status = TO_PROVE
      to_prove = Implies(
         constrs + invs_proved + [Not(guard)],
         post)
      match abduct(to_prove):
         case Valid:
            pending[-1].status = PROVED
         case [*suggs]:
            assum = suggest_missing(suggs)
            closing = implies(assum, to_prove)
            pending[-1].status = \
               PROVED_COND if closing
               else TO_PROVE_NEXT
            prove_missing(assum, as_inv=True)
            prove_post()

   def suggest_missing(suggs: List[Formula]):
      suggs += conjectures(guard, body)
      num_disjs = choose([1, 2, 3])
      disjs = []
      for i in range(num_disjs):
         d = choose(suggs)
         disjs.append(d)
         if is_conjecture(d):
            event(CONJECTURING_EVENT)
         else:
            event(ABDUCTION_EVENT)
      return Or(*disjs)

   def prove_missing(f: Formula, as_inv: bool):
      if meta_only(f):
         constrs.append(f)
         assert sat(constrs)
      else:
         assert as_inv
         inv, fresh = strengthen(f)
         for c, _ in fresh:
            constrs.append(Ge(Metavar(c), 0))
         pending.append(
            Pending(INV, inv, TO_PROVE))
         prove_init(inv)
         prove_preserved(inv)
         invs_proved.append(inv)
         pending.pop()
         for c, btype in fresh:
            cval = abduct_refinement(
               c, btype, constrs)
            subst(invs_proved, c, cval)
            subst(constrs, c, cval)

   def prove_init(inv: Formula):
      pending[-1].status = TO_PROVE
      to_prove = Implies(
         constrs + invs_proved + [init],
         inv)
      match abduct(to_prove):
         case Valid: return
         case [*suggs]:
            assum = choose(suggs)
            prove_missing(assum, False)

   def prove_preserved(inv: Formula):
      pending[-1].status = TO_PROVE
      to_prove = Implies(
         constrs +
         invs_proved + [guard, inv],
         body.wlp(inv))
      match abduct(to_prove):
         case Valid: return
         case [*suggs]:
            assum = suggest_missing(suggs)
            closing = implies(assum, to_prove)
            pending[-1].status = \
               PROVED_COND if closing
               else TO_PROVE_NEXT
            prove_missing(assum, as_inv=True)
            prove_preserved(inv)

   pending.append(
      Pending(POST, post, TO_PROVE))
   prove_post()
   pending.pop()
   return invs_proved
```

Figure 2.7: Full Strategy for the Solver Agent. The highlighted code is only useful for providing the network with contextual information and can be disregarded on first reading.

```
x = 0;
while (x < 5) {
  x = x + 1;
  if (y > z) {
    y = z;
  }
}
assert y <= z;
```

Problem 3

Invariant: $x < 5 \vee y \leq z$.

This problem can be solved using our proposed strategy by directly abducting the correct disjunctive invariant when attempting to prove the postcondition.

```
assume x <= 10;
assume y >= 0;
while (*) {
  x = x + 10;
  y = y + 10;
}
assume x == 20;
assert y != 0;
```

Problem 7

Invariant: $x - y \leq 10$.

The star (*) corresponds to a nondeterministic boolean value. In this case, the loop body can be executed an arbitrary number of times. This problem can be solved by conjecturing an invariant of the form $x - y \leq c_?$ and then using abduction to refine $c_?$.

```
assume n >= 0;
i = 0;
x = 0;
y = 0;
while (i < n) {
  i = i + 1;
  if (*) {
    x = x + 1;
    y = y + 2;
  } else {
    x = x + 2;
    y = y + 1;
}}
assert 3*n == x + y;
```

Problem 93

Invariant: $3i = x + y \wedge i \leq n$.

This problem can be solved by conjecturing an invariant of the form $3i - x - y = c_?$, refining $c_?$ through abduction and then abducting $i \leq n$ as a missing invariant while trying to prove the postcondition.

```
s = 0;
i = 1;
while (i <= n) {
  i = i + 1;
  s = s + 1;
}
assume s != 0;
assert s == n;
```

Problem 110

Invariant: $i - s = 1 \wedge (i \leq n + 1 \vee s = 0)$.

This problem can be solved by first conjecturing an invariant of the form $i - s = c_?$, then using abduction to refine $c_?$ and finally abducting the disjunctive invariant $i \leq n + 1 \vee s = 0$ while trying to prove the postcondition.

Table 2.3: Examples of Solved Code2Inv Problems.

a template is first selected of the form $x_? \odot c_?$ or $x_? \odot y_?$ where $x_?$ and $y_?$ are variable placeholders, $c_?$ is a constant placeholder and $\odot \in \{<, \leq, >, \geq, =, \neq\}$. Each variable placeholder is then nondeterministically substituted by an existing or a fresh variable. Constant placeholders are either instantiated with concrete constants (a set of 6 available numerical constants is sampled at the start of the teacher strategy along with constraints) or parameters (special variables that cannot be modified by the program). Constant placeholders can also be left as-is and refined later using abduction (e.g. before invariant preservation is checked, abduction is used to suggest values for the remaining constant placeholders). Abduction is also used to suggest required parameter assumptions that are added to `init` (e.g. $n > 0$ where n is a variable not modified in the program).

- **Refining programs:** Subprograms are refined by selecting a sequence of assignment templates of the form: $x_? := c_?$, $x_? := y_?$, $x_? := x_? + d_?$, $x_? := x_? - d_?$, $x_? := x_? + y_?$ and $x_? := c_? - y_?$ ($d_?$ is a placeholder for a strictly positive constant). A special `skip` template can be selected to stop adding assignments. Variable and constant placeholders are handled in the same way they are handled in formulas. Which templates are available is determined by the `assignment-templates` constraint (see Table 2.4).
- **Extra refinement suggestions:** Extra suggestions are added to the standard templates when refining some formulas. When adding a disjunct to the main invariant, the loop guard itself is added as a suggestion along with a relaxed version of it (using a placeholder constant). When refining the last disjunct of the postcondition, abduction is used to suggest candidates that are consequences of the current assumptions in the associated proof obligation. Abduction is also used to suggest conjuncts for `init`.
- **Detecting constraint violations early:** Constraint violations are detected as early as possible to allow early feedback during search. For example, whether or not the invariant is satisfiable is checked right after the invariant has been refined and before the loop body is refined in turn. Most constraints must be checked again whenever a new parameter assumption is added. For example, an invariant $0 \leq x \wedge x < n$ may be initially judged as satisfiable. However, abduction may later on suggest $n < 0$ as a parameter assumption, making it unsatisfiable.
- **Applying random transformations to generated problems:** For increased diversity, a sequence of random transformations is applied to any problem generated by the teacher before it is returned. We provide a list of all such transformations in Table 2.6.

2.4 Related Work

Leveraging nondeterministic programming for proof search. The idea of combining nondeterministic expert strategies with neural oracles was proposed by Selsam [82] as a possible angle for tackling the IMO Grand Challenge [81]. However, how to train such oracles remains an open problem. One proposal [83] is to use supervised learning for training a universal oracle that is conditioned on the description of arbitrary search problems and can therefore use training data from a large number of heterogeneous sources. In contrast, we propose using reinforcement learning with the insight that nondeterministic search strategies can be used to

Name	Type	Description
num-preserved-term-vars	none|2|3	If sampled as an integer n, then inv_lin is refined with an invariant of the form $\sum_{i=1}^{n} a_i x_i = c$.
num-inv-main-disjuncts	none|1|2	If sampled as an integer n, then inv_main is refined into a disjunction of n atomic formulas.
num-inv-aux-conjuncts	none|1|2	If sampled as an integer n, then inv_aux is refined into a conjunction of n atomic formulas.
num-post-disjuncts	1|2	Number of desired atomic disjuncts for post.
has-conditional	bool	Whether body must include a conditional statement.
has-else-branch	bool	Whether the conditional in body has an else branch.
has-cond-guard	bool	Whether the conditional in body has a guard. If not, the guard is refined with the nondeterministic expression *.
body-implies-main-inv	bool	If true, then inv_main always holds after executing body regardless of whether or not it holds before.
loop-guard-useful-for-inv	bool	Whether assuming the loop guard is useful in proving that inv_main is preserved.
loop-guard-useful-for-post	bool	Whether assuming the negation of the loop guard is useful in proving the postcondition post.
use-params	bool	Whether or not to use variables that have a constant value throughout the program.
eq-only-for-init	bool	Whether or not to use equalities only in init.
loop-guard-template	template	The template to be used to refine the loop guard (e.g. a constant upper bound on a variable).
assignment-templates	templates	Allowed assignment templates for the body (e.g. constant var increment, assigning a var to another one…)
allow-vcomp-in-inv-main	bool	Whether inv_main's first disjunct can feature a comparison between two variables modified by the program.

Table 2.4: Complete List of Soft Teacher Constraints. Every constraint type is associated with a separate violation event. The associated reward is -0.5 for all constraint types except the last four ones where it is -0.2. A total reward of at least $r_{\texttt{min}} = -0.5$ is delivered in case of a success.

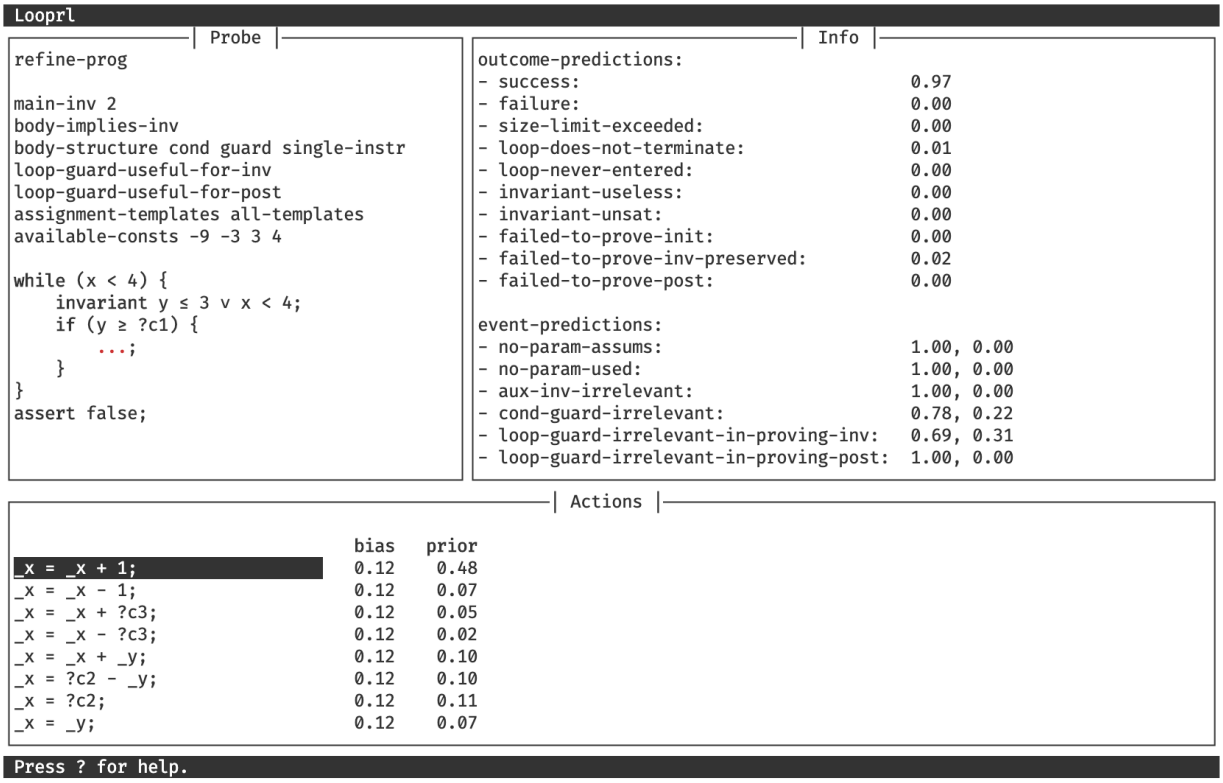


Figure 2.8: Visualizing the Teacher Strategy with the Looprl UI. This screenshot captures a choice point where the network is tasked with adding an assignment to the true branch of the conditional within the loop body. Several templates are proposed, which are going to be refined in turn. The upper left pane (i.e. the *Probe* pane) features all contextual information that is sent to the network to help it make a choice. Among this information, we can see a list of all constraints that the generated problem should ideally satisfy. The *Info* pane gives us some insights into the network's prediction about future events and outcome. For example, the network estimates with 0.97 probability that a valid problem will be generated (along with a 0.02 probability that a failure will be encountered due to the invariant not being preserved by the loop body). The network also estimates a 31% risk that the generated problem will violate the soft constraint according to which the loop guard should be relevant for proving the invariant.

Name	Description
`correctness`	The problem is correct, meaning that the invariant (i.e. the conjunction of `inv_lin`, `inv_main`, and `inv_aux`) satisfies the three properties defining a valid invariant (i.e., it holds initially, is preserved by the loop body, and implies the postcondition).
`{inv_main,post,init}` `-not-valid-unsat` `-or-redundant`	Disjunctive or conjunctive formulas such as `inv_main`, `post`, and `init` must not be valid, unsatisfiable, or redundant in the sense that they can be simplified (e.g. $x > 0 \vee x = 0$ is redundant because it simplifies to $x \geq 0$, and $x > 1 \wedge x > 2$ is redundant because it simplifies to $x > 2$).
`inv-sat`	The invariant must be satisfiable.
`loop-terminates`	The loop guard must not be preserved by the loop body when assuming the invariant, in which case the loop would never terminate once entered. (Note this constraint only rules out a subset of nontermination cases.)
`loop-entered`	The `init` formula does not imply the negation of the loop guard.

Table 2.5: Complete List of Hard Teacher Constraints. Violation of a hard constraint leads to an immediate failure and to a reward of -1.

generate problems in addition to solving them.

Learning to generate synthetic theorems. Procedural generation techniques have been used for producing synthetic theorems [3, 56, 75, 77, 100]. These usually proceed in a backwards fashion by generating random proof trees from a given set of axioms and rules. Wang *et al.* proposed to use supervised learning to learn how to guide this process towards generating more interesting theorems that are similar to those of a reference dataset [92].

Using reinforcement learning for loop invariant synthesis. Reinforcement learning has already been applied to the problem of loop invariant synthesis [85, 86], albeit in a very different way. In the aforementioned work (which introduces the Code2Inv benchmark), a *separate* reinforcement learning agent is trained from scratch on every benchmark problem, using counterexamples from an SMT solver as a training signal. This process takes minutes to hours for each problem to be solved. In contrast, we train a *single* agent to generalize across problem instances so that new problems can be solved in a matter of milliseconds.

Using reinforcement learning for theorem proving. The HOList Zero [4] and TacticZero [99] systems use reinforcement learning to learn how to interact with tactic-based theorem provers without relying on human proofs. However, they still rely on large human-produced corpora of formalized mathematical statements to be used as training tasks and are not yet competitive with approaches based on imitation learning. The use of reinforcement learning has also been explored in the context of saturation-based or tableau-based provers for first-order logic [17, 51].

Name	Description
`add-useless-loop-guard`	If the loop guard is irrelevant, assign a random formula in its place.
`add-useless-init`	Add a random conjunct to `init`.
`add-useless-post`	Add a random disjunct to `post`.
`add-useless-cond`	Replace `body` by `if(cond){body}` where `cond` is a random formula.
`rearrange-commutative`	Shuffle the order of disjunctions and conjunctions.
`move-conditional`	Commute the conditional statement in `body` with other instructions.
`shuffle-instrs`	Shuffle the order of consecutive assignments.
`randomize-comparisons`	Randomize comparisons by changing variable order or converting strict inequalities to nonstrict inequalities and vice versa.
`move-param-assum`	Remove a parameter assumption and add its negation to `post`.
`make-post-assums`	Rewrite a disjunctive final assertion into a sequence of atomic assumptions with a final atomic assertion (e.g. rewrite "`assert x>0||y>0`" into "`assume x<=0; assert y>0`").
`make-init-instrs`	Replace the `init` conjunctive assumption by a sequence of variable assignments and atomic assumptions.
`weaken-post`	Weaken the final (atomic) postcondition (e.g. replace "`assert x>0`" by "`assert x!=0`").

Table 2.6: Complete List of the Final Problem Transformations Implemented by the Teacher. Some transformations may or may not trigger based on a fixed probability. A transformation is canceled if it would violate a hard constraint or increase the number of soft constraint violations.

However, learned heuristics in such settings must operate at a very low-level and are subject to a speed-accuracy tradeoff that is unfavorable to deep learning.

2.5 Conclusion

We took a bold stride toward the challenge of self-learning automated theorem proving without relying on example theorems and proofs. We advocated a hybrid approach to theorem proving where experts are provided a flexible language to formalize their domain-specific knowledge in the form of generic nondeterministic teacher and solver strategies, leaving blanks to be filled by learning. We demonstrated our framework by applying it to the problem of loop invariant synthesis. Our agent solves all Code2Inv challenges. More importantly, it learns to solve a majority of problems with no search at all despite never seeing these problems during training.

None of our core contributions, however, are specific to invariant synthesis. These include: *i)* our key insight that *nondeterministic programming* and *reinforcement learning* can be similarly combined to implement solvers and teachers, *ii) conditional generative strategies* as a general template to write teachers, *iii) abductive reasoning* as a design principle that is made scalable by self-learned guidance, *iv) strategy events* for easier reward engineering and better sample-efficiency, and *v)* an implementation that establishes engineering foundations for making the whole approach practical.

Revisiting This Work in the Era of Large Language Models. This chapter introduced a general method for refining arbitrary nondeterministic programs with self-learned oracles, without *any* examples or supervision. However, it has two significant limitations. First, it is restricted to *finite-choice nondeterminism*, due to the reliance on AlphaZero-style learning. Writing strategies in this setting can be very engineering intensive, even for relatively narrow tasks and especially for teacher agents (see Sections 2.3.3 and 2.3.4). Second, learning oracles from scratch using reinforcement learning is computationally expensive.

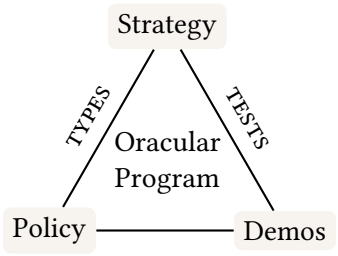
Large Language Models (LLMs) provide a new avenue for implementing universal oracles for nondeterministic programs, which we explore throughout the rest of this thesis. Their ability to generate structured data enables richer forms of nondeterminism, while in-context learning allows them to generalize from a handful of examples of correct and incorrect decisions. LLMs can also have their weights updated via reinforcement learning to improve their performance as oracles on specific tasks, using the same methodology presented in this chapter. Improved sample efficiency can be expected due to their pretraining, although their sheer size makes reinforcement learning expensive. We leave such experiments to future work.

Finally, our proposal of *conditional generative strategies*, which produce reinforcement-learning curricula by repeatedly generating problem instances *conditional* on randomly sampled constraints, has since been independently proposed and explored in the LLM setting. For example, TinyStories [27] generates a synthetic dataset of short stories to be used for distillation, by first sampling constraints and then tasking an LLM to generate stories that satisfy them. Possible constraints include the occurrence of specific words, the inclusion of a particular sentence in the story, or specific story features (e.g., dialogue, a bad ending, a plot twist, a conflict).

3

The Triad of Oracular Programming

This chapter motivates and defines the three languages that together constitute the triad of *oracular programming*, highlighting how their designs naturally emerge from the principles outlined in Section 1.2. These principles include: the complete and modular separation of search and core logic (allowing the latter to freely evolve without breaking the former), and the treatment of few-shot examples as *grounded* and *evolvable* program components (allowing their consistency with the rest of the program to be enforced through its evolution, with breakages easily identifiable and repairable).



An *oracular program* is defined by three orthogonal components: a *strategy*, a *policy*, and a set of *demonstrations*. A *strategy* is a nondeterministic program that denotes a high-level plan for solving a particular class of problems. It induces a search tree that can be navigated with LLM guidance, as specified by a *policy*. A *demonstration* bundles relevant examples of answering specific LLM requests with unit tests, in such a way as to describe concrete scenarios of successfully or unsuccessfully navigating the search tree for a particular problem instance. Consistency between *strategies* and *policies* is enforced through types, while consistency between *strategies* and *demonstrations* is enforced through *navigation tests*. By design, *policies* and *demonstrations* are fully decoupled and so a change in one cannot affect the other. Each of these components can be described in its own dedicated language. The remainder of this section motivates and presents these three languages, which together constitute the triad of *oracular programming*.

We define the *strategy* and *policy languages* via *shallow embeddings* in Haskell, a natural host language due to its purity, its expressive type system and its syntactic facilities for expressing monadic code. Full definitions are available in the appendices, and also as type-checkable Haskell code in the supplementary material. However, our *main implementation*—the Delphyne framework—uses a Python embedding instead, opting for a different trade-off in terms of static type safety, ecosystem integration and accessibility (Chapter 4). Indeed, although Haskell is better suited to formalization and exposition, Python is more widely used and offers advantages

```
data Tree a = Success a | Failure | forall b. Branch (Query b) (b → Tree a)
data Query b = Query { prompt :: String, parseAnswer :: String → Maybe b }
```

Figure 3.1: Naive Definition for a Search Tree. A tree producing values of type `a` (`Tree a`) can be either a *success leaf*, a *failure leaf* that represents a contract violation, or a *branching node* that contains a *query* and one *subtree* for every possible answer to this query. A *query* represents a question being asked to an external oracle and is defined by a prompt along with an answer-parsing function. Note that `b` is existentially quantified in the definition of a branching node, meaning that each such node can have children indexed by a different, locally-defined type (perhaps surprisingly, existential types are introduced in Haskell using the `forall` keyword. This is because any constructor type $(\exists\alpha\, T(\alpha)) \to \tau$ is isomorphic to $\forall\alpha\, (T(\alpha) \to \tau)$).

```
instance Monad Tree where
  return = Success
  Success a >>= f = f a
  Failure >>= f = Failure
  Branch q k >>= f =
    Branch q (\b → k b >>= f)
```

```
type Strategy = Tree

branch :: Query b → Strategy b
branch q = Branch q Success
ensure :: Bool → Strategy ()
ensure b = if b then return () else Failure
```

Figure 3.2: A Naive Language for Defining Search Trees. The tree type defined in Figure 3.1 can be equipped with a monadic structure, allowing trees to be defined directly and naturally using Haskell's *do-notation* (see Figure 3.3 for an example). For readers unfamiliar with monads, understanding this technical definition is not crucial: what matters more is an intuitive understanding of how a tree (as defined in Figure 3.1) can be defined via a nondeterministic program (as the one in Figure 3.3).

in tooling and reflection. Moreover, its static type system is expressive enough to type the strategy language precisely, as well as the upper layer of the policy language (gradual typing is still necessary for writing policy primitives that directly manipulate search trees).

3.1 The Strategy Language

We introduce our proposed strategy language in three steps, starting with a naive design based on well-known techniques for defining search trees via monadic programs [33], and then successively adding support for *modularity* and *extensibility*.

3.1.1 Initial Design Attempt

A minimal strategy language can be defined in just a dozen lines of Haskell, as shown in Figures 3.1 and 3.2. Trees are defined via an algebraic data type and equipped with a monadic structure, allowing them to be constructed using Haskell's *do-notation*, as we demonstrate in

```
generateProg :: Spec → Strategy Prog
generateProg spec = do
  prog ← branch (conjectureProg spec)
  proof ← branch (generateProof spec prog)
  ensure (checkProof spec prog proof)
  return prog
```

```
conjectureProg
  :: Spec → Query Prog
generateProof
  :: Spec → Prog → Query Proof
checkProof
  :: Spec → Prog → Proof → Bool
```

Figure 3.3: A Minimal Strategy for Program Synthesis. This strategy is expressed using the naive strategy language defined in Figure 3.2. To generate a program that provably meets specification `spec`, strategy `generateProg` first issues a query to conjecture a program (via `conjectureProg`, whose type but not definition is provided), then issues a query to obtain a proof that the program indeed meets this specification (via `generateProof`), and finally ensures that the proof is correct. For all values of its `spec` argument, `generateProg` produces a tree with two levels of branching, followed by either success or failure leaves.

```
dfs :: (String → IO [String]) → Tree a → MaybeT IO a
dfs _ (Success x) = return x
dfs _ Failure = mzero
dfs oracle (Branch (Query prompt parse) k) = do
  answers ← lift (oracle prompt)
  msum (map (dfs oracle . k) (mapMaybe parse answers))
```

Figure 3.4: Defining Depth-First Search for Naive Search Trees. See Figure 3.1 for the definition of naive search trees. The `dfs` function takes as its first argument an *oracle* in the form of a function that maps a prompt to a list of possible answers. Such an oracle can be implemented by sampling multiple answers from a large language model. The `MaybeT` monad transformer adds failure capability to a monad, `mzero` denotes a failure, `msum` returns the first successful value out of a list of alternatives, and `lift` wraps an IO value into `MaybeT`.

Figure 3.3. Once constructed, trees can be explored by independently definable search algorithms that pattern-match on their structure (see Figure 3.4 for an example). This initial design has the advantage of simplicity but is severely limited in terms of *extensibility* and *modularity*. The extensibility issue is straightforward: as noted in the introduction, advanced search algorithms leverage a wide variety of specific annotations and structural variations in trees (e.g., value annotations for MCTS [11]), and thus an explicit extension mechanism is needed to accommodate this diversity. The *modularity* issue is more subtle.

Modularity requires every query to be *locally* and *transparently* replaceable by a dedicated strategy. Indeed, a powerful workflow for designing strategies consists in starting with broad queries and then iteratively *refining* them when and where more control is needed. It might appear that our naive design allows this. For example, in the `generateProg` strategy from Figure 3.3, a program is conjectured via a single query. We can update `conjectureProg` to use a dedicated strategy instead, as shown below on the right. Doing so is akin to inlining the tree associated

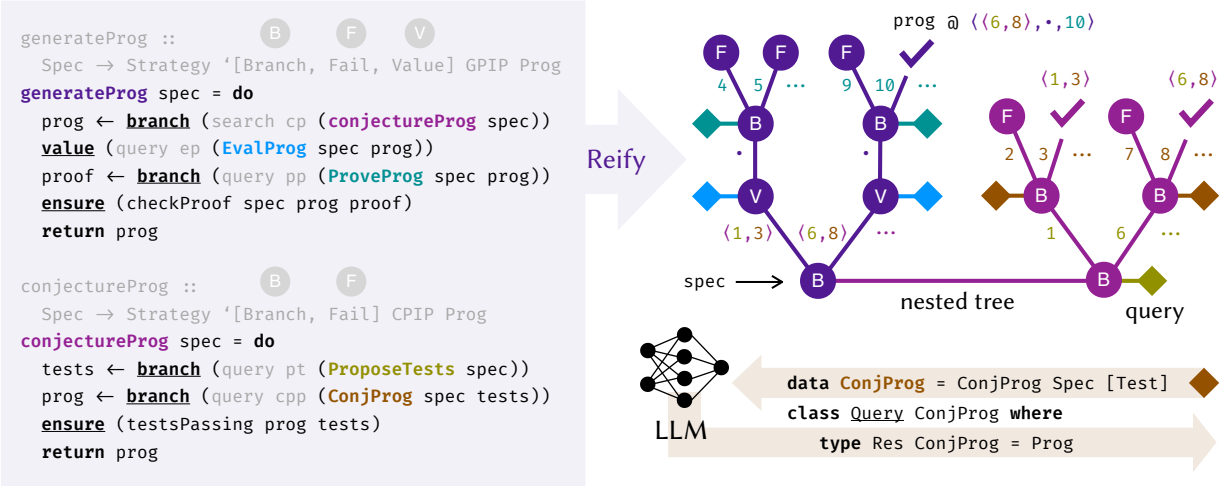


Figure 3.5: Example of a Modular Strategy For Program Synthesis. Strategies are nondeterministic programs that can be reified into search trees. They can issue *queries* to be answered by external oracles (queries are represented by diamonds ◇ and have constructors starting with uppercase letters). *Branching*, *failure*, and *value* nodes are labeled `B`, `F`, and `V` respectively. Horizontal lines denote *nesting* while other lines denote parent-child relations. Child edges and success leaves are decorated with *local value references*, with numbers being used as shortcuts for individual query answers. Grayed-out elements can be ignored on first reading.

```
- conjectureProg :: Spec → Query Prog
+ conjectureProg :: Spec → Strategy Prog
...
- prog ← branch (conjectureProg spec)
+ prog ← conjectureProg spec
```

with `conjectureProg` into the tree associated with `generateProg`. However, such inlining is hardly transparent and deeply alters the shape of the outer tree, typically requiring changes to policies searching it. For example, assuming the tree induced by `generateProg` is explored using depth-first search at depth d, such inlining would likely require updating the value of d. In addition, a single search algorithm may not necessarily be best adapted to both `generateProg` and `conjectureProg`: these two strategies should be allowed to produce different types of trees and to leverage different, independent policies.

3.1.2 A Modular Strategy Language

We introduce a novel strategy language that addresses the aforementioned challenges of *modularity* and *extensibility*. We begin by tackling *modularity*, proposing a monadic DSL for defining *modular* search trees that may feature *branching*, *failure*, and *value* nodes (value nodes provide quantitative information on how promising a branch is). We later present an extensibility mechanism for defining new kinds of trees (Section 3.1.6).

Figure 3.5 illustrates a strategy for *program synthesis* expressed in this language, while Figure 3.6 defines its typed syntax in the form of a Haskell module signature. Consider the example strategy from Figure 3.5, ignoring the grayed-out elements for now. Given a specifi-

```
data Strategy s p a
instance Monad (Strategy s p)
  return :: a → Strategy s p a
  (>>=) :: Strategy s p a → (a → Strategy s p b) → Strategy s p b
class Query q where { type Res q }

fail :: (Fail ∈ s) ⇒ Strategy s p a
branch :: (Branch ∈ s) ⇒ Opaque p a → Strategy s p a
value :: (Value ∈ s) ⇒ Opaque p Double → Strategy s p ()

query :: (Query q) ⇒ (p → PromptingPolicy q) → q → Opaque p (Res q)
search :: (p → SearchPolicy s' p') → Strategy s' p' a → Opaque p a
```

Figure 3.6: A Modular Strategy Language, as a Haskell Monadic DSL. Section 3.1.6 demonstrates how to extend this language with new effect types beyond `Fail`, `Branch`, and `Value`. The ∈ operator expresses membership of a type within a list of types [88]. An `ensure` function can be defined from `fail`, as in Figure 3.2.

cation, `generateProg` nondeterministically generates a program that satisfies it. It proceeds in four steps: *(i)* conjecturing a program, *(ii)* independently estimating the likelihood that this program is correct (providing valuable quantitative information to potential search algorithms), *(iii)* generating a machine-checkable correctness proof, and *(iv)* checking this proof. The first step is itself implemented via a separate strategy named `conjectureProg`, which generates executable unit tests before it produces a program and only returns this program if it passes all tests. In contrast to the naive strategy language from Section 3.1.1, a key innovation already emerges: programs can branch *not only* on the answer to a query, but *also* on the result of another strategy. This is reflected in the structure of the produced tree (Figure 3.5), where branching nodes can contain queries (represented as diamonds) but also *nested trees*, in which case children are indexed by the success leaves of this tree.

A *query* is a value that represents a question posed to an oracle. Queries are stratified by type, and each query type is associated with a *response type*. How exactly a query gets answered is *not* the concern of the strategy language but of the *policy language*, which allows defining *prompting policies* that map queries to streams of possible answers (e.g., by repeatedly sampling and parsing answers from an LLM using a given prompt and a set of examples; see Figure 3.12a for a preview). The policy language further allows defining *search policies*, which map strategy-induced trees to streams of values attached to success leaves of these trees. By *modularity*, search policies must remain agnostic to whether the candidate space of a branching node arises from a query or from a nested strategy tree. Exposing this distinction would violate the abstraction barrier: if a search policy had direct access to a query, it would break once that query were refined into a dedicated strategy, contradicting the modularity principle established in Section 3.1.1. Instead, search algorithms such as *depth-first search* or *MCTS* should operate over a lazy stream of candidates at each branching node, regardless of whether this stream is induced by a query or by a strategy (see Figure 3.12b for a preview and Section 3.2.1 for details on the exact nature of these *search*

streams, which is irrelevant for now). However, by the principle of separation between core and search logic, such streams cannot be directly attached to strategy trees, as they encapsulate policy-specific information. We resolve this tension by introducing the concepts of an *inner policy type* and of an *opaque space*.

3.1.3 Inner Policy Types and Opaque Spaces

Every strategy is associated with an *inner policy type*, which appears in its type signature. Values of this type—called *inner policies*—capture all the information required to turn inner queries and sub-strategies into proper streams of elements. For example, Figure 3.7 defines the inner policy type for the `generateProg` strategy from Figure 3.5, named `GPIP` as an acronym. A value of type `GPIP` specifies two prompting policies `ep` and `pp`, which respectively handle inner instances of the `EvalProg` and `ProveProg` queries, as well as a search policy `cp` for managing the inner call to `conjectureProg`. In turn, `cp` can be defined by combining a search algorithm such as *depth-first search* (`dfs`) with a nested inner policy of proper type (`CPIP`). Given that `dfs` has type

```
data GPIP = {
  cp :: SearchPolicy '[Branch, Fail] CPIP,
  ep :: PromptingPolicy EvalProg,
  pp :: PromptingPolicy ProveProg }
```

Figure 3.7: Inner Policy Type for `generateProg`

```
dfs :: DFSOptions → p → SearchPolicy '[Branch, Fail] p,
```

a possible policy for `generateProg` has shape `bestFirst _ (GPIP (dfs _ (CPIP _ _)) _ _)`, where `bestFirst` is a search algorithm capable of handling trees with value nodes, and where `_` holes stand for either search hyperparameters or prompting policies.

An *opaque space* abstracts away a query or a strategy, intentionally hiding its internal structure and exposing only a mapping from the ambient inner policy to a stream of elements. More information about the internal structure of opaque spaces remains accessible to the demonstration interpreter, but *not* to policies (Section 3.3). With these intuitions in place, we can now examine the formal signature of our proposed strategy language, as defined in Figure 3.6.

3.1.4 Language Signature

A *strategy value* is a computation that can be reified into a search tree. We use the more general term *strategy* to denote either a strategy value or a function that returns one, such as `generateProg`. A strategy value has type `Strategy s p a` (Figure 3.6, Line 1). The first parameter, `s`, is its *signature*: the list of effects that it is allowed to invoke—that is, the list of all node types that can occur in the associated tree. The second parameter, `p`, is its associated *inner policy type*, and the third parameter, `a`, is its *return type* (e.g., `Prog` in the case of `generateProg`). Strategies are equipped with a monadic structure (Lines 2-4), allowing them to be composed via the bind operator `>>=` for as long as their signature and inner policy types coincide (only trees with the same type of nodes can be inlined into each other).

Branching occurs over elements of an *opaque space* (Line 7). In turn, an opaque space can

be defined via a query (using the `query` function, Line 9) but *also* via a strategy of arbitrary signature (using the `search` function, Line 10), through pairing with a function that maps a choice of *inner policy* for the surrounding strategy to an adapted *sub-policy*. Often, this mapping consists in accessing a specific field of the inner policy (`cp`, `ep` or `pp` in our example from Figure 3.5). Finally, the `value` function also takes an opaque space as an argument (Line 8), allowing both queries and strategies to be used for computing value estimates. Associated `Value` nodes have a unique child (Figure 3.5), since value estimates are accessible to the policy but *not* to the strategy continuation (enabling transformations that remove value nodes from a tree; see Section 3.2).

3.1.5 Locality, References, and Traces

From the perspective of strategies, branching over an opaque space with element type `a` yields a value of type `a` (Figure 3.6, Line 7). However, in the induced tree, the children of the corresponding node are *not* indexed by values of type `a` themselves, but by *local references* to such values. When the opaque space is induced by a query, a local reference corresponds to an answer of type `a`. When it is induced by a nested tree, it corresponds to a path within that tree leading to a success leaf that contains a value of type `a`. Figure 3.5 shows these references for all child edges and success leaves in an example tree.

This distinction is not a mere technicality: it ensures that every success leaf can be traced back to a concrete sequence of query answers leading to it. This property is essential both for establishing the completeness of our demonstration language (Section 3.3) and for extracting learning data from successful runs of an oracular program (Section 5.2.3). When encountering a branching node, policies cannot produce values out of thin air; they must instead select values that are *explicitly* present in the enclosing opaque space, from which they can derive a stream of valid local references.

Whenever a policy explores a search tree, the finite pruned tree containing all visited nodes can be automatically extracted as a *trace*. Traces are serializable, due to references being serializable themselves. They can be visualized using proper tooling (Section 4.2), offering a general debugging mechanism. Moreover, they can be inspected to automatically extract learning data from runs of oracular programs (Section 5.2.3). This ability to reify the outcome of search as a trace, and to let dedicated *teacher programs* analyze it (e.g., by identifying subproblems that required many failed attempts and contrasting them with the ultimately successful attempt) is what we call *search reflection*. Search reflection provides a general foundation for self-improving oracular programs and relies critically on the separation of strategies and policies (Section 5.2.3).

3.1.6 Defining New Effects

Advanced search algorithms often exploit specific structure and annotations in search trees for efficiency. To accommodate arbitrary algorithms, our strategy language is extensible with new *effects*—each effect introducing a new kind of tree node. We motivate and illustrate this extension mechanism through two examples, and then provide a general definition.

Example: Compare-and-Branch Branching nodes provide search policies with a space of candidates to explore, yet no quantitative information is available to compare their quality and prioritize the search. Moreover, branching on syntactically distinct but semantically equivalent candidates (e.g., formulas $x > 0$ and $-x < 0$) is wasteful. The strategy language allows us to remedy this by defining a new *compare-and-branch* effect using the following syntax (not standard Haskell):

```
effect cbranch :: (CBranch ∈ s) ⇒                                          3.1.1
  { cands :: Opaque p a, compare :: [a] → Opaque p [Double] } → Strategy s p a
```

An effect is defined using the `effect` keyword, followed by the type of its triggering function, with named arguments (see the types of `branch`, `fail`, and `value` in Figure 3.6 for comparison). From such a declaration, a corresponding node type is automatically derived and can be used in strategy signatures (`CBranch` in this case, see Figure 3.10). Compared to `branch`, `cbranch` takes as an additional argument a comparison function that maps lists of branching candidates to probability distributions over them. This comparison function can be implemented using a query but also via a dedicated substrategy, since it returns an opaque space. In the reified tree, a `CBranch` node does not only contain an opaque space of branching candidates (as `Branch` does), but *also* a family of opaque spaces indexed by candidate lists containing comparison outcomes. The search policy can use this information in different ways (e.g., generating candidates lazily and performing a comparison for every new candidate, computing a set of candidates first and then only performing comparison once, using a single estimate for each comparison or aggregating several) but, crucially, how it does so is not the concern of the strategy triggering the effect (by our separation principle).

Example: Concurrency in Strategies Many search tasks can be decomposed into smaller tasks that can be solved concurrently. For example, in theorem proving, applying a tactic to a goal typically results in multiple subgoals that can each be proved independently. Many proof search algorithms exploit this structure by navigating variants of *conjunction-disjunction* trees [57, 78, 95]. We can model this with a `Join` effect that is defined as follows:

```
effect join :: (Join ∈ s) ⇒                                                3.1.2
  { left :: Strategy s p a, right :: Strategy s p b } → Strategy s p (a, b)
```

The `join` function takes as its arguments a strategy producing a value of type `a` and a strategy producing a value of type `b` and returns a strategy producing a value of type `(a, b)`. Note that it does *not* take opaque spaces as arguments, but instead *strategies* with identical signatures and inner policy types. Indeed, `join` does not abstract away the structure of its arguments, by design. A `Join` node in the reified tree contains two nested trees that are directly accessible to the search policy, thus enabling algorithms such as HyperTree Proof Search [57] to exploit the resulting bipartite structure. Still, one can combine opaque spaces with `join` by first wrapping them with `branch`. A `Join` node has one child for every pair of success values from its two nested trees.

$$\langle \text{decl} \rangle ::= \textbf{effect}\ \langle \text{trigger-name} \rangle\ ::\ (\langle \text{node-name} \rangle \in \textbf{s}) \Rightarrow \{\, \langle \text{arg} \rangle^* \,\} \to \textbf{Strategy s p}\ \langle \text{res-type} \rangle$$
$$\langle \text{arg} \rangle ::= \langle \text{name} \rangle\ ::\ \big(\langle \text{space} \rangle \mid \langle \text{type} \rangle \to \langle \text{space} \rangle\big)$$
$$\langle \text{space} \rangle ::= \textbf{Opaque p}\ \langle \text{type} \rangle \mid \textbf{Strategy s p}\ \langle \text{type} \rangle$$

Figure 3.8: Grammar of Admissible Effect Declarations. Terminal symbols are rendered in brown, while ⟨res-type⟩ and ⟨type⟩ denote type expressions with no free variables in {**p**, **s**}. A node type can be automatically derived from such a declaration, as exemplified in Figure 3.3 and formally defined in Figure 3.9. The value of ⟨res-type⟩ determines the node's *action type* (first line), each effect argument defines a member space or parametric space (second line), and spaces can be either opaque spaces or embedded trees (third line).

$$\frac{\forall i \;\; f_i \rightsquigarrow f_i'}{\textbf{effect}\ e\ ::\ (E \in \textbf{s}) \Rightarrow \{\, \vec{f} \,\} \to \textbf{Strategy s p}\ \alpha \;\rightsquigarrow\; \textbf{data}\ E\ \textbf{p t n a where}\ E ::\ \{\, \vec{f'} \,\} \to E\ \textbf{p t n}\ \alpha}$$

$$\frac{s \rightsquigarrow s'}{l :: s \rightsquigarrow l :: s'} \qquad \frac{s \rightsquigarrow s'}{l :: (\tau \to s) \rightsquigarrow l :: (\textbf{Local n}\ \tau \to s')}$$

$$\frac{}{\textbf{Opaque p}\ \tau \rightsquigarrow \textbf{LocalOpaque p n}\ \tau} \qquad \frac{}{\textbf{Strategy s p}\ \tau \rightsquigarrow \textbf{t n}\ \tau}$$

Figure 3.9: Inference Rules for Deriving Node Types from Effect Declarations. See Figure 3.8 for the grammar of effects, and Figure 3.10 for concrete examples of this transformation. The first line shows how to derive node types as GADTs after translating effect arguments. The second line shows how to translate effect arguments into local spaces and local parametric spaces, respectively. Finally, the third line details the translation of spaces into local spaces.

3.1.7 Generic Strategy Trees

We now give an informal definition of generic strategy trees, and then relate it to the examples from the previous section as well as to the grammar of admissible effect declarations (Figure 3.8). A formal type definition appears in Figure 3.11, but it can be safely skipped on a first read. Strategy trees are introduced through three mutually recursive definitions, which we present together before discussing them in detail.

Tree A *tree* is either a *success leaf* or an *effect node* (*node* for short). Nodes have types (e.g., `Branch`, `Join`), and a tree can only feature nodes whose types belong to its signature. Each node is associated with an *action type*, and has child trees indexed by *local values* of this type. It may contain *local spaces*, as determined by its type.

Local Value For a given node, a *local value* is either an element of a *local space*, or obtained from other local values through introduction and elimination of lists, tuples (possibly empty), and sum types (e.g., a list of local values is a local value; see Appendix B.1).

Local Space A *local space* is either an *opaque space*, an *embedded tree*, or the result of instantiating a *parametric opaque space* with a local value. An *opaque space* maps inner policies to

```
data CBranch p t n a where CBranch :: {
    cands :: LocalOpaque p n a,
    compare :: Local n [a] → LocalOpaque p n [Double] } → CBranch p t n a

data Join p t n a where Join :: {
    left :: t n a,
    right :: t n b } → Join p t n (a, b)

getStream :: LocalOpaque p n v → p → Stream IO (Local n v)
```

Figure 3.10: Examples of Derived Node Types. Each effect declaration induces a GADT node type with four parameters: the node's ambient inner policy type `p`, the type `t` of embedded trees, a phantom type `n` uniquely identifying the node for enforcing locality, and the node's action type `a`. The inference rules for deriving node types are defined in Figure 3.9 but are better understood through examples, which we provide here for `cbranch` (3.1.1) and `join` (3.1.2). `Local n a` denotes the type of local values attached to node `n`, while `LocalOpaque p n a` is the type of a *local* opaque space producing such values (see `getStream`).

streams of local values, while an *embedded tree* is a tree that shares the same signature and inner-policy type as its surrounding node, and whose success leaves carry local values.

Let us illustrate these definitions with `CBranch` and `Join`. From declaration (3.1.1), `CBranch` nodes contain an opaque space `cands` and a parametric opaque space `compare`. The associated action type coincides with the element type of the enclosed opaque space. From declaration (3.1.2), `Join` nodes contain two embedded trees, `left` and `right`. The associated action type is the product of their result types. More generally, the grammar of admissible effect declarations (Figure 3.8) mirrors our definition of generic search trees: each argument of the trigger function defines a (possibly parametric) space, while its return type determines the action type of the corresponding node.

Locality and References Generalizing our discussion from Section 3.1.5, the children of a node are indexed by *local values* that cannot be produced by policies out of thin air, and which can only be obtained by combining local space elements. Our formal type definition enforces this property using phantom types (Figure 3.11). Local values carry generalized references that trace their origin.

Reifying Strategies into Trees Together, Figures 3.6 and 3.8 define the syntax of our strategy language. Its semantics is given by a *reification* function that maps strategies to trees. The exact definition of this function is largely determined by the types of strategies and trees, and is therefore not particularly insightful. A Haskell implementation is provided in the supplementary material, where strategies are internally represented using free monads [88].

```
data Tree s p n v
  = Success (Local n v)
  | forall n'. SomeNode (NodeOf s p (Tree s p) n' (Tree s p n v))

data NodeOf s p t n k where
  Z :: Node e p t n k → NodeOf (e ': s) p t n k
  S :: NodeOf s p t n k → NodeOf (e ': s) p t n k

data Node e p t n k =
  forall a. (Effect e) ⇒ Node { effect :: e p t n a, child :: Local n a → k }
```

Figure 3.11: Definition of a Generic Strategy Tree. Type `Tree s p n v` denotes a tree with signature `s`, inner policy type `p`, return type `v`, and contained in a node identified by phantom type `n`. Such a tree is either a *success leaf* that contains an `n`-local value, or an *effect node* (Figure 3.10) whose higher-kinded type `e` belongs to `s` (`NodeOf` expresses such membership, with constructors that encode a Peano integer pointing to an element of `s`) and whose children are indexed by local values compatible with the node's action type `a`.

3.2 The Policy Language

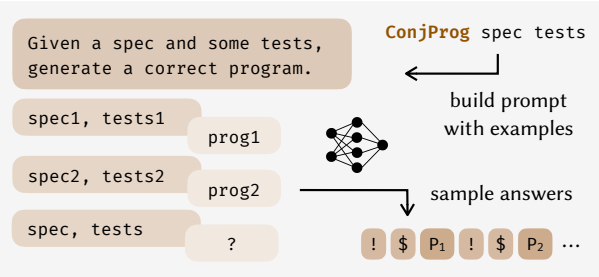
An oracular program is defined by three components: a *strategy*, a *policy* and a set of *demonstrations*. The previous section explored the first component, defining a language for expressing high-level problem-solving strategies as nondeterministic programs that can be reified into search trees. We now focus on the specification of *policies* that navigate such trees with the assistance of LLM oracles. This separation between *strategies* and *policies*—between *core logic* and *search logic*—provides major practical benefits, as it allows *vastly different* prompting techniques and search algorithms to be explored concurrently without requiring *any* changes in strategies (or in demonstrations).

We propose a *layered* policy language, which can be used at two levels. At a low level, it offers a series of combinators for building and combining *search streams*, guarding *by construction* against resource misuse (e.g., losing track of LLM API consumption and accidentally overspending). At a higher level, it provides a library of reusable abstractions for assembling policies by combining basic search algorithms and prompting policies with *stream transformers* and *tree transformers*.

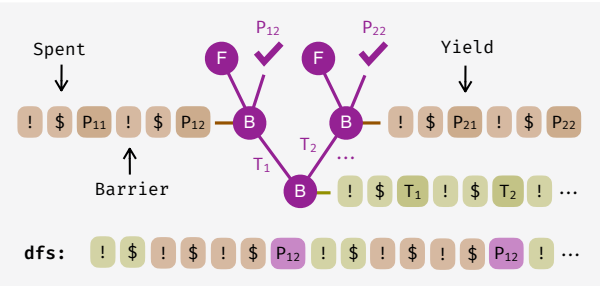
3.2.1 Policies and Search Streams

In the previous section, we defined *prompting* and *search policies* as functions mapping queries and trees, respectively, to *streams* of local values, while remaining intentionally abstract concerning the nature of these streams.

As a first approximation, a *search stream* is simply a (possibly infinite) iterator over local values. When a search policy encounters an opaque space, it can obtain such an iterator and use it to extract space elements *on demand*. Already we encounter a problem, though, since the search



(a) The *few-shot* prompting policy



(b) The *depth-first-search* search policy

Figure 3.12: Prompting and search policies map queries and strategies, respectively, to search streams. Section 3.2.1 defines `dfs`, which, in this example involving the `conjectureProg` strategy from Figure 3.5, accesses (possibly infinite) streams at each branching node and produces the stream shown at the bottom.

policy may be unwilling to relinquish full control to such an iterator, indefinitely waiting for an element that might never arrive. Instead, it might want to apply a *budget limit* to that stream (e.g., how many LLM requests or what inference cost it is willing to incur). Doing so requires search streams to be resource-aware. An intuitive view of streams is given in Figure 3.12: they can be seen as iterators producing messages of three kinds. A `Yield` message produces a new value; a `Barrier` message (shown as `!`) requests authorization to spend an estimated amount of search budget; and a `Spent` message (shown as `$`) reports actual resource consumption. Search budgets can be expressed in various metrics, such as number of requests, token usage, or dollar cost for a given LLM API. A consumer of a search stream is therefore capable of monitoring resource usage, and interrupting or resuming the stream whenever it sees fit.

Supporting concurrency in policies requires a more complex representation of streams. When a policy leverages concurrency internally, consumers of the resulting stream must be able to *selectively* deny spending requests without blocking the entire stream, which in turn calls for bi-directional communication. Details of this representation are provided in Section 3.2.3. However, this complexity is abstracted away from users: the policy language hides the underlying representation and lets users build and combine search streams using a combinator language that enforces proper resource management by construction. A representative subset of this language is shown in Figure 3.13.

Stream Combinators

Search streams are monad transformers (Figure 3.13, Lines 1-3) and can therefore be composed through binding (with a behavior analogous to the `concatMap` function on lists). Streams can be concatenated with `<|>`, for which the empty stream `mzero` is the neutral element. The `parallel` combinator is a variant, which runs its arguments concurrently, possibly interleaving their values. Operations that consume resources such as LLM inference credits can be lifted into streams via `spend` (Line 6). The `spend` function takes as its first argument an estimated over-approximation of the cost that will be incurred. This estimate can be used to proactively abort if not enough resources are available. Its second argument is the resource-consuming

```
1  return :: a → Stream m a
2  (>>=) :: Stream m a → (a → Stream m b) → Stream m b
3  lift :: m a → Stream m a
4  mzero :: Stream m a
5  (<|>), parallel :: Stream m a → Stream m a → Stream m a

6  spend :: Budget → m (a, Budget) → Stream m a
7  take :: Int → Stream m a → Stream m a
8  withBudget :: Budget → Stream m a → Stream m a
9  collect :: Stream m a → m ([a], Budget)  -- only call at top level
10 partial :: Stream m a → Stream m ([a], Budget, Stream m a)
```

Figure 3.13: A Combinator Language for Building Search Streams. The `(Monad m)` constraint applies implicitly everywhere, and `parallel` additionally requires `(MonadIO m)`.

operation to be executed, which must return a value along with the *actual* amount of resources that was consumed. The `Budget` type can be thought of representing a multi-dimensional vector of nonnegative real metrics, with possibly infinite values (to represent the absence of a limit). The `withBudget` combinator takes a budget limit and a stream as arguments and returns a transformed stream that cannot spend more than the allocated budget, terminating early if needed. Finally, the `collect` function (Line 9) exhausts a stream, returning all yielded values in a list along with the total spent budget. Note that it is only safe to use at top-level since invoking it *within* another stream loses track of the consumed resources due to the result type not being wrapped into `Stream`. A safe variant to use within streams is `partial` (Line 10), which runs a stream until no more budget is available and then returns the generated elements, the spent budget (for information only, resources are still tracked), and the stream continuation in case one wants to resume it later. In particular, `withBudget b (partial s)` gathers as many elements as possible with budget `b` and then returns these elements, along with a continuation for `s` that is unaffected by this budget limit.

The language presented in Figure 3.13 for defining search streams offers strong correctness guarantees in terms of resource management:

Property 3.2.1 (Correct Resource Management) *Let `s` be a search stream defined without `collect`. Then, collecting stream `withBudget b s` consumes a budget of `b` at most, assuming that all calls to `spend` provide a correct consumption over-estimate. Failing this assumption, the amount of budget spent over the limit is at most $n \times \delta$, with n the maximum concurrency level of the stream (1 if `parallel` is never used) and δ is the maximal estimation error of a call to `spend`.*

A proof sketch is provided in Section 3.2.3.

Example: Depth-First Search

We can now define the `dfs` search policy illustrated in Figure 3.12b, which we do in Figure 3.14. The `dfs` function is defined inductively on the structure of a strategy tree whose non-success nodes can have type `Branch` and `Fail`. The `p` argument is the ambient inner policy. As defined

```
dfs :: p → SearchPolicy '[Branch, Fail] p
dfs p tree = case tree of
  Success x → return x
  SomeNode (Z (Node (Branch cands) child)) →
    getStream cands p >>= dfs p . child
  SomeNode (S (Z (Node Fail _))) → mzero
```

Figure 3.14: Defining *Depth-First Search*.

in Figure 3.11, the `S` and `Z` constructors are used to encode the position of different effects in the `[Branch, Fail]` list, thereby encoding extensible sum types in Haskell. Encountering a success node yields an element, encountering a failure node yields no element, while encountering a branching node leads to lazily exploring the `cands` space while recursively calling `dfs` on each generated element, which we do via the monadic bind combinator. Note that adding a `maxBranching` parameter to `dfs` is unnecessary, since the prompting policies and search policies in `p` can be composed with the `take n` stream transformer, as we explain next.

3.2.2 Assembling Policies

Search and prompting policies can be written from scratch using the search stream combinators introduced in Section 3.2.1. More typically, however, they are assembled by composing standard building blocks (e.g., `fewShot`, `dfs`, `mcts`) with *stream transformers* and *tree transformers*. *Stream transformers* are functions that map streams to streams of the same type. Examples include `withBudget b`, `take n`, and `majorityVote`, which exhausts a stream and yields the most frequently observed element, if any. A *tree transformer* maps a tree to another tree, often with a different signature so that it can be used with a specific search policy.

For example, the `generateProg` strategy from Figure 3.5 must be paired with a search policy that handles branching, failure, and *value* nodes. This is the case for `bestFirst` [23] and `mcts` [11], which leverage value information to prioritize branches to explore. Alternatively, `dfs` can be used by right-composing it with the `dropValues` transformer—which removes value nodes entirely, preventing value computations—or with the `threshold` transformer—which converts value nodes whose estimates fall below a given threshold into failure nodes. As another example, the `elimJoin` transformer eliminates all `Join` nodes in a tree by inlining their subtrees in order (Figure 3.15).

3.2.3 Implementing Search Streams

This section describes a possible implementation for search streams, along with a proof sketch for Property 3.2.1. It can be skipped on first reading.

We propose an internal representation for search streams in Figure 3.16. A stream is a monadic value producing a *stream element* (search streams are often effectful—e.g., sending LLM requests). There are four kinds of stream elements: `Done` indicates that the stream has terminated, `Yield` produces a value, and `Barrier` and `Spent` handle resource management. A `Barrier` element requests authorization to spend some resources, up to a provided upper estimate. The client

```
elimJoin :: Tree (Join ': s) p n v → Tree s p n v
elimJoin = elimEffect (\(Node (Join l r) child) →
  bindTree (elimJoin l) (\lv →
    bindTree (elimJoin r) (\rv →
      elimJoin (child (trackPair (lv, rv))))))
```

Figure 3.15: Implementation of the `elimJoin` Transformer. The `bindTree` function is the tree equivalent of the monadic bind operation on strategies. It also uses the `elimEffect` utility, which handles the plumbing of recursively applying a given local transformation to the full tree.

```
type Skipped = Int
data StreamElem m a =
      Done
    | Yield a (Stream m a)
    | Barrier Budget (Bool → Stream m a)
    | Spent Budget Skipped (Stream m a)
newtype Stream m a = Stream {runStream :: m (StreamElem m a)}
```

Figure 3.16: Internal Representation of Search Streams.

must answer with a boolean—granting or denying the request—before the stream can resume[1]. Each `Barrier` element must be later paired with a `Spent` element that reports *actual* resource consumption. When using `parallel`, a `Spent` element is not necessarily paired with the latest `Barrier` element in the stream, due to calls to `spend` happening concurrently. Thus, `Spent` elements carry an integer that indicates how many previous barrier elements are to be skipped before encountering the matching `Barrier` element. We now explain how to implement two particular examples of stream combinators, and then provide a proof sketch for Property 3.2.1.

Implementing "withBudget". The `withBudget b` transformer intercepts and retransmits all elements from the underlying stream, while passing back all answers to resource requests. It overrides these answers whenever granting a request risks exceeding spending limit `b`. To do so, it not only memorizes how much budget has been spent already, but it also maintains an estimate of how much spending is *pending* at any moment in time—that is, what amount of resources is currently frozen by currently executing `spend` operations. Every granting of a `Barrier` request increases this pending amount by the associated consumption estimate, which is then subtracted upon encountering the matching `Spent` element. A full implementation is provided in Figure 3.17.

Implementing "parallel". The `parallel` combinator spawns one thread for each of its arguments. Each thread terminates upon encountering the `Done` element. `Yield` and `Spent`

[1]One cannot just interrupt the stream for denying resource requests since other requests for smaller amounts may follow (along with `Spent` elements that must not be dropped), especially when `parallel` is used.

```
withBudget :: (Monad m) ⇒ Budget → Stream m a → Stream m a
withBudget lim = aux 0 []
  where
    aux :: (Monad m) ⇒
      Budget → [(Skipped, Budget)] → Stream m a → Stream m a
    aux spent pending (Stream me) = Stream $ do
      e ← me
      case e of
        Done → return Done
        Yield x cont → return (Yield x (aux spent pending cont))
        Barrier b cont →
          return (Barrier b (\allow →
            let allow' =
              allow && (spent + b + sum (map snd pending) <= lim) in
            let reserved = if allow' then b else 0 in
            let pending' =
              (0, reserved) : map (\(s, v) → (s + 1, v)) pending in
            aux spent pending' (cont allow')))
        Spent b skipped cont →
          let spent' = spent + b in
          let freed = fromJust (lookup skipped pending) in
          let pending' = delete (skipped, freed) pending in
          return (Spent b skipped (aux spent' pending' cont))
```

Figure 3.17: Implementation of `withBudget`.

elements encountered by any thread are transmitted back (although care must be taken to update the `Skipped` argument of `Spent` messages to account for interleaving). Handling `Barrier` messages is more subtle. When a thread encounters a `Barrier` message and the client desires to allow it, the thread does so. However, if the client denies the request, the thread does *not* transmit this answer right away. Rather, if the *other* thread has `spend` operations pending (i.e., unmatched barrier elements), it waits until these operations are finished and then asks the client *again* (whose answer might be different upon seeing an updated estimate of pending expenses).

Proof Sketch for Property 3.2.1. To show *correct resource management*, we can prove that all combinators from Figure 3.13 preserve the following invariants on streams:

- Every `Spent` message is associated with a unique `Barrier` message and vice versa.
- If all spending requests are denied after seeing a given prefix from the stream, then the *total amount spent* after exhausting the stream (i.e., the sum of all amounts in `Spent` messages) is at most the sum of *(i)* the amount spent in the prefix, *(ii)* the pending amount (i.e., the sum of all amounts in `Barrier` messages that are not matched to a `Spent` message in the prefix), and *(iii)* the number of pending `Barrier` messages times the maximum resource estimation error δ.

The proof can be concluded by noticing that the number of pending `Barrier` messages in any stream prefix is at most its concurrency level n.

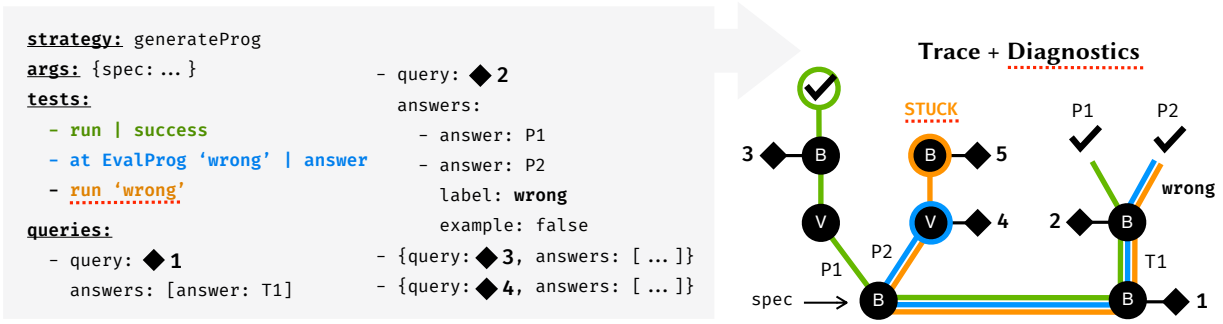


Figure 3.18: Example of a Demonstration for the Strategy from Figure 3.5. A demonstration bundles a set of query answers with a sequence of unit tests that describe concrete scenarios of navigating a specific strategy tree using these answers. Each test describes a path in the tree, highlighted with a matching color.

3.3 The Demonstration Language

Although queries can sometimes be answered successfully via *zero-shot* prompting, LLMs generally perform better when given examples of how to answer queries of the same type. Such examples are therefore central to LLM-enabled programs, yet are typically stored as raw query–answer pairs and treated as isolated data, disconnected from the logic that links queries together in multi-prompt pipelines. As a result, they are often difficult to *read*, *write*, and *maintain*. They are difficult to *read* because individual examples are often collected with little or no information about the context in which they arise, and are not grounded in concrete narratives of solving end-to-end problems. They are difficult to *write* because determining *what* questions need to be answered can be nontrivial for queries that occur in the middle of complex pipelines—not to mention the cost of crafting high-quality answers. Finally, they are difficult to *maintain* because changes in the pipeline can render some examples obsolete or incorrect, without clear ways for users to identify which ones.

We introduce a novel language for writing and evolving examples in oracular programs. In this language, related examples are bundled with unit tests into coherent *demonstrations*. Tests themselves can be specified concisely in a dedicated sublanguage. The language is designed to support a rich editor experience through an interactive, test-driven workflow for writing and repairing demonstrations, either manually or with LLM assistance. In addition, demonstrations are policy-agnostic and thus never break as a result of policy changes.

3.3.1 Overview of Demonstrations

Figure 3.18 shows a demonstration for the `generateProg` strategy (Figure 3.5). A demonstration consists of a *strategy value* (i.e., a strategy function with all its arguments instantiated), a set of *navigation tests*, and a set of *answered queries*. It is typically written in YAML syntax. In our example, the strategy value is specified by the `strategy` and `args` fields, which together induce a unique search tree. *Answered queries* are listed in the `queries` field. An *answered query* is a pair consisting of a query and a list of *answers*. Each answer may optionally carry a *label* (e.g.,

⟨test⟩ ::= ⟨instr⟩ | ⟨test⟩ | ⟨test⟩
⟨instr⟩ ::= **run** ⟨hints⟩? | **at** ⟨node-sel⟩ ⟨hints⟩? | (**go** | **answer**) ⟨space-ref⟩ |
take ⟨val-ref⟩ | **success** | **save** node-var | **load** node-var
⟨hints⟩ ::= **'**hint*'
⟨node-sel⟩ ::= node-tag(#int)? | ⟨space-sel⟩**/**⟨node-sel⟩
⟨space-sel⟩ ::= space-tag(#int)?
⟨space-ref⟩ ::= space-id(**(**⟨val-ref⟩**)**)?
⟨elt-ref⟩ ::= ⟨space-ref⟩ **{**⟨hints⟩**}** | ⟨hints⟩ | %node-var
⟨val-ref⟩ ::= ⟨elt-def⟩ | **[**⟨val-ref⟩ **,** … **,** ⟨val-ref⟩**]** | ⟨val-ref⟩**[**int**]**

Figure 3.19: Grammar of Demonstration Tests

`wrong`) and a flag indicating that it must *not* be used as a few-shot example (we return to this distinction shortly). Figure 3.18 depicts queries as diamonds, with identifiers locating them in the search tree, while actual demonstrations store serialized representations of query values.

Crucially, the answered queries are assembled into coherent scenarios of navigating the search tree via *navigation tests*. Three such tests are shown in Figure 3.18. Each test describes a path through the tree, starting at the root and ending at a specific node. Tests can either succeed, as in the first two cases, or fail, as in the third. Also, each test is composed of a sequence of instructions that are chained together using the `|` pipe operator. The first test (`run | success`) ensures that the provided examples are sufficient for solving the particular problem instance that the demonstration is about. The `run` instruction can be informally interpreted as "walk through the tree, using the first listed answer as a response every time a listed query is encountered"; the `success` instruction checks that the node at the end of this path is a success leaf. The second test demonstrates how to recover from a suboptimal choice by assigning it a low value. The `at EvalProg 'wrong'` instruction is similar to `run`, except that the answer labeled `wrong` must be selected when applicable and navigation must stop upon reaching a node associated with a query of type `EvalProg`; the `answer` instruction ensures that this query is answered in the demonstration. Since the answer labeled with `wrong` is not optimal and only included for the purpose of demonstrating how to reflect on a bad choice, it is marked as unsuitable for use as an example. Finally, the third test aims for a full walk through the tree, selecting the `wrong`-labeled answer when appropriate. However, it hits a query that is not listed in the demonstration and thus fails as *stuck*, providing the user with appropriate information on what query must be answered for navigation to proceed. With proper tooling, this mechanism allows writing and repairing demonstrations interactively (Section 3.3.4).

3.3.2 Navigation Tests

Figure 3.19 provides a full grammar for navigation tests. A test consists of a sequence of *instructions*. Each instruction takes as an input a node in the tree (initially the root) and returns a new node. The most important instruction is `run`. Starting at the current node, the `run`

```
navigate (CBranch cands _) = Just (\choose → choose cands)
navigate Fail = Nothing
navigate (Value _) = Just (\ _ → return nil)
navigate (Join l r) = Just (\choose →
    do { vl ← choose l ; vr ← choose r ; return (liftPair (vl, vr)) })
```

Figure 3.20: Examples of Standard Navigation Functions. A type signature for the `navigate` type class method is provided in Appendix B.2. The `liftPair` function lifts a pair of local values into a local value (Appendix B.1).

instruction uses answered queries to walk through the tree until a leaf node is reached or an answer is missing. But how should this walking behavior be defined? Since our strategy language is extensible, it must accommodate arbitrary effect nodes. Also, it must not depend on a specific policy since demonstrations are meant to be policy-agnostic.

Walking Through The Tree

The answer is to have each effect define a canonical *navigation function* that, at any given node, maps a *choice function* capable of selecting elements from local spaces to an action. The type for navigation functions is formally defined in Appendix B.2. Here, we explain this concept through examples, by looking at the navigation functions of standard effects (Figure 3.20). Navigating a `CBranch` node consists in selecting an element from the `cands` space and using it as an action. Failure nodes are leaf nodes and thus cannot be navigated. Value nodes have a unique child, which can be selected without further inspection. Finally, `Join` nodes can be navigated by selecting elements from the `left` and `right` spaces and pairing them up to form an action.

Whenever `run` needs to select an element from a space defined by a query, it looks for this query in the demonstration's `queries` section and picks the first provided answer. If none is found, it fails at the current node. When `run` encounters a space defined by a tree, it recursively navigates this tree. The `run` instruction stops when reaching a leaf at the same level of nesting where it started.

The `run` instruction can be passed a sequence of answer labels as *hints*, so as to specify alternate paths through the tree. Whenever a query is encountered, it is checked whether or not an answer is available whose label matches the first provided hint. If so, this answer is used and the hint is consumed. For example, instruction `run 'foo bar'` can be interpreted as: "walk through the tree, using answer `foo` whenever applicable and then `bar`". The `at` instruction demonstrated in Figure 3.18 works like `run`, except that it allows specifying a node at which the walk must stop. Our design allows describing paths concisely, by only specifying the few places in which they differ from a default path. This works well for specifying demonstrations, which typically describe a central scenario around which side explorations occur (e.g., showing how a bad decision leads to a low value score—as demonstrated in Figure 3.18—or demonstrating how redundant candidates can be removed at a particular step).

Although our demonstration language can express a wide variety of tests (see Figure 3.19 for a full grammar), `run ... | success` plays a special role. In most cases, nothing more is

needed to reach arbitrary success leaves in the search tree, which we capture as a completeness property in Section 3.3.3. Other test instructions are still useful for exploring parts of the search tree that are *not* on any shortest path to a solution (e.g., a substrategy that computes value estimates—useful for guiding search but not strictly necessary), or for cases not covered by our completeness result. We describe these instructions below.

Other Test Instructions

Stopping at particular nodes. The `at` instruction works like `run`, except that it allows specifying a node at which the walk must stop. All nodes in a tree are associated with a set of tags. The `at` instruction takes as an additional argument a node selector, the simplest form of which denotes a tag to match. For example, in Figure 3.18, instruction `at EvalProg 'wrong'` behaves similarly to `run 'wrong'`, except that it stops when encountering a node tagged with `EvalProg`. All *spaces* are tagged with the name of the associated query or strategy, and each node inherits the tags of its *primary space* if it has one (effect nodes can optionally define at most one *primary space*—see the definition of `Effect` in Appendix B.2). Effect nodes can also define custom node tags (Figure B.2, Line 10).

Importantly, `at` can only stop within the same tree that it started in and *not* inside a nested tree. In the example from Figure 3.18, `at ConjProg` will error instead of stopping at the unique node that contains a `ConjProg` query (query 2). This design choice is mandated, once again, by *modularity*. Indeed, individual strategies can be made responsible for setting unambiguous tags for nodes that they control but cannot be made responsible for ensuring the absence of clashing tags in *other* strategies. In order to stop at the node mentioned earlier, one must use instruction `at conjectureProg/ConjProg` instead. The node selector that is used here refers to the first node with tag `ConjProg` *within* the first space with tag `conjectureProg`. Finally, `at foo#2/bar#3` stops at the *third* node with tag `bar`, within the *second* space with tag `foo`.

Entering nested spaces. The `go` instruction allows entering a tree nested within the current node. For example, if the current node is a `CBranch` node, `go cands` enters the tree that defines the `cands` space or errors if `cands` is defined by a query. This instruction can be shortened as `go`, since `cands` is the primary space of `CBranch` nodes. More interestingly, suppose that the demonstration already explores two paths within `cands` that reach different success leaves and thus correspond to two different branching candidates. As previously discussed, each of these paths can be described through a sequence of *hints* so let us assume that the first candidate is identified by `''` (no hints, or default path) and the second candidate is identified by `'foo'` (use answer `'foo'` when appropriate). Then, instruction `go compare([cands{''}, cands{'foo'}])` can be used to enter the strategy tree comparing those two candidates. It can be shortened into `go compare(['', 'foo'])` since `cands` is a primary space. In general, any element of a local space can be referred to via a (possibly empty) sequence of hints. For spaces defined by queries, at most one hint is expected that indicates which answer to use. For spaces defined by trees, a sequence of hints is expected that leads to a success leaf by calling `run` recursively.

The `answer` instruction is similar to `go`. It takes a space selector as an argument but then expects to find a query instead of a tree when entering this space. It then succeeds if the corresponding query is answered in the demonstration and fails otherwise.

Visiting a child. The `take` instruction takes a local value as an argument and updates the current node to the corresponding child. For example, at a `Join` node, the instruction `take [left{''}, right{'foo bar'}]` visits the child associated with the pair whose first component is the default element from the `left` space and whose second component is the element from the `right` space described by hints `'foo bar'`.

Loading and saving nodes. The `save` and `load` instructions allow saving nodes into named variables and later recovering them. For example, `save id` saves the current node in a variable named `id` while `load id` ensures that the `id` variable is set (possibly by a previous test) and sets the current node to the corresponding value. When specifying a local space or a local value in the `take` and `go` commands, `%id` can be used to refer to the value attached to success leaf `id` (and errors if `id` does not refer to a success leaf).

3.3.3 Completeness

Our demonstration language offers *two* completeness guarantees. The first (*weak completeness*) states that any node in a search tree can be reached by a demonstration. The second (*strong completeness*) states that, under mild assumptions, any success node can be reached using only the `run` instruction. Weak completeness ensures that users cannot get stuck when demonstrating a known solution to a problem due to limitations in the expressiveness of the demonstration language. Strong completeness implies that, *in most cases*, the `run` instruction suffices, enabling a maximally simple and interactive demonstration-writing workflow.

Weak completeness is not hard to obtain *in principle*, since *Haskell* itself provides a trivially complete testing DSL. The challenge is to achieve it while keeping the testing DSL simple, and without allowing arbitrary Haskell values to be explicitly constructed. This is enabled by *locality* (Section 3.1.5), which guarantees that actions can only result from combining local space elements, which can themselves be identified by answer labels (for spaces induced by queries) or recursive sequences of actions (for spaces induced by trees). Thus, any path in a tree can be described using the `take`, `go`, `load`, and `save` instructions.

Property 3.3.1 (Weak Completeness) *For any node in a search tree, there exists a demonstration with a test (expressed using the grammar defined in Figure 3.19) that reaches this node.*

The *strong* completeness guarantee is more interesting and subtle. It states that, under a suitable assumption about navigation functions, and for any success leaf of a tree, there exists a demonstration with a single `run` test that reaches a success leaf with an *equivalent* value. Two values (of type `Local n v`, as defined in Appendix B.1) are said to be *equivalent* if they have the same content, despite possibly having different references.

The required assumption on navigation functions is *invertibility*. A navigation function is said to be *invertible* if, for any valid action in an associated node, there exists a sequence of local space elements such that the navigation function generates an equivalent action when passed an *inverse* choice function that picks all these elements in sequence.

All effects introduced so far have invertible navigation functions. For example, `CBranch` has an invertible navigation function since any valid action is an element from the `cands` space that can be picked by the inverse choice function.[2] In addition, the navigation function for `Value` is trivially invertible since value nodes have a single child. Our first case study introduces a new `Abduction` effect that is only *conditionally* invertible (Figure 5.1).

Property 3.3.2 (Strong Completeness) *For any success node in a tree, and assuming that all involved effects have invertible navigation functions, there exists a demonstration with a test of the form* `run <hints>?` *that reaches a success node carrying an equivalent value.*

A proof sketch proceeds as follows. Given a success leaf in the tree, the attached reference specifies a precise path leading to it as a sequence of actions. At every step along this path, and by invertibility, an equivalent action can be produced by `navigate`. Whenever an inverse choice function selects an element from a space defined via a query, the corresponding answer is added to the demonstration based on the element's reference. For spaces based on nested trees, the process is invoked recursively. Passing hints to `run` is only useful in the rare cases where the same query must be answered differently at different steps. In such cases, labels can be introduced to distinguish the different answers and provided as hints in the appropriate order.

3.3.4 Writing and Repairing Demonstrations

Demonstrations can be written interactively. A typical workflow is to start with an empty `queries` section and a single `run | success` test. Evaluating the demonstration results in the test getting *stuck* at a given node. Proper editor support allows visualizing this node, along with the attached unanswered query, which can then be added to the demonstration with a single click. An answer can be written manually, or generated by an LLM and then edited, after which the demonstration can be evaluated again. The next chapter illustrates this through the Delphyne VSCode extension (Section 4.2.1), with supporting screenshots (Figure 4.5). More advanced workflows are also possible, where demonstrations are partially written, with holes automatically filled by running external policies and extracting query answers from successful searches (Section 4.2.2).

By design, policy changes cannot break demonstrations. Many strategy changes are also guaranteed to preserve them. For example, adding a call to `value` in a strategy cannot cause a previously passing test to fail, nor can introducing or eliminating a call to `join`. In the event that a strategy change *does* break a demonstration, the breakage is clearly indicated by one or more failing tests.

[2]Technically, it is possible for an action at a `CBranch` node to be obtained through a nontrivial combination of elements from `cands`, such as `fst (untrackPair (trackPair (e, e')))` with `e` and `e'` elements from `cands`. However, using a *parametricity* argument [91], any action built by a policy (which does not have knowledge of the concrete action types of nodes, themselves hidden by existentials in the definition of trees; see Figure 3.11) must be equivalent to an element from `cands` (`e` in our example). Intuitively, this is because without further knowledge about `a`, there is no way to combine values of type `a` into a value of type `a` that is not *equal* to one of them; thus, there is no way to combine local values of type `Local n a` into a local value of type `Local n a` that is not *equivalent* to one of them. Alternatively, one can enforce at runtime that valid `CBranch` actions must be elements of `cands`, by implementing the optional `validAction` method (see Appendix B.2).

4

The Delphyne Framework

This chapter introduces *Delphyne*, an open-source[1] framework for oracular programming based on Python.[2] Beyond being a popular and accessible programming language with a rich ecosystem, Python has several technical properties that make it suitable for implementing an oracular programming framework. First, its *reflection capabilities* allow the definition of new effects and the enrichment of trees with debugging information with minimal boilerplate. Second, its ecosystem includes powerful libraries for structured data serialization and parsing (e.g., Pydantic) and for writing prompts as string templates (e.g., Jinja). Finally, and perhaps surprisingly, its optional static type system is expressive enough to precisely type the strategy language, along with all standard components of the policy language. Gradual typing is only necessary within the internals of primitive policy components (e.g., `dfs`), which typical users rarely, if ever, need to implement.[3] Delphyne offers rich tooling support for writing demonstrations and inspecting search trees, in the form of a VSCode extension. It consists of about 23,000 lines of Python and TypeScript (excluding blank lines and comments), and about 45,000 words of documentation.

We first describe how the strategy and policy languages are embedded in Python (Section 4.1) and outline the resulting trade-offs in terms of simplicity and type safety. We then present editor and tooling support, in particular for writing and repairing demonstrations (Section 4.2). Finally, we demonstrate Delphyne's power as an extensible, foundational framework by highlighting components of its standard library (Section 4.3).

[1] Delphyne is available at: `https://github.com/jonathan-laurent/delphyne`.

[2] Delphyne was the monstrous serpent, also known as *Python*, that guarded the Delphi oracle.

[3] Haskell remains a superior choice for technical exposition, as its type system allows for a fully precise definition of strategy trees, using *generalized algebraic data types* and *higher-kinded types* (Figure 3.11).

```python
@strategy
def generate_prog(spec: Spec) → Strategy[Branch | Fail | Value, GPIP, Prog]:
    prog = yield from branch(conjecture_prog(spec).using(lambda p: p.gp))
    yield from value(EvalProg(spec, prog).using(lambda p: p.ep))
    proof = yield from branch(ProveProg(spec, prog).using(lambda p: p.pp))
    yield from ensure(check_proof(spec, prog, proof))
    return prog
```

Figure 4.1: An Implementation of `generateProg` (Figure 3.5) in Delphyne. Strategies are described via coroutines instead of monads (hence `yield from`) and the `using` method corresponds to `query` and `search`. See Figure 4.2 for a definition of the `ProveProg` query.

4.1 Oracular Programming in Python

We discuss Python embeddings of the *strategy* and *policy* languages in Sections 4.1.1 and 4.1.2.

4.1.1 Embedding the Strategy Language

Delphyne strategies are not expressed with monads (for which Python has poor syntactic support) but using coroutines instead.[4] An example is provided in Figure 4.1. Since Python coroutines cannot be cloned, reification is implemented via thermometer continuations [55]: a tree node is identified by the sequence of actions leading to it and the strategy is replayed from scratch every time a child is computed. Doing so does not raise practical performance issues since expensive computations can be cached (see Section 4.3.1).

Defining Queries. Queries producing answers of type `T` can be defined as dataclasses inheriting from `Query[T]` (Figure 4.2). By default, the docstring is used as a system prompt, and instance prompts include a serialized JSON representation of the query object produced by Pydantic. LLMs are tasked with producing *structured output* [35] from an automatically generated schema, with the resulting JSON parsed by Pydantic. All these behaviors can be customized. Python's reflection capabilities are particularly useful here.

```python
@dataclass
class ProveProg(Query[Proof]):
    """
    Given a specification and a,
    program produce a checkable proof
    that the program meets the spec.
    """
    spec: Spec
    prog: Prog
```

Figure 4.2: Defining a Query in Delphyne.

[4] Python does not offer a syntactic mechanism like Haskell's *do*-notation to write monadic code, and does not allow writing multiline anonymous functions, making explicit uses of `bind` inconvenient. However, it provides *coroutines* (also called *generators* in Python), which are generalized functions that can be interrupted and resumed at specific yield points. Strategies can be defined by coroutines that yield each time an effect is triggered.

Strategies and Mutable State. In general, strategies must be *pure* and not depend on global, mutable state. Otherwise, reifying the same strategy computation twice could yield different trees. Haskell allows enforcing this discipline through types, but Python does not. We explicitly allow *internal* side effects and mutations inside strategies, which are idiomatic in Python, *as if* the associated coroutine never yielded control. Thus, the following is acceptable within a strategy, where `xs` and `ys` are local list variables:

```
ys = yield from branch(Query(xs).using(...)) ; xs.append(0) ; ys.append(0)
```

To ensure soundness, we restrict the arguments and return values of effects to be either deep-copyable values or pure functions. This is partially enforced by runtime checks, although these cannot distinguish pure functions from impure ones. In our experience, this soundness hole is very difficult to trigger accidentally in idiomatic Python code. Impure, stochastic, or non-replicable computations (e.g., calls to external theorem provers with wall-clock timeouts) can be performed inside strategies via the `Compute` effect, introduced shortly (Section 4.3.1).

Static Type Safety. Perhaps surprisingly, the strategy language can be fully typed using Python's static type system. For example, the `branch` effect trigger is typed as follows:

```
def branch[P, T](cands: Opaque[P, T]) → Strategy[Branch, P, T]: ...
```

Strategy signatures can be represented using *union types* (see Figure 4.1), which is more ergonomic than our Haskell encoding based on *type lists*, since unions are automatically commutative. The `Strategy` type is covariant with respect to its first and third arguments, and so `Strategy[N, P, T]` is a subtype of `Strategy[N | M, P, T]`. Overall, writing strategies in Delphyne with type checkers such as Pyright enabled in strict mode delivers an experience comparable to that of a statically typed language such as Haskell. Strategies can also be written without type annotations, providing the feel of a dynamic language.

One limitation of current Python type checkers is that they often struggle to infer the argument types of anonymous functions, which cannot be annotated in Python (although this may change in the future). This becomes especially problematic when building opaque spaces through the `using` method (Figure 4.1). As a workaround, several functions and methods, such as `branch` and `using`, feature an optional argument `inner_policy_type` that allows the inner policy type to be explicitly specified when necessary.

Adding New Effects. New effects can be defined by inheriting from the `Node` class and defining a corresponding trigger function. We show a definition of the `Join` effect in Figure 4.3. Effect node types cannot be typed precisely (as in the Haskell embedding in Figure 3.10), since Python does not support GADTs or higher-kinded types, hence the use of `Any`. However, this only affects the internals of policy primitives that explicitly manipulate trees (see Section 4.1.2) and does not affect strategy authors, since the trigger function `join` is precisely typed. Navigation functions are defined slightly differently than in Haskell: instead of taking a choice function as an argument, they return a Python generator that yields local spaces and receives elements from them (hence the use of `yield`). Node methods such as `spaces` and `map_embedded` (see Ap-

```
@dataclass
class Join(Node):
    subs: Sequence[EmbeddedTree[Any, Any, Any]]

    def navigate(self) → Navigation:
        ret = []
        for sub in self.subs:
            ret.append((yield sub))
        return tuple(ret)

def join[N: Node, P, T](
    subs: Sequence[StrategyComp[N, P, T]]) → Strategy[N, P, Sequence[T]]:
    recv = yield spawn_node(Join, subs=subs)
    return recv.action
```

Figure 4.3: Defining the `Join` Effect. Unlike the version defined in Figure 3.10, `join` can take a *list* of computations as arguments (not just two), all with the same type.

pendix B.2) are automatically inherited from the `Node` parent class, where they are implemented using reflection by inspecting dataclass field annotations.

4.1.2 Embedding the Policy Language

On the side of the policy language, policy components (i.e., search and prompting policies, stream and tree transformers) can be given a precise external signature, enforcing their correct use and composition. However, their *implementation* typically uses gradual typing since search trees are not fully typed. For example, the `Branch` class (representing branching nodes) has no parameter constraining the type of value being branched on or the type of the surrounding inner policy (see also Figure 4.3 for the definition of `Join`). In addition, locality (Section 3.1.5) is only enforced at runtime. Writers of functions such as `dfs` must therefore be careful in ensuring that their implementation truly matches the advertised type. Overall, this is still a good trade-off since typical users are expected to spend much more effort writing strategies and assembling policy components than defining new effects and atomic policies (especially given a rich standard library).

Type-Directed Policy Writing and Inner Policy Dictionaries. Inner policy types are typically defined as dataclasses in Delphyne, and policies are written by composing primitives (search algorithms, stream transformers, and tree transformers) in a type-directed process, as demonstrated in Section 3.1.3. This approach offers strong static type safety, but at the cost of verbosity: each strategy must be followed by an explicit inner policy type definition and each creation of an opaque space within a strategy requires passing a mapping from the ambient inner policy to a proper sub-policy, often in the form of an anonymous function (Figure 4.1).

```
@search_policy
def par_dfs[P, T](
    tree: Tree[Branch | Fail, P, T], env: PolicyEnv, policy: P,
) → StreamGen[T]:
    match tree.node:
        case Success(x): yield Solution(x)
        case Fail(): pass
        case Branch(cands):
            cands = yield from cands.stream(env, policy).all()
            yield from Stream.parallel(
                [par_dfs()(tree.child(a.tracked), env, policy) for a in cands])
```

Figure 4.4: Defining a Parallel Variant of Depth-First Search in Delphyne.

Delphyne offers an alternative that trades static type safety for concision: *inner policy dictionaries*. An inner policy dictionary (with type `IPDict`) is simply a Python dictionary that maps strings denoting space tags (by default, the name of the query or sub-strategy inducing the space; see Section 3.3.2) to sub-policies. Strategies can elect to adopt `IPDict` as their associated inner policy type, in which case no additional inner policy type needs to be defined, nor any mapping passed to `using` (the ellipsis value `...` can be used instead). The `guess` operator used in our introductory example (Section 1.1) and defined in Section 4.3.3 leverages inner policy dictionaries for concision. Finally, note that the trade-off made here between static type safety and concision is not fundamental but stems from limitations of Python. In principle, macros could be used to automatically derive inner policy types and mappings from syntactic conventions. Although Python supports runtime code generation and evaluation, it does not do so in a way that is compatible with static type checking.

In any case, defining inner policy types and mappings is generally a small price to pay, and the resulting explicitness is valuable in its own right. It also offers additional expressiveness. For instance, an opaque space created inside a loop can have its associated policy indexed by the loop iteration number, in which case the corresponding inner policy type would include a function field.

Defining New Search Algorithms. Figure 4.4 shows how to define a new search algorithm in Delphyne, here a *parallel* variant `par_dfs` of depth-first search. The `par_dfs` function has a precise type signature but internally uses gradual typing. It returns a search stream defined via a Python generator. The `all` method collects all elements from a stream while forwarding resource-consumption messages (similar to `partial` in Figure 3.13). The `Stream.parallel` combinator takes a list of search streams and returns a stream (similar to `parallel` in Figure 3.13). As in our embedding of strategies in Python, we use Python coroutines (generators) instead of monads to define search streams.

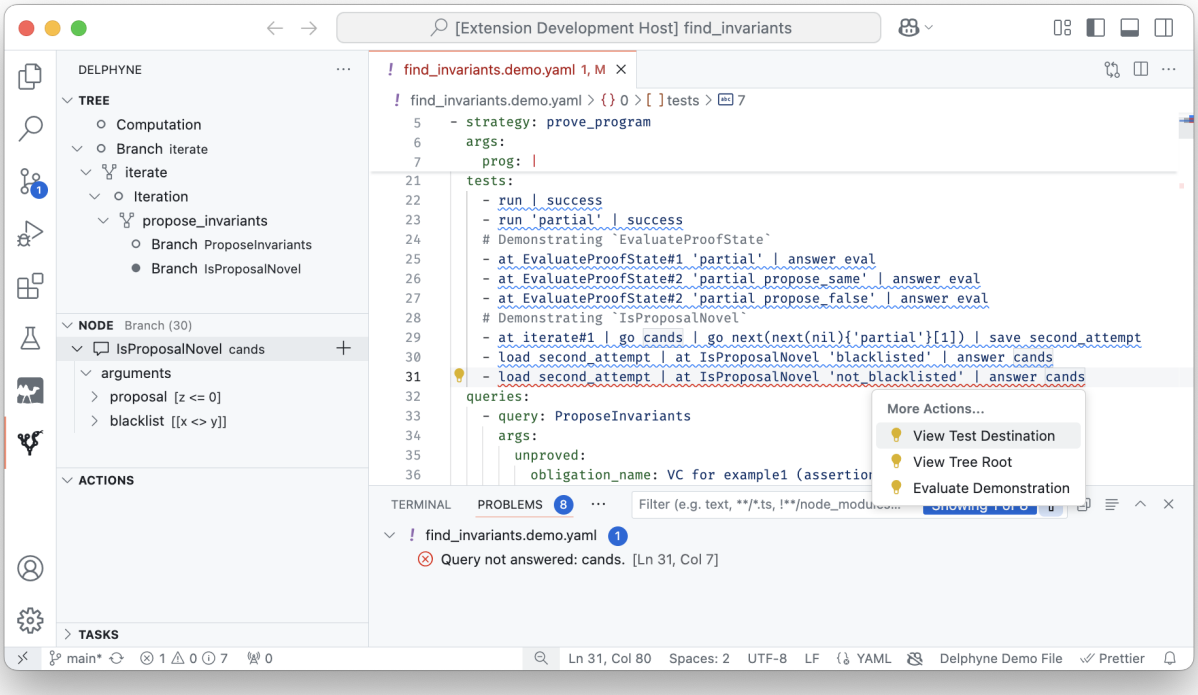


Figure 4.5: Using the Delphyne VSCode Extension to Write a Demonstration. Passing tests are underlined in blue while failing tests are underlined in red. Code actions allow inspecting the destination node of each test. Doing so for the failing test reveals that an `IsProposalNovel` query needs to be answered. This query can be added to the demonstration body by clicking on the `+` icon next to the query name. In general, the three views in the left pane allow inspecting arbitrary traces and describe one selected node at any moment in time. The `Tree` view shows the position of this node in the trace, the `Node` view shows all spaces attached to it, and the `Actions` view shows all associated actions. The trace can be navigated by clicking on actions to access children, on spaces to access nested trees, or on ancestor nodes from the `Tree` view.

4.2 Editor and Tooling Support

Our proposed demonstration language (Section 3.3) is explicitly designed to enable rich tooling support for interactively writing and repairing demonstrations. We discuss how this support is implemented in Delphyne via a dedicated VSCode extension. The separation of strategies and policies also enables structured inspection of runs of oracular programs by visualizing the resulting traces (i.e., the pruned trees containing all visited nodes; see Section 3.1.5), a capability facilitated by Delphyne's VSCode extension.

4.2.1 Writing and Repairing Demonstrations

Figure 4.5 illustrates the interactive writing of demonstrations using the Delphyne VSCode extension. In Delphyne, demonstrations are written in YAML, making many standard YAML

editing features available for free (collapsing, structural navigation, schema-based linting, anchor jumping). Demonstrations are gathered in demonstration files, and can be evaluated[5] all together or individually. Users must *explicitly* request the evaluation of demonstrations, since doing so may involve running expensive computations, but demonstrations can be evaluated concurrently. Once a demonstration has been evaluated, diagnostics are shown in the editor. When modifications to a demonstration file invalidate particular diagnostics, they automatically disappear, and the affected demonstration can be evaluated again.

As mentioned in Section 3.3.4, a typical workflow for writing a demonstration is to start with an empty `queries` section and a single `run | success` test. Evaluating the demonstration causes the test to become *stuck* at a given node. The user can then visualize this node in Delphyne's *Tree View* (Figure 4.5), along with the full trace generated by the test, and add the corresponding unanswered query to the demonstration body with a single click. An answer can then be written manually, possibly with accompanying explanations, after which the demonstration can be evaluated again. Alternatively, the user can use the *"Answer Query"* code action to open a pane showing the associated prompt and send it to LLMs, whose answers can be filtered, edited, and pasted back into the demonstration.

When a strategy change breaks a demonstration (policy changes *cannot* break demonstrations by construction), users can use Delphyne's diagnostics and tree view to investigate and resolve the issue. In some cases, Delphyne can even suggest fixes, as demonstrated in Figure 4.6.

4.2.2 Hybrid Workflows for Writing Demonstrations

More advanced workflows are also possible for writing demonstrations, where demonstrations can be partially specified, with holes automatically filled by running external policies and extracting query answers from successful searches. This allows demonstration writers to retain tight control over some parts of the writing, while outsourcing other parts already tractable for LLMs to automated generation, possibly followed by auditing and revision. Figure 4.7 demonstrates such a scenario for the Lean theorem-proving strategy developed in our second case study (Section 5.2). Such hybrid workflows are enabled by the following Delphyne features:

Policy Overriding. Given a set of query–answer pairs, policies can be wrapped so that whenever a query from the set is encountered, the specified prompting policy is bypassed and the associated answer is returned instead. This allows oracular programs to be run so that they follow partial demonstrations for all specified decisions, while performing normal search elsewhere. In Figure 4.7, this feature is enabled via the `using` clause shown in the right pane (itself generated by the *"Run Strategy"* code action).

Answer Fetching. Given a set of query–answer pairs, the demonstration interpreter can be configured to use answers from this set whenever an unanswered query is encountered. Such a set can be sourced from other demonstrations, but also from traces produced by running oracular programs. By default, an answer in a trace is considered *good* (and thus fetchable) if it occurs on a success path in the tree, but this behavior can be overridden by defining custom feedback hooks for self-improvement (see Section 5.2.3). In Figure 4.7, this feature is enabled via the `using` clause in the demonstration shown in the left pane.

[5] Evaluating a demonstration consists in running its tests (Section 3.3).

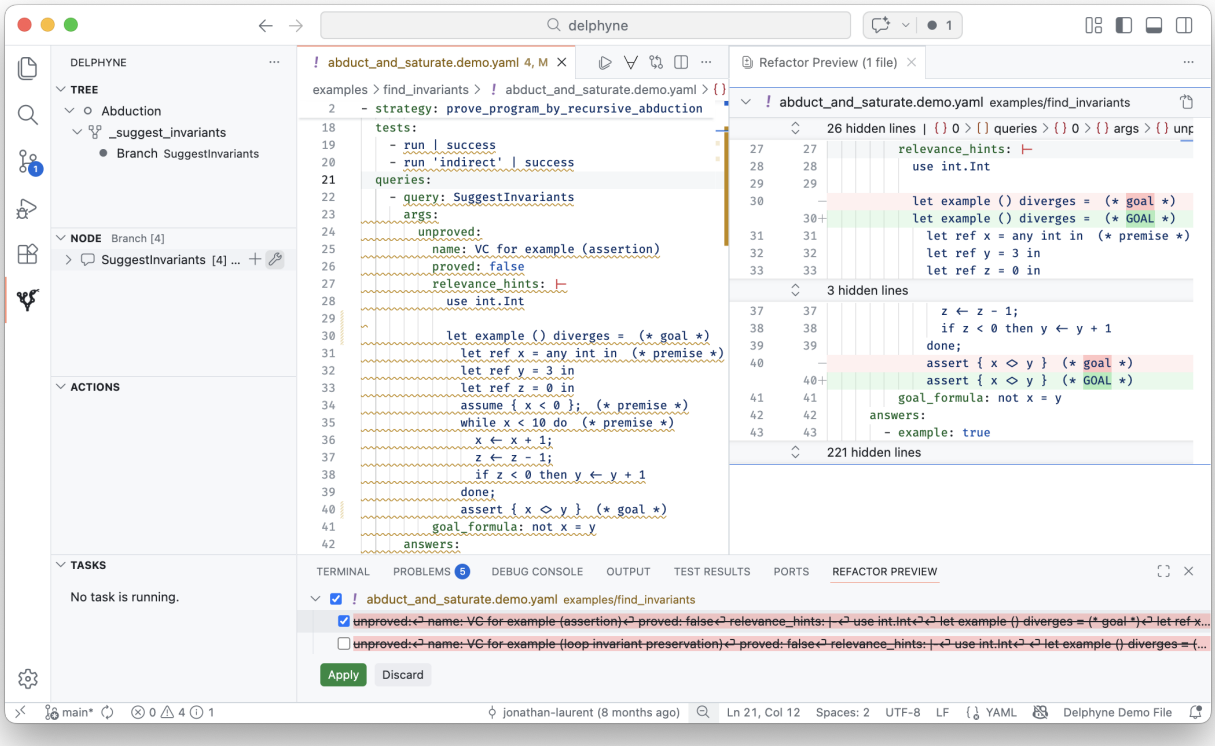


Figure 4.6: Repairing a Broken Demonstration Interactively. This screenshot illustrates a simple demonstration repair scenario that involves a program verification strategy that we developed in our first case study (Section 5.1). In this scenario, a change to the rendering of proof obligations causes a demonstration to break, as some of its queries include proof obligations. For simplicity, we consider a minor change in which only the casing of the word `goal` is altered. Evaluating the demonstration after the change causes its tests to become stuck, and some queries to be diagnosed as *unreachable*. Jumping to the node where the first test is stuck reveals the first unanswered query (`SuggestInvariants`). However, because a very similar unreachable query of the same type already exists in the demonstration body, Delphyne suggests a possible fix (revealed by clicking on the wrench icon): simply updating the existing query. The user can then inspect the proposed patch, confirm that the change to the query does not require modifying the answer, and validate the patch.

Implicit Answers. Whenever the demonstration interpreter obtains an answer to a query that does not appear in the demonstration's body, it remembers this answer as *implicit*. Whenever a demonstration evaluation uses implicit answers, a diagnostic is emitted and the user is presented with the option of explicitly adding these answers to the demonstration's body. Implicit answers are useful in combination with answer fetching, but also in other contexts such as handling the `Compute` effect (Section 4.3.1).

4.2.3 Debugging Oracular Programs

Oracular programs can be debugged in different ways. First, from any run of an oracular program, a *trace* can be extracted that describes what parts of the search tree were visited (Section 3.1.5). Traces can be serialized, deserialized, and visualized using Delphyne's *Tree View* (Figure 4.5). In addition, the `message` effect from Delphyne's standard library can be used to attach debugging messages and metadata to trees. Second, when running an oracular program, Delphyne allows the results of all LLM requests to be automatically cached, as well as the outcomes of impure, non-replicable computations performed via the `compute` effect (Section 4.3.1). This allows any run to be easily and quickly replicated, possibly with a debugger attached to follow the execution of strategies and policies step by step.

4.3 Standard Library Highlights

Delphyne provides a large, *batteries-included*, standard library. Beyond essential utilities (e.g., provider-agnostic model interfaces, parsers, prompt templates, search algorithms, example selectors, retrieval algorithms), this section highlights more original components that showcase the strength of Delphyne as an extensible, foundational framework on top of which powerful new abstractions can be built.

Section 4.3.1 introduces the standard `Compute` effect that allows including impure, stochastic, or irreplicable computations in strategies. Section 4.3.2 introduces the higher-order `interact` strategy for building ReAct-style agents [103]. Section 4.3.3 introduces the concept of *universal queries*, along with the `guess` operator used in our introductory example (Figure 1.1). Chapter 5 introduces additional abstractions for building self-improving oracular programs (Section 5.2.3).

4.3.1 Performing Impure Computations in Strategies

As explained in Section 4.1.1, strategy computations must be *pure*, so that they always reify into the same tree. Thus, strategies cannot *directly* call impure functions, including external tools such as automated theorem provers with wall-clock timeouts (which may not always return the same result on a given input). Fortunately, Delphyne allows performing such computations within strategies by wrapping them in the `Compute` effect (Figure 4.8).

Invoking `compute` is akin to branching on the answer of a special `__Compute__` query that specifies the computation to be performed, as a fully qualified function identifier obtained via reflection along with a serialized dictionary of arguments. Such a query is not to be answered by calling an LLM oracle, but rather by executing the computation.

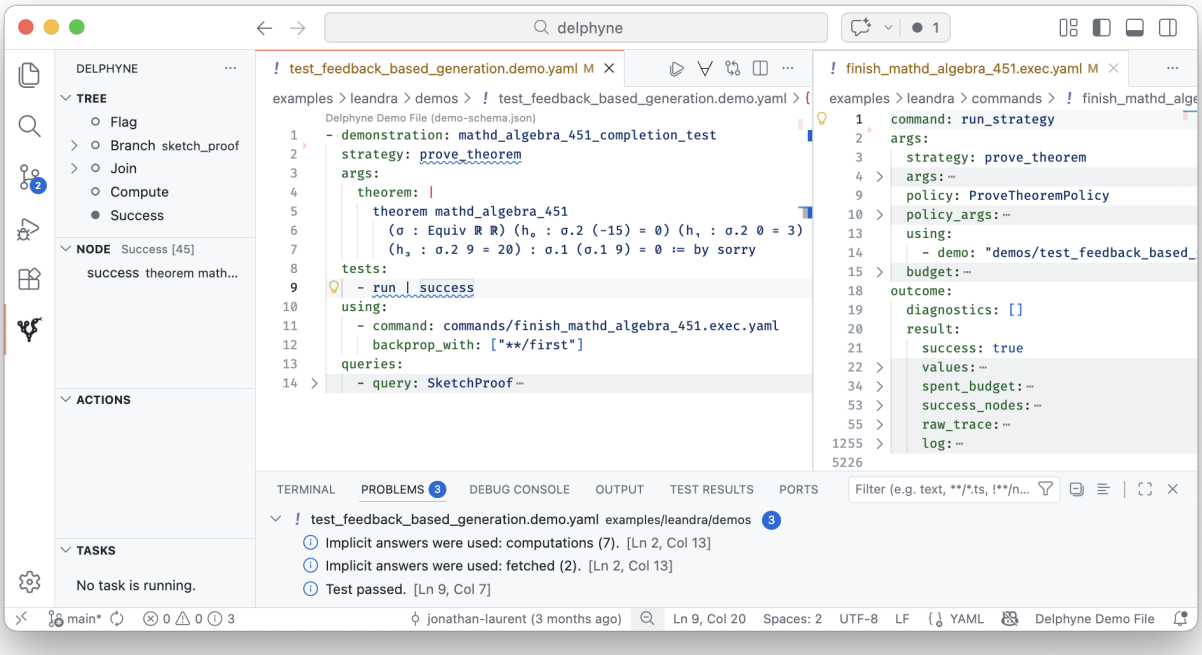


Figure 4.7: Hybrid Writing of Demonstrations. This screenshot illustrates a concrete scenario of hybrid demonstration writing (Section 4.2.2) for the Lean theorem proving strategy developed in our second case study. As a reminder, the `prove_theorem` strategy works by first generating a proof sketch and then filling in the remaining holes (Section 5.2). In this scenario, the user first provides a proof sketch for a particular problem instance by answering the `ProofSketch` query (collapsed on Line 14). Then, they use an existing policy to try to finish the proof, as follows: *(i)* they use the *"Run Strategy"* code action to create a template for a *command file* specifying a run of the strategy, *(ii)* they fill in policy and budget details in the template, *(iii)* they launch the run, which ultimately results in an `outcome` section being added to the file that includes a trace (leading to the result shown in the right pane), *(iv)* they add a `using` section to the demonstration referring to the command file (Lines 10–12), and finally *(v)* they re-evaluate the demonstration. During re-evaluation, whenever unanswered queries are encountered, the demonstration interpreter tries to fetch answers from successful paths in the run's trace, which it treats as implicit answers (Section 4.2.2). Once the tests are passing and upon inspection, the user can use the *"Add Implicit Answers (fetched)"* code action to explicitly add all implicit answers to the demonstration. Finally, note that the `backprop_with` clause on Line 12 instructs the demonstration interpreter to fetch implicit answers using *custom feedback hooks* defined by `prove_theorem` for self-improvement (see Section 5.2.3). In particular, this ensures that answers leading to correct proofs of subgoals are identified as fetchable, even if other subgoals cannot be proved and the overall run fails.

```
def compute[**A, T, P](
    f: Callable[A, T]) → Callable[A, dp.Strategy[Compute, P, T]]: ...

res = yield from compute(call_z3_prover)(formula)  # Example
```

Figure 4.8: Wrapping Impure Computations with the `Compute` Effect.

```
def interact[P, A, B](
    step: Callable[[AnswerPrefix], Opaque[P, Response[A]],
    process: Callable[[A], Opaque[P, B | Error]],
    tools: Mapping[type, Callable[[Any], Opaque[P, Any]]]
) → Strategy[Branch, P, B]: ...
```

Figure 4.9: Simplified Signature for the Standard `interact` Strategy. The `step` argument maps a conversation history to either an answer proposal or a tool call request, and can be implemented either via a query or a strategy (hence the `Opaque` return type). The `process` argument maps a proposed answer (of type `A`) to either a validated answer (of type `B`) or a feedback message (of type `Error`). The `tools` argument is a dictionary mapping types that specify tool interfaces to their implementations as oracular programs.

In demonstrations, `__Compute__` queries are seamlessly handled through the *implicit answer* mechanism introduced in Section 4.2.2. When such a query is encountered during a navigation test, the underlying computation is run in the background and the result is treated as an implicit answer. A diagnostic indicates when implicit answers were used while executing a test (Figure 4.7). At any time, the user can use the *"Add Implicit Answers (computations)"* code action to explicitly add the results of all computations to the demonstration body by inserting `__Compute__` entries into its `queries` section. This ensures that the demonstration is replicable (and can be evaluated again without rerunning the computations).

In policies, `Compute` nodes can be eliminated from a tree via an `elim_compute` tree transformer, which returns a new tree that transparently executes computations whenever needed.

Beyond including impure computations in strategies, the `Compute` effect can also be used to include *expensive* computations, ensuring their proper caching.

4.3.2 Writing ReAct Agents

Oracular programming is naturally suited for building *vertical* LLM pipelines, where traditional computations orchestrate calls to LLMs. Importantly, it also encompasses the dual *horizontal* paradigm, where an LLM orchestrates traditional computations via tool calls. These two approaches make fundamentally different trade-offs (see Section 6.1 for a complete discussion), and oracular programming allows integrating both (e.g., giving a horizontal agent access to tools that are themselves implemented as oracular programs, either horizontal or vertical).

I am executing a program that contains nondeterministic assignments along with assertions (e.g., in the form of `ensure` and `fail` statements). I am stuck at one of these nondeterministic assignments and your goal is to generate an assigned value, in such a way that the program can go on and not fail any assertion. More specifically, I'll give you three pieces of information:

- A nondeterministic program.
- The name of the variable being assigned at the program location where I am stuck.
- Some values for a number of local variables.

You must generate a correct value to assign. The expected type of this value is indicated inside the nondeterministic assignment operator. Terminate your answer with a code block (delimited by triple backquotes) that contains a YAML object of the requested type.

Figure 4.10: System Prompt for Universal Queries.

The Delphyne standard library includes a higher-order strategy named `interact` that generalizes ReAct-style [103] horizontal agents. A simplified type signature is provided in Figure 4.9. The `interact` strategy repeatedly invokes a query or sub-strategy (`step` argument) to try to produce a valid answer to a question, resending the full conversation history each time. Another query or strategy is tasked with evaluating answers and providing feedback when an answer is rejected (`process` argument). In addition, the first query or strategy may return tool call requests instead of answer candidates, with each available tool implemented as its own query or strategy (`tools` argument). The `interact` strategy is used in both our case studies (Chapter 5): to implement a baseline agent for invariant synthesis, and several times within our Lean theorem-proving agent, which integrates vertical and horizontal oracular programs in a nested fashion.

Although `interact` allows implementing basic ReAct agents, it is significantly more general. Not only can all its arguments be implemented as arbitrary oracular programs (including the `step` function that generates each additional conversation message), it also induces a tree that branches at every step of the conversation. This tree can itself be explored via arbitrary policies. For example, although standard ReAct agents explore only a single child at each level, it is also possible to *clone* the conversation state and explore continuations in parallel.

4.3.3 Universal Queries

```
x, y, z ← branch (query _ (GenTriple n))
ensure (x^2 + y^2 == z^2 && x + y + z == n)
return (x, y, z)
```

Consider the strategy computation on the right and imagine being tasked with writing a prompt for the `GenTriple` query. What would such a prompt say? Most likely, it would only restate the information already present in the surrounding strategy—namely that the oracle must find a Pythagorean triple that sums to `n`. In general, what

qualifies as an acceptable answer to a query is implicitly specified by the strategy's *continuation* at the query call site: a good answer is one after which a winning path still exists in the strategy tree. This continuation is fully determined by the strategy's source code and the current state of its stack, raising the question: can a *universal oracle* [83] make appropriate decisions directly from this information, eliminating the need to design individual prompts? Delphyne implements this idea in the form of a `guess` operator that uses runtime stack inspection to issue a *universal query*, whose default system prompt is shown in Figure 4.10. For concision, `guess` is designed to work with inner policy dictionaries (`IPDict`; see Section 4.1.2). It is used in our introductory example (Figure 1.1).

5
Case Studies

Having established oracular programming as a *principled* foundation for LLM-enabled software—with strong theoretical guarantees of separation, modularity, consistency, completeness, and evolvability—we now turn to empirical validation. This chapter demonstrates the approach and the Delphyne framework through two case studies.

Advanced Search for Loop Invariant Discovery (Section 5.1). The first case study showcases Delphyne's ability to seamlessly leverage advanced search algorithms. It demonstrates how an oracular program can be written in mere hours that is competitive with a standard ReAct baseline that uses comparable prompts, while showing a fourteenfold reduction in inference cost by combining smaller models with search. Such development speed is enabled by *(i)* the composition of advanced standard-library components, which are themselves enabled by Delphyne's extensible effect system, *(ii)* an interactive demonstration-writing workflow that allows high-quality examples to be written in mere minutes, and *(iii)* the ability to concurrently tune and test vastly different search algorithms, enabled by the separation of strategies and policies.

Self-Improvement of a Lean Prover Agent (Section 5.2). The second case study demonstrates how oracular programming enables universal yet customizable self-improvement mechanisms by automatically analyzing traces of successful and unsuccessful runs to extract retrievable examples and improve prompts. It also shows how oracular programming supports the nested integration of both *horizontal* and *vertical* pipelines, in which LLMs orchestrate traditional computations and vice versa.

Full code for both case studies is available in the Delphyne repository. All quantitative experiments can be replicated without incurring inference cost, since all LLM answers and external tool results (from Why3 and Lean) are stored in a human-readable cache that allows oracular program runs to be replayed deterministically.

5.1 Advanced Search for Loop Invariant Discovery

Our first case study showcases Delphyne's ability to leverage advanced search algorithms through the problem of *loop invariant synthesis* in Why3 [31]. LLM-enabled invariant synthesis is an active area of research [6, 52, 73, 93, 94, 97, 98] (see the recent survey by Wei et al. [93, Introduction]), and state-of-the-art performance is currently achieved by generic ReAct-style agents [103], in which large reasoning models interact with a prover within a single feedback loop [93]. We implement such an agent using Delphyne, along with an alternative agent that uses similar prompts but leverages smaller models and advanced search to drastically reduce inference costs while remaining competitive on the standard Code2Inv benchmark suite [85] (Section 5.1.1). As we demonstrate, Delphyne is uniquely suited to the rapid development of such advanced pipelines (Section 5.1.2).

5.1.1 Experimental Setting and Results

ReAct Baseline

A ReAct-style baseline agent can be implemented in a couple of lines (ignoring custom prompts and demonstrations) using the `interact` strategy (Section 4.3.2) from Delphyne's standard library. Given an input Why3 program, it repeatedly tries to suggest invariant annotations (Figure 5.3) until the program is successfully proved, receiving feedback from Why3 on failed attempts, and periodically clearing its context at a frequency determined by the policy (Figure 5.4).

A Strategy Based on Recursive Abduction

We propose an alternate strategy for invariant synthesis (Figure 5.5) that revolves around a query whose role is to suggest auxiliary invariants that may facilitate the proof of a given goal. The query takes as input: a goal (assertion or invariant) that Why3 failed to prove, the associated prover feedback, and a set of already-established invariants. Based on this information, it attempts to generate one or more candidate invariants that could plausibly serve as intermediate lemmas before eventually retrying the original proof obligation.

Delphyne defines a dedicated `Abduction` effect in its standard library that generalizes this recursive abduction pattern (Figure 5.1). Decisions on which abduction candidates to explore, in what order, and assuming which pre-established facts, are left to policies.

Two Vastly Different Policies

We define two policies for our abduction-based strategy. The first (Figure 5.6) makes short, repeated, and concurrent attempts to generate invariants from scratch, each time exploring a small number of abduction candidates (e.g., two) recursively up to some depth. The second (Figure 5.7), used in our Code2Inv experiment, is designed to handle *large* numbers of abduction candidates and uses a saturation-based search algorithm from Delphyne's standard library.

This search algorithm performs repeated *abduction rollouts* where, starting from the main goal and up to some depth, a large number of suggestions are sampled from an LLM and

Agent	Problems Solved	Average Price (¢)	Median Price (¢)
Abduction (gpt-4o-mini)	**124.0 ± 0.0**	**0.12 ± 0.02**	**0.05 ± 0.00**
Baseline (gpt-4o)	113.0 ± 2.0	3.19 ± 0.31	0.80 ± 0.17
Baseline (gpt-4o-mini)	120.0 ± 1.7	1.64 ± 0.24	0.10 ± 0.01
Baseline (o3)	**124.0 ± 0.0**	1.90 ± 0.03	1.58 ± 0.13

Table 5.1: Experimental Results on Code2Inv (124 problems, excluding incorrect ones). Standard deviations are estimated based on three random seeds. A 20¢ budget limit is fixed for each problem (see Table C.1 in Appendix C.1 for details on model pricing).

semantically deduplicated before one of them is recursively examined, based on how many times it was suggested and explored before (in a way reminiscent to the UCT formula [11]). Established facts are accumulated and systematically assumed once proved. The whole process is restarted periodically and runs until some allocated budget is consumed. The invariant suggestion prompt asks for *concise* and *constrained* answers, as lists of pairs that each consist in *(i)* a short identifier referring to a relevant piece of advice from the system prompt (Appendix C.1.2) and *(ii)* a proposed invariant. Such concision allows sampling more answers at any given budget, trading precision for quantity and diversity, to be leveraged by search.

Results on Code2Inv

We compare our abduction-based oracular program to three ReAct agent baselines, each using a different model. All agents receive demonstrations for the same three example problems, with comparable levels of explanation, and all receive identical advice in their system prompt, including advice on using abduction to identify missing invariants (Appendix C.1.2). The key difference therefore lies not in how much domain-specific knowledge is shared in prompts, but in how model calls are chained and orchestrated. Each agent is tasked to solve all 124 problems from the Code2Inv benchmark suite [85], with a 0.20$ budget limit on LLM API spending per problem. All baselines are properly tuned (Appendix C.1.1). Results are shown in Table 5.1. The abduction-based agent solves all problems, with an average spending per problem that is over an order of magnitude lower than baselines. Interestingly, the ReAct agent based on the smaller GPT-4o-mini model outperforms the one based on GPT-4o on all metrics, by virtue of being cheaper and thus being afforded more trials per problem within its budget limit.

These results illustrate the effectiveness of orchestrating well-scoped LLM queries and prover feedback through advanced search. More importantly, Delphyne allows building such pipelines in mere hours, by composing reusable components and writing demonstrations interactively.

5.1.2 A Case for Oracular Programming

Delphyne allowed us to express the core logic of our abduction-based oracular program in less than fifty lines of strategy code (Figure 5.5), which directly express domain-specific knowledge about invariant generation, and which we never had to revisit (in contrast to the search

logic, as we discuss next). It allowed us to write three demonstrations in mere minutes, by simply *(i)* choosing example programs to verify, *(ii)* writing empty demonstrations with single `run | success` tests, and *(iii)* following the guidance of the demonstration interpreter to answer relevant queries until all tests passed (the universality of this workflow is guaranteed by the strong completeness property of the demonstration language; Section 3.3.3). Some of these queries were large—featuring Why3 feedback in the form of failing proof obligations—but all were automatically generated and so we only had to *answer* them. Finally, after a strategy update made more detailed Why3 feedback available, the Delphyne VSCode extension enabled us to update demonstrations with almost no effort. It identified breakages—in the form of stuck tests and unused demonstration queries—and proposed query updates in the form of inspectable diffs. By reviewing these diffs, we were able to validate the proposed changes and, in some cases, refine the explanations accompanying our answers to better leverage the new feedback information (see simplified scenario in Figure 4.6).

By leveraging the standard `Abduction` effect, which is definable in its full generality using Delphyne's extensible effect system, our strategy accommodates a wide variety of policies. We iteratively improved our saturation-based policy, introducing additional hyperparameters to better control the computational cost of propagating known facts. This process was most time-intensive, but—crucially—no change was ever needed to the strategy code, and thus to the demonstrations, allowing us to iterate fast and to maintain and optimize multiple policies concurrently. Had our program been written in a traditional way, without a strict separation between core and search logic, frequent refactorings would have been required, and supporting multiple search algorithms at the same time would have introduced significant additional complexity.

5.1.3 Details on Strategies and Policies

In this section, we provide more details on the `Abduction` effect and then provide code highlights for our proposed strategies and policies.

Recursive Abduction and Invertibility

We show a signature for the `Abduction` effect in Figure 5.1, along with the associated navigation function in Figure 5.2. This navigation function is invertible (Section 3.3.3) *if (i)* all proofs of the same fact are indistinguishable (proof irrelevance), *(ii)* the `suggest` function is *surjective* in the sense that it can output any valid fact on any input, and *(iii)* the `prove` function is *monotonic* in the sense that for any prefix `fs` of `fs'`, if there exists a proof element in `prove fs f`, then there exists a proof element in `prove fs' f`. A proof sketch proceeds as follows. From the reference of any action (i.e., a local value representing a proof of the main goal), one can extract a directed acyclic graph representing the dependencies between all involved auxiliary facts, where each fact depends on all other facts used to establish it via `prove`. The `navigate` procedure can then produce a proof of the main goal by having `suggest` output all dependencies of each fact that fails to be proved. Monotonicity is important, since `navigate` may attempt to prove each fact while assuming more auxiliary facts than its original dependencies.

```
effect abduction :: forall fact proof feedback. (Abduction ∈ s, Eq fact) ⇒
  { prove :: ([(fact, proof)], Maybe fact) → Opaque p (Either proof feedback)
  , suggest :: feedback → Opaque p [fact]
  , searchEquivalent :: ([fact], fact) → Opaque p (Maybe fact)
  , redundant :: ([fact], fact) → Opaque p Bool } → Strategy s p proof
```

Figure 5.1: The `Abduction` Effect for Recursive Abduction. The `prove` function takes as an argument a list of pre-established facts along with a new fact to prove (or `Nothing` to refer to the main goal). It returns a proof, or, failing that, some feedback. The `suggest` function maps such feedback to a list of suggested auxiliary facts to prove. The `searchEquivalent` function determines whether a given fact is equivalent to another one from a given list (useful for removing semantic duplicates), and the `redundant` function determines whether a fact is implied by a list of other facts. Actions are proofs of the main goal. Figure 5.2 describes the associated navigation function, which is invertible under mild assumptions (Section 5.1.3).

Code Highlights

Figure 5.3 shows the strategy for our baseline ReAct agent, while Figure 5.4 shows the associated policy. Our abduction-based strategy is shown in Figure 5.5, with its two associated policies in Figures 5.6 and 5.7. All featured strategies and policies are fully typed. In policies, the `@` operator composes policies with streams or tree transformers, while the `&` operator pairs a search policy parameterized by an inner policy (e.g., `dfs`) with a concrete inner policy. In addition, policies usually do not include an explicit parameter for the global budget limit, since the Delphyne experiment launcher provides facilities for automatically composing policies with the `with_budget` stream transformer.

5.2 Self-Improvement of a Lean Prover Agent

Every oracular program is automatically equipped with a natural self-improvement mechanism reminiscent of Expert Iteration [2]. Each time a problem is solved through search, examples of correct decisions can be extracted along the tree path leading to success, which can then be used to improve future executions—for instance through few-shot prompting, retrieval [34], or fine-tuning. Delphyne allows domain-specific knowledge to be leveraged for refining this coarse mechanism—for example, by integrating negative feedback, assigning credit for the successful solving of independent subproblems, or retroactively revisiting answers in strategies that perform iterative repair. It does so by defining a `Feedback` effect that strategies can use to declare *hindsight knowledge* about the outcomes of previous choices.

In this second case study, we first provide an overview of a Lean theorem-proving oracular program on which we demonstrate self-improvement (Section 5.2.1), and then reflect on how Delphyne facilitates its construction (Section 5.2.2). Next, we describe how self-improvement mechanisms are implemented in the Delphyne standard library and can be customized via the

```
1  @dataclass
2  class Abduction(dp.Node):
3      prove: ...  # types omitted for brevity
4      suggest: ...
5      search_equivalent: ...
6      redundant: ...
7
8      def navigate(self):
9
10         def aux(fact):
11             res = yield self.prove([], fact)
12             status, payload = res[0], res[1]
13             if status.value == "proved":
14                 return [(fact, payload)]
15             elif status.value == "disproved":
16                 return []
17             else:
18                 assert status.value == "feedback"
19                 feedback = payload
20                 suggestions = yield self.suggest(feedback)
21                 proved = []
22                 for s in suggestions:
23                     extra = yield from aux(s)
24                     proved.extend(extra)
25                 res = yield self.prove(proved, fact)
26                 status, payload = res[0], res[1]
27                 if status.value == "proved":
28                     proved.append((fact, payload))
29                 return _remove_duplicates(
30                     proved, by=lambda x: drop_refs(x[0]))
31
32         proved = yield from aux(None)
33         main_proof = _find_assoc(proved, None)
34         assert main_proof is not None, "Failed to prove the main goal"
35         return main_proof
```

Figure 5.2: Navigation Function for `Abduction` nodes (Figure 5.1). This navigation function first attempts to prove the main goal. If that fails, it recursively proves each abducted suggestion in order before trying again, while maintaining a global list of proved facts. If the main goal cannot be proved at the end of the recursion, an error is raised (Line 34), providing feedback to demonstration writers. This navigation function is conditionally invertible (Section 5.1.3).

```python
@strategy
def prove_program_interactive(
    prog: why3.File,
) -> Strategy[Branch, dp.PromptingPolicy, why3.File]:
    annotated = yield from dp.interact(
        step=lambda prefix, _:
            AnnotateWithInvs(prog, prefix).using(dp.ambient_pp),
        process=lambda invs, _:
            check_invariants(prog, invs).using(dp.just_compute))
    return annotated

@strategy
def check_invariants(
    prog: why3.File, invariants: Sequence[why3.Formula]
) -> dp.Strategy[Compute, None, why3.File | dp.Error]:
    annotated = why3.add_invariants(prog, invariants)
    feedback = yield from dp.compute(why3.check)(prog, annotated)
    if feedback.success:
        return annotated
    feedback.obligations = [
        o for o in feedback.obligations if not o.proved]
    return dp.Error(label="feedback", meta=feedback)

@dataclass
class AnnotateWithInvs(dp.Query[dp.Response[Sequence[why3.Formula], Never]]):
    prog: why3.File
    prefix: dp.AnswerPrefix
    __parser__ = dp.last_code_block.yaml.response
```

Figure 5.3: Baseline Strategy for Finding Invariants. The `interact` function is a higher order, standard library strategy that allows implementing ReAct agents (Section 4.3.2). The `prove_program_interactive` strategy directly takes a prompting policy as its inner policy, since it only issues a single query. The `check_invariants` strategy only uses the `Compute` effect and so does not need an inner policy. The `AnnotateWithInvs` query has a `prefix` field that is used to store past conversation history. Its answer type is wrapped in `Response`, allowing `interact` to access such history (the `Never` argument means that no tool call is allowed). System and instance prompts are stored in external Jinja2 template files.

```python
def prove_program_interactive_policy(
    model_name: str,
    temperature: float | None = None,
    max_feedback_cycles: int = 3
):
    model = dp.standard_model(model_name)
    sp = dp.loop() @ dp.dfs(max_depth=max_feedback_cycles+1)
    pp = dp.few_shot(model, temperature=temperature, max_requests=1)
    return sp & pp
```

Figure 5.4: Policy Associated with the Baseline Strategy for Invariant Synthesis (Figure 5.3). This policy repeatedly starts a new conversation with at most `max_feedback_cycles` rounds of feedback until success or budget exhaustion. See Section 5.1.3 for explanations on the syntax.

`Feedback` effect (Section 5.2.3). Finally, we present the strategies and policies involved, with an emphasis on concrete code examples (Section 5.2.4).

5.2.1 Experimental Setting and Results

We implement a strategy for proving theorems in Lean, inspired by Draft-Sketch-Proof [46]. Given a theorem, a proof sketch is first produced, and the resulting goals are then processed independently by a dedicated substrategy (using the `Join` effect). This substrategy has access to a tool for finding relevant theorems in Mathlib, itself implemented as a custom strategy that interacts with Loogle. An example execution is provided in Appendix C.2.1. Evaluated on the MiniF2F [106] *test set*, our agent solves 140 problems (57%). It then undergoes an automatic self-improvement cycle that proceeds as follows. First, it attempts to solve each problem from the MiniF2F *validation set*. Then, examples from these attempts are collected and made available for few-shot prompting through a prompting policy that uses MMR retrieval [14]. In addition, a *meta-strategy* examines all search traces (Section 3.1.5) to identify queries that were ultimately correctly answered, but only at the cost of substantial search effort. For each such query, positive and negative feedback is aggregated and distilled into reusable pieces of advice, to be added back to the agent's prompts (Appendix C.2.2). The improved agent solves 148 problems (60%) on the MiniF2F test set, while exhibiting a better scaling trend (Figure 5.9).

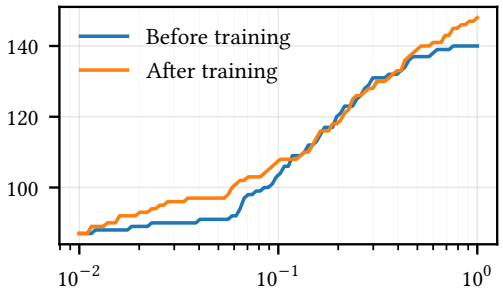


Figure 5.9: Number of MiniF2F test problems solved as a function of the per-problem inference budget (in dollars), *before* and *after* training.

```python
@strategy
def prove_program_by_recursive_abduction(
    prog: why3.File,
) -> Strategy[dp.Abduction, ProveProgIP, why3.File]:
    invs = yield from dp.abduction(
        prove=lambda proved, goal:
            _prove_goal(prog, proved, goal)
                .using(lambda p: p.prove),
        suggest=lambda feedback:
            _suggest_invariants(feedback)
                .using(lambda p: p.suggest),
        search_equivalent=lambda facts, fml:
            _search_equivalent(facts, fml)
                .using(lambda p: p.search_equivalent),
        redundant=lambda proved, fml:
            _is_redundant(proved, fml)
                .using(lambda p: p.is_redundant))
    return why3.add_invariants(prog, invs)

@strategy
def _suggest_invariants(
    unproved: Sequence[why3.Obligation],
) -> Strategy[Branch | Fail, dp.PromptingPolicy, Sequence[Formula]]:
    assert len(unproved) > 0
    # We focus on the first unproved obligation
    answer = yield from dp.branch(
        SuggestInvariants(unproved[0]).using(lambda p: p))
    return [s.invariant for s in answer.suggestions]

@dataclass
class SuggestInvariants(dp.Query[Sequence[InvariantSuggestions]]):
    unproved: why3.Obligation

@dataclass
class InvariantSuggestion:
    trick: str
    invariant: str
```

Figure 5.5: An Abduction-Based Strategy for Finding Invariants. See Figures 5.1 and 5.2 for the definition of the `Abduction` effect, and Figures 5.6 and 5.7 for associated policies.

```
def prove_program_by_repeated_recursive_abduction(
    model_name: str,
    max_suggestions: int | None = 2, max_abduction_depth: int = 2,
    num_parallel: int = 3, temperature: float | None = None,
):
    ip = ProveProgIP.make(
        dp.few_shot(
            dp.standard_model(model_name),
            temperature=temperature, max_requests=1))
    sp = dp.abduct_recursively(
        max_suggestions=max_suggestions, max_depth=max_abduction_depth)
    return dp.loop() @ dp.parallel([sp for _ in range(num_parallel)]) & ip
```

Figure 5.6: A Simple Parallel Policy for Abduction-Based Invariant Synthesis. From the last line, a fixed number of concurrent attempts are repeatedly performed in parallel. The `abduct_recursively` policy handles `Abduction` nodes using the same logic implemented by their navigation function (Figure 5.2), by recursively calling itself on abduction candidates while maintaining a global list of proved facts. The `ProveProgIP.make` function creates an inner policy for `prove_prog` from a prompting policy for suggesting invariants (other optional arguments are available to configure prover timeouts).

```
def prove_program_by_saturation(
    model_name: str,
    num_suggestion_completions: int = 4, max_abduction_depth: int = 2,
    max_requests_per_attempt: int = 4, temperature: float | None = None
):
    ip = ProveProgIP.make(
        dp.few_shot(
            dp.standard_model(model_name), temperature=temperature,
            num_concurrent=num_suggestion_completions, max_requests=1))
    sp = dp.abduct_and_saturate(
        max_rollout_depth=max_abduction_depth+1,
        max_proved=64, max_candidates=64, remember_disproved=False,
        max_raw_suggestions_per_step=8*num_suggestion_completions)
    per_attempt = dp.BudgetLimit({dp.NUM_REQUESTS: max_requests_per_attempt})
    return dp.with_budget(per_attempt) @ dp.loop() @ sp & ip
```

Figure 5.7: A Saturation-Based Policy for Abduction-Based Invariant Synthesis. This policy was used in our Code2Inv experiment. See Section 5.1.3 for explanations. See Appendix C.1.1 for details on how parameters were tuned. All hyperparameters for `abduct_and_saturate` past the first one were set to limit the computational cost of propagating facts (not involving LLMs).

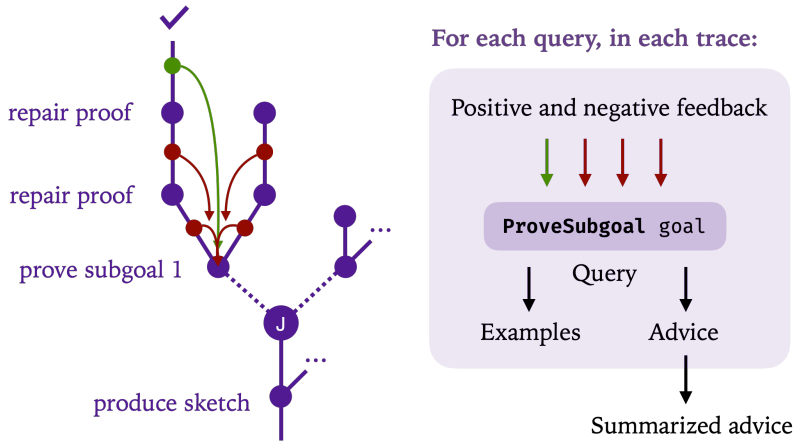


Figure 5.8: Custom Feedback Propagation for our Lean Theorem Proving Strategy. Small green and red dots correspond to custom `Feedback` nodes (Section 5.2.3), while `J` denotes a `Join` node. Positive feedback is issued for each proved subgoal (green arrows), while Lean errors result in negative feedback (red arrows). Importantly, these custom feedback nodes can be specified by adding only four lines of code to our theorem-proving strategy (Figure 5.10). For each query in each trace, the received feedback is used to generate retrievable examples and also generalized into reusable advice, which is then aggregated and added to prompts.

5.2.2 A Case for Oracular Programming

Beyond the quantitative result above, the separation of strategies and policies enables generic yet customizable self-improvement mechanisms, by allowing the outcomes of arbitrary runs to be reified into structured traces and reflected upon (see Section 5.2.3 for details). In the present case, Delphyne allowed us to implement self-improvement for our theorem-proving oracular program by merely inserting *four* new lines of strategy code (to collect positive feedback when a subgoal is solved even if the overall proof fails, record prover errors as negative feedback, and link successful proofs obtained after multiple rounds of feedback to all intermediate failed attempts; see Section 5.2.4) and defining custom prompts for generating and aggregating advice.

In addition, our theorem-proving strategy exploits Delphyne's ability to integrate *horizontal* and *vertical* LLM pipelines, where LLMs orchestrate traditional computations and vice versa. This is achieved using Delphyne's standard higher-order `interact` strategy (Section 4.3.2). At the top level, our proving agent implements a vertical sketch–prove pipeline (Section 5.2.4). The component responsible for closing subgoals is implemented as a horizontal agent with access to a tool for finding relevant Mathlib lemmas, itself implemented with a custom strategy.

Finally, the demonstration language and its associated tooling allowed us to write a handful of high-quality examples to bootstrap self-improvement in mere minutes. In addition to the interactive writing workflow illustrated in our invariant synthesis case study (Section 5.1), we leveraged a more automated writing workflow (Section 4.2.2), where we provided proof sketches and let our existing policy try to close as many proof goals as possible, only supplying manual proofs where it failed, refining the style, and adding explanations.

5.2.3 Self-Improving Oracular Programs

Every oracular program is automatically equipped with a natural self-improvement mechanism: each time a problem is solved through search, examples of correct decisions can be extracted along the tree path leading to success, which can then be used to improve future executions. Oracular programming is particularly amenable to self-improvement, since the separation of strategies and policies ensures that *any* run of an oracular program can be reified into a highly structured *trace* object (Section 3.1.5), which can itself be analyzed in a process that we call *search reflection*. The nested tree structure of traces carries rich information for learning. For example, answers on success paths can be automatically labeled as good, and the number of unsuccessful alternative paths explored at each decision level provides information about how *challenging* the decision was and how valuable an extracted example would be. In addition, traces can incorporate further information by allowing strategies to encode domain-specific insights through custom feedback nodes.

Emitting Feedback Messages. Traces can be enriched with additional feedback nodes, refining and customizing the coarse self-improvement mechanism described above. Delphyne's standard library[1] defines a `Feedback` node type for this purpose, together with two trigger functions, `emit_feedback` and `backprop_feedback`, that induce such nodes.

Strategies can use `emit_feedback` to provide custom hindsight feedback about any *decision* made in a search tree. By *decision*, we refer to the selection of an element from an opaque space. For any decision, four types of feedback messages can be emitted: `GoodValue`, indicating that the decision was good; `BetterValue`, specifying a better decision that could have been made; `BadValue`, indicating that the decision was bad; and `BadValueAlso`, specifying an alternative decision that would also have been bad. All messages can carry explanations as metadata, and the `BetterValue` and `BadValueAlso` messages additionally include alternative values.

For example, after successfully proving a subgoal, our Lean theorem-proving strategy emits a `GoodValue` message (Figure 5.10, Lines 26–27), registering a partial success in the event that the surrounding proof is not completed. In addition, the standard `interact` strategy (Section 4.3.2), which is used several times in our Lean theorem-proving strategy, can be configured to emit a `BadValue` message when a proof sketch or tactic is rejected by Lean, and to emit a `BetterValue` message when a successful sketch or tactic is found after several repair iterations, indicating that such a proposal could have been made from the start (Figure 5.10, Lines 17 and 39).

Propagating Feedback in Traces. Feedback messages can be automatically propagated through traces until they reach queries. Messages concerning the answer to a query stop there, while those concerning the outcome of a strategy are processed by feedback backpropagation handlers. Strategies can define such handlers via the `backprop_feedback` function. These handlers receive a message about the final strategy outcome and return messages about inner decision points. When no custom handler is defined, the default backpropagation handler only recognizes the `GoodValue` message and automatically forwards it to all inner decision points.

[1]This also demonstrates the power of oracular programming as a *foundational* framework: all self-improvement utilities can be defined as library components that leverage Delphyne's extensible effect system.

The standard `interact` strategy (Section 4.3.2) defines two different backpropagation handlers, which can be selected externally. For example, upon receiving a `GoodValue` message about its final outcome, it can either forward this message to the *last* call to `step` that produced it, or convert it into a `BetterValue` message for the *first* call to `step`.

Leveraging Feedback. A standard way to perform search reflection is to selectively enable a number of feedback emitters and handlers, propagate feedback, and gather it at the level of each query (Figure 5.8). In our Lean case study, positive feedback received at each query is used to generate few-shot examples that are later selected via retrieval. Examples are extracted with priority from queries that also received the greatest amount of negative feedback, indicating that significant search was required to reach a good decision. In addition, for each query with both positive and negative feedback, an LLM is tasked with generalizing this feedback into a short piece of reusable advice. All feedback generated across a large number of problem instances is then sent to another LLM, which aggregates it into a small number of maximally useful and relevant advice items to be added to prompts (see Appendix C.2.2 for examples).

Custom feedback nodes can also be leveraged by the demonstration interpreter in hybrid demonstration writing workflows (Section 4.2.2). When adding `using` clauses to demonstrations to fetch implicit answers from a given trace (Figure 4.7), feedback is propagated and one fetchable answer is generated for each `GoodValue` or `BetterValue` message reaching a query. The `backprop_with` argument can be used to customize this behavior by selectively enabling emitters and backpropagation handlers.

5.2.4 Details on Strategies and Policies

Strategy Overview. We provide a sketch for our Lean theorem-proving strategy in Figure 5.10. An example run is discussed in Appendix C.2.1. At the top level, the `prove_theorem` strategy first generates a sketch via the `sketch_proof` sub-strategy and then concurrently fills every hole in the proof via the `fill_hole` sub-strategy, using the `Join` effect (Line 4). Both `sketch_proof` and `fill_hole` use the standard `interact` strategy (Section 4.3.2) to let oracles orchestrate Lean calls (Lines 11 and 32). In addition, `fill_hole` has access to an additional tool for finding auxiliary theorems in Mathlib, implemented via a custom `find_theorem` strategy (Lines 37-38) that internally uses `interact` to orchestrate Loogle requests (the wrapping in the standard `nofail` combinator ensures that failures of the `find_theorem` strategy do not result in failures of `fill_hole`, but instead cause an empty tool response to be returned).

Four lines of strategy code were added to customize self-improvement feedback (Section 5.2.3). In `fill_hole`, Lines 26 and 27 ensure that feedback is obtained from successful proofs of subgoals, even when the surrounding proof fails, by emitting hindsight feedback about the prior outcome of `prove_subgoal`. This outcome originates from Line 22, where `branch` is called with the special option `with_ref=True` so that a reference to the corresponding decision is returned and can later be used for targeting feedback. In addition, an additional `produce_feedback=True` argument is passed to `iterate` (Lines 17 and 39), so as to instruct it to *(i)* emit negative feedback when an answer is not validated by `process` and *(ii)* backpropagate positive feedback to the initial call to `step` when a good answer is generated after several rounds of feedback (the `unprocess` argument allows converting a final validated answer into the unvalidated answer

that should, in hindsight, have been returned by the initial call to `step`, and should yield the identity function when left-composed with `process`).

Policy Overview. We provide a sketch of the policy associated with our Lean theorem-proving strategy in Figure 5.11. Its structure mirrors that of the associated strategy (Figure 5.10), and it is defined using Delphyne's `PolicyRecord` pattern, in which each sub-policy is represented as a serializable dataclass with an `instantiate_with` method that produces an actual policy from its record representation. We provide more details on the implementation of `SketchProofPolicy` (Lines 17–21): to produce a proof sketch, it makes a series of attempts, each time using a different random procedure for selecting few-shot examples (Figure 5.12). Such randomization improves the robustness of our policy, since different problem instances may benefit from different example-selection policies (e.g., prioritizing relevance or diversity in different settings).

Defining a Custom Example Selector. We define the aforementioned example selection procedure in Figure 5.12, combining primitives from the Delphyne standard library. The `ExampleSelector` dataclass can be instantiated into a stream of example selectors, which is used in the implementation of `SketchProofPolicy` (Figure 5.11, Lines 17–21). Each example selector in the stream first selects a number of high-quality, human-written examples from the main demonstration file, then uses MMR retrieval [14] with relevance weight λ to select n_s examples originating from self-improvement runs, and finally randomly samples n_i examples from them. The parameters n_s, n_i, and λ are initially set to fixed values and are then repeatedly and independently sampled from finite lists of choices (see the definition of `Randomized`).

```
@strategy
def prove_theorem(theorem) :
    sketch, goals = yield from dp.branch(sketch_proof(theorem).using(...))
    subproofs = yield from dp.join(
        [fill_hole(theorem, sketch, i, goal)
        for i, goal in enumerate(goals)])
    return fill_sketch(theorem, sketch, subproofs)

@strategy
def sketch_proof(theorem):
    sketch_and_goals = yield from dp.interact(
        step=lambda prefix, _:
            SketchProof(theorem, prefix).using(...),
        process=lambda sketch, _:
            check_sketch(theorem, sketch).using(...),
        # ask 'interact' to issue feedback once a sketch is found
        produce_feedback=True, unprocess=lambda sg: sg[0])
    return sketch_and_goals

@strategy
def fill_hole(theorem, sketch, hole_index, goal):
    proof, proof_ref = yield from dp.branch(
        prove_subgoal(theorem, sketch, hole_index, goal).using(...),
        return_ref=True)
    # We can issue partial feedback even if the surrounding proof fails
    yield from dp.emit_feedback("subproof", [
        dp.send(dp.GoodValue(), proof_ref)])
    return proof

@strategy
def prove_subgoal(theorem, sketch, hole_index, goal):
    proof = yield from dp.interact(
        step=lambda prefix, _:
            ProveGoal(goal, prefix).using(...),
        process=lambda proof, _:
            process_proof(theorem, sketch, hole_index, proof).using(...),
        tools={TheoremRequest: lambda call:
            dp.nofail(find_theorem(call).using(...), default=None)},
        produce_feedback=True, unprocess=lambda proof: proof)
    return proof
```

Figure 5.10: A Strategy for Proving Theorems in Lean. See Section 5.2.4 for explanations.

```python
@dataclass
class ProveTheoremPolicy(dp.PolicyRecord[...]):
    sketch: SketchProofPolicy | None = None
    subgoal: ProveSubgoalPolicy | None = None

    def instantiate_with(self, env: dp.PolicyEnv): ...

@dataclass
class SketchProofPolicy(dp.PolicyRecord[...]]):
    model_name: str = "gpt-5"
    effort: dp.ReasoningEffort = "medium"
    examples: ExampleSelectorStream | None = None
    max_full_attempts: int = 4
    max_feedback_rounds_per_attempt: int = 4
    lean_timeout: float = 8.0

    def instantiate_with(self, env: dp.PolicyEnv):
        examples = self.examples or ExampleSelectorStream()
        selectors = itertools.islice(
            examples.instantiate(env.random), self.max_full_attempts)
        return dp.sequence((... for sel in selectors))

@dataclass
class ProveSubgoalPolicy(dp.PolicyRecord[...]):
    model_name: str = "gpt-5-mini"
    effort: dp.ReasoningEffort = "low"
    max_full_attempts: int = 3
    max_feedback_rounds_per_attempt: int = 3
    max_requests_per_attempt: int = 7
    examples: ExampleSelectorStream | None = None
    lean_timeout: float = 8.0
    find_theorem: FindTheoremPolicy | None = None
    ...

@dataclass
class FindTheoremPolicy(dp.PolicyRecord[...]):
    model_name: str = "gpt-5-mini"
    effort: dp.ReasoningEffort = "low"
    max_requests: int = 4
    ...
```

Figure 5.11: Policy Sketch for the Lean Theorem-Proving Strategy from Figure 5.10.

```
@dataclass
class ExampleSelectorStream:
    """
    Specification for a stream of example selectors that include a certain
    amount of permanent examples along with extra ones selected using MMR.
    """
    model_name: dp.StandardOpenAIEmbeddingModel = "text-embedding-3-large"
    num_selected: Randomized[int] = Randomized(5, [5])
    num_included: Randomized[int] = Randomized(5, [5, 15])
    mmr_lambda: Randomized[float] = Randomized(0.5, [0.3, 0.5, 0.7])

    def instantiate(self, rng: random.Random):
        nss = self.num_selected.stream(rng)
        nis = self.num_included.stream(rng)
        mls = self.mmr_lambda.stream(rng)
        for ns, ni, ml in zip(nss, nis, mls):
            permanent = dp.all_examples.such_that(
                lambda ex: ex.demo_file is not None
                and MAIN_DEMO_FILE in ex.demo_file.stem)
            extra = dp.maximum_marginally_relevant(
                k=ns, lambda_param=ml, model_name=self.model_name
            ).random(ni)
            # We want the closest examples returned by MMR to appear last
            yield (extra + permanent).reverse()

@dataclass
class Randomized[T]:
    """
    Specification for a hyperparameter that is first picked
    deterministically and then randomly.
    """
    first: T
    then: Sequence[T]

    def stream(self, rng: random.Random) -> Iterable[T]:
        yield self.first
        while True:
            yield rng.choice(self.then)
```

Figure 5.12: Example Selection Procedure Used by the Lean Theorem-Proving Agent. It is used by the policy from Figure 5.11 and explained in Section 5.2.4.

6
Discussion and Conclusion

We introduced a principled framework for composing traditional computations with LLM oracles, combining the modularity and clear semantics of the former with the inductive capabilities of the latter. By separating core logic and search logic into strategies and policies, respectively, this framework enables rapid and concurrent experimentation with advanced search algorithms, while providing a foundation for self-improvement via search reflection. It also elevates few-shot examples as grounded, evolvable program components, deserving of their own language.

We conclude this thesis by situating oracular programming within recent trends in agentic design (Section 6.1), discussing its inherent trade-offs and limitations while identifying opportunities for future work (Section 6.2). We then review related work in LLM programming, effectful programming, and probabilistic programming (Section 6.3). Finally, we close with a call for creativity, in the hope that our framework may foster the discovery of new idioms for weaving generative AI with symbolic computation (Section 6.4).

6.1 Comparison with ReAct Agents

An increasingly popular paradigm for leveraging LLMs is the ReAct-style agent framework [103], exemplified by systems such as Claude Code, in which a large reasoning model with an append-only context coordinates calls to external tools. Although the power, versatility, and simplicity of this framework have led to its wide adoption, it raises persistent challenges in terms of reliability, steerability, security, and cost:

- Long contexts incur substantial inference costs and can degrade model performance through context distraction, confusion or poisoning [15].
- Composing tool calls through the model's context[1] is flexible but expensive, and raises fundamental security vulnerabilities via prompt injection [22]. For this reason, the field

[1] A standard framework for agentic design is to give a model access to a variety of tools (e.g., through MCP servers). Tool calls are then combined by letting the model issue an initial call, adding the result to the conversation context, and using this information to produce a subsequent tool call. This approach is not only expensive but also

is increasingly moving away from MCP servers and toward having agents compose tool calls by generating traditional programs, which can themselves be analyzed and executed in sandboxed environments. In some cases, these programs can themselves issue calls to LLMs [22, 105], presenting a clear opportunity for oracular programming.

- For improved steerability and predictability, many agents rely on prompts that specify highly specific workflows, which are essentially algorithms described in natural language (e.g., Numina-Lean-Agent [65]). However, there is no guarantee that such workflows will be faithfully followed and none of the standard software tooling is available for exploring and maintaining them.

Having traditional computations orchestrate LLM calls, which oracular programming enables with unprecedented flexibility and generality, offers tight control over semantics and context curation. As demonstrated in our invariant synthesis case study (Section 5.1), it also has the potential to reduce inference cost by leveraging smaller models and search. On the other hand, the dual ReAct paradigm of having LLMs orchestrate traditional computations offers complementary strengths, notably its universality and its ability to tackle broad classes of problems with unanticipated edge cases. We do not call for one paradigm to supplant the other, but for the two to be combined, through *composition* (e.g., our Lean case study in Section 5.2 combines vertical and horizontal strategies) and *staging* (e.g., having ReAct agents generate, analyze, and execute oracular programs, to be explored in future work).

6.2 Limitations and Future Work

Physical and Conceptual Overhead. Separating strategies from policies is powerful, but also introduces overhead—both physical and conceptual—that may not always be justified, especially for building simple pipelines. Physically, fully separating policies introduces overhead through the need to define inner policy types, and redundancy, since the structure of a policy must mirror the structure of the associated strategy (a constraint enforced through types). Conceptually, leveraging oracular programming in its fullest requires users to learn many new abstractions and idioms, which can be a barrier to adoption. Even for advanced users, it can add cognitive overhead, since there are often multiple ways to split a program's logic between strategies and policies with subtle trade-offs[2].

Such overhead can be mitigated, by building abstraction layers on top of oracular programming's foundations that offer a simpler interface for expressing common pipelines. Universal queries and inner policy dictionaries (Section 4.3.3) play this role in our introductory example (Section 1.1). In addition, although our framework allows full separation of strategies and policies, it does not *mandate* it: policies can be specified inline within strategies, as arguments

raises security concerns, since tool outputs may contain sensitive data or injected prompts.

[2]Consider our abduction-based invariant synthesis strategy. Here, we chose to shift as much of our program's logic as possible to the policy side through the use of the `Abduction` effect, so as to enable the greatest variety of search algorithms to be used. However, a more elementary strategy can also be written in the style of Figure 2.2 that only uses `branch` and `ensure`, hardcoding the structure of recursive abduction calls into the strategy. Doing so still leaves plenty of room for policies to implement diverse search algorithms, but abstracts away key information about the relationship between branches—information that our saturation-based, `abduction`-aware policy exploits.

of the `using` methods (Figure 4.1). Even when writing simple pipelines that do not exploit the full expressive power of Delphyne, there is value in doing so within a framework that provides a continuous path toward greater customization and more advanced search. Finally, coding agents could lower the barrier to entry by partially or fully automating the writing of oracular programs. Oracular programs are particularly well suited to this setting because they support strong consistency enforcement mechanisms that provide useful feedback to the agent.

An Incomplete Enforcement of Consistency. Oracular programming provides strong mechanisms for enforcing consistency between strategies, policies, and demonstrations, via types and tests. However, such mechanisms are incomplete and can be further improved.

For example, policies can be composed in ways that are well-typed yet still inconsistent (e.g., using depth-first search on an infinitely branching tree, sequencing two search algorithms when the first is nonterminating, requesting multiple LLM completions with a temperature of zero). Future work may explore ways to detect such inconsistencies through improved type systems, abstract interpretation, or more sophisticated runtime checks.

In addition, although demonstration tests ensure that the provided decision examples suffice to solve actual problem instances, they cannot guarantee consistency with the more informal prompting instructions that some policies may provide, especially regarding unparsed chains of thought or reasoning tokens. Future work may investigate using small language models to perform fast sanity checks on the consistency of prompts and examples.

Advanced Pipelines Require Advanced Tuning. Oracular programming particularly shines when building advanced pipelines involving many well-scoped queries and sophisticated search. However, such pipelines require careful tuning: each additional search hyperparameter or prompt introduces a potential weak link that may cause the entire pipeline to fail if misconfigured. This highlights a key strength of the ReAct paradigm: it introduces very few search parameters and is relatively forgiving of imperfect prompts, since models operate with access to the full problem context and can often disregard prompt inconsistencies.

We believe that self-improvement is key for oracular programming to reach its full potential. Our second case study (Section 5.2) demonstrated a limited form of self-improvement, where retrievable examples and prompt augmentations are extracted from runs. Beyond those, future work may consider updates to search parameters, policies, and even strategies themselves.

6.3 Related Work

LLM Programming Frameworks

Many existing frameworks aim at bridging prompting and programming and facilitating the development of LLM-enabled software (see survey [66]). Ours is unique in leveraging the full power of nondeterministic programming to integrate prompts with general computations, while allowing arbitrarily advanced search logic to be specified independently. It is also the first to identify and address the challenge of treating few-shot examples as *grounded* and *evolvable* program components, deserving of their own language.

Languages such as LMQL [7] and SGLang [107] enable guiding LLM inference by enforcing strict answer templates, within a single completion. They naturally complement our framework, which allows orchestrating multiple requests.

LangGraph [1] provides graph-based orchestration abstractions for LLM agents and tool calls. Its runtime offers useful capabilities such as scheduling, observability, replay, and support for human-in-the-loop interactions.[3] However, its graph abstractions restrict how control flow can be expressed, reproducing only a limited subset of standard programming constructs in an ad-hoc manner (e.g., routing operators used in place of conditionals). By contrast, Delphyne leverages the full expressive power of Python to express control logic directly. LangGraph does offer a functional API that allows graphs to be constructed dynamically from Python code, but doing so reduces the benefits provided by its runtime. Delphyne remains unique in allowing the full separation of core and search logic and in offering a demonstration language.

Mellea [45] emphasizes treating LLM requests as unreliable, nondeterministic operations whose outcomes must be rigorously validated, and provides a rich library of utilities and prompting templates for doing so. However, LLM queries are fundamentally treated as guarded function calls, whereas Delphyne goes further by explicitly reifying strategies into trees that are amenable to advanced search. Mellea also revisits a key idea from object-oriented programming in the context of LLM programming by introducing the concept of an `MObject`, which packages data together with LLM-powered operations.

Finally, the DSPy framework [53] allows building LLM pipelines by composing declarative modules with natural-language–typed signatures. Notably, it allows automatically optimizing prompts and examples based on an objective function, drawing a similarity to backpropagation-based learning frameworks such as PyTorch [71], and inspiring our own self-improvement mechanism. DSPy's separation of LLM computations into *signatures* and *optimizable modules* parallels our own separation of *queries* and *prompting policies*. However, absent a corresponding distinction at the higher level at which such modules are composed (as achieved by our *strategies* and *search policies*), it only supports primitive forms of search.

Effectful and Nondeterministic Programming

Our proposed extensible effect system draws inspiration from the Freer monad [54] and interaction trees [101]. It combines these ideas with open unions for representing signatures and, more importantly, with additional structure imposed on effects in the form of member spaces, a locality constraint over actions, and local navigation. More generally, and beyond expressing nondeterminism, effect systems enable the separation of pure computations from abstract interactions with the external world, whose meaning can be defined independently and modularly while still supporting precise reasoning about the pure aspects of programs.

The idea of defining search spaces via nondeterministic programs is long-standing [33], but we push it further by demonstrating the potential of LLMs to serve as universal oracles (an idea pioneered by Selsam [83]) and by enabling an unprecedented separation between strategies and policies while preserving modularity and type safety via inner policy types. For example, although miniKanren [13] allows some separation between the specification of

[3] Delphyne also provides strong observability and replay facilities (Section 4.2.3). Integrating human oracles is an interesting avenue for future work.

modeling constraints and search heuristics, the two are still typically interleaved within the same program (e.g., via choices between `conde`, `conda`, or `condi` to express nondeterminism).

Probabilistic Programming

Our use of nondeterministic programming to separate core and search logic draws parallels with *probabilistic programming* [89]. For example, languages such as Pyro [8] or Gen [18] allow the separate specification of probabilistic programs and inference policies. These policies involve not only global decisions (e.g., whether to use MCMC or variational methods), but also local ones (e.g., which guide distribution to use at each sampling location), making our concept of *inner policy type* relevant for building new, statically type-safe probabilistic programming languages.

Gen [18] in particular bears further similarities with Delphyne, by allowing the modular composition of generative functions (i.e., probabilistic programs) through hierarchical address spaces, and offering the full expressiveness of its host language (i.e., Julia) to express them. Still, Gen policies operate on completely different abstractions (choice maps instead of trees), have no resource-awareness, and are not as composable (no direct counterpart to our stream and tree transformers).

6.4 Conclusion

LLMs and traditional programs are natural complements: the former provide inductive capabilities that symbolic computation lacks, leveraging common-sense knowledge and generalization to perform tasks implicitly specified through natural-language instructions and examples, while the latter offer reliability, modularity, and precise semantics. Yet bringing these strengths together presents unique software-engineering challenges that have so far constrained the complexity of LLM-enabled programs. In this thesis, we identified the separation of *core logic* and *search* as a key principle for addressing these challenges, along with the elevation of examples into *grounded*, *evolvable* program components deserving of their own language.

As we demonstrated both theoretically through strong guarantees and empirically through case studies, separating strategies and policies enables the concurrent exploration of advanced search algorithms while providing a strong foundation for self-improvement via search reflection. In addition, our novel demonstration language enables high-quality examples to be written and repaired in mere minutes through an interactive, test-driven development workflow.

Our approach complements recent trends in ReAct-style agentic design, trading off simplicity and universality for increased control over semantics, parallelism, security, and cost. We expect future agents to weave these two paradigms in increasingly elaborate ways, not only through composition but also through staging. For example, personal assistants could translate natural-language prompts into oracular programs that are auditable, provably secure, resource-efficient, and capable of self-improvement across repeated use. We hope that oracular programming and its practical implementation, Delphyne, can serve as an extensible foundation for building such intersymbolic AI systems and inspire the discovery of new idioms for agentic design.

A

Details about the Looprl Experiment

A.1 Implementing Abduction

Both the teacher and solver strategies we use as examples in this paper rely on an `abduct` function that takes as input a formula F and then either proves it valid or returns a (possibly empty) set of assumptions A such that $A \rightarrow F$ can be proved valid.

Implementing such a function is hard in the general case and so an implementation of `abduct` may leverage nondeterminism. However, because we are only dealing with arithmetic of limited complexity in this work, we implemented a fully-specified abduction procedure for linear integer arithmetic that relies on Fourier-Motzkin elimination [21].

When given a formula F, our abduction procedure first rewrites it in conjunctive normal form as $F = \bigwedge_{i=1}^{n} \bigvee_j F_{ij}$ where F_{ij} are atomic formulas of the form $\sum_k a_k x_k \odot c$ where $\odot \in \{\geq, =\}$ and c, a_k are integer constants.

Case where $n = 1$. If F can be expressed as a disjunction of atomic formulas, we can compute abduction suggestions as follows: *i)* one considers the negation of F, obtaining a set of atomic assumptions and *ii)* one derives as many consequences as possible from those assumptions using Fourier-Motzkin elimination (linearly combining inequalities so as to eliminate variables). If a contradiction is derived, then F is valid. If no contradiction is derived given some timeout, then the negation of any derived consequence can be considered as an abduction candidate.

For example, suppose we want to compute abduction candidates for $F = x \geq 0 \rightarrow x+y \geq 1$. The conjunctive normal form of F is $(x < 0 \vee x + y \geq 1)$. Taking the negation yields $(x \geq 0 \wedge x + y < 1)$, which we normalize into the set of assumptions $\{x \geq 0, -x - y \geq 0\}$ (all variables are integers). Then, we can take a Fourier-Motzkin step by adding these two assumptions and derive the following consequence: $-y \geq 0$. After this, no other reasoning step is applicable and we end up with the following set of facts: $\{x \geq 0, -x - y \geq 0, -y \geq 0\}$. We therefore suggest the following abduction candidates: $x < 0$, $x + y > 0$ and $y > 0$.

Case where $n > 1$. If the conjunctive normal form of F has two conjuncts or more, we apply the procedure above to each conjunct separately. For example, suppose that $F = G \wedge H$, G admits a set of abduction candidates $\{A_i\}_i$, and H admits a set of abduction candidates $\{B_i\}_i$. One possibility would be to return all combinations $\{A_i \wedge B_j\}_{ij}$ as abduction candidates for F. However, doing so can quickly result in a combinatorial explosion. Therefore, our implementation does something different and returns the union of the sets $\{A_i\}_i$ and $\{B_i\}_i$ instead. By doing so, it cannot guarantee that any resulting abduction candidate A is sufficient to imply F. Rather, abduction candidates are seen as suggestions to unblock one part of the proof (i.e. enable proving one conjunct of F), but not necessarily the whole proof.

A.2 Training Hyperparameters

We provide an exhaustive list of all hyperparameter values used in our experiments in Table A.1.

Table A.1: Training hyperparameters for the experiments described in Section 2.3. Hyperparameters are organized according to a hierarchical structure that is represented using indentation.

Parameter	Value	Description
`params`		All hyperparameters.
`· teacher`		Hyperparameters for the teacher agent.
`· · agent`		Hyperparameters common to all AlphaZero agents.
`· · · num-iters`	20	Total number of training iterations.
`· · · num-problems-per-iter`	8000	Number of data generation episodes per iteration.
`· · · num-validation-problems`	800	Number of validation-data generation episodes per iteration.
`· · · num-workers`	600	Number of episodes simulated in parallel or asynchronously during data generation.
`· · · num-processes`	none	Number of distinct CPU processes spawned for data generation (by default, this value is set to the number of available physical CPU cores).
`· · · search`		Proof search limits.
`· · · · max-proof-length`	60	Maximum number of allowed environment steps before failing automatically.
`· · · · max-probe-size`	80	Maximum allowed size of state encodings. Bigger state encodings result in immediate failures.
`· · · · max-action-size`	12	Maximum allowed size of action encodings. Actions with bigger encodings are discarded.

`· · · encoding`		State and action encoding hyperparameters.
`· · · · d-model`	128	Embedding size for tokens, which is also equal to the neural network's state dimension.
`· · · · pos-enc-size`	32	Maximal size of the tree positional encoding added to each token embedding.
`· · · · uid-emb-size`	16	Maximum number of unique identifiers that can be encoded.
`· · · · const-emb-size`	0	Maximum number of bits used to encode the binary value of numerical constants. No encoding is done if a value of 0 is provided.
`· · · · enable-numerical-edges`	true	Add comparison edges between numerical constants.
`· · · · add-reverse-edges`	true	Add reverse edges for all edge types.
`· · · network`		Hyperparameters for the Dynamic Graph Transformer network.
`· · · · num-heads`	4	Number of transformer attention heads.
`· · · · probe-encoder-layers`	6	Number of transformer blocks used to encode states.
`· · · · action-encoder-layers`	3	Number of transformer blocks used to encode actions
`· · · · combiner-layers`	1	Number of transformer blocks used in the combiner network that combines state and action encodings.
`· · · · dropout-rate`	0.05	Dropout rate.
`· · · · num-head-layers`	2	Number of layers in the feed-forward networks used for the value and policy heads.
`· · · · head-dim`	256	Hidden dimension of layers in the feed-forward networks used for the value and policy heads.
`· · · mcts`		MCTS hyperparameters.
`· · · · num-simulations`	64	Number of MCTS simulations used for planning every environment step.
`· · · · num-considered-actions`	8	The number of top-ranked actions that are considered in the sequential halving Gumbel exploration process.
`· · · · value-scale`	0.1	The c_{scale} hyperparameter (see original Gumbel AlphaZero paper).
`· · · · max-visit-init`	50	The c_{visit} hyperparameter (see original Gumbel AlphaZero paper).
`· · · · fpu-red`	0.1	Value estimates for unvisited children are decreased by this value so as to encourage deeper search trees (see LC0 AlphaZero implementation).

`· · · · reset-tree`	true	Whether the MCTS tree is reset after an environment move is taken.
`· · · · max-tree-size`	256	Maximal size of the MCTS tree. This parameter is relevant to avoid out-of-memory errors when `reset-tree` if false.
`· · · · dirichlet-alpha`	10	Concentration parameter for the Dirichlet exploration noise (see original AlphaZero paper).
`· · · · dirichlet-eps`	0.25	Magnitude of the Dirichlet exploration noise. No Dirichlet exploration noise is normally used with Gumbel AlphaZero but we add some anyway in the teacher for increased diversity.
`· · · training`		Network training hyperparameters.
`· · · · max-epochs`	6	The maximum number of training epochs.
`· · · · improvement-required`	1	Gradient updates are stopped if the validation loss fails to decrease for more than this number of epochs.
`· · · · batch-size`	400	Training batch size.
`· · · · lr-base`	0.0005	Maximal learning rate used in a cosine learning rate schedule with warm-up.
`· · · · warmup-epochs`	0.2	Length of the warm-up part of the learning rate schedule (in epochs).
`· · · · weight-decay`	0.01	Weight decay parameter.
`· · · · outcome-loss-coeff`	0.7	Coefficient for the loss term evaluating outcome prediction accuracy (e.g. success or failure).
`· · · · event-loss-coeff`	3	Coefficient for the loss term evaluating event prediction accuracy.
`· · · · policy-loss-coeff`	1	Coefficient for the loss term evaluating the divergence from policy priors to their targets.
`· · · training-window`	1, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6, 7	The number of previous iterations from which samples are used to update the network at each iteration. If the provided list is shorter than the total number of iterations, the last number in this list is duplicated as many times as necessary.
`· solver`		Hyperparameters for the solver agent.
`· · agent`		Hyperparameters common to all AlphaZero agents.
`· · · num-iters`	20	Total number of training iterations.
`· · · num-problems-per-iter`	20000	Number of data generation episodes per iteration.

`· · · num-validation-problems`	5000	Number of validation-data generation episodes per iteration.
`· · · num-workers`	600	Number of episodes simulated in parallel or asynchronously during data generation.
`· · · num-processes`	none	Number of distinct CPU processes spawned for data generation (by default, this value is set to the number of available physical CPU cores).
`· · · search`		Proof search limits.
`· · · · max-proof-length`	12	Maximum number of allowed environment steps before failing automatically.
`· · · · max-probe-size`	80	Maximum allowed size of state encodings. Bigger state encodings result in immediate failures.
`· · · · max-action-size`	12	Maximum allowed size of action encodings. Actions with bigger encodings are discarded.
`· · · encoding`		State and action encoding hyperparameters.
`· · · · d-model`	128	Embedding size for tokens, which is also equal to the neural network's state dimension.
`· · · · pos-enc-size`	32	Maximal size of the tree positional encoding added to each token embedding.
`· · · · uid-emb-size`	16	Maximum number of unique identifiers that can be encoded.
`· · · · const-emb-size`	0	Maximum number of bits used to encode the binary value of numerical constants. No encoding is done if a value of 0 is provided.
`· · · · enable-numerical-edges`	true	Add comparison edges between numerical constants.
`· · · · add-reverse-edges`	true	Add reverse edges for all edge types.
`· · · network`		Hyperparameters for the Dynamic Graph Transformer network.
`· · · · num-heads`	4	Number of transformer attention heads.
`· · · · probe-encoder-layers`	6	Number of transformer blocks used to encode states.
`· · · · action-encoder-layers`	3	Number of transformer blocks used to encode actions
`· · · · combiner-layers`	1	Number of transformer blocks used in the combiner network that combines state and action encodings.
`· · · · dropout-rate`	0.1	Dropout rate.
`· · · · num-head-layers`	2	Number of layers in the feed-forward networks used for the value and policy heads.

· · · · `head-dim`	256	Hidden dimension of layers in the feed-forward networks used for the value and policy heads.
· · · `mcts`		MCTS hyperparameters.
· · · · `num-simulations`	32	Number of MCTS simulations used for planning every environment step.
· · · · `num-considered-actions`	8	The number of top-ranked actions that are considered in the sequential halving Gumbel exploration process.
· · · · `value-scale`	0.1	The c_{scale} hyperparameter (see original Gumbel AlphaZero paper).
· · · · `max-visit-init`	50	The c_{visit} hyperparameter (see original Gumbel AlphaZero paper).
· · · · `fpu-red`	0	Value estimates for unvisited children are decreased by this value so as to encourage deeper search trees (see LC0 AlphaZero implementation).
· · · · `reset-tree`	false	Whether the MCTS tree is reset after an environment move is taken.
· · · · `max-tree-size`	256	Maximal size of the MCTS tree. This parameter is relevant to avoid out-of-memory errors when `reset-tree` if false.
· · · · `dirichlet-alpha`	10	Concentration parameter for the Dirichlet exploration noise (see original AlphaZero paper).
· · · · `dirichlet-eps`	none	Magnitude of the Dirichlet exploration noise. No Dirichlet exploration noise is normally used with Gumbel AlphaZero but we add some anyway in the teacher for increased diversity.
· · · `training`		Network training hyperparameters.
· · · · `max-epochs`	1	The maximum number of training epochs.
· · · · `improvement-required`	1	Gradient updates are stopped if the validation loss fails to decrease for more than this number of epochs.
· · · · `batch-size`	300	Training batch size.
· · · · `lr-base`	0.0003	Maximal learning rate used in a cosine learning rate schedule with warm-up.
· · · · `warmup-epochs`	0.2	Length of the warm-up part of the learning rate schedule (in epochs).
· · · · `weight-decay`	0.01	Weight decay parameter.
· · · · `outcome-loss-coeff`	1	Coefficient for the loss term evaluating outcome prediction accuracy (e.g. success or failure).

`* * * * event-loss-coeff`	1	Coefficient for the loss term evaluating event prediction accuracy.
`* * * * policy-loss-coeff`	1	Coefficient for the loss term evaluating the divergence from policy priors to their targets.
`* * * training-window`	1, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6, 7	The number of previous iterations from which samples are used to update the network at each iteration. If the provided list is shorter than the total number of iterations, the last number in this list is duplicated as many times as necessary.
`* num-teacher-iters-used-by-solver`	5	The number of teacher training iterations from which solver training samples are collected.
`* extra-teacher-problems`	10000	Number of extra teacher problems that are generated once the teacher is trained, with no exploration noise.

A.3 Network Architecture and Choice Points Encoding

Network Architecture

Neural networks that are used as oracles for nondeterministic strategies take as an input a *choice point* and return *i)* a probability distribution over all available choices, *ii)* some success and failure probability estimates and *iii)* some event occurrence probability estimates (see Section 2.1.4). In turn, a choice point is represented as *i)* a *probe* [82] that encodes all relevant state information provided to `choose` (see Figure 2.7 for examples) and *ii)* a list of possible choices that were passed as arguments to `choose`.

Our proposed network architecture works as follows. Given a choice point, the probe is encoded using a Dynamic Graph Transformer (DGT) neural network [83]. DGT networks are similar to Transformers [90] but they allow leveraging some additional graph structure over the source tokens by associating edge types to learned attention biases. Each choice is encoded separately using another DGT. It is then concatenated with the probe encoding and passed to a combiner network that outputs a score. Scores are normalized into a probability distribution using a softmax operation. The probe encoding is also passed to a value head that produces outcome and event predictions. Batches of choice points can be evaluated efficiently in parallel using scatter operations [29].

Encoding Programs and Formulas

All data that is to be passed to the neural network must be encodable as a sequence of tokens with an optional graph structure. This is the case for programs and formulas in particular. We use standard techniques to encode them [42, 84]:

- Abstract syntax trees (ASTs) are encoded into a sequence of tokens in polish notation order (e.g. $x + 3y$ is encoded as "`PLUS VAR(x) MUL CONST(3) VAR(y)`"). The edges in the

Edge type	Description
PARENT	Connect any token to its parent in the syntax tree.
PREV_SIBLING	Connect any token to its previous sibling in the syntax tree.
PREV_LEXICAL_USE	Connect any token for name s to the previous occurrence of s.
LAST_READ	Connect a variable occurrence to places where it was possibly read last.
LAST_WRITE	Connect a variable occ. to places where it was possibly written last.
GUARDED_BY	Connect a variable occ. to assumptions made about it.
GUARDED_BY_NEG	Connect a variable occ. to negated assumptions made about it.
COMPUTED_FROM	Connect the lhs of any assignment to the variables in its rhs.
SAME_CONST	Connect two identical numerical constants.
SMALLER_CONST	Compare two different numerical constants.

Table A.2: Edge types used to encode formulas and programs. We organize these edges into three groups: syntactic edges, semantic edges and numerical edges. Within a conditional statement, every variable in the `if` branch is connected to the guard using a GUARDED_BY edge whereas every variable in the `else` branch is connected to the guard using a GUARDED_BY_NEG edge.

original AST are preserved as graph edges to be passed to the DGT network.

- We use a different positional encoding scheme for tokens that leverages the tree structure of the underlying AST [84]. Doing so has been demonstrated to result in better generalization capabilities [36].
- Identifier names are randomly mapped into a finite number of unique ids for each choice point. Unique ids are encoded using a one-hot encoding scheme. If a choice point introduces more names than there are unique ids available (rare), the last occurring names are made to share a single uid that is flagged to indicate a naming conflict.
- Numerical constants with an absolute value greater than 3 are represented with generic POS_CONST and NEG_CONST tokens. A binary encoding of their value is also provided to the network. Moreover, special edges are added to compare all numerical constants involved in a program or formula (see Table A.2).
- Following [42], we also add semantic edges to the encoding of programs and formulas that reflect the way information flows within them. Details can be consulted in Table A.2.

B

Oracular Programming in Haskell

We defined the triad of oracular programming in Chapter 3 via an embedding in Haskell. This appendix provides more details on this embedding.

Full type-checked code is also available in the supplementary material, sometimes with small, inconsequential variations (e.g., additional `Typeable` constraints that were omitted from the main text for simplicity). The automatic derivation of node effect types is not implemented. Instead, node effect types are manually defined for several examples, as are normally auto-generated methods such as `spaces` and `mapEmbedded`.

B.1 Local Values and References

Figure B.1 provides type definitions for *local values* and *references* (Sections 3.1.5 and 3.1.7). Local values index the children of a node and are obtained by combining elements of local spaces.

Local Values. A *local value* (Lines 2-4) is a pair consisting of a value and a *value reference* that indicates its origin relative to a tree node identified by a phantom type parameter `n`. Local values cannot be constructed directly; instead, they can only be obtained by combining local space elements via a number of combinators (Lines 18-27).

References. A *value reference* (Line 7) is an *assembly* (Line 8) of local *space element references*—i.e., it refers to a value obtained by combining space elements through the introduction and elimination of lists, pairs, `Maybe`, `Either`, and `()`, all of which are encoded in references using lists.[1] A *space element reference* (Lines 12-14) denotes an element of a local parametric space (all spaces are considered parametric, be it over unit type) and is defined by a *space name* (a string denoting a field of the effect type), a *value reference* for the space parameter, an *answer* in the case of a query, and a *node reference* to a success node in the case of a nested tree. A *node*

[1]For example, a reference to `(Just _)` is encoded as `[_]`, while a reference to `Nothing` is encoded as `[]`.

```
1  -- Local value
2  data Local n a  -- No constructor is exposed
3  getRef :: Local n a → ValueRef
4  dropRef :: Local n a → a
5
6  -- Local value reference (relative to a given node)
7  type ValueRef = Assembly SpaceElementRef
8  data Assembly a = Atom a | List [Assembly a] | Element Int (Assembly a)
9  data SpaceRef = Main | SpaceRef SpaceName ValueRef
10 data SpaceElementRef
11   = Answer SpaceRef String
12   | Result SpaceRef NodeRef
13
14 -- Node reference, relative to the root of a (possibly nested) tree
15 data NodeRef = Root | Child NodeRef ValueRef
16
17 -- Combining local values
18 nil :: Local n ()
19 liftPair :: (Local n a, Local n b) → Local n (a, b)
20 liftList :: [Local n a] → Local n [a]
21 liftMaybe :: Maybe (Local n a) → Local n (Maybe a)
22 liftEither :: Either (Local n a) (Local n b) → Local n (Either a b)
23 unliftPair :: Local n (a, b) → (Local n a, Local n b)
24 unliftList :: Local n [a] → [Local n a]
25 unliftMaybe :: Local n (Maybe a) → Maybe (Local n a)
26 unliftEither :: Local n (Either a b) → Either (Local n a) (Local n b)
27 castLocal :: (Typeable a, Typeable b) ⇒ Local n a → Maybe (Local n b)
28
29 -- Recovering a success value from a reference
30 successValue :: Tree s p n v → NodeRef → Maybe (Local n v)
```

Figure B.1: Definition of References and Local Values

reference (Line 15) consists of a list of *value references* denoting a sequence of actions to follow from the root.

B.2 Effect Types and Associated Methods

All effect types (derived using the rules defined in Figure 3.9) must implement the `Effect` type class defined in Figure B.2. Methods of `Effect` are only relevant to the demonstration language (Section 3.3), with the exception of `mapEmbedded`, which is useful for defining tree transformers, since tree transformers must *generically* update all embedded trees in nodes that contain them (e.g., `Join`). The `spaces` and `mapEmbedded` methods can be automatically generated from the effect declaration. All other methods have default implementations, so the only method that must be defined manually in every case is `navigate` (Appendix B.2).

```
1  class Effect e where
2    spaces :: (Space t, Typeable a) ⇒  -- auto-generated
3      e p t n a → [(SpaceName, SomeParametricSpace n)]
4    mapEmbedded :: (Space t, Space t') ⇒  -- auto-generated
5      (forall v. t n v → t' n v) → e p t n a → e p t' n a
6    navigate :: (Space t, MonadFail m) ⇒  -- mandatory
7      e p t n a → Maybe (ChoiceFun m n → m (Local n a))
8    validAction :: e p t n a → Local n a → Bool  -- optional
9    hasPrimarySpace :: e p t n a → Bool  -- optional
10   nodeTags :: e p t n a → [Tag]  -- optional
11
12 class Space sp where  -- Instances: ‘Tree s p‘ and ‘LocalOpaque p‘
13   source :: sp n v → SpaceSource n v
14   spaceTags :: sp n v → [Tag]
15
16 data SpaceSource n v
17   = forall s p. SourceTree (Tree s p n v)
18   | forall q. (Query q, Res q ∼ v) ⇒ SourceQuery (AttachedQuery n q)
19
20 data SomeParametricSpace n =
21   forall sp i v. (Space sp, Typeable i, Typeable v) ⇒
22   SomeParametricSpace (Local n i → sp n v)
23
24 type ChoiceFun m n = forall sp v. (Space sp) ⇒ sp n v → m (Local n v)
```

Figure B.2: Definition of the `Effect` Type Class.

As a reminder, an effect type `e` is a type constructor with parameters `p` (the inner policy type associated with the surrounding tree), `t` (the type of embedded trees), `n` (the phantom type identifying the current node), and `a` (the action type of the current node).

Spaces. The `spaces` function returns a list of *parametric* and *nonparametric spaces*, each indexed by a *local value* (Appendix B.1) of some type (`()` for nonparametric spaces). Spaces can be of two kinds—*opaque spaces* and *embedded trees*—corresponding to the two instances of the `Space` type class (Lines 12-14). The `spaces` function is useful for manipulating *generic* trees with *unknown* or *arbitrary* effects. Its output features runtime type annotations that can be used to perform safe casting and runtime type checks, hence the `Typeable` instances (some `Typeable` constraints were omitted in the main text for simplicity; see the full Haskell embedding described in Appendix B for details). These features are useful for implementing an interpreter for the demonstration language (Section 3.3), which is dynamic in nature. However, specific policies such as `dfs` can access nested spaces directly via the fields of specific effects, thereby benefiting from strong static typing. For the purposes of the demonstration language, a *space* (Lines 12-14) must specify a *source*, which is either a *tree* or a *query*. Spaces can also specify *tags*, which by default are derived from the name of the attached strategy or query.

Navigation Functions. Navigation functions must be implemented for every effect type via the `navigate` method (Lines 6-7). For leaf nodes such as `Fail`, `navigate` must return `Nothing`. For other nodes, it must return a navigation function that maps a *choice function* (Line 24) to an action. A choice function maps a local space to one of its elements. Note that choice functions (and hence `navigate`) are allowed to be monadic, since the demonstration interpreter uses choice functions that inspect answered queries from demonstrations and that may fail in the case of missing answers. In addition, navigation functions themselves may fail, hence the `MonadFail` constraint on Line 6 (see the navigation function for the `Abduction` effect in Figure 5.2 for an example, Line 34).

C

Details on Case Studies

C.1 Advanced Search for Loop Invariant Discovery

This appendix provides details on our invariant synthesis case study (Section 5.1). Appendix C.1.1 elaborates on the experimental protocol, while Appendix C.1.2 shows the system prompts used for both the baseline and abduction-based agents. Table C.1 provides details on model pricing.

C.1.1 Experimental Protocol Details

Sensitive policy parameters were tuned for all agents by selecting the values that led to the highest number of solved Code2Inv problems (the average cost is only used in cases of ties, although both metrics have a strong inverse correlation).

Tuning of the abduction-based agent. Parameters for our abduction-based agent were chosen as follows. The `num_suggestion_completions` parameter was set to 8, aiming to balance the cost of LLM input and output tokens. The `max_abduction_depth` parameter was set to 2 (performing more than 2 nested abduction steps seems unnecessary for single-loop invariant generation). Values of 4 and 8 were tried for the `max_requests_per_attempt` parameter (i.e., the frequency at which all established facts are forgotten). Finally, both temperatures of 1.5 and 1.7 were tried (a high temperature is clearly desirable since this agent relies on generating a lot of diverse candidates).

Tuning of baselines. For large/costly models, the temperature was set to 1 for o3 (no other value is supported) and to 0.7 for 4o (a standard value used in related work to balance reasoning ability and diversity [52, 94]). Values of 0, 1, and 3 were tried for the `max_feedback_cycles` parameter (which is particularly sensitive for large models given a relatively small budget). For the *small* model 4o-mini, the `max_feedback_cycles` parameter was set to 3 (it is much less

Model	Input	Cached Input	Output
gpt-4o	$2.50	$1.25	$10.00
gpt-4o-mini	$0.15	$0.075	$0.60
o3	$2.00	$0.50	$8.00

Table C.1: OpenAI API Pricing (dollars per 1M tokens). Reported costs in our invariant synthesis experiment are computed based on these prices (Table 5.1).

important since a very high number of attempts can be performed from scratch in any case), but temperature values of 0.7, 1, and 1.5 were evaluated.

C.1.2 Complete System Prompts

System prompt used for the abduction-based strategy (`SuggestInvariants`, Figure 5.5)

Your goal is to prove the correctness of WhyML programs using the Why3 theorem prover. To do so, you must annotate programs with loop invariants in such a way that all assertions in the program can be proved automatically via the weakest-precondition calculus.

We only allow invariants that feature logical combinations of **linear** arithmetic expressions. In particular, we only accept multiplication by constant numerical literals. Thus, we accept invariants such as `x + y > 0` or `a > 0 || 2*x + y < n` but not `x * y > 0` or `x >= y * (y + 1) / 2`. I insist, and this is **very important**, you must **not** propose invariants involving complex arithmetic such as `2 * x >= y * (y + 1)`.

I will show you an annotated program in which Why3 did not manage to prove a particular assertion or invariant. Your task is to suggest a list of new invariant candidates that may unlock the proof. In order to help you diagnose the problem, I added comments to the WhyML program indicating what assertion or invariant failed to be proved (`GOAL`) and what parts of the program provide relevant assumptions (`premise`). The name of the failing proof obligation also provides a hint about the nature of the problem, which is either:

- `assertion`: the final assertion is not implied by the invariants
- `loop invariant init`: an invariant does not hold initially
- `loop invariant preservation`: an invariant cannot be proved to be preserved by the loop body (it may not be preserved or an additional invariant may be needed to complete the proof)

Please suggest new invariant candidates. Each invariant candidate must be obtained by following one of the *tricks* discussed below. A trick can be used multiple times. Do not suggest candidates that are already established invariants. Answer as a JSON object representing a list of `(trick_name, suggested_invariant)` pairs and **nothing else**. Examples are provided that include additional explanations for clarity. Do not include such explanations in your answer.

Notes on Why3 The `any T` construct generates an arbitrary object of type `T`. In particular, a loop whose guard is `any bool` can run for an arbitrary number of times.

Tricks Each trick is identified by a unique name. For each trick, we discuss *when* it is applicable and *what* the corresponding recipe is.

propose_post If the final assertion fails to prove but appears to hold through the whole program, propose it as an invariant.

monotone Whenever a variable `x` is only incremented (resp. decremented), propose invariant `x >= c` (resp. `x <= c`) for `c` some numerical constant.

linear Whenever a linear equality or inequality between variables appears to hold throughout the program (e.g. `x - y >= 0`, `3*x + 2*y = 1`...), consider proposing it as an invariant.

abduct_post If the final assertion fails to prove, look for a missing assumption that implies it when assuming all established invariants (along with the negation of the loop guard). Propose this assumption as a new invariant candidate.

abduct_inv If an invariant cannot be proved to be preserved, look for a missing assumption and propose it as a new invariant.

strengthen_inv If an invariant cannot be proved to be preserved, consider making it stronger (e.g. proposing `x < y` as a replacement for `x <> y`).

guard_inv If proving an invariant `inv` requires assuming a global assumption `assum` that is only made *after* the loop, consider proposing `assum -> inv` as an invariant instead.

true_or_continue If a property `P` always holds after the loop but cannot be proved as an invariant because `P` does not hold initially, consider proposing `P || loop_guard` as an invariant instead.

cond_guard If proving the preservation of an invariant candidate requires proving that a specific branch in the code cannot be taken (or is always taken), consider proposing an invariant that establishes this fact.

relax_loop_guard Suppose the loop guard is an inequality such as `expr < c` with `expr` an expression and `c` a constant. Then, if quantity `expr` cannot increase more than a constant amount `d` at each iteration, consider proposing `expr < c + d` as an invariant (and similarly for `<=`, `>` and `>=`).

System prompt used for the baseline strategy (`AnnotateWithInvs`, Figure 5.3)

The `AnnotateWithInvs` query uses the same system prompt as `SuggestInvariants` from the abduction-based strategy, except for the following differences:

- Instead of generating a list of invariant candidates labeled with trick names, the model is instructed to output a copy of the original program with added invariant annotations.
- The same tricks are presented for abducting invariant candidates from Why3 feedback, but they are not attributed names and following them is not presented as mandatory.

All the rest is identical: the short introduction about Why3, the instruction to only generate linear invariants, and the explanations for interpreting Why3's feedback. The full system prompt (along with all code for the case studies) is provided in the supplementary material.

C.2 Self-Improvement of a Lean Prover Agent

This appendix provides details on our Lean theorem-proving case study (Section 5.2). Appendix C.2.1 presents an example scenario where a proof is discovered with our proposed strategy, and Appendix C.2.2 shows examples of learned advice.

C.2.1 Discovering a Proof: A Concrete Scenario

We provide an example scenario of finding a Lean proof, which we reconstructed by browsing an actual search trace using the Delphyne VSCode extension. For proving the following theorem:

```
theorem algebra_sqineq_unitcircatbpabsamblt1
  (a b : Real)
  (h0 : a ^ 2 + b ^ 2 = 1) :
  a * b + abs (a - b) <= 1 := by sorry
```

Our strategy produces this sketch (after some search):

```
have eq_sq : (a - b) ^ 2 = (1 : Real) - 2 * (a * b) := by sorry
have abs_eq_sqrt : abs (a - b) = Real.sqrt ((1 : Real) - 2 * (a * b)) := by sorry
have ab_le_half : a * b <= (1 : Real) / 2 := by sorry
have rhs_nonneg : 0 <= (1 : Real) - a * b := by sorry
have square_ineq : (1 : Real) - 2 * (a * b) <= (1 - a * b) ^ 2 := by sorry
```

It is then tasked with producing proofs for all subgoals that do not close directly with `grind`, including the second one above. In doing so, it hallucinates a Mathlib theorem that does not exist (`Real.sqrt_eq_abs`). After receiving an error from Lean, it calls the `FindTheorem` tool:

```
description: "lemma saying |x| = sqrt (x^2) for real x"
```

This tool is implemented via a dedicated strategy, which itself uses Loogle to find good matches. After several unsuccessful requests yielding no relevant results (e.g., `Real.sqrt, abs, "abs_eq"`), it discovers a promising match using the query `"eq_abs", Real.sqrt, "sq"` and returns the corresponding documentation entry. The caller then manages to close the subgoal with:

```
rw [Real.sqrt_sq_eq_abs (a - b) |> Eq.symm, eq_sq]
```

C.2.2 Example of Learned Advice

This section presents examples of *learned advice*, automatically generated via search reflection on the MiniF2F validation set and subsequently incorporated into the prompt. See the supplementary material for a complete list.

Advice for proving subgoals

- **Control casts: work in one type, then come back by injectivity.** Avoid mixing $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{R}$ mid-proof. Cast early to a single target type and do all algebra there; when you need to return to $\mathbb{N}$, rewrite to a single cast (e.g., `←Nat.cast_add` / `←Nat.cast_mul`) and use `Nat.cast_injective` (or `Rat.cast_injective`) on the resulting equality. Normalize casts before ring-like tactics with `simp [Nat.cast_add, ...]`. For congruences, stay in $\mathbb{N}$ if your goal is `Nat.ModEq`; introducing $\mathbb{Z}$ gratuitously causes type mismatches without adding power.
- **Use $\leftrightarrow$ lemmas via `.mp` / `.mpr` (or `.1` / `.2`), not as functions.** Many important lemmas return an equivalence $A \leftrightarrow B$. To use them, first instantiate any parameters to obtain the $\leftrightarrow$, then pick a direction with `.mp` ($A \to B$) or `.mpr` ($B \to A$); `.1` and `.2` are equivalent shorthands. Don't try to "apply the lemma to a hypothesis" like a function. This pattern shows up everywhere (divisibility, order facts, `Real.sqrt`/log lemmas, set membership equivalences). It both fixes type errors and makes the intended direction explicit.

Advice for generating proof sketches

- **Avoid $\mathbb{N}$ subtraction: rewrite to an additive invariant and induct on that.** Natural number subtraction is partial and awkward in Lean. When a goal has terms like $a^{n+1} - (n+2)$, recast it as an additive equality (move the subtrahend to the other side) and prove the cleaner statement by induction. For instance, define `v n := u n + (n + c)` so the recurrence on `u` turns into a simple multiplicative or additive recurrence on `v`, which is easy to solve by `Nat.rec`. This both simplifies algebra (no case splits from `n - 1`) and exposes the right invariant: prove a closed form for `v`, then convert back to `u` by rearranging. The same idea works with helper sequences tailored to your recurrence so that the induction step is tautological.
- **Prepare prerequisites before using `sqrt` / `log` / `floor` / `ceil`, then use their characterization lemmas.** For square roots, first extract $0 \le x$ or $0 \le a$, then use theorems `Real.sqrt_eq_iff_sq_eq` and `sqr_sqrt` to move cleanly between $x^2 = a$ and $x = $ `a`. When bounding roots, use `Real.lt_sqrt` and `Real.sqrt_lt` to reduce inequalities to squared ones, supplying the required nonnegativity hypotheses. For logs, isolate side conditions in advance (`0 < base`, `base != 1`, `0 < argument`). Use `Real.log_rpow` to take logs of powers, rewrite the equality into a cancellable form, and cancel with `mul_right_cancel0` using `log != 0` (from `Real.log_pos_iff`). For floor/ceil, sandwich the number between consecutive integers and finish with `Int.floor_eq_iff` or `Int.ceil_eq_iff`. When square roots are involved, first obtain the bounds using the sqrt inequalities above, then apply the floor/ceil characterization.

Bibliography

[1] LangChain AI. LangGraph. https://github.com/langchain-ai/langgraph, 2023. 6.3

[2] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. *Advances in neural information processing systems*, 30, 2017. 1.2, 5.2

[3] Eser Aygün, Zafarali Ahmed, Ankit Anand, Vlad Firoiu, Xavier Glorot, Laurent Orseau, Doina Precup, and Shibl Mourad. Learning to prove from synthetic theorems. *CoRR*, abs/2006.11259, 2020. 2.4

[4] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, and Christian Szegedy. Learning to reason in large theories without imitation. *CoRR*, abs/1905.10501, 2019. 2, 2.4

[5] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. HOList: an environment for machine learning of higher order logic theorem proving. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 454–463. PMLR, 2019. 2

[6] Bernhard Beckert, Jonas Klamroth, Wolfram Pfeifer, Patrick Röper, and Samuel Teuber. Towards combining the cognitive abilities of large language models with the rigor of deductive program verification. In *International Symposium on Leveraging Applications of Formal Methods*, pages 242–257. Springer, 2024. 5.1

[7] 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. 6.3

[8] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. Pyro: Deep universal probabilistic programming. *Journal of machine learning research*, 20 (28):1–6, 2019. 6.3

[9] Pierre Boutillier, Mutaamba Maasha, Xing Li, Héctor F Medina-Abarca, Jean Krivine, Jérôme Feret, Ioana Cristescu, Angus G Forbes, and Walter Fontana. The Kappa platform for rule-based modeling. *Bioinformatics*, 34(13):i583–i592, 2018. 1.5

[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*,

33:1877–1901, 2020. 1.2, 1.4

[11] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo Tree Search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012. 2.1.2, 2.3, 3.1.1, 3.2.2, 5.1.1

[12] Rudy Bunel, Matthew J. Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. 2.1.3

[13] William E Byrd. *Relational programming in miniKanren: techniques, applications, and implementations*. PhD thesis, Indiana University, 2009. 6.3

[14] Jaime Carbonell and Jade Goldstein. The use of MMR, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 335–336, 1998. 5.2.1, 5.2.4

[15] Comanici et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv:2507.06261*, 2025. 6.1

[16] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978. 2

[17] Maxwell Crouse, Ibrahim Abdelaziz, Bassem Makni, Spencer Whitehead, Cristina Cornelio, Pavan Kapanipathi, Kavitha Srinivas, Veronika Thost, Michael Witbrock, and Achille Fokoue. A deep reinforcement learning approach to first-order logic theorem proving. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 6279–6287. AAAI Press, 2021. 2.4

[18] Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th acm sigplan conference on programming language design and implementation*, pages 221–236, 2019. 6.3

[19] Ivo Danihelka, Arthur Guez, Julian Schrittwieser, and David Silver. Policy improvement by planning with Gumbel. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. 2.1.2, 2.2, 2.3.1

[20] Vincent Danos, Jérôme Feret, Walter Fontana, Russell Harmer, and Jean Krivine. Rule-based modelling of cellular signalling. In *International conference on concurrency theory*, pages 17–41. Springer, 2007. 1.5

[21] George B. Dantzig and B. Curtis Eaves. Fourier-Motzkin elimination and its dual. *J. Comb. Theory, Ser. A*, 14(3):288–297, 1973. doi: 10.1016/0097-3165(73)90004-6. A.1

[22] Edoardo Debenedetti et al. Defeating prompt injections by design. *arXiv preprint arXiv:2503.18813*, 2025. 6.1

[23] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM (JACM)*, 32(3):505–536, 1985. 3.2.2

[24] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. Inductive invariant generation via abductive inference. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 443–456. ACM, 2013. doi: 10.1145/2509136.2509511. 2.1.1

[25] Mnacho Echenim, Nicolas Peltier, and Yanis Sellami. Ilinva: using abduction to generate loop invariants. In Andreas Herzig and Andrei Popescu, editors, *Frontiers of Combining Systems - 12th International Symposium, FroCoS 2019, London, UK, September 4-6, 2019, Proceedings*, volume 11715 of *LNCS*, pages 77–93. Springer, 2019. doi: 10.1007/978-3-030-29007-8_5. 2.1.1

[26] Thomas Eiter and Georg Gottlob. The complexity of logic-based abduction. *J. ACM*, 42(1): 3–42, 1995. doi: 10.1145/200836.200838. 2.1.1

[27] Ronen Eldan and Yuanzhi Li. TinyStories: How small can language models be and still speak coherent english? *arXiv preprint arXiv:2305.07759*, 2023. 1.4, 2.5

[28] Yao Feng, Jun Zhu, André Platzer, and Jonathan Laurent. Adaptive shielding via parametric safety proofs. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA1):816–843, 2025. 1.5

[29] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019. A.3

[30] Jean-Christophe Filliâtre. One logic to use them all. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *LNCS*, pages 1–20. Springer, 2013. doi: 10.1007/978-3-642-38574-2_1. 2

[31] Jean-Christophe Filliâtre and Andrei Paskevich. Why3—where programs meet provers. In *European symposium on programming*, pages 125–128. Springer, 2013. 5.1

[32] Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1229–1241, 2023. 1

[33] Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. Purely functional lazy nondeterministic programming. *Journal of Functional programming*, 21(4-5):413–465, 2011. 1.2, 3.1, 6.3

[34] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2023. 1.2, 5.2

[35] Saibo Geng, Hudson Cooper, Michał Moskal, Samuel Jenkins, Julian Berman, Nathan

Ranchin, Robert West, Eric Horvitz, and Harsha Nori. Generating structured outputs from language models: Benchmark and studies. *arXiv e-prints*, pages arXiv–2501, 2025. 4.1.1

[36] Christopher Hahn, Frederik Schmitt, Jens U. Kreber, Markus Norman Rabe, and Bernd Finkbeiner. Teaching temporal logics to neural networks. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. A.3

[37] Reiner Hähnle and Peter H. Schmitt. The liberalized δ-rule in free variable semantic tableaux. *J. Autom. Reasoning*, 13(2):211–221, 1994. 2

[38] Joseph Y Halpern and Spencer Peters. Reasoning about causal models with infinitely many variables. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 5668–5675, 2022. 1.5

[39] Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W. Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. 2

[40] David Harel. *First-Order Dynamic Logic*, volume 68 of *LNCS*. Springer, 1979. ISBN 3-540-09237-4. doi: 10.1007/3-540-09237-4. 2

[41] Jules Hedges. Monad transformers for backtracking search. In Paul Blain Levy and Neel Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*, volume 153 of *EPTCS*, pages 31–50, 2014. doi: 10.4204/EPTCS.153.3. 2.2

[42] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. A.3

[43] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10): 576–580, 1969. doi: 10.1145/363235.363259. 2.2

[44] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. GamePad: A learning environment for theorem proving. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. 2

[45] IBM Research. Mellea. `https://docs.mellea.ai`, 2024. A programming framework for reliable generative programs. 6.3

[46] Albert Q Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. *arXiv preprint arXiv:2210.12283*, 2022. 1, 5.2.1

[47] Aditi Kabra, Jonathan Laurent, Stefan Mitsch, and André Platzer. CESAR: control envelope synthesis via angelic refinements. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 144–164. Springer, 2024. 1.5

[48] Aditi Kabra, Jonathan Laurent, Sagar Bharadwaj, Ruben Martins, Stefan Mitsch, and André

Platzer. Can large language models autoformalize kinematics? In *Proceedings of the 25th Conference on Formal Methods in Computer-Aided Design, FMCAD 2025, Menlo Park, CA, USA, October 6-10, 2025*, 2025. 1.5

[49] Aditi Kabra, Jonathan Laurent, Ruben Martins, Stefan Mitsch, and André Platzer. LLM-powered automatic theorem proving and synthesis for hybrid systems and game. *arXiv preprint arXiv:2603.00737*, 2026. 1.5

[50] Aditi Kabra, Jonathan Laurent, Stefan Mitsch, and André Platzer. Hybrid game control envelope synthesis. *Proceedings of the ACM on Programming Languages*, 10(OOPSLA1): 850–876, 2026. 1.5

[51] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olsák. Reinforcement learning of theorem proving. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 8836–8847, 2018. 2.4

[52] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. Finding inductive loop invariants using large language models. *arXiv preprint arXiv:2311.07948*, 2023. 5.1, C.1.1

[53] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into self-improving pipelines. 2024. 6.3

[54] Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. *ACM SIGPLAN Notices*, 50(12):94–105, 2015. 6.3

[55] James Koppel, Gabriel Scherer, and Armando Solar-Lezama. Capturing the future by replaying the past (functional pearl). *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–29, 2018. 4.1.1

[56] Guillaume Lample and François Charton. Deep learning for symbolic mathematics. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. 2.1.3, 2.4

[57] Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. *Advances in neural information processing systems*, 35:26337–26349, 2022. 3.1.6, 3.1.6

[58] Jonathan Laurent. Alphazero.jl: A generic, simple and fast AlphaZero implementation. `https://github.com/jonathan-laurent/AlphaZero.jl`, 2021. 1.5

[59] Jonathan Laurent and André Platzer. Learning to find proofs and theorems by learning to refine search strategies: the case of loop invariant synthesis. *Advances in Neural Information Processing Systems*, 35:4843–4856, 2022. 1.4, 1.5, 2

[60] Jonathan Laurent, Hector F Medina-Abarca, Pierre Boutillier, Jean Yang, and Walter Fontana. A trace query language for rule-based models. In *International Conference on Computational Methods in Systems Biology*, pages 220–237. Springer, 2018. 1.5

[61] Jonathan Laurent, Jean Yang, and Walter Fontana. Counterfactual resimulation for causal analysis of rule-based models. In *IJCAI*, pages 1882–1890, 2018. 1.5

[62] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems with language models. *Advances in Neural Information Processing Systems*, 35:3843–3857, 2022. 1

[63] Wenda Li, Lei Yu, Yuhuai Wu, and Lawrence C. Paulson. IsarStep: a benchmark for high-level mathematical reasoning. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. 2

[64] 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. 1

[65] Junqi Liu et al. Numina-Lean-Agent: An open and general agentic reasoning system for formal mathematics. *arXiv preprint arXiv:2601.14027*, 2026. 6.1

[66] Xiaoxia Liu, Jingyi Wang, Xiaohan Yuan, Jun Sun, Guoliang Dong, Peng Di, Wenhai Wang, and Dongxia Wang. Prompting frameworks for large language models: A survey. *ACM Comput. Surv.*, 2026. ISSN 0360-0300. doi: 10.1145/3789253. 6.3

[67] Hector F Medina Abarca. *A combinatorial view of canonical Wnt signaling*. PhD thesis, Harvard University, 2023. URL https://dash.harvard.edu/entities/publication/b7c44683-e981-4438-8063-a23e0d77daa5. 1.5

[68] Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. Sympy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103, 2017. 1.1

[69] Michael Meyer. Abduction – a logical view for investigating and initiating processes of discovering mathematical coherences. *Educational Studies in Mathematics*, 74(2):185–205, 2010. 2.1.1

[70] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 561–577. USENIX Association, 2018. 2.2

[71] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019. 2.2, 6.3

[72] Spencer Peters and Joseph Halpern. A unifying framework for causal modeling with

infinitely many variables. *Journal of Artificial Intelligence Research*, 83, 2025. 1.5

[73] Muhammad AA Pirzada, Giles Reger, Ahmed Bhayat, and Lucas C Cordeiro. LLM-generated invariants for bounded model checking without loop unrolling. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1395–1407, 2024. 5.1

[74] André Platzer. Differential dynamic logic for hybrid systems. *Journal of Automated Reasoning*, 41(2):143–189, 2008. 1.5

[75] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *CoRR*, abs/2009.03393, 2020. 2.4

[76] George Polya. *How to solve it: A new aspect of mathematical method*, volume 85. Princeton university press, 2004. 2.1.1

[77] Yury Puzis, Yi Gao, and Geoff Sutcliffe. Automated generation of interesting theorems. In Geoff Sutcliffe and Randy Goebel, editors, *Proceedings of the Nineteenth International Florida Artificial Intelligence Research Society Conference, Melbourne Beach, Florida, USA, May 11-13, 2006*, pages 49–54. AAAI Press, 2006. 2.4

[78] David W Renshaw, Sarah M Loos, and André Platzer. Distributed theorem proving for distributed hybrid systems. In *International Conference on Formal Engineering Methods*, pages 356–371. Springer, 2011. 3.1.6

[79] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. CLN2INV: learning loop invariants with continuous logic networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. 2.3

[80] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. A systematic survey of prompt engineering in large language models: Techniques and applications. *arXiv preprint arXiv:2402.07927*, 2024. 1

[81] Daniel Selsam. The IMO grand challenge. `http://aitp-conference.org/2020/slides/DS.pdf`, 2020. Talk at AITP 2020. 2.4

[82] Daniel Selsam. Beyond the tactic-state automaton. *Mathematical Reasoning in General Artificial Intelligence Workshop, ICLR*, 2021. 2, 2.1.1, 2.2, 2.4, i

[83] Daniel Selsam, Jesse Michael Han, Leonardo de Moura, and Patrice Godefroid. Universal policies for software-defined MDPs. *arXiv preprint arXiv:2012.11401*, 2020. 2.3.1, 2.4, 4.3.3, 6.3, A.3

[84] Vighnesh Leonardo Shiv and Chris Quirk. Novel positional encodings to enable tree-based transformers. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 12058–12068, 2019. A.3

[85] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In *Advances in Neural Information Processing Systems*, pages 7762–7773, 2018. 2.3, 2.4, 5.1, 5.1.1

[86] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. Code2Inv: A deep learning framework for program verification. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *LNCS*, pages 151–164. Springer, 2020. doi: 10.1007/978-3-030-53291-8_9. 2.3, 2.4

[87] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018. 1.4, 2, 2.1.2, 2.1.2, 2.2

[88] Wouter Swierstra. Data types à la carte. *Journal of functional programming*, 18(4):423–436, 2008. 3.6, 3.1.7

[89] Jan-Willem Van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An introduction to probabilistic programming. *arXiv preprint arXiv:1809.10756*, 2018. 6.3

[90] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017. A.3

[91] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, 1989. 2

[92] Mingzhe Wang and Jia Deng. Learning to prove theorems by learning to generate theorems. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. 2, 2.4

[93] Anjiang Wei, Tarun Suresh, Tianran Sun, Haoze Wu, Ke Wang, and Alex Aiken. InvBench: Can LLMs accelerate program verification with invariant synthesis? *arXiv preprint arXiv:2509.21629*, 2025. 5.1

[94] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. Enchanting program specification synthesis by large language models using static analysis and program verification. In *International Conference on Computer Aided Verification*, pages 302–328. Springer, 2024. 5.1, C.1.1

[95] Daniel Whalen. Holophrasm: a neural automated theorem prover for higher-order logic. *arXiv preprint arXiv:1608.02644*, 2016. 3.1.6

[96] David J. Wu. Accelerating self-play learning in Go. *CoRR*, abs/1902.10565, 2019. 2.1.4

[97] Guangyuan Wu, Weining Cao, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. LLM meets bounded model checking: Neuro-symbolic loop invariant inference. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 406–417, 2024. 5.1

[98] Haoze Wu, Clark Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification. *arXiv preprint arXiv:2310.04870*, 2023. 5.1

[99] Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. TacticZero: learning to prove theorems from scratch with deep reinforcement learning. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 9330–9342, 2021. 2, 2.4

[100] Yuhuai Wu, Albert Q. Jiang, Jimmy Ba, and Roger Baker Grosse. INT: an inequality benchmark for evaluating generalization in theorem proving. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. 2.4

[101] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in Coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, 2019. 6.3

[102] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 6984–6994. PMLR, 2019. 2

[103] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022. 1, 1.1, 1.4, 4.3, 4.3.2, 5.1, 6.1

[104] 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. 1

[105] Alex L Zhang, Tim Kraska, and Omar Khattab. Recursive language models. *arXiv preprint arXiv:2512.24601*, 2025. 6.1

[106] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. MiniF2F: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*, 2021. 5.2.1

[107] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. SGLang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024. 6.3