Codeplay: Autotelic Learning through Collaborative Self-Play in Programming Environments

Laetitia Teodorescu, Cédric Colas, Matthew Bowers, Thomas Carta, Pierre-Yves Oudeyer

Codeplay: Autotelic Learning through Collaborative Self-Play in Programming Environments

Laetitia Teodorescu[*]
Inria

Cédric Colas
MIT, Inria

Matthew Bowers
MIT

Thomas Carta
Inria

Pierre-Yves Oudeyer
Inria

Abstract

Autotelic learning is the training setup where agents learn by setting their own goals and trying to achieve them. However, creatively generating freeform goals is challenging for autotelic agents. We present Codeplay, an algorithm casting autotelic learning as a game between a Setter agent and a Solver agent, where the Setter generates programming puzzles of appropriate difficulty and novelty for the solver and the Solver learns to achieve them. Early experiments with the Setter demonstrates one can effectively control the tradeoff between difficulty of a puzzle and its novelty by tuning the reward of the Setter, a code language model finetuned with deep reinforcement learning.

1 Introduction

Large Language Models (LLMs) have achieved substantial general reasoning abilities emerging from unsupervised pretraining on massive corpora of text scraped from the Internet [Radford et al., 2019, Brown et al., 2020]. While their performance is in part driven by sheer scale [Kaplan et al., 2020, Hoffmann et al., 2022, OpenAI, 2023], these models are fundamentally limited by the data they have access to: crafting datasets of high-quality text has become a difficult art, and the performance of these models is limited in domains that they have not been trained on, or where human data is lacking. They cannot master skills beyond their training set, they cannot explore and discover, create their own concepts, invent and master novel skills.

To allow agents to explore their environments and master an open-ended skill repertoire, the framework of autotelic learning has been proposed [Colas et al., 2022]. An autotelic agent samples its own goals and is intrinsically rewarded for achieving them. The goals are ideally sampled according to classic measures of intrinsic motivation: novelty, intermediate competence, or learning progress. As training unfolds the agent masters an ever-expanding set of skills, allowing it to cover an increasing subspace of the environment. However, previous instantiations of autotelic learning have been limited by the simplicity of the environment and the lack of flexibility, expressiveness and adaptation of the goal sampler [Colas et al., 2020, Teodorescu et al., 2023, Colas et al., 2023]. An interesting recent exception is the Voyager agent of [Wang et al., 2023] who use LLMs to build an embodied autotelic agent; however the model never gets finetuned on its own experience. Voyager impressively exploits and organizes the LLM's original Minecraft knowledge but does not go beyond.

In this work we propose an approach that is both a way to implement autotelic agents discovering a truly open-ended set of goals, and a way to allow language models to master novel coding skills in interaction with an interpreter. We ground LM-agents in a coding environment that provides straightforward binary rewards through code execution. We set ourselves in the Python Programming Puzzles domain [Schuster et al., 2021],

[*]Correspondance to: laetitia.teodorescu@inria.fr

Preprint. Under review.

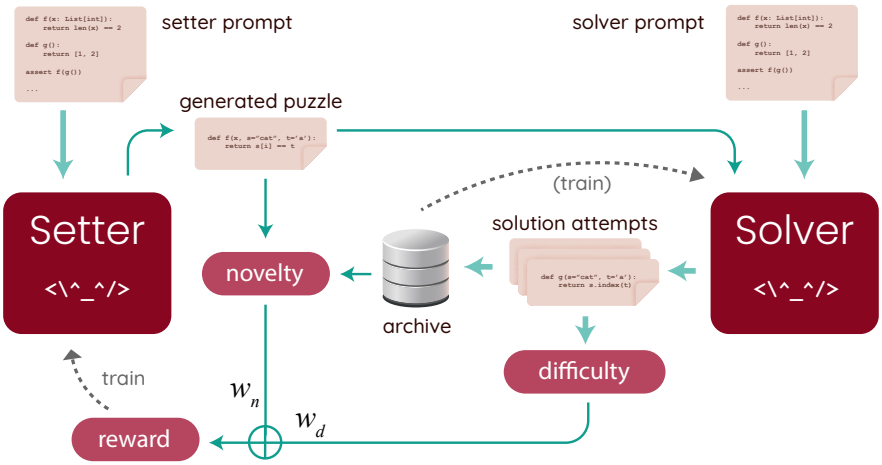

Figure 1: Overview of the proposed Codeplay algorithm. The Setter, a language model, takes in a few-shot prompt and emits a puzzle. This puzzle is given to the problem Solver who appends it to its few-shot prompt, and generates N_a candidate solutions. These problem-solution pairs are given to the Python interpreter, this gives us success or failure information on the solution. The number of successful attempts allows us to compute a difficulty reward. The generated puzzle is also compared with all previously generated puzzles to compute its novelty. Those two rewards are weighted and summed and used as the reward in a deep RL algorithm to train the Setter.

and we define an autotelic agent composed of 2 sub-agents: a **Setter** that generates puzzles, and a **Solver** whose objective is to solve the generated puzzles. Both agents are LM policies. They play a collaborative game where the Setter has to create puzzles that push the solver to learn, and the solver sees and tries to solve puzzles in its zone of proximal development (ZPD): hard but still solvable. First experiments presented here (with a fixed Solver) highlight the possibility to tradeoff difficulty of the generated puzzles and their novelty by tuning the reward experienced by the Setter, trained with deep RL (PPO: Schulman et al. [2017]).

Other related work Closely related to this work are approaches to autotelic learning or automatic curriculum learning involving goal setter agents [Florensa et al., 2018, Sukhbaatar et al., 2017, Campero et al., 2020, OpenAI et al., 2021, Mu et al., 2022], as well as the PowerPlay framework of Schmidhuber [2013]. Very closely related to this work as well are approaches for generating novel code puzzles for augmenting the capabilities of code-puzzle-solving language models [Haluptzok et al., 2022], as well as recent attempts to cast program synthesis as a reinforcement learning problem [Le et al., 2022, Yang et al., 2023].

2 Methods

Python programming puzzles We use as our testbed the Python Programming Puzzle (P3) domain of Schuster et al. [2021]. Each puzzle is defined by a test program `f` designed to verify the validity of solution programs `g` such that valid solutions satisfy `f(g()) is True` when run in an interpreter, see an example for the classical Towers of Hanoi in Appendix A. P3 puzzles span problems of various difficulties, from trivial to open questions.

Setter We instantiate our Setter as a pretrained language model, finetuned on the P3 domain. We cast the puzzle-generating problem as an MDP where the observation space is the list of all possible sequences of tokens, the action space is the list of all tokens and the reward is the intrinsic motivation measure we compute based on the difficulty of a puzzle. Transitioning from one (fully-observable) state to another is simply appending the emitted token to the observation: the environment is purely deterministic. This training setup is reminiscent of the one in reinforcement learning from human feedback (RLHF: Ouyang et al. [2022]), except in our case we do not use a reward function trained from human preferences but an intrinsic motivation metric based on the Solver's abilities. Our Setter agent's stochastic policy is given by the pretrained language model's logits which are used to sample the tokens (temperature of 0.8 in our

experiments). We use PPO [Schulman et al., 2017] as our training deep RL algorithm with an implementation based on the RL4LMs library [Ramamurthy et al., 2022]. As the value head in PPO we use a separate untrained MLP head on top of the language model backbone. In our preliminary experiments we keep the Solver fixed and investigate different rewards for the Setter. We use the small GPT-Neo-125M[2] pretrained on the Codex-generated dataset of Haluptzok et al. [2022].

Setter difficulty reward We want to reward the Setter for producing puzzles that are hard, while not being empirically unsolvable. To do so, we compute a reward based on the number of solutions generated by our fixed Solver within a maximum number of attempts `num_attempts` and use this signal to compute a difficulty score $\mathcal{D}(p)$ for a given puzzle p. We say a puzzle is invalid when the Python parser rejects it; we say a puzzle is (empirically) unsolvable when the Solver did not manage to find one valid solution. The difficulty score is defined for valid, solvable puzzles as the inverse of the probability for the Solver to generate a solution in one attempt: the difficulty of a puzzle that is always solved on each attempt is 1, and it tends to infinity as the difficulty increases; the difficulty is infinite for unsolvable puzzles. We define the *difficulty reward* $R_d(\mathcal{D}(p))$ of a puzzle as a quantity between -1 and 1, that is zero for trivial puzzles ($\mathcal{D} = 1$), -1 for invalid and unsolvable puzzles, and interpolates smoothly (cosine) between $\mathcal{D} = 1$ and $\mathcal{D} = 3$.

Setter novelty reward Because the Solver is fixed, optimizing only R_d will lead to Setter collapse on a single puzzle (or a small set). To encourage diversity in the generated puzzles we define the *novelty reward* $R_n(p, \mathcal{A})$ of a puzzle p as the average distance between p and its 5 closest neighbors in the archive of previously generated puzzles $\mathcal{A}$ in an embedding space. We normalize this distance by the average pairwise distance of puzzles in the P3 train set in that same space (so R_n is roughly between 0 and 1).

The total reward for the Setter is a weighted sum of the difficulty and novelty rewards:

$$R(p) = w_d \, R_d(\mathcal{D}(p)) + w_n \, R_n(p, \mathcal{A}) \tag{1}$$

In the following experimental results (Section 3) we investigate the impact of different values of the weights.

Solver We do not train the Solver in our experiments, but in principle there are two options: behavioral cloning (BC) from positive puzzle solution pairs, and RL from puzzle-solving signal. In preliminary experiments we have found RL training with PPO to be very unstable (presumably because samples in the PPO buffer are very correlated, since we repeatedly solve the same puzzles). Furthermore PPO is an on-policy algorithm, meaning that we cannot train on old samples and thus we cannot replay old puzzles to mitigate forgetting. This suggests that BC might be a better option. Training language models with off-policy Deep RL methods is still an open field of study. In our preliminary experiments we only present the fixed-solver setup to demonstrate the effect of RL training on the Setter.

Prompts We use a fixed few-shot prompt for both the solver and the setter. The few-shot prompt simply includes a series of puzzles and solutions from the tutorial set of the P3 training set (which are pretty short, so we can fit all of them). The puzzles and solutions are separated by `assert(f(g()))` statements. The prompt for the Setter finishes after an assertion, the prompt for the Solver (evaluated N_a times to produce the difficulty estimation) additionally includes the puzzle to be solved at the end. Part of the prompt can be found in Appendix B.

3 First experimental results

We present here a series of results studying Setter training (training details can be found in Appendix D). We report the learning curves for both reward components R_d and R_n in Figure 2, as well as the total Setter reward, the quantity effectively being optimized. It is the weighed sum of both components (see Equation 1).

The gray curve provides or baseline: no optimization whatsoever is taking place. On the left-hand side, we see that the puzzles generated in this case are close to trivial (R_d close to 0) and that they are relatively repetitive: the gray novelty reward tends to zero. This is because the novelty reward is non-stationary: the more puzzles you generate in the same distribution, the denser the puzzles become in the embedding space used and so the smaller the distance between one puzzle and the next become. The mauve curve shows what happens when only the difficulty reward gets optimized ($w_d = 1, w_n = 0$): the latter goes up pretty quickly

[2] https://github.com/EleutherAI/gpt-neo

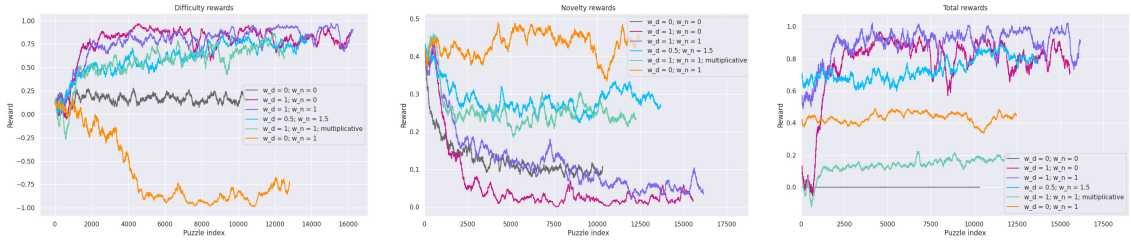

Figure 2: Competence and novelty rewards for different experiments. The values are smoothed over 200 puzzles. The colors correspond to different pairs of weights for Equation 1. See main text for analysis.

but the novelty reward goes to zero fast, indicating highly repetitive generated puzzles (and in effect that is what happens, see examples in Appendix C). The orange curve is what happens when only the novelty reward gets optimized ($w_d = 0, w_n = 1$): on the center figure we see that the novelty stays high throughout, which is impressive given that new puzzles are generated continuously, but on the left the difficulty reward goes to -1 quickly. This indicates either generation of unsolvable or of invalid puzzles, and manual inspection reveals the latter to be true. Optimizing only for novelty yields a Setter unable to generate valid programming puzzles. A reward with equal weights ($w_d = 1, w_n = 1$, purple curve) wields puzzle with appropriate difficulty and as repetitive as not optimizing at all (as far as the novelty reward is concerned). Biasing the reward towards novelty ($w_d = 0.5, w_n = 1.5$, blue curve) yields puzzles that maintain a fixed amount of novelty while still eventually converging towards difficult but solvable puzzles. The same effect can be achieved by multiplying the two reward components with a weight of 1 instead of summing them. From these curves we see that there is a tradeoff in the novelty and appropriate difficulty of puzzles for a fixed Solver, and that we are successfully able to navigate this tradeoff by tuning different reward components. We provide 10 puzzles generated by each variant at the end of training in Appendix C.

4 Further work and discussion

In this preliminary work, we have cast the autotelic learning problem in the space of Python Programming Puzzles as a collaborative game between two agents: a Solver learning to solve python puzzles and a Setter learning to output puzzles that are intermediately difficult for the Solver. In first experiments examining the behavior of the Setter, we have seen in these early experiments that we are able to train language models with deep RL to optimize classical measures of intrinsic motivation such as intermediate difficulty or novelty of a sample. This allows us to combine the paradigm of autotelic agents trained with deep RL with the flexibility, generality, and massively multitask nature of pretrained language models.

Of course, these results need to be extended by training the Solver as well, which we have actually found more difficult than training the Setter. The setting needs to be scaled, because the diversity and quality of the generated samples improve radically as the model capacity and training are scaled up. There are two final objectives for this work: the first being generating a sequence of puzzles that are increasingly difficult for the fixed, untrained Solver but that maintain a constant difficulty for the Solver as it is trained; the second objective would be to find a way to steer the process towards the generation of artifacts that are interesting to humans. If these two conditions are fulfilled we will have demonstrated the embryo of a truly open-ended linguistic agent, steerable towards problems that humans find interesting.

References

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.

Andres Campero, Roberta Raileanu, Heinrich Küttler, Joshua B Tenenbaum, Tim Rocktäschel, and Edward Grefenstette. Learning with amigo: Adversarially motivated intrinsic goals. *arXiv preprint arXiv:2006.12122*, 2020.

Cédric Colas, Tristan Karch, Nicolas Lair, Jean-Michel Dussoux, Clément Moulin-Frier, Peter Dominey, and Pierre-Yves Oudeyer. Language as a cognitive tool to imagine goals in curiosity driven exploration. *Advances in Neural Information Processing Systems*, 33:3761–3774, 2020.

Cédric Colas, Tristan Karch, Olivier Sigaud, and Pierre-Yves Oudeyer. Autotelic agents with intrinsically motivated goal-conditioned reinforcement learning: a short survey. *Journal of Artificial Intelligence Research*, 74:1159–1199, 2022.

Cédric Colas, Laetitia Teodorescu, Pierre-Yves Oudeyer, Xingdi Yuan, and Marc-Alexandre Côté. Augmenting autotelic agents with large language models. *arXiv preprint arXiv:2305.12487*, 2023.

Carlos Florensa, David Held, Xinyang Geng, and Pieter Abbeel. Automatic goal generation for reinforcement learning agents. In *International conference on machine learning*, pages 1515–1528. PMLR, 2018.

Patrick Haluptzok, Matthew Bowers, and Adam Tauman Kalai. Language models can teach themselves to program better. *arXiv preprint arXiv:2207.14502*, 2022.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

Jesse Mu, Victor Zhong, Roberta Raileanu, Minqi Jiang, Noah Goodman, Tim Rocktäschel, and Edward Grefenstette. Improving intrinsic exploration with language abstractions. *Advances in Neural Information Processing Systems*, 35:33947–33960, 2022.

OpenAI. Gpt-4 technical report. *ArXiv*, abs/2303.08774, 2023.

OpenAI OpenAI, Matthias Plappert, Raul Sampedro, Tao Xu, Ilge Akkaya, Vineet Kosaraju, Peter Welinder, Ruben D'Sa, Arthur Petron, Henrique P d O Pinto, et al. Asymmetric self-play for automatic goal discovery in robotic manipulation. *arXiv preprint arXiv:2101.04882*, 2021.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

Rajkumar Ramamurthy, Prithviraj Ammanabrolu, Kianté Brantley, Jack Hessel, Rafet Sifa, Christian Bauckhage, Hannaneh Hajishirzi, and Yejin Choi. Is reinforcement learning (not) for natural language processing?: Benchmarks, baselines, and building blocks for natural language policy optimization. 2022. URL https://arxiv.org/abs/2210.01241.

Jürgen Schmidhuber. Powerplay: Training an increasingly general problem solver by continually searching for the simplest still unsolvable problem. *Frontiers in psychology*, 4:313, 2013.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Tal Schuster, Ashwin Kalyan, Oleksandr Polozov, and Adam Tauman Kalai. Programming puzzles. *arXiv preprint arXiv:2106.05784*, 2021.

Sainbayar Sukhbaatar, Zeming Lin, Ilya Kostrikov, Gabriel Synnaeve, Arthur Szlam, and Rob Fergus. Intrinsic motivation and automatic curricula via asymmetric self-play. *arXiv preprint arXiv:1703.05407*, 2017.

Laetitia Teodorescu, Eric Yuan, Marc-Alexandre Côté, and Pierre-Yves Oudeyer. A song of ice and fire: Analyzing textual autotelic agents in scienceworld. *arXiv preprint arXiv:2302.05244*, 2023.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *arXiv preprint arXiv:2306.14898*, 2023.

A Towers of Hanoi in P3

To illustrate the P3 domain we provide here an implementation of the classic Towers of Hanoi problem and a possible solution below:

```
def f(moves: List[List[int]]):
    """
    Eight disks of sizes 1-8 are stacked on three towers, with each
        tower having disks in order of largest to
    smallest. Move [i, j] corresponds to taking the smallest disk off
         tower i and putting it on tower j, and it
    is legal as long as the towers remain in sorted order. Find a
        sequence of moves that moves all the disks
    from the first to last towers.
    """
    rods = ([8, 7, 6, 5, 4, 3, 2, 1], [], [])
    for [i, j] in moves:
        rods[j].append(rods[i].pop())
        assert rods[j][-1] == min(rods[j]), "larger␣disk␣on␣top␣of␣
            smaller␣disk"
    return rods[0] == rods[1] == []

def sol():
    # taken from https://www.geeksforgeeks.org/c-program-for-tower-of
        -hanoi/
    moves = []
    def hanoi(n, source, temp, dest):
        if n > 0:
            hanoi(n - 1, source, dest, temp)
            moves.append([source, dest])
            hanoi(n - 1, temp, source, dest)
    hanoi(8, 0, 1, 2)
    return moves

assert f(g()) is True
```

B Few-shot prompt

```
from typing import List

def f(s: str):
    return "Hello␣" + s == "Hello␣world"

def g():
    return "world"

assert f(g())

def f(s: str):
    return "Hello␣" + s[::-1] == "Hello␣world"

def g():
    return "world"[::-1]

assert f(g())

...
```

C Example generated puzzles for all versions of the reward

In this version we show some examples of generated puzzles for each weight combination we train in the main body of the paper. For all versions, we randomly sample without replacement 10 puzzles from the last 100 generated puzzles in the run of the algorithm.

C.1 No optimization: $w_d = 0; w_n = 0$

```
def f(s: str, a='hello', b='world'):
    return a in s and b in s and (len(s) == len(a + b))

def f(s: str, a='hello', b='world', c='hello world'):
    return a in s and b in s and (c in s)

def f(s: str, a=12345678):
    return int(s) == a

def f(li: List[int]):
    return len(li) == 3 and all((i in li for i in range(1, 4)))

def f(s: str, n=5):
    return len(s) == n and all((c in 'aeiou' for c in s))

def f(s: str, a=9, b=9):
    return a <= s.count('a') <= b and a <= s.count('b') <= b

def f(s: str, t='hello world'):
    return t in s

def f(s: str, n=10):
    return len(s) == n and all((c.isalpha() or c.isdigit() for c in s
        ))

def f(s: str, n=5):
    return len(s) == n and all((s[i] == s[i + 1] for i in range(n -
        1)))

def f(n: int, a=9, b=9, c=9):
    return n == a * b * c
```

C.2 Difficulty reward only: $w_d = 1; w_n = 0$

```
def f(x: int):
    return x - 10000 == 1

def f(x: int):
    return x - 1 == 0

def f(x: int):
    return x - 10000 == 1

def f(x: int):
    return x - 3450000 == 0

def f(x: int):
    return x - 100000000 == 0

def f(x: int):
```

```
    return x - 1 == 0

def f(x: int):
    return -x == 1

def f(x: int):
    return x - 1 == 1

def f(x: int):
    return -x == 1

def f(x: int):
    return x - 1 == 1
```

C.3 Novelty reward only: $w_d = 0; w_n = 1$

```
def f(x1: float):
    return 1 <= x1 < 2 ** 16

def f(x1stones: Listesthetic, xplomastones):
    return len(xplomastones) == xplomastones

def f(stones: List[void], n=1024):
    return 2 ** n - 1 == 2 ** n

def f(n: float, sr=1000000):
    return n * sr + float.imag == sr

def f(x1: float, x2='01', n=1000):
    return x1 * n + x2 == x * n * n

def f(n: float, square=1000000):
    return 2 * n + 2 == square

def f(x1):
    return 2 ** 100 - 2 ** chance < 2 ** chance

def f(x: float, r=10000000000000000000000000000):
    return 1 - x ** r == x

def f(coins: float=1 << 3 | 2 | 1):
    return coins * 2 <= 36 * 36

def f(n: float=1000000000.0):
    return n * n + float.imag == float.imag
```

C.4 Both rewards, balanced: $w_d = 1; w_n = 1$

```
def f(s: int):
    return s < 111

def f(s: int):
    return s < 111

def f(s: int):
    return s < 256
```

```
def f(s: int):
    return s < 65535

def f(s: int):
    return s < 256

def f(s: int):
    return s < 256

def f(s: int):
    return s < 111

def f(s: int):
    return s < 65535

def f(s: int):
    return s < 256

def f(s: int):
    return s < 256
```

C.5 Both rewards, unbalanced: $w_d = 0.5; w_n = 1.5$

```
def f(x: float, t=1, tt=2):
    return x > t * 2 * 2

def f(x: float, y=2, delta=2):
    return x > y * 3 - y

def f(x: float, y=2, delta=1):
    return x > 3 * y - delta

def f(n: float, m=2, m_nines=2):
    return n > 3 * m_nines

def f(x: float, n=2, delta=1):
    return x > n * n - n + delta

def f(x: float, n=2, lower_bound=1):
    return x > n * 2 - n

def f(n: float, m=2, m_nppos=2):
    return n > m and n > m_nppos

def f(x: float, lower_bound=500):
    return x > lower_bound * 2

def f(x: float, y=1, z=2):
    return abs(x) * z > 10 * y

def f(x: float, t=2, ttl=1):
    return t * x > t
```

D Training details

We train the Setter for 50 PPO iterations [Schulman et al., 2017]. Each PPO epoch is composed of a data-gathering phase on 64 parallel environments, each environment being stepped for 128 steps (token generations). Once one of the parallel Setter agents has finished generating a puzzle p (generated a function `f` and output a double newline), we store p as a list of tokens in the PPO buffer and restart the episode. Once all the steps have been taken we perform 5 PPOs epoch on the collected buffer with the Adam optimizer and a learning rate of `1e-6`. We then discard the buffer and start again. We use a discount factor of 1, and generalized advantage estimation [Schulman et al., 2015] with $\lambda = 1$.