Domain Agnostic Techniques for Scaling Up Program Synthesis

Théo Matricon

LaBRI

université de BORDEAUX

THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR DE L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE
DE MATHÉMATIQUES ET D'INFORMATIQUE

SPÉCIALITÉ : INFORMATIQUE

par **Théo MATRICON**

Techniques agnostiques au domaine pour l'accélération de la synthèse de programmes

---

Domain Agnostic Techniques for Scaling Up Program Synthesis

Sous la direction de :
Nathanaël **Fijalkow**

Soutenue le 10 décembre 2024

Devant la commission d'examen composée de :

M. Jean-Rémy Falleri	Professeur, Bordeaux INP	Président du jury
M. Sebastijan Dumancic	Associate Professor, TU Deft	Rapporteur
M. Colin De la Higuera	Professeur, Université de Nantes	Rapporteur
Mme Elizabeth Polgreen	Associate Professor, University of Edimburgh	Examinatrice
M. Cazenave Tristan	Professeur, Université Paris-Dauphine	Examinateur
M. Jansen Nils	Professor, Ruhr-University Bochum	Examinateur
M. Nathanaël Fijalkow	Chargé de Recherche, CNRS	Directeur

Contents

Résumé

Ces denières années, la synthèse automatique de programmes, un saint-graal de l'informatique, a vu un intérêt grandissant. Du développement de code à la découverte automatique, les applications sont nombreuses.

Dans cette thèse, nous nous intéressons à la synthèse de programmes dans le cadre de spécifications par des exemples. Nous nous concentrons sur des techniques indépendantes du domaine ce qui nous amène à de la recherche guidée par coûts. Les coûts permettent de guider l'énumération vers les programmes les plus probables comme solutions. Nous avons trois contributions principales. En premier, nous introduisons Heap Search le premier algorithme d'énumération par le bas par coût avec délai logarithmique. Puis nous cherchons à construire un algorithme avec délai constant, ce que nous faisons avec Eco Search, le premier, et nous montrons qu'il est aussi meilleur en pratique. Nous nous intéressons à la parallélisation de la recherche avec des algorithmes d'échantillonages et introduisons l'algorithme optimal parmi ceux-ci pour la recherche par coûts. Nous décrivons aussi un algorithme de division de grammaires en grammaires disjointes afin de facilement paralléliser la recherche. Les programmes redondants peuvent être éliminés par équivalence à l'exécution, néanmoins une majeur partie de ses équivalences sont dûes à la grammaire et non à la tâche. Nous proposons un processus qui trouve automatiquement ces programmes redondants en amont et les compile en un filtre pour les éliminer sans les exécuter. Enfin, nous nous intéressons à intégrer des connaissances en dehors de la grammaire dans les programmes. Nous décrivons plusieurs niveaux de difficulté pour ces tâches et résolvons le premier niveau.

Mots-clés : synthèse de programmes, recherche par coût, énumération

---

Abstract

In recent years, the automated synthesis of programs from specifications, a holy grail of computer science, has seen tremendous interest. From developing code, automating proofs, optimisations to automatic discovery, the applications are numerous.

In this thesis we focus on program synthesis in the programming by examples framework. We will focus on domain agnostic techniques which leads us to cost-guided search. We describe Heap Search the first best-first search algorithm with logarithmic delay. Then we investigate whether we can build a constant delay, or no-delay, best-first search algorithm. We answer positively with Eco Search the first such algorithm and prove it outperforms its competitors in practice. We also look into parallelising the search procedure with sampling algorithms and introduce the optimal best-first search algorithm among sampling algorithms. We also look into splitting the grammar into disjoint grammars in order to easily parallelise the search, and describe such an algorithm. Redundant programs can be eliminated with pruning, most of these redundancies are part of the grammar not the task. Therefore we develop a pre-processing technique enabling to find these redundant programs and compile them, once and for all, in a filter that enables to prune them at runtime. Finally we also look into knowledge powered programs: programs that talk about relational knowledge outside of the grammar. We show there are different difficulty levels for such tasks and describes a solution to solve the first level of difficulty.

Keywords: Program Synthesis, Best-first search, Enumeration

Laboratoire Bordelais de Recherche en Informatique
351, cours de la Libération F-33405 Talence cedex

Acknowledgements

When I started my PhD thesis, I was very much afraid I would not be able to work on the same subject seriously during three years. In hindsight, this was not a difficulty at all, instead I faced many other challenges I was not prepared for. It took me quite some time and the support of my colleagues, friends and people I met along the way to push through.

Thank you.

I would like to thank my PhD advisor, Nathanaël for your guidance, for your patience in dealing with me, for the insights you shared with me, and for believing in me more than I did without which I would have not finished this thesis. Thank you for showing me an academic path that is both human, ethic and of scientific excellence.

I want to thank Sebastijan Dumancic and Colin de la Higuera for agreeing to review this document. I would also want to thank Jean-Rémy Falleri, Elizabeth Polgreen, Tristan Cazenave and Nils Jansen for accepting to be part of the jury.

I would like to thank Laurent Simon for pushing me towards research, your passion for my ideas and for enabling me to work on program synthesis. I would like to thank the many outstanding people I had the honour to work with: Laurent Simon, Marie Anastacio, Holger Hoos, Guillaume Lagarde, Alexandre Blanche, Guruprerana Shabadi, Axel Martin, Djamel Eddine Khelladi, Mathieu Acher, Roman Kniazev, Pierre Vandenhove, Gabriel Bathie, Baptiste Mouillon and the many interns I worked with: Gaëtan Margueritte, Priscilla Tissot, Félix Yvonnet, Chaimaa Radiousse, Gianni Padula, Sylvain and many others. Thank you to all the people in LaBRI that foster an inclusive and fun atmosphere, this has been a pleasure. Specifically to the PhD students and the AFoDIB who have organised numerous activities making life much more fun. Thank you Guillaume Lagarde for listening me weight the pros and cons of an academic career for way too many hours, you always bring a smile to the table, it has been fun discussing with you.

Je voudrais aussi remercier ma famille pour leur soutien inconditionnel, plus particulièrement à mon petit frère, ton soutien me pousse à repousser mes limites.

I would like to thank all my friends for bearing with me, I am not very good at communicating my feelings but I can say that you make me feel completely at home, so thank you. The order of thanks does not matter at all, it is very random. I would like to thank my office mates, whether I stayed in your office only temporarily due to cold or more generally. Thanks Clara and Gabriel for your unyielding support. Merci Gabriel de m'avoir montré la voie vers le meilleur ~~culte~~ langage de programmation. Merci Clara d'avoir ouvert mes horizons et pour ta patience avec les jeunes, tout le monde n'a pas trois millénaires. I would like to thank Elsa for your support and for randomly talking about your life, it led to very interesting discussions, and mainly all the fun we had talking about Luc. Merci Sarah pour toutes les discussions, tout le soutien que tu m'as apporté,

les parties de jeux, l'escalade et détective Conan! Merci aussi à Azur pour les calins. Merci Sara Riva, tu as été la première à me pousser à continuer dans la recherche et je n'aurais probablement pas poursuivi sans ton soutien. Grazie per aver creduto in me e ti prego di continuare a guidarmi nel mondo accademico. Thanks Rémi for bearing with me for three years, expanding my horizon, being involved in the decisions that matter in the laboratory. Thanks Sarah, Clément, Gabriel, Joséphine, Adrien, Benoit, Dylan, Emile, Mathieu for all the climbing sessions. Thanks Clément for all the advice and the climbing hype and discoveries. Merci Thibault pour les cookies, pour les discussions et ton aide du haut de ton HDR. Merci Corto de m'avoir évité de gâcher toutes ces pommes. Merci Emile de toujours croire en mes annonces au bluff et pour la vidéo de 3h sur Twilight. No te olvidé Antonio, nos dejaste demasiado pronto... Merci Pierre Vandenhove pour tous ces quiz face aux quizmestres et ses sprints de gare de l'est à Montparnasse. Merci Gaëtan pour ta motivation a toucher le code des autres et même un parseur. Merci à Julien M., Julien S., Dylan et d'autres d'avoir joué à des jeux débiles avec moi, merci pour ces parties, merci pour tous ces escapes. Merci Juliette de m'avoir lancé dans toute une série de questionnements et de m'avoir recommandé le podcast le coeur sur la table, c'est incroyable. Merci Françoise pour ta bonne humeur, ça aurait été sympa de passer plus de temps dans le même bureau. Thanks Arka for making me feel so productive. Thanks Roman for all the interesting discussions. Merci Edern pour les échanges. Merci Aline de nous avoir montré comment être une personne incroyablement humaine. Merci Waris, ça été un plaisir de te mentorer quand tu n'étais qu'un étudiant et de voir ce que tu accomplis. Merci Marc d'être venu faire un stage et d'avoir discuter avec nous sur ce que c'était que la recherche, j'espère que ta thèse te plait. Thanks Kacper for the climbing session, I hope I converted you to the best sport ever: climbing. Thanks Thimothé for the translations of the Polish songs during the karaoke, you truly are a poet. I would like to thank Luc for your support and all the fun we had talking about Elsa. I would like to thank Meghna for her cards in the card game Little Happy Dinosaurs that I stole twice with the help of a card in order to take the win. Merci Nacim pour l'idée de demander le consentement aux personnes avant de les inclure.

I believe I have forgotten a lot of people and ommitted a lot of details, but if you are reading this you probaly ought to have my thanks.

à mon petit-frère, Alex.

Résumé étendu en français

Nous présentons ici un brief résumé de ce manuscrit en langue française, pour une analyse plus approfondie nous invitons la personne lisant ce manuscrit à lire le contenu en langue anglaise qui contient l'intégralité du contenu de ce manuscrit.

1 Contexte

Le monde d'aujourd'hui se noie dans le logiciel de notre téléphone aux coeurs artificiels. La demande de logiciel augmente inlassablement dans des environements de plus en plus complexes. De plus développer des logiciels dans des environments très variables est notablement dur. Cela demande une vision à long-terme et de la pensée critique afin d'obtenir un logiciel robuste et sans bogues. Enfin, le logiciel n'est pas un objet statique tout au contraire c'est un objet en constant changement. C'est pourquoi la synthèse de programmes est un des saint-graal de l'informatique: générer automatiquement du code correct qui respecte la spécification fournie. C'est un vieux problème qui existe depuis les débuts de l'informatique [Chu57] et qui attire de plus en plus d'intérêt. Aujourd'hui la plupart des personnes ayant touché à un ordinateur ont utilisé des outils de synthèse de programmes à leur insu avec FlashFill [Gul11a] ou SmartFill [Goo21]: ces outils permettent d'aider les individus à faire des manipulations syntactiques de texte.

Plus formellement, la synthèse de programmes chercher à résoudre le problème suivant:

Définition ▶ Synthèse de programmes

La synthèse de programmes est le problème qui étant donné :

- une spécification φ
- un ensemble de programmes $\mathcal{P}$

de trouver $p \in \mathcal{P}$ tel que p satisfait φ *i.e.* $\forall x, \varphi(x, p(x))$

C'est un problème très générique, nous nous intéressons au cas où l'ensemble de programmes est défini par une grammaire non contextuelle. De même, les spécifications peuvent être très variables. Il y a trois formes principales de spécifications : par une formule logique, par du texte naturel ou par des exemples. Nous nous intéressons au dernier cas: on peut exécuter le programme pour vérifier s'il satisfait la spécification contrairement au langage naturel et il est relativement accessible et simple de spécifier son problème.

Exemple ▶ La synthèse à partir d'exemples

Spécification:

$$f([1,2,4]) = [1,4,16] \qquad f([9,3]) = [81,9]$$
$$f([]) = [] \qquad f([9,5,5,6]) = [81,25,25,30]$$

Une fonction f qui serait solution ici est une fonction qui prendrait chaque élément de la liste donnée en entrée et le mettrai au carré. Ce qui peut donner cela dans la syntaxe de Python:

```
def f(elements: list[int]) -> list[int]:
    return [x**2 for x in elements]
```

On appelle le langage décrit par la grammaire: le langage spécifique au domaine (LSD). De nombreuses approches à la synthèse de programmes emploient beaucoup de techniques spécifiques au domaine [Gul11b, SBS20a, CGL+23]. Pour tenter de faire avancer le domaine de manière général, nous nous intéresserons plus particulièrement aux techniques indépendantes du domaine pour la synthèse de programmes.

Nous nous concentrons sur un modèle de grammaire particulier : les grammaires hors-contextes.

Exemple ▶ Grammaire pour les formules booléennes à trois variables

Considérons la grammaire qui décrit les formules booléennes àavec ces trois variables : x_0, x_1, x_2. Voici les règles de production P:

$$S \to And(S,S) \qquad S \to Or(S,S)$$
$$S \to Not(S) \qquad \forall i \in [0,2],\ S \to x_i$$

Donc $G = (\{S\}, \{And, Or, Not, x_0, x_1, x_2\}, P, S)$.
Pour tester si par exemple $And(x_0, Or(x_2, Not(x_3)))$ appartient au langage décrit par notre grammaire, nous allons chercher une suite d'applications de règles de productions qui permet de produire le programme. Voici comment faire, nous avons souligné le non-terminal sur lequel était appliqué la règle à chaque étape :

$S \to And(\underline{S}, S)$	avec $S \to And(S,S)$
$S \to And(x_0, \underline{S})$	avec $S \to x_0$
$S \to And(x_0, Or(\underline{S}, S))$	avec $S \to Or(S,S)$
$S \to And(x_0, Or(x_1, \underline{S}))$	avec $S \to x_1$
$S \to And(x_0, Or(x_1, Not(\underline{S})))$	avec $S \to Not(S)$
$S \to And(x_0, Or(x_1, Not(x_2)))$	avec $S \to x_2$

Donc $And(x_0, Or(x_1, Not(x_2)))$ appartient au langage.

2 Contributions

Nous présentons les contributions principales de la thèse très brièvement, des hyperliens référencent les chapitres concernés. Pour une description détaillée de nos contributions, nous invitons à lire les chapitres concernés.

Code Ce manuscrit implémente toutes les méthodes décrites et montrent des résultats pratiques, il serait donc incomplet sans le code associé au cours de ces trois années. Une ancienne version du code est disponible à `https://github.com/nathanael-fijalkow/DeepSynth`, en effet le dépôt ne contient pas les résultats plus récents. Le dépôt à `https://github.com/Theomat/ProgSynth` contient la quasi totalité des travaux de synthèse fait au cours de cette thèse, à la date d'écriture de ce manuscrit, août 2024, le dernier commit est le `ef20aa31d6f6f8e137d36aa20ee6c0cc6c308dfe`.

Approche Générale En premier lieu, avec le chapitre 2, pour attaquer le problème avec des techniques indépendantes du domaine, nous nous dirigeons vers la recherche guidée par coût. La figure 2.2 illustre notre architecure, les exemples sont passés dans un modéle prédictif qui va donner des probabilités ou des coûts que l'on peut projeter sur la grammaire. Cela nous donne une grammaire avec des coûts, le but avec la recherche par coûts est que cela guide la recherche. Un programme de plus petit coût ayant *a priori* plus de chances d'être une solution au problème. Dans ce chapitre, nous expliquons les difficultés lié au fait que ce modèle prédictif, souvent un réseau de neurones doit être entrainé, et que peu de données sont disponibles pour les LSDs. Nous expliquons comment générer des données de manière synthétique de meilleure qualité.

Enumération En second lieu, avec le chapitre 3, nous attaquons les contributions principale de cette thèse: les algorithmes d'énumération. L'avantage c'est que ce sont des algorithmes complètement indépendants du domaine et qu'ils se basent juste sur les coûts. Nous nous intéresseons aux algorithmes qui énumèrent les programmes par coûts croissant et nous cherchons à faire des algorithmes plus rapides en pratique et avec un meilleur délai. Le délai représente la complexité temporelle entre l'énumération du $i$ème programme et du suivant. Historiquement, A^* [LHAN18] était le meilleur algorithme avec un délai logarithmique mais il était de la racine vers les feuilles, ce qui limite par exemple sa capacité à tirer profit de l'équivalence observationnelle: quand deux programmes ont les mêmes sorties sur les exemples, on peut en ne garder qu'un. Nous avons introduit Heap Search, le premier algorithme avec délai logarithmique, des feuilles vers la racine, et avions montré qu'il était bien plus efficace en pratique. Ameen et Leli [AL23] ont introduit Bee Search un autre algorithme avec délai logarithmique mais plus efficace en pratique et qui propose des idées intéressantes. En reprenant une partie de ces idées que nous combinons avec Heap Search, nous produisons Eco Search. Puis avec une analyse théorique non triviale, en utilisant des *bucket queue* [Tho00], nous adaptons l'agorithme pour avoir le premier algorithme en délai constant. Avec des expériences nous montrons que Eco Search est plus rapide et en étudiant la mise à l'échelle avec la complexité de la grammaire, nous montrons que Eco Search supporte mieux la mise à l'échelle.

Parallélisation Dans le chapitre 4 nous nous intéressons à la parallélisation de la recherche. D'abord, nous étudions les algorithmes d'échantillonage et introduisons SQRT

Sampling un algorithme d'échantillonage qui minimise l'espérance de tomber sur le bon programme. Les algorithmes d'échantillonage n'utilisant pas de mémoire sont en effet trivialement parallélisables. Néanmoins, avec des expériences nous montrons que cela ne reste pas compétitif avec les algorithmes d'énumération. Puis nous cherchons à paralléliser en séparant la grammaire en sous grammaires disjointes, permettant d'énumerer chaque sous grammaire indépendamment. Nous décrivons un algorithme permettant de faire cela sur des grammaires décrivant des ensembles de programmes finis.

Filtrage Dans le chapitre 5, nous discutons le fait que lors de la recherche de solutions, de nombreux programmes sont éliminer par élagage: les programmes syntactiquement différents mais sémantiquements équivalents sont éliminés. Une majeure partie de ces programmes redondants est dûe à des propriétes inhérentes au LSD et non à la tâche exécutée. Nous montrons donc comment en amont, on peut générer ces programmes redondants pour les compiler dans un automate d'arbre. Cet automate d'arbre permet ensuite d'éliminer ces programmes redondants lors de la recherche sans les évaluer. En effet, pour une grande partie des domaines, l'énumération coûte peu cher comparée à l'évaluation qui prend la majeure partie du temps. Nous montrons aussi que si les algorithmes peuvent s'adapter à des grammaires non contextuelles non ambiguës, alors ce filtrage peut directement se faire dans la grammaire. Nous montrons l'efficacité expérimentale du filtrage au niveau de la grammaire.

Connaissances Externes Dans le chapitre 6, nous nous intéressons à introduire de la connaissance dans nos programmes, cette connaissance est extérieure dans le sens où elle n'est pas dans le LSD. La majeure partie des approches ne considère que les transformations syntactiques pourtant, les personnes utilisant les outils tels que FlashFill [Gul11a] parlent du monde extérieur. Nous définissons donc la synthèse de programmes avec des connaissances. Nous montrons qu'il existe trois niveaux de difficulté. Nous décrivons WikiCoder un outil permettant de résoudre le premier niveau de difficulté. En premier lieu, nous décomposons la tâche en sous-tâches puis chaque sous-tâche est résolue en parallèle avec un outil de synthèse classique et un graphe de connaissances. Nous utilisons le graphe de connaissances pour trouver des chemins permettant de partir des noeuds en entrée et arriver aux noeuds en sortie. Nous montrons que cela fonctionne en pratique puis nous discutons des similarités et différences avec des approches basées sur des grands modèles de langages.

Futures Un axe immédiat est l'améliroation des méthodes énumératives qui actuellement doivent se souvenir de tous les programmes précédemment énumérés. L'objectif étant de diminuer cette quantité de mémoire nécessaire sans perdre trop de vitesse d'énumération.

A plus long terme, plusieurs axes vont revêtir une importance majeure dans le future de la synthèse de programmes, notamment les gros modèles de langages (GML). Certains travaux [SDES22, SDL+24, LPP24] essaient déjà de combiner ces modèles plus lent mais précis avec les techniques classiques de synthèse mais tout reste à faire. On peut notamment aussi les utiliser pour construire un LSD à la volée, une piste encore non explorée.

Un autre axe est la synthèse de programmes guidée par des récompenses, indépendamment des coûts, un programme est par exemple évalué dans un environnement, comme un circuit de voiture, et donné une récompense en fonction de la réussite de la tâche.

Le but de cette approche est d'attaquer l'apprentissage par renforcement en travaillant uniquement avec des stratégies, ici des programmes, explicables. Une idée sous jacente est qu'*a priori* les programmes devraient mieux généraliser et être plus robustes.

Introduction

The world of today is crawling with software from your phone to artificial hearts. The demand for software keeps growing in more and more complex environments. While a few years ago software was running on specific architectures; there is nowadays a high variability in the environments on which software can run. These days you can even run Twitter on your fridge. The domains in which software becomes keys are steadily increasing. For example, every scientist needs to know how to code basic analysis for their domain. This is coherent with the rise of data; which becomes apparent in every single field including the humanities. There is a relentless increasing quantity of data that is available to our disposal and in order to tackle such amounts; automated processing of data is essential.

However building software in highly variable environments is notoriously hard. It requires long term thinking in order to have robust, efficient, modular, bug-free software. Furthermore, it would be an error to think of software as a static object; on the contrary it is an ever changing object. With modern packagers for languages such as Python or JavaScript, it is easy to accumulate hundreds of dependencies; requiring frequent updates. On the other side, business is going faster than ever, in other words the expectations towards software are changing faster than ever.

In order to tackle these difficulties, programmers have developed tools and methodologies to alleviate some of the burdens that come with designing software. For example, modern integrated development environments offer refactoring features for most languages, debugging environments, autocompletion and a plethora of other features. But these tools often mainly focus on helping and not on the complete automation of the task. For example, fuzzers exist to help test your programs but they do not alleviate the need for extensive testing in order to check the robustness of your code nor to prove it is bug free. There are tools such as Coq [Tea24] in which you write code and prove it is correct at the same time; often only a part of the burden of proof is done by the software. Modern tools, while empowering developers, they fail to scale to the challenges developers face.

There are tools for automated verifications which aim to ensure that a given program is bug free, however they are complex to use, there is a translation step from code to logic and then the tool cannot always prove the desired property by itself and need the users to prove some steps by themselves. Instead we can dream of automated software synthesis: from specifications software synthesis aims to automatically generate software that meets the desired specifications in a bug free manner. Therefore by encoding in the constraints the performance and robustness, we can have everything we ever wanted: this is one of the holy grail of computer science.

Program Synthesis has been there since the beginning of computer science; its practical impact has been growing in the recent years. Nowadays most computer users have used automated synthesis without knowing about it in the form of either FlashFill [Gul11b] or

SmartFill [Goo21]. This thesis focuses mainly on making program synthesis work with as few assumptions as possible; so that it can be applied to all scenarios. Of course, this comes at the cost of efficiency compared to alternatives with more assumptions.

3 Program Synthesis: Dreams and Reality

Program Synthesis aims to generate a program satisfying a given specification. One of the main reason efficient program synthesis remains a challenge is that the specification only describes the goals that should be achieved, the *what*, but not how to achieve it: the *how*. The synthesiser should figure out how to achieve the goal by itself. General program synthesis is an unreachable dream: it is much too hard to tackle. It is often possible to encode complex problems which we do not yet know how to solve.

Therefore, researchers have focused on specific domains, where they would be able to produce algorithms that figure out the how. Intuitively, the smaller the domain, the easier it will be to synthesise a solution. This is the approach of FlashFill [Gul11b], the first resounding success of program synthesis. It only focus on syntactic string transformations and assumes that these transformations will be small in terms of program size. Let us look at the example below which shows a basic string transformation.

Example ▶ FlashFill Example

We describe this example of string transformation using Python syntax. Here is the specification as input output examples:

```
f("John Lerouge") = "John"
f("Didier Troispattes") = "Didier"
f("Bon Nethese") = "Bon"
```

A human will quickly find that the user wants to keep the first word, but this is much more complex from a computer's point of view. It has no concept of words. Rather it will leverage the fact that there is a space after the word that is going to be in the output and offer a solution like:

```
def f(s: str) -> str:
    return s[:s.find(" ")]
```

which keeps everything before the first space.

One other direction of research is interactive program synthesis with the user. This is well illustrated with the concept of *sketching* [BSL08], the idea is that the user should give some hints to help the process. Asking for specific information from the user is hard but the user usually has some high level idea of the structure of the solution such as a *sketch*: a *sketch* is a partial program with holes. Let us look at the sketching example below. Sketching can be viewed as doing the easy lifting from a user's point of view, in order for the synthesiser to do the heavy lifting. The synthesis difficulty from the synthesiser's point of view is radically different from the humans', the main factor of difficulty is the length of the programs generated. Therefore, helping the synthesiser with partial solutions dramatically helps the synthesis process.

Example ▶ Sketching

sketching Let us synthesise the factorial function, here are the examples:

```
f(1)=1
f(3)=6
f(4)=24
f(5)=120
```

However, generating this function is hard with only this much information, since we know it can be easily framed recursively, we can help with the following sketch:

```
def sketch(n: int) -> int:
    if n == ?: #terminal case
        return ?
    else:
        return ?
```

This will dramatically speed up the search by only synthesising the hard part of the code, the part where the user has to think in this case, leading the synthesiser to the following program:

```
def f(n: int) -> int:
    if n == 1:
        return 1
    else:
        return n * f(n-1)
```

There are multiple ways to specify a problem in program synthesis; one can for example give a logical specification, or pairs of inputs outputs as it was the case in previous examples or even natural language. More cases are possible depending on the domain but these are the three most popular. Natural language specification despite being the most accessible lacks one very important property: we cannot check that the synthesised program is correct with respect to the specification. Logical specification requires a high level of mathematical understanding and intuition in order to describe what we want; writing logic specification is a highly non trivial task. In the middle we have pairs of inputs outputs, which can be checked and is easy to write. It lacks the complete control that logical specification offers and is a bit more tedious than natural language but comes out as the correct trade-off to us. This is the type of specification we focus on in this thesis.

There are three main areas of program synthesis: deductive approaches, inductive approaches and local search such as genetic algorithms. They can all be applied to most specifications with variable success. They all rely on a key component: search. More precisely, a lot of different methods rely on enumeration as a subroutine. Thanks to the exponential growth of hardware capabilities, improving our search algorithms has proven more prolific [CHhH02, SSS+17] than including more human knowledge in our algorithm. This is why a large part of this thesis focus on improving the scaling of enumerative algorithms.

As mentioned previously, we want to work with as few assumptions as possible, that will translate for us as doing enumerative search of programs since it only requires the ability to evaluate programs and knowing the space of programs which is our search space.

4 Motivating Applications

Going back to the practical uses of program synthesis. There is a plethora of applications. We list four that we believe to be important.

Speeding-up Development In an approach where developers write code, test driven development is a way to code which prone writing tests first and then developing according to these tests. Instead, developers would just write tests and program synthesis would try to generate the code according to the tests. In fact, for business software since tests will be written at some point, writing tests first and getting as much code out of program synthesis could enable a dramatic speed-up. The example below shows a more business related task. While we are focusing on the business side, program synthesis has tremendous advantages in critical applications such as medicine or aeronautics, usually a large part of the time developing is spent ensuring correctness and soundness of the programs but program synthesis can guarantee these for free.

Example ▶ Development

Here are examples of a function that computes the total cost of a basket:

```
f([("Pikachu Onesie size M", 1)],
   {"Pikachu Onesie size M": 50}) = 50
f([], {"Pikachu Onesie size M": 15}) = 0
f([("Olympics Ticket: Sport Climbing - Semifinals W", 8),
   ("French Olympics Flag", 4),
   ("Olympics Ticket: Judo - Final - <63kg", 0)
   ],
   {"Olympics Ticket: Sport Climbing - Semifinals W": 200,
    "French Olympics Flag": 15,
    "Olympics Ticket: Judo - Final - <63kg": 800
    }) = 1660
```

which can be used to discover a simple implementation with program synthesis:

```
def total_cost(items: list[tuple[str, int]],
               cost: dict[str, int]) -> int:
    total = 0
    for name, qty in items:
        total += cost[name] * qty
    return total
```

Super Optimization Optimising programs is hard and tedious, it is common knowledge that developers should profile their code before optimising. Indeed it is often the case that bottlenecks are seldom the ones developers thought. Even when the code to optimise is clearly defined, a large part of the optimisation process is trial and error: a new variant of the algorithm is developed and tested to see if it is better. In other words, optimising code is hard. In fact, this process is very close to enumerative search. Automated program synthesis can leverage the current specification of the code in order to test large quantities of relevant programs to find better programs. A good example is shown below, we have the `min` function which uses a conditional statement and we want to optimise it because the function branches and in specific cases it can dramatically slow

down the processor. The compiler is not able to find such an optimisation but program synthesis can find a bit twiddling hack that enables to have a branchless `min`.

Example ▶ Super Optimization

Here are the examples of the `min` function:

```
f(1, 3)=1
f(5, 30)=5
f(47, 256)=47
f(5894, 151)=151
```

Here is the classic implementation of such a function:

```
def min(x: int, y: int) -> int:
    if x < y:
        return x
    else:
        return y
```

It is the trivial `min` function but perhaps in some intensive code, you want a faster version, one that does not branch: without any `if`. Using program synthesis we find this bit twiddling hack that you probably do not want to find yourself:

```
def f(x: int, y: int) -> int:
    return y ^ ((x ^ y) & -(x < y))
```

Automatic Discovery The idea behind this term is to discover novel approaches to solve a problem. This is often intertwined with super optimization in the sense that we want to find better approaches. A property of program synthesis is that it does not know *how* to solve the solution therefore it must create something from scratch: potentially novel. This is the approach used in [FBH+22], they use their program synthesis approach in order to find new algorithms to matrix multiplication. They successfully find a better version for matrix multiplication in a specific scenario.

Automated Proof Thanks to the Curry-Howard bijection [H+80, Cur34] writing programs is the same as writing constructive proofs. Therefore solving a program synthesis task is actually proving an equivalent statement. Let us walk through the example below, we will almost prove that there is an infinite number of prime numbers. In order to do that, we use the program synthesiser to find a way to create new primes that are not in a finite set of primes, showing that this finite set of primes does not contain all primes. The example shows us that by just knowing how to check our end property, program synthesis can help us find a constructive proof in order to prove our claim.

Example ▶ Automated Prof

We will prove that there is an infinite number of prime numbers using program synthesis. The idea is to say that we have a finite set X that contains all prime numbers. Using program synthesis we will try to build a new prime number that is not in X, that is a y that is prime to every element of X. We will change a bit our specification and describe it with an easily checkable property:

```
f([2, 3, 5]) = ?
f([2, 7, 9]) = ?
f([2, 7, 9, 91, 101]) = ?
# since we do not not how to build the solution
# instead we give a checker:
def check_is_sol(X: list[int], y: int) -> bool:
    return all(is_prime(y, z, X) for z in X)
```

with search we can find the following solution:

```
def f(X: list[int]) -> int:
    total = 1
    for y in X:
        total *= y
    return total + 1
```

which is a constructive proof that there is another prime number not included in the initial set. Note that here we would need to prove that this holds generally but deductive solvers can do that during synthesis.

5 Our Contributions

This thesis focuses on a specific context of the large domain which is program synthesis. Our contributions are listed there and will be listed again at the beginning of each chapter.

Code This thesis tackles not only some theoretical problems but also practical ones, thus it would not be complete without the associated code. The code has been developed over three years. An old version for the first experiments is available at `https://github.com/nathanael-fijalkow/DeepSynth` but is outdated and contains a small part of our work. The repository at `https://github.com/SynthesisLab/DeepSynth2` contains almost all of the work that was done during this thesis as of the date of this thesis the last commit is `ef20aa31d6f6f8e137d36aa20ee6c0cc6c308dfe`.

In this manuscript First, in Chapter 1, we will introduce the program synthesis problem we are considering. We will introduce most of the notions used throughout the thesis and introduce the different types of specifications; along with a literature review linked to the different kinds of specifications. The literature review contained there is rather small, since in later chapters, when we will focus on a specific subject, we will also provide a review of related works.

Second, in Chapter 2, we will introduce the program synthesis framework in which we are working; that is the pipeline we will be using to solve the problem and to use the relevant information provided by the specification. We will see that, in order to tackle

with as few hypotheses as possibles, many possible scenarios, search-as in enumeration-is the solution we chose. In order to accelerate the search, we use an oracle, in the form of a neural network, to bias the search towards some programs by adding costs on the grammar describing the program space. Our contributions in this chapter are linked to this neural network. First, the network must be trained, but data is often scarce for such specific domains so we have developed an algorithm to generate synthetic data of better quality. Second, the grammars are type specific, so we also developed an abstraction in order to train a single model for multiple grammars at the same time.

Third, in Chapter 3, we tackle the main contributions of this thesis: cost-guided search. Their main advantage is that they are completely agnostic to the domain because they are only based on costs. We focus on algorithms that enumerate programs in non-decreasing order of costs: these are best-first search algorithms. Our goal is to make these algorithms faster theoretically and also in practice. Theoretically, this is characterized by the delay which is the time complexity between the enumeration of the i^{th} program and the next. Historically, A^* [FMBD18] was the best top-down best first search algorithm with logarithmic delay. Being top-down limited its ability to profit from observational equivalence: you remove a program if it produces the same outputs on the examples as another program previously enumerated. We introduced, Heap Search the first bottom-up best-first search algorithm with logarithmic delay, being bottom-up enables it to fully leverage observational equivalence, but even without it, we showed that practically it outperforms A^*. Ameen and Leli [AL23] introduced Bee Search another bottom-up best-first search algorithm with logarithmic delay but faster in practice and that offered interesting ideas. By merging ideas from both, we developed Eco Search. Then with a non-trivial theoretical analyse, using *bucket queue* [Tho00], we adapt the algorithm to get the first best-first search algorithm with constant delay. With practical experiments, we show that Eco Search is faster, then we study the scaling laws and throughput with the complexity of the grammar and show that Eco Search is better than its competitors.

In Chapter 4, we consider parallelising the search process. First, we introduce a sampling algorithm, memoryless, therefore it can easily be parallelized and is optimal among sampling algorithms as a best-first search algorithm. Then, we describe another direction for effortless parallelisation: splitting the grammar into sub grammars, each grammar can then be enumerated independently enabling for a linear speed-up with the number of processors. We then describe an algorithm that can split grammars.

In Chapter 5, we address the idea that we should remove semantically equivalent but syntactically different programs in the grammar. Commonly done at runtime in a process called pruning. We show how we can generate automatically equations describing these semantic equalities. Then we show that incorporating these equations in the grammar model leads to a more powerful grammar model. However, having a more expressive model also slow downs the enumeration speed, despite that results show it largely improves solving rate and time. We end up with a pre processing step that can be done once and for all, enabling to generate equations which we compile into a grammar that only describe non redundant programs.

Then, in Chapter 6, we consider the specific domain of string transformations. Most approaches consider only syntactic transformations, but users reason about the world. They need external knowledge, that is knowledge that is not contained either in the language nor in the specification. This is what we call *knowledge powered* program synthesis. First, we decompose *knowledge powered* program synthesis into different levels of difficulty and target the first level: knowledge is not pre processed nor post processed, it can

be directly extracted as-is from the specification. We propose an approach that stacks on top of any synthesis pipeline. The idea is to decompose the task into sub tasks. Each sub task can be then solved in parallel either by the classic program synthesis pipeline or by the knowledge solver. Here our knowledge solver is a program that makes simple queries to a knowledge graph in order to find paths that enable to go from the inputs to the outputs. We show our approach solves numerous knowledge powered tasks. We briefly discuss the advantages and drawbacks compared to using large language models.

Finally, in Chapter 7 we will briefly go through what has been done in this thesis. We will discuss what this thesis has changed and opportunities for the futures of program synthesis.

Chapter 1

Background

TL;DR 1

This thesis focuses on programming by examples, in other words the specification of the program synthesis task is given as examples of input and outputs pairs and the space of programs is described by a grammar giving us functional programs. The synthesised program should be correct on the given input and outputs pairs, that is given an input it must produce the matching output. Our programs will be typed, thus each task is also typed. We also distinguish different classes of context-free grammars (CFGs): non-deterministic CFGs which represent the whole class of CFGs and det-CFGs which are non-ambiguous and locally deterministic.

In this chapter, we define the program synthesis problem. Then we describe the most common types of specifications which often imply different methods for solving the problem. First, in Section 1 we define the program synthesis problem. Then we discuss the different kind of specifications for program synthesis: Section 2 tackles logical specification then Section 3 treats natural language specification and finally Section 4 focuses on specification by examples.

1 Program Synthesis

When manipulating concepts it is easier to check that a solution is correct than to find a solution. One such well known example is NP-complete problems, here we will consider the boolean satisfiability problem (SAT) [Coo71, Tra84] which aims to find a correct assignment of the variables of a given boolean propositional formula. Because of its NP completeness it is very easy to manipulate and create a *verifier* however it is hard to find a solution. We can also see the solution as a program. One of the goal of the program synthesis problem is to build from this *verifier*, which is both easier to implement and easier to manipulate, a solution to the problem which passes the verifier. Program Synthesis aims to produce a solution from a program space that matches the given specification, which as in the previous example can be described by a *verifier*. Note that one might deduce that program synthesis is thus NP-complete, while it is true for some instances of the problem, it is generally false. Instead think of program synthesis as a meta problem parametrised by a specification space and a program space.

Definition 1.1 ▶ Program Synthesis Problem

The *program synthesis problem* is the problem that given:

- a specification φ
- a program space $\mathcal{P}$

of finding a program $p \in \mathcal{P}$ such that p satisfies φ *i.e.* $\forall x, \varphi(x, p(x))$

The program synthesis problem is a very generic problem introduced first in the context of circuit synthesis in 1957 by Alonzo Church [Chu57], this is why it is also known as Church's problem. This formulation is very generic in the terms of defining a program and a specification thus many problems can be framed as a program synthesis problem however it is not always relevant to do so.

Program Space In order to tackle this humongous variability, we will restrict program space to grammars. A grammar enables to generate a set of words which we interpret as trees here and we call programs.

Definition 1.2 ▶ Word

A *word* over a finite alphabet Σ is a sequence of elements of Σ.

The grammar defines syntactically which words belong to the language associated to the grammar or not. The grammar focus solely on syntactically describing words.

Definition 1.3 ▶ Grammar [Cho56, Cho57]

A *grammar* is a tuple $G = (N, T, P, S)$ where:

- N is a finite set of *nonterminal* symbols,
- T is a finite set of *terminal* symbols, note that $N \cap T = \emptyset$,
- P is the set of *production rules*, also called *derivation rules*, of the form $u \to v$ where both u and v are words over the alphabet $N \cup T$ and u contains at least one element of N,
- $S \in N$ is the start symbol.

It generates words over the alphabet T.

Example 1.4 ▶ Grammar for boolean formula with three variables

Let us consider a grammar that generates all boolean formula of three variables: x_0, x_1, x_2. Here is the set of production rules P:

$$S \to And(S,S) \qquad S \to Or(S,S)$$
$$S \to Not(S) \qquad \forall i \in [0,2],\ S \to x_i$$

Thus $G = (\{S\}, \{And, Or, Not, x_0, x_1, x_2\}, P, S)$.
We will show how we can check for example whether $And(x_0, Or(x_2, Not(x_3)))$ belongs to the language associated to the grammar. In order to do that, we look for a sequence of applications of production rules in order to generate the target program. We have underlined the nonterminal to which we apply the derivation rule in the following:

$$\begin{aligned}
&S \to And(\underline{S}, S) && \text{using } S \to And(S,S)\\
&S \to And(x_0, \underline{S}) && \text{using } S \to x_0\\
&S \to And(x_0, Or(\underline{S}, S)) && \text{using } S \to Or(S,S)\\
&S \to And(x_0, Or(x_1, \underline{S})) && \text{using } S \to x_1\\
&S \to And(x_0, Or(x_1, Not(\underline{S}))) && \text{using } S \to Not(S)\\
&S \to And(x_0, Or(x_1, Not(x_2))) && \text{using } S \to x_2
\end{aligned}$$

Therefore $And(x_0, Or(x_1, Not(x_2)))$ belongs to the language associated to the grammar.

Actually, the trace described in example 1.4 is called a derivation.

Definition 1.5 ▶ Derivation

A *derivation* of a word w in a grammar G is a sequence of applications of production rules from the start symbol to obtain w.

The definition of a derivation allows us to naturally define the ambiguity of a grammar.

Definition 1.6 ▶ Ambiguity

A grammar is *ambiguous* if there exists a word w for which there exists at least two possibles different derivations.

However the model of grammar described above is unrestricted and too large for program synthesis, in fact such grammars are as expressive as Turing machines [HU79]. We will consider a smaller model of grammars: context-free grammars. Their main advantage is the simplicity of computing their derivations, making them an ideal candidate in practice. We will see that it is a actually a trade-off between the expressive power of the grammars and their practical efficiency.

Definition 1.7 ▶ Context-free grammar(CFG) [HU79]

A *context-free grammar* (CFG) is a grammar $G = (N, T, P, S)$ with the restriction that production rules are of the form $u \to v$ with $u \in N$ and v, as in an unrestricted grammar, being a word over $N \cup T$.

Remark 1. *Notice that loops can still be created in a CFG meaning that they can describe infinite set of words or programs, this will often be the case.*

Going back to example 1.4, the grammar defined here is actually a CFG, there is only a nonterminal symbol on the left hand side. We will say *finite grammar* for a grammar that describes a language with a finite number of words, and denote by *infinite grammar* its infinite counterpart.

Definition 1.8 ▶ Local Ambiguity for CFG

A CFG is *locally ambiguous* if there exists a nonterminal S for which there exists at least two production rules which differ only by their nonterminals. If a CFG is not locally ambiguous then it is locally deterministic.

Example 1.9 ▶ Ambiguous and Locally ambiguous

Here is a CFG that is equivalent to the grammar of example 1.4 however, this one is locally ambiguous but not ambiguous. The change made was that the nonterminal S_1 was extracted to produce x_0 and $Not(x_0)$ leading to this local ambiguity.

$$
\begin{array}{ll}
S \to And(S,S) & S \to Or(S,S) \\
S \to And(S_1,S) & S \to Or(S_1,S) \\
S \to And(S_1,S_1) & S \to Or(S_1,S_1) \\
S \to And(S,S_1) & S \to Or(S,S_1) \\
S \to Not(S) & \forall i \in [1,2],\ S \to x_i \\
S_1 \to Not(S_1) & S_1 \to x_0
\end{array}
$$

Here deciding which production rule to use when we see an *And* or an *Or* in a program is not locally deterministic, we have to go further in order to see which production rule needs be used.

Through example 1.9, we see that the property of being locally deterministic seems like an interesting property in order to have efficient algorithms.

Definition 1.10 ▶ det-CFG

A *det-CFG* is a CFG that is locally deterministic.

We will refer later to the general set of all CFGs including the ambiguous ones as the *non-deterministic CFGs* (ND-CFGs). Now, that we have described our program space, let us move towards the specification space. In order to illustrate each specification, each will start with an example (see examples 1.11, 1.13 and 1.12) that provides a specification for the same exact task of mapping each element of a list to its square.

2 Logical Specifications

Example 1.11 ▶ Program Synthesis from a logical formula

Specification:

$$\varphi(x, y) = |x| = |y| \wedge \bigwedge_{i \in [1,|x|]} x[i] \times x[i] = y[i]$$

Historically, this is the first type of specification that was considered. The idea is to give the specification in the form of a logical formula. Church's seminal paper [Chu57] indeed aims to generate a circuit that matches a formula in monadic second order logic. There are also works in the same line [Rob65].

A few years later, Buchi and Handweber [BL69] framed program synthesis as an automata synthesis problem. Given a condition in sequential calculus they provide an algorithm in order to synthesise a strategy in the form of an automaton that satisfies the specification. This gave birth to the field of reactive synthesis [BCJ18] with the goal of synthesising state machines from specifications, usually in linear temporal logic. While this area is very interesting, this is not the goal of this thesis. The goal is not to produce finite state machines but programs. However, a part of the techniques deployed in their field can be applied to program synthesis. This led to the work on abstraction refinement for program synthesis [WDS17, GJJ+19]. The idea is to describe the space of solutions programs as an automaton; this is an abstract space. This abstract space is built based on the semantic of the programs making full use of the fact that different syntax does not mean different semantic. Then at each step, take a program from this abstract space, if it is a solution then the job is done, if not we find a counter-example and extract predicates that are relevant for our abstract automaton in order to refine it. At the end of the process, either a solution was found or we have proven that there is no solution in the current DSL.

Another approach to this specification type was started by the work of Manna and Waldinger [MW80] where program synthesis is seen as a theorem proving task relying on the Curry-Howard correspondence between constructive proofs and algorithms [H+80, Cur34]. The specification is given as first order logical formula and by proving the formula, a solution program is generated or a proof that no solution exists is provided. This is what is called *deductive synthesis*, the program space is axiomatized and only potentially correct programs are considered until a proof is made of either the correctness of a program or that there is no program that can solve the task under the current axiomatization. We will cover more precisely deductive program synthesis later on. Logic specification is not the target of this thesis. As mentioned in the introduction, logical specification often requires a certain skill from the user and that is not easily obtainable for people without a computer science background.

In recent years, SyGuS [ABJ+13, AFSSL16a, AFSSL16b, AFSSL17] introduced a specification format and a competition which was organised from 2013 until 2019, has had a large impact on the development of new solvers for program synthesis.

3 Natural Language Specifications

Example 1.12 ▶ Program Synthesis from natural language

Specification: Write a program that map each element of the list to its square.

Generating programs from natural language specifications is not necessarily new. The rise of high level programming languages such as Python are already big steps towards such an endeavour. The main cause of such developments however is the advent of machine learning and more precisely neural networks. Indeed, since AlexNet [KSH12] in 2012, there is an ever growing interest in machine learning.

While being easily interpretable by humans, this specification is very vague and unclear. By making it easy to manipulate for people even without a background in computer science, this comes at the cost of accuracy. The largest drawback is that the specification itself does not allow for checking that the program has the desired behaviour. Therefore, it is often the case that tests are also provided in order to test the behaviour of the program. Furthermore, one might bias the produced program by what is written, perhaps the process described is not the correct one in order to realise the current task. Or the specification might be ambiguous or underspecified, in fact there are papers [LXWZ23] adding additional tests to underspecified tests in NLP and fixing ambiguous wordings. The measure of what is correct is completely broken, currently the solution is to use a set of tests that checks that the solution is correct.

Neural networks however do not see code as code but as text, for them there is no difference in the way words are transcribed to the model just like text. Back in 2016, *an eternity ago for deep learning*, the work of Iyer *et.al.* [IKCZ16] used neural networks in order to summarize code. A few years later, others used deep neural networks in order to provide clever code recommendations [LYB+19]. Code2Seq [ABLY19] also tackled code summarization but came with the idea of treating code as a sequence. Then works emerged focusing on generating code for domain specific languages. This work focused on generating code sequentially, the code is then neuro-executed and fed back to the generator [CST21]. The neuro-execution is a deep learning model that takes code as input and provides a vector in an abstract space that should represent the current state of the execution.

In parallel, transformers [VSP+17] emerged in 2017, these are architectures that were specifically trained for text sequence prediction. Bert [DCLT19] was the first large model that beat a large part of natural language processing leaderboards. Until GPT-2 [RWC+19] beat it at its own game, then came GPT-3 [BMR+20a] which was ten times bigger and produced even better results but it was still far from being able to code making too many syntax mistakes. Then ChatGPT was released with no paper explaining how it works, a chat interface and an improved version of GPT-3, it was able to produce code for quite a range of programs. This is the advent of large language models (*LLMs*), which are neural networks with at least billions of parameters, usually costing millions to train on most of the text available on internet. Last year, GPT-4 [AAA+23] was released, and CEOs are starting to question the need to teach children how to code. While their ability to produce code under certain conditions are great; a lot of issues remain and it does not look like as if they will be solved anytime soon [BTK+23, KUIKZ24, LLS+24]. This account may lead the reader to believe that the only large models are the GPT family but there are numerous others such as LLama [TLI+23], DeepSeek Coder [GZY+24],

Gemini [TAB+23], we just exposed the story through the most prominent one.

Datasets such as HumanEval [CST21], Mostly Basic Python Programs (MBPP) [AON+21] have become the standard for evaluating the capabilities of such models in generating code. However, it is hard to check that the generated code is correct this is why these datasets offer tests in order to check the behaviour of the produced program. Despite this, critics [LXWZ23] have been raised due to the incompleteness of such tests and offered corrections by adding more tests and offered fixes because of ambiguous formulations in natural language.

This thesis does not consider natural language specifications because of its ambiguity.

4 Examples Specifications (PBE)

Example 1.13 ▶ Program Synthesis from Examples

Specification:

$$f([1,2,4]) = [1,4,16] \qquad f([9,3]) = [81,9]$$
$$f([]) = [] \qquad f([9,5,5,6]) = [81,25,25,30]$$

This the *programming by examples* (PBE) framework. This specification emerged from inductive programming which was first introduced by the work of Biermann [Bie78] with the goal of generating LISP programs from examples.

One of the main advantage in specifying from examples is that anybody can write these kind of specifications. However, there is bias in the selection of examples. Providing the right examples in order to get the desired program is a complex open problem. The current approach is to ask the user to add a new example until the desired program is generated in order to remove undesirable candidate solution programs. Another advantage compared to the current NLP approach of describing the program, is that the user do not have to describe how to solve the task, only the solution that should come out of the program. In machine learning terms, the natural language specification is akin to supervised learning whereas PBE is akin to reinforcement learning where we only know the consequences that we want to bring about.

Despite previous works [LWDW03, LBCO04, DRJDLM10, SL08] on the subject, the boom of this specification came with the seminal paper: FlashFill [Gul11a] which was the first application of program synthesis in real life. It was used in Microsoft Excel, and has been used by most people who have used a computer in the last 15 years. The idea between FlashFill is to help users autocomplete their spreadsheet by selecting cells and then dragging them to extend them upon new cells. The cells are captured and each row is a set of inputs to produce one output, producing a PBE specification. In order to tackle the enormous complexity of syntactic string manipulations, they use the idea of witness functions. They are close to inverse mappings, the idea is that given a target output a witness function gives a set of inputs that are coherent with the output. This is an overapproximation of the preimage of the output through a given primitive. It enables in some cases to drastically reduce the search space since only this approximation needs to be searched. The approach was adapted to other uses in spreadsheets [GHS12, SG12, SG16, DUB+17].

Then a second generation of approaches for PBE sprouted. With the Deepcoder [BGB+17]

paper, the community merged deep learning with search. The idea behind DeepCoder is to have a neural network that takes as input the examples for a task and gives the probability for each primitive that it appears in the solution program. Once the probabilities are obtained, it induces a probabilistic grammar, search is performed in this grammar according to the probabilities in order to find the best result first. The principle is that a large issue of PBE techniques rely on search, which has a hard time scaling with the number of primitives, therefore reducing the number of primitives is key. Others [CPS20, LMPS23] have taken up that idea and pushed it further. PCCoder [ZW18] introduced the idea of learning with a garbage collector, instead of viewing the program as a functional program, one can view it as an imperative program, with one statement per line. Then the question of dropping variables identifier from memories appear. They learn a neural net that learns to drop identifiers from memories, this enable them to generate longer programs than their competitors.

Another seminal paper is DreamCoder [EWN+21] which takes the ideas of previous works and goes further with bayesian wake-sleep programming. During wake, they solve tasks of their DSL. During sleep, they learn new lambda terms functions based on the tasks solved during wake in order to learn new primitives, this is more than just refactoring since this increase the expressivity of their DSL. They prove the efficiency of their approach on various domains achieving better results; furthermore starting from only basic functional primitives such as `cons`, `car`, `cdr` and `map` they manage to rediscover high level primitives such as `filter` or `nth_largest` eventually learning `sort`. This idea of library learning to augment expressivity is a ongoing idea that has been applied to other contexts such as quantum circuits [SEM24].

The quality of predictions is key, for example focusing on a specific domain such as in TFCoder [SBS22a], leads to better models. CrossBeam [SDES22] instead view program synthesis as a process that should be guided at all steps; instead of having a model that gives information to guide the search; the model is queried each time a new primitive of the DSL is applied. At the cost of speed, they manage to have highly relevant combinations of primitive leading to great results on benchmarks. LambdaBeam [SDL+24] uses the same idea in order to build lambda terms, notoriously very hard since it usually make the program synthesis problem explodes in complexity.

There are many different lines of work; as mentioned the quality of the predictions is key to obtaining good performances, HySynth and others [BGK+24, LPP24] suggest that large language models capture all the necessary information for most DSLs and are even better than models trained solely for this purpose. FlashFill++ [CGL+23] takes the idea of FlashFill but adds cut functions that enables to split the program synthesis tasks into subtasks, leveraging more domain knowledge than only with witness functions. Ideally, cut functions enable to cut the grammar to allow only coherent subprograms. For example a cut for `lowerCase` would be that the input must be a string and the input must contain all characters of the output. There is also a line of work on probabilistic programs [MDK+18] which is useful in applications like biology and is so far the best way people have found to tackle noisy data and uncertainty in the context of program synthesis.

Chapter 2

A Generic Program Synthesis Framework

TL;DR 2

A good overview of this chapter is Figure 2.2 which describes the full pipeline used at a high level. We have an inductive enumerative approach. Examples are fed into a prediction model which gives us a probabilistic grammar which enables domain agnostic solving with cost-guided search which will be discussed in the next chapter. The advantage of such a framework is that it is completely agnostic to the domain in which it is used contrarily to most other approaches which require a lot of domain specific knowledge such as deductive approaches. We describe how to train a predictive model, the different DSLs that will be considered throughout this manuscript. We show how we can use one prediction model for different grammars and describe an algorithm to generate synthetic data of better quality.

This chapter will introduce the idea of genericity for program synthesis in Section 1 then we will describe the proposed solution in Section 2. The ideas described here are present in numerous of works but they are introduced in [FLM+22].

Our contributions:

- An single prediction model that can be used to predict multiple PCFGs at a desired granularity.
- A better than random PBE task generation system.

Program Synthesis from examples to find a solution program contained in a CFG is our objective. Successful approaches [Gul11a] to program synthesis have been focusing on a particular domain. In order to target a specific domain, specific techniques have been employed. One of which is essential, is the language in which programs are written. This is what we call a domain specific language.

Definition 2.1 ▶ Domain Specific Language (DSL)

A *domain specific language* is a tuple $L = (G, E)$ where G is a grammar that describes syntactically the language and E is an evaluation function which describes the semantic of the language. E takes as inputs a program $P \in G$ and an input x and return P evaluated on x.

The advantage of DSLs is that they provide a good trade-off between expressive power and abstraction. The larger the language the more complex the program synthesis is. Conversely, the smaller the programs the easier program synthesis is. Large languages can provide us with a lot of functions making it easy to generate small programs however their size would make the problem much harder. Most DSLs used in practice are on the smaller side because language size is the most limiting factor.

In this thesis, we will consider a DSL syntax by its primitives that is the functions that are provided in the DSL. Then based on the types of the functions along with the type of the function requested for the synthesis problem, we can compile the DSL automatically into a CFG. This process is described in Figure 2.1, the list of primitives contain both functions and constants provided by the DSL. Then the grammar obtained when the type request is $int \to int$ is shown on the right. That means we want to generate a program that takes one argument of type *int* and returns an *int*. The compilation shown here is actually quite simple, nonterminals only keep the type information. However, more advanced compilation can be used. As seen in example 2.2, nonterminals can keep bigrams, that is nonterminal will keep memory of the last primitive used and which argument this nonterminal is deriving from.

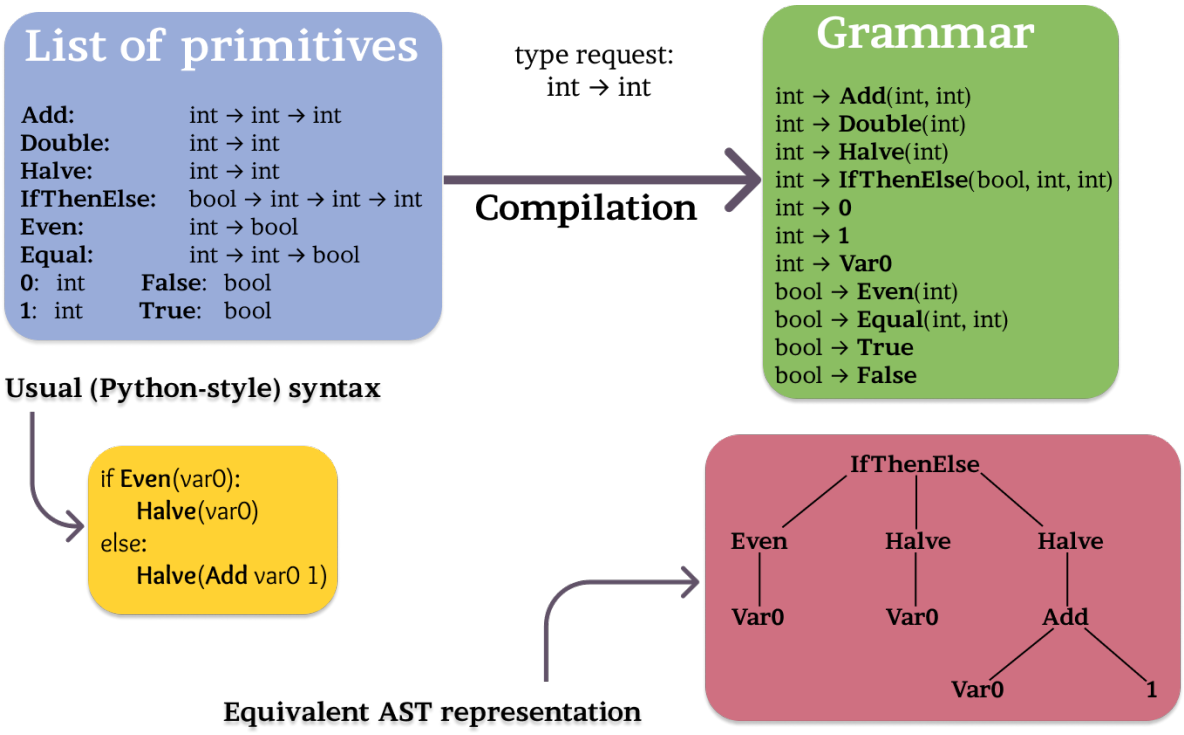


Figure 2.1: Compilation of a grammar from the primitives

Underneath the list of primitives in Figure 2.1, we show a sample program in a Python style syntax and its equivalent representation as a tree, this tree representation is called *abstract syntax tree* (AST), it is relevant to remember to see programs as trees.

Example 2.2 ▶ Bigrams

Here is an example grammar where nonterminals correspond to bigrams:

$$
\begin{array}{ll}
(I,\ ()) \to +(I,\ (+,0))(I,\ (+,1)) & (I,\ ()) \to -(I,\ (-,0))(I,\ (-,1)) \\
(I,\ (+,0)) \to +(I,\ (+,0))(I,\ (+,1)) & (I,\ (+,0)) \to -(I,\ (-,0))(I,\ (-,1)) \\
(I,\ (+,1)) \to -(I,\ (-,0))(I,\ (-,1)) & \\
(I,\ (-,0)) \to +(I,\ (+,0))(I,\ (+,1)) & (I,\ (-,1)) \to +(I,\ (+,0))(I,\ (+,1)) \\
(I,\ ()) \to var_0 & (I,\ ()) \to 1 \\
(I,\ (+,0)) \to var_0 & (I,\ (+,1)) \to var_0 \\
(I,\ (+,0)) \to 1 & (I,\ (+,1)) \to 1 \\
(I,\ (-,0)) \to var_0 & (I,\ (-,1)) \to var_0 \\
(I,\ (-,0)) \to 1 & (I,\ (-,1)) \to 1
\end{array}
$$

We start with $(I,\ ())$ where we can use both $+$ and $-$, then if we chose a $+$ since $(I,\ (+,1))$ does not allow $+$ then $+$ is forced to be left associative. If we choose $-$ then both $(I,\ (-,0))$ and $(I,\ (-,1))$ do not allow for $-$ so we can only use $+$. All nonterminals lead to 1 and var_0 which corresponds to the input given to the program. While the grammar is much more verbose, we see that with a larger grammar we could express some constraints such as left associativity or forbidding the use of a $-$ after a $-$.

1 Domain Agnostic Framework

The goal of this thesis is to tackle domain agnostic techniques for program synthesis. That means that all techniques should be useful and general enough so that they can be used for any domain. This brings about a lot of issues. There are a lot of different domains in which program synthesis can be applied: string manipulations [Gul11a, DUB+17], data completion scripts [WDS17], graphic programs [ERSLT18], symbolic regression [UT20], quantum circuits [SEM24], *etc.*

Deductive Program Synthesis During the 1990's the research community was looking at transformations from high level specifications [WB89, Gon98, GGGS86]. The advent of inductive logic programming was simultaneous [Sha82, Qui90, Mug91]. It enabled to answer basic reasoning questions from a base of facts along with logic relations.

Example 2.3 ▶ Prolog Questions

This example is taken from [CKPR73] and translated in english.

```
Input:
    EVERY PSYCHIATRIST IS A PERSON .
    EACH PERSON THEY ANALYSE IS SICK .
    *JACQUES IS A PSYCHIATRIST IN *MARSEILLES.
    IS *JACQUES A PERSON?
    WHERE IS *JACQUES?
    IS *JACQUES SICK?
Answer:
    YES
    IN MARSEILLE
    I DO NOT KNOW
```

Every single phrase is translated into first order logic. The first two phrases enable us to deduce knowledge, they are implications. Then the phrase about `Jacques` gives us facts. From this data, a deducer is called to reason on the questions asked in order to answer them.

This led Bundy to work on the automation of inductive proofs [Bun88, IB96, Bun99] which are programs thanks to the Curry-Howard equivalence [H+80, Cur34]. This is what the works of Manna and Walding [MW80] exactly do, they propose to solve program synthesis as theorem proving; this is the emergence of deductive program synthesis. This was also linked to intuitionist type theory [ML82, HML75].

This led to dramatic improvements in the 2000s. Proof assistants were developed such as Coq [CH85] or Isabelle [Pau94]. Program synthesis distinguished itself a bit more from theorem proving with the landmark paper describing SyGus [ABJ+13]. Given a syntax and a logical specification in the SyGus format, you need to find a program satisfying the specification in the given syntax. The syntax specification is actually a context-free grammar; more precisely we have found that in practice it is always a det-CFG. In order to solve these tasks, people use SMT solvers such as z3 [DMB08] or yices [DDM06]. With the SyGus competitions [AFSSL16a, AFSSL16b, AFSSL17] pushing forward the development of such tools; highly performant tools were developed like EUSolver [ARU17] which was state of the art in 2017. That culminated with cvc4sy [RBN+19] which uses cvc4 as a subroutine [BCD+11]. With specific subroutines for cvc4 for fast and smart enumerations of relevant program candidates, they won the subsequent competitions. The focus of a part of the research community has now shifted towards mixing deductive approaches with neural approaches.

While these techniques are promising; a few key properties are that the semantic should be encodable in first order logic. Users must be able to know how to reason about the semantics of the language, which is in fact a high requirement. Most developers do not how to write first order logic specifications. We will rely on less information but that comes at a cost: when translating in first order logic is possible, deductive approaches will be the best choice; but in the more general case, we need another approach. As a side note, some subroutines that we improve upon are shared with the deductive approaches.

2 One Solution: Cost Guided Search

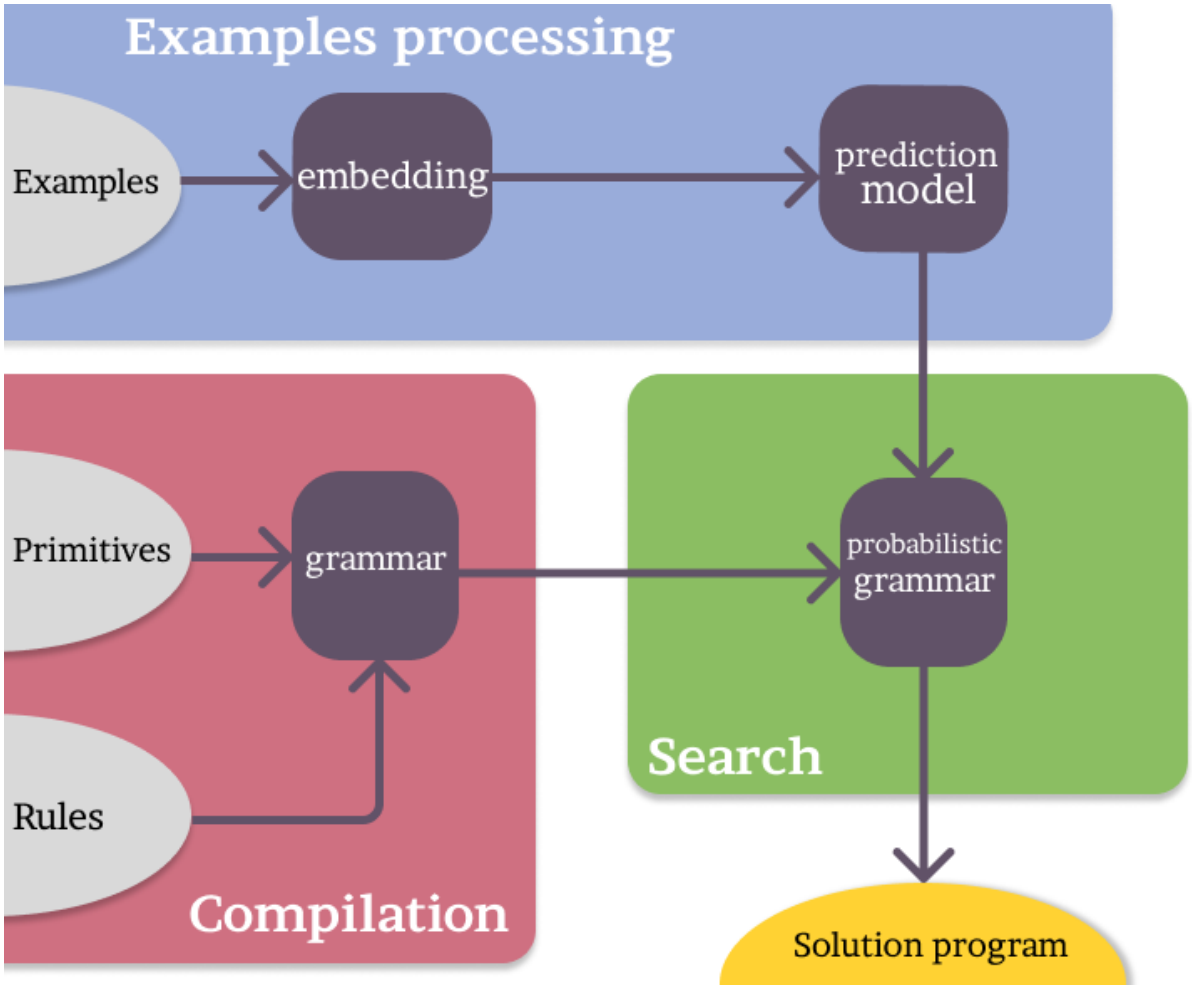


Figure 2.2: Cost-Guided Search Program Synthesis Pipeline

Instead of adding more bias by providing more information, we choose a cost guided approach which is so far the most generic approach to program synthesis. The idea behind cost guided search is to enumerate programs and check if they are a solution to the problem, until a solution is found. In order to do that we must be able to check if a program is a solution. The specification is given as examples, therefore we only need to check that the program agrees on the examples. In order to do that we need to be able to evaluate a program, since we have a DSL that comes with semantic, then we can evaluate programs.

Remark 2. *Notice that enumeration will never terminate on infinite grammars, this means that this approach cannot offer a proof that there is no solution. Furthermore, we will see that even on finite grammars, chances are that there are so many programs that it will not terminate in reasonable time even if we were to only enumerate semantically unique programs.*

Enumerating programs is a brute force approach. The order in which programs are enumerated is key to the performance of such methods. Therefore we will view this as if enumeration was guided by an oracle cost function, this is why this is called cost-guided search.

Definition 2.4 ▶ Probabilistic CFG (PCFG)

A *probabilistic context-free grammar* is a CFG where each production rule is tagged with a real in $[0; 1]$. Given a nonterminal $n \in N$, then the sum of all tags from this nonterminal sums up to 1.

This leads us to obtain the pipeline described in Figure 2.2. First, examples are processed in order to give a prediction which will be used to order programs for the enumeration. Second, the primitives along with the syntax rules are used in order to compile a grammar. Finally, the grammar is combined with the prediction model to obtain a probabilistic grammar which will be used by the cost guided search. In this context, we call them probabilistic grammar but this is a special case of cost grammars, however in the following all our algorithms will treat the probabilistic grammar as if it were a cost grammar, it is easier conceptually to see this as a probabilistic grammar than a cost grammar.

2.A Prediction model

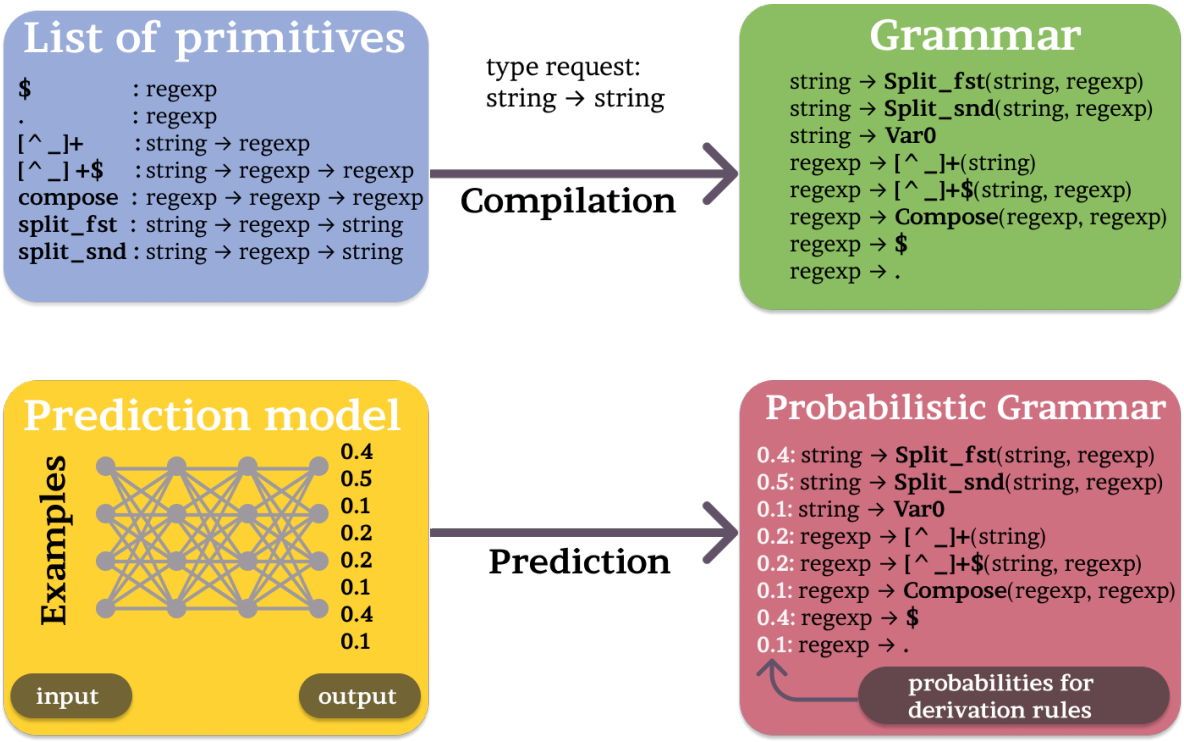


Figure 2.3: Illustration of the prediction model.

This prediction process is illustrated in Figure 2.3. In other words, it defines a probabilistic distribution over programs, and the prediction model is trained to maximise the probability that a solution program is generated. This effectively leverages the prediction power of neural networks without sacrificing correctness: the prediction model biases the search towards most likely programs but does not remove part of the search tree, hence – theoretically – a solution will be found if there exists one. This transforms the context-free grammar representing programs in a probabilistic context-free grammar, which is a stochastic process generating programs.

Let us briefly explain the framework for building a prediction model. First, we start with an input grammar. Then we have an abstraction of the grammar for the model. That is as in DreamCoder [EWN+21] we may want to predict only the presence of a primitive or we may want to predict the probability of deriving primitives in context [FLM+22]. This choice is streamlined by this abstraction, it is a function that maps the nonterminals to a set of symbols. The prediction model will learn probabilities for this set of symbols and then when producing a probabilistic grammar, all nonterminals mapping to the same symbol will share the same probability. Therefore the abstraction enables to adjust the granularity with which we want to learn probabilities. Notice that the abstraction can only lower the granularity it cannot increase it. In the following, we will only have two types of abstractions: one equivalent to DreamCoder [EWN+21] to predict at program level the presence of a primitive and one that do not lose any information from the grammar, that is it predicts in context, with bigrams, the probabilities of using primitives [FLM+22]. There is one thing we choose to not predict, it is the usage of arguments of the task, their presence is likely to be mandatory and in context since it is likely the first time we have done this task we have little to no prior information in the usage of arguments therefore we choose to set their probability to 0.2 which is quite high to ensure their usage.

In order to be able to use a single model per DSL we need to be able to tackle multiple grammars since a grammar corresponds to a single program type. For example, on Figure 2.3, we showed only the grammar for the type request $string \to string$, but we could consider types $string \to string \to string$ or $string \to regexp$, a concrete example can be seen in example 2.5.

Example 2.5 ▶ Two grammars from the same DSL

Let us take the following grammar with type request $int \to int$:

$$\begin{aligned} &I \to + \ I \ I && I \to - \ I \ I \\ &I \to x_0 && I \to 1 \\ &I \to 2 && I \to 3 \end{aligned}$$

and the grammar with type request $string \to string \to int$:

$$\begin{aligned} &I \to + \ I \ I && I \to 1 \\ &I \to length \ S && I \to parseInt \ S \\ &S \to + \ S \ S && S \to substring \ S \ I \ I \\ &S \to x_0 && S \to x_1 \end{aligned}$$

These two grammars are part of the same DSL. Another version of these grammars of limited depth is to make a product of the nonterminals with the depth so far: we start at $(I,\ 0)$ then at each derivation rule that is not a constant we increase the number by 1 until we reach the limit where we only allow constants. For brevity, we only show the first one with depth limited to 3:

$$\begin{aligned} &\forall d \in [0,2], && (I,\ d) \to + \ (I,\ d+1) \ (I,\ d+1) && (I,\ d) \to - \ (I,\ d+1) \ (I,\ d+1) \\ &\forall d \in [0,3], && (I,\ d) \to x_0 && (I,\ d) \to 1 \\ &\forall d \in [0,3], && (I,\ d) \to 2 && (I,\ d) \to 3 \end{aligned}$$

To tackle this, we just need to use the abstraction on all of the grammars provided at the same time, given the type request we can decode the correct grammar without any ambiguity. The prediction model might produce probabilities for symbols that do not map to any nonterminals for the requested grammar but we can just ignore that information. The two last concepts are illustrated through example 2.6. It shows that both abstractions offer a trade-off, predicting the primitive presence is easy and requires few data but provides very limited information for the search whereas the second abstraction offer a lot of information for the search but is much harder to predict and requires more data for training.

Example 2.6 ▶ Abstractions and predictions

Considering the grammars of example 2.5 with depths limited to 3 and that we want to use a single model, then the abstraction that only predict probability presence would yield:

$$
\begin{array}{ll}
X \to + \ I \ I & X \to - \ I \ I \\
X \to 1 & X \to 2 \\
X \to 3 & X \to \mathit{length} \ S \\
X \to \mathit{parseInt} \ S & X \to \mathit{substring} \ S \ I \ I \\
X \to + \ S \ S &
\end{array}
$$

Our notation indicates that for each production rule in the above a probability is computed, note the disappearance of variables. Note that while there are two primitives that share the same name they are not merged since they take arguments of different types and produce different types, to be merged these have to match. Furthermore, since all the nonterminals have been renamed to the same symbol without the depth information all that information is lost. Using the same notation as previously, the abstraction that do not lose any information, *i.e.* the identity abstraction, would yield the following rules to be predicted:

$$
\begin{array}{ll}
\forall d \in [0,2], & (I,\ d) \to + \ (I,\ d+1) \ (I,\ d+1) \\
\forall d \in [0,2], & (I,\ d) \to - \ (I,\ d+1) \ (I,\ d+1) \\
\forall d \in [0,2], & (I,\ d) \to \mathit{length} \ (S,\ d+1) \\
\forall d \in [0,2], & (I,\ d) \to \mathit{parseInt} \ (S,\ d+1) \\
\forall d \in [0,3], & (I,\ d) \to 1 \\
\forall d \in [0,3], & (I,\ d) \to 2 \\
\forall d \in [0,3], & (I,\ d) \to 3 \\
\forall d \in [0,2], & (S,\ d) \to + \ (S,\ d+1) \ (S,\ d+1) \\
\forall d \in [0,2], & (S,\ d) \to \mathit{substring} \ (S,\ d+1) \ (I,\ d+1) \ (I,\ d+1)
\end{array}
$$

There is a huge contrast between the two abstractions, for the first abstraction there are 10 rules to predict while for the latter there are 30 rules to predict.

Following [EWN+21, FLM+22], we build a prediction model in the form of a neural network: it reads embedding of the examples and outputs probabilities on the derivation rules of the grammar representing all programs. Figure 2.4 describes our architecture

with an example of how the model looks like when there are only two examples.

In order for the prediction model to work on our examples, they need to be converted to a vector of float. Examples are one-hot encoded, that means we need a lexicon describing all the symbols in our vocabulary, and that we have special symbols reserved to indicate that we move to the output of the example or to the next argument of the input. Then all examples are padded to be of matching length. Once they are one-hot encoded, they go through an embedding layer. Now for each of the k examples we have one vector of floats. To manage this variable number of examples we feed them sequentially in a one layer LSTM [HS97]. After this step we now have a single vector which we can feed to a simple stack of two linear layers with ReLU activations [Fuk69]. Finally, we end up at the last layer which given the grammars and the abstraction will automatically produce a vector of the matching size. The log softmax is applied on a abstracted symbol level and not directly on the vector. More precisely, it is applied at the symbol level so that we get probably distributions in the log space. This means that given a type request we produce a log probabilistic grammar after a normalisation procedure to ensure probabilities sum up to 1 since we need to add variables which we did not include. For learning, we can simply maximise the log probability of generating the solution program, which can easily be computed using our log probabilistic grammar.

The architecture shown is not random. We tried improving the program synthesis performance by:

- swapping the LSTM with a GRU [CvMG+14] or a RNN [RHW86],
- increasing the number of linear layers,
- changing the activations functions,
- changing the weight initialisations.

But none of the changes above offered significant performance changes, decreasing the size of the current network however has a negative impact on performances, therefore the current architecture is quite robust to changes. This is also a negative result in the sense that we do not manage to provide enough information to the network in order to improve its performances. An idea was to use attention [VSP+17] a mechanism that is key to the success of LLMs but preliminary experiments showed no improvements so we did not pursue the idea further. Our conclusion is that the encoding of our examples is of poor quality, we did not have the time to look into improving it but an emerging idea in the field is to leverage LLM tuned on code generation, they should be able to embed correctly code even if they are from a DSL and that means they are capable to extract relevant information out of their encoding and embeddings. One example could be using the embedding part of StarCoder2 [LLA+24] a subpart of the network that is relatively small compared to the full network and use it instead of our embedding.

Training A main issue is training because of the lack of data. Most existing datasets were either written by humans therefore have few tasks or they have been randomly generated and are of poor quality. It is known that the quality of the examples is of major importance [CMF+20]. In order to tackle this issue, we built a pipeline in order to generate synthetic tasks of better than purely random quality. First, we start from the test dataset, one written by humans and of relatively high quality, this will be our assumption.

We learn probabilistic grammars for each type request in the original dataset empirically if solutions are present otherwise we will use uniform probabilistic grammars. What we mean by learning here is the empirical mean among all the solutions for this type request, see example 2.7.

Example 2.7 ▶ Empirical Learning

Consider the following three tasks f_1, f_2, f_3:

$$f_1(1) = 2 \qquad f_1(2) = 3$$
$$f_2(4) = -1 \qquad f_2(9) = 4$$
$$f_3(4) = 2 \qquad f_3(7) = 2$$

The solutions would be:

```
def f1(x):
    return x + 1
def f2(x):
    return x - (1 + 3)
def f3(x):
    return 2
```

Then empirically for examples we would generate *int* uniformly in the range $[-1, 4]$ and the grammar we would sample programs from is, if we learn from a grammar with no bigrams nor depth information:

$$I \xrightarrow{2} + \; I \; I \qquad I \xrightarrow{1} - \; I \; I$$
$$I \xrightarrow{2} x_0 \qquad I \xrightarrow{2} 1$$
$$I \xrightarrow{1} 2 \qquad I \xrightarrow{1} 3$$

notice that the grammar is not normalised, we would need to divide by the sum so by 9. An argument that can be made is that a long program contributes more than smaller ones in which case we can normalise by the size of the program (viewed as a tree). For example, f_1 has size 3 therefore it contributes $\frac{1}{3}$ to $+, 1$ and 3.

Simultaneously, we also capture a lexicon of all symbols contained in the dataset. For numbers, we capture the minimum and maximum value and will assume that all values within that range are allowed. For lists, we capture the empirical distribution of lists lengths. Given these distributions we can sample inputs of certain types and we can sample programs. This learning for numbers and lists lengths is the only domain specific technique employed, the rest is domain agnostic. Example 2.7 describes this process.

Generating synthetic data of better quality Generating a dataset of better quality than random is quite hard since we assume no knowledge of the specific domain since we are targeting a domain agnostic technique. We describe an algorithm that tries to improve the quality of the tasks by ensuring that they are unique up to their input output signatures. Now let us describe Algorithm 1. First, SampleInputs($n_{examples}, n_{inputs}$) sample n_{inputs} set of inputs with each containing $n_{examples}$ examples. All programs should share the same inputs and be mainly differentiated by their outputs, enabling the model to grasp deeper semantics. At the same time the inputs should not be constant that is

why we produce n_{inputs} different of such sets. Then we try up to 100 times to get the desired number of programs, that are unique with respect to their outputs. To do so UniqueProgramsByOutputs(I, P, E) evaluates all programs of P on all inputs in I with E, if two programs match we put them in the same equivalence class, then only one program per class is returned. Finally, MakeTasks(P, I, E) is an abstraction that constructs the tasks objects with the given data.

Algorithm 1 Tasks Generation

procedure GenerateTasks($n_{examples}$, n_{inputs}, n_{tasks}, E: evaluator)
  $I \leftarrow$ SampleInputs$(n_{examples}, n_{inputs})$
  $P \leftarrow \emptyset$
  for $j = 0$ to 100 **do**
    $P \leftarrow P \cup$ SamplePrograms$(n_{tasks} - |P|)$
    $P \leftarrow$ UniqueProgramsByOutputs(I, P, E)
    if $|P| \geq n_{tasks}$ **then**
      return MakeTasks(P, I, E)
    end if
  end for
  return MakeTasks(P, I, E)
end procedure

3 DSLs

Now that everything has been defined, we will introduce the DSLs that will be used throughout all of this thesis.

3.A DeepCoder or Integer List Manipulation

DeepCoder [BGB+17] is an integer list manipulation DSL, it has specific versions of classic functional operators such as `MAP`, `FILTER`. It is thus rather large containing nearly 40 primitives without any constants. It contains the following primitives:

```
primitives = {
    "HEAD": "int list -> int",
    "TAIL": "int list -> int",
    "ACCESS": "int -> int list -> int",
    "MINIMUM": "int list -> int",
    "MAXIMUM": "int list -> int",
    "LENGTH": "int list -> int",
    "COUNT[<0]": "int list -> int",
    "COUNT[>0]": "int list -> int",
    "COUNT[EVEN]": "int list -> int",
    "COUNT[ODD]": "int list -> int",
    "SUM": "int list -> int",
    "TAKE": "int -> int list -> int list",
    "DROP": "int -> int list -> int list", # drop n elements
    "SORT": "int list -> int list",
    "REVERSE": "int list -> int list",
    "FILTER[<0]": "int list -> int list",
    "FILTER[>0]": "int list -> int list",
```

```
    "FILTER[EVEN]": "int list -> int list",
    "FILTER[ODD]": "int list -> int list",
    "MAP[+1]": "int list -> int list",
    "MAP[-1]": "int list -> int list",
    "MAP[*2]": "int list -> int list",
    "MAP[/2]": "int list -> int list",
    "MAP[*-1]": "int list -> int list",
    "MAP[**2]": "int list -> int list",
    "MAP[*3]": "int list -> int list",
    "MAP[/3]": "int list -> int list",
    "MAP[*4]": "int list -> int list",
    "MAP[/4]": "int list -> int list",
    "ZIPWITH[+]": "int list -> int list -> int list", # zip with
                                        combine two lists, by applying
                                        the operator to each element
    "ZIPWITH[-]": "int list -> int list -> int list",
    "ZIPWITH[*]": "int list -> int list -> int list",
    "ZIPWITH[min]": "int list -> int list -> int list",
    "ZIPWITH[max]": "int list -> int list -> int list",
    "SCAN1L[+]": "int list -> int list", # does the same as zip with
                                        but both arguments are the same
                                         list
    "SCAN1L[-]": "int list -> int list",
    "SCAN1L[*]": "int list -> int list",
    "SCAN1L[min]": "int list -> int list",
    "SCAN1L[max]": "int list -> int list"
}
```

3.B DreamCoder or List Manipulation

DreamCoder [EWN+21] contains many DSL, but we will mainly be targeting the list manipulation DSL, it is very generic and contain polymorphic functions which are supported by their framework and is supported by all of our tools. The polymorphic types are denoted with a ', here there is only two: 'a and 'b. It is usually a harder DSL than DeepCoder usually because a lot of relevant tasks need the introduction of lambda expressions to be solved. It contains the following 32 primitives including 6 constants:

```
primitives = {
    "cons": "'a -> 'a list -> 'a list",
    "car": "'a list -> 'a",
    "cdr": "'a list -> 'a list",
    "empty?": "'a list -> bool",
    "max": "int -> int -> int",
    "min": "int -> int -> int",
    "gt?": "int -> int -> bool",
    "le?": "int -> int -> bool",
    "not": "bool -> bool",
    "if": "bool -> 'a -> 'a -> 'a",
    "eq?": "int -> int -> bool",
    "*": "int -> int -> int",
    "+": "int -> int -> int",
    "-": "int -> int -> int",
    "length": "'a list -> int",
    "range": "int -> int list",
    "map": "('a -> 'b) -> 'a list -> 'b list",
```

```
    "iter": "int -> ('a -> 'a) -> 'a -> 'a",
    "append": "'a -> 'a list -> 'a list",
    "unfold": "'a -> ('a -> bool) -> ('a -> 'b) -> ('a -> 'a) -> 'b
                                        list",
    "index": "int -> 'a list -> 'a",
    "fold": "'a list -> 'b -> ('a -> 'b -> 'b) -> 'b",
    "is-mod": "int -> int -> bool",
    "mod": "int -> int -> int",
    "is-prime": "int -> bool",
    "is-square": "int -> bool",
    "filter": "('a -> bool) -> 'a list -> 'a list",
    "0": "int",
    "1": "int",
    "2": "int",
    "3": "int",
    "4": "int",
    "5": "int",
}
```

3.C String Transformations

This *string transformation* DSL is adapted from SyGus [ABJ+13], enables string and int manipulation, it also uses polymorphic types but to a lesser degree. Indeed, `'a[string|int]` indicates a polymorphic type that can only be instantiated into a `string` or an `int`. It contains the following primitives:

```
primitives = {
    "concat": "string -> string -> string",
    "replace": "string -> string -> string -> string",
    "substr": "string -> int -> int -> string",
    "ite": "bool -> 'a[string|int] -> 'a[string|int] -> 'a[string|int]"
                                        ,
    "int2str": "int -> str",
    "at": "string -> int -> string",
    "lower": "string -> string",
    "upper": "string -> string",
    "str2int": "string -> int",
    "+": "int -> int -> int",
    "-": "int -> int -> int",
    "len": "string -> int",
    "indexof": "string -> string -> int -> int",
    "firstindexof": "string -> string -> int",
    "*": "int -> int -> int",
    "%": "int -> int -> int",
    "=": "'a[string|int] -> 'a[string|int] -> bool",
    "contains": "string -> string -> bool",
    "prefixof": "string -> string -> bool",
    "suffixof": "string -> string -> bool",
    ">": "int -> int -> bool",
    "<": "int -> int -> bool",
}
```

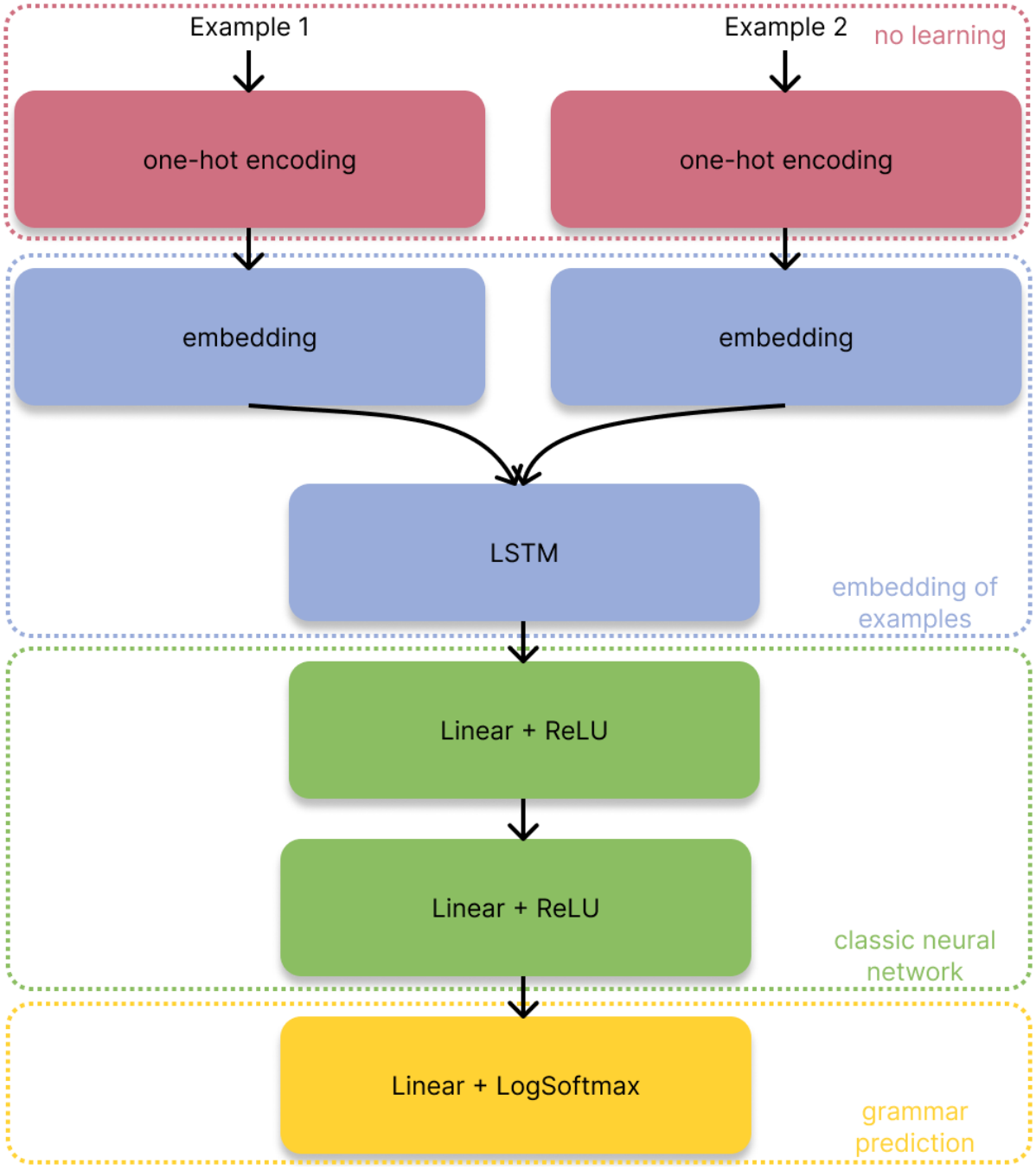


Figure 2.4: Architecture of our prediction model.

Chapter 3

Cost Guided Search

TL;DR 3

This chapter describes enumeration algorithms, we show there has been tremendous progress in enumeration algorithms in the last few years. Algorithms that enumerate programs in order of non-decreasing cost are called best-first search algorithms. We first introduced Heap Search the first bottom-up best-first search algorithm with logarithmic delay. The delay is the complexity between enumerating the i^{th} program and the next, it is a major indicator of the performance of the algorithm. Then Bee Search was proposed by others, it also had logarithmic delay, introducing new insights and became the new state of the art. We investigate whether **there exist best-first search algorithms with constant delay**, also called "no-delay". We answer this question positively by constructing the first no-delay best-first search algorithm called Eco Search and demonstrate the effectiveness of Eco Search in two classical domains.

This chapter focuses on the search part of the cost-guided search also called best-first search. That is it is assumed that a probabilistic grammar as been obtained (see Chapter 2 Section 2.A for how to).

Best-first search algorithms explore the space in the exact order induced by the cost function: this significantly reduces the portion of the program space to be explored. Since the A^* algorithm [LHAN18], several best-first search algorithms have been constructed [SBS22b, EWN+21, FLM+22, AL23].

The major issue of best-first search algorithms is that they *slow down over time*. This is because in order to ensure optimality they need to consider a growing frontier of potentially next-to-be-generated programs in their data structures, which quickly become enormous. The notion of *delay* captures this behaviour: it quantifies the amount of compute required between outputting two programs. The first best-first search algorithm had logarithmic delay [LHAN18].

We will first discuss enumerative approaches describing in chronological order Heap Search, Bee Search and Eco Search. The content of this chapter is close to our published work [FLM+22] and a submitted work on Eco Search.

Our contributions:

- A theoretical framework called distribution-based search for evaluating and comparing search algorithms in the context of cost guided search.

- Heap Search the first bottom-up best-first search algorithm with logarithmic delay.

- Eco Search the first no-delay best-first search algorithm and it is much more performant than current alternatives.

- Comprehensive experiments showcasing their performance and their scaling laws.

In the following we will be talking about probabilistic grammars instead of cost grammars but the same arguments apply.

To make our pseudocode as readable as possible we use the generator syntax of Python. In particular, the **yield** statement is used to return an element (a program in our case) and continue the execution of the code.

The PCFG obtained through the probability measure defines a probabilistic distribution $\mathcal{D}$ over programs. We make the theoretical assumption that the program we are looking for is actually sampled from $\mathcal{D}$, and construct algorithms searching through programs which find programs sampled from $\mathcal{D}$ as quickly as possible. Formally, the goal is to minimise the expected number of programs the algorithm outputs before finding the right program.

We write $A(n)$ for the n^{th} program chosen by the algorithm A; since A may be a randomised algorithm $A(n)$ is a random variable. The performance $(A, \mathcal{D})$ of the algorithm A, which we call its loss, is the expected number of tries it makes before finding x:

$$(A, \mathcal{D}) = \mathbb{E}_{x \sim \mathcal{D}} \left[\inf\{n \in \mathcal{N} : A(n) = x\} \right].$$

Definition 3.1 ▶ Loss Optimal

An algorithm A^* is *loss optimal* if $(A^*, \mathcal{D}) = \inf_A (A, \mathcal{D})$, the algorithm is also called a best-first search algorithm.

Let us state a simple fact: an algorithm is loss optimal if it generates each program once and in non-decreasing order of costs. Depending on $\mathcal{D}$ constructing an efficient loss optimal algorithm may be challenging, pointing to a trade off between quantity and quality: is it worth outputting a lot of possibly unlikely programs quickly, or rather invest more resources into outputting fewer but more likely programs?

Example 3.2 ▶ Loss Optimality

To illustrate the definitions let us consider the distribution $\mathcal{D}$ over the natural numbers such that $\mathcal{D}(n) = \frac{1}{2^{n+1}}$; it is generated by the following PCFG:

$$S \to^{.5} f(S) \quad ; \quad S \to^{.5} x,$$

when identifying n with the program $f^n(x)$. Let us analyse a few algorithms.

- The algorithm A_1 enumerates in a deterministic fashion the natural numbers starting from 0: $A_1(n) = n$. Then $(A_1, \mathcal{D}) = \sum_{n\geq 0} \frac{n+1}{2^{n+1}} = 2$. This enumeration algorithm A_1 is loss optimal.
- The algorithm A_2 samples the natural numbers using the distribution $\mathcal{D}$. For $n \geq 0$, the value of $\mathbb{E}\left[\inf\{n' : A_2(n') = n\}\right]$ is 2^{n+1}: this is the expectation of the geometric distribution with parameter $\frac{1}{2^{n+1}}$. Then $(A_2, \mathcal{D}) = \sum_{n\geq 0} \frac{2^{n+1}}{2^{n+1}} = \infty$. Hence the naive sampling algorithm using $\mathcal{D}$ has infinite loss.
- The algorithm A_3 samples the natural numbers using a distribution that we call $\sqrt{\mathcal{D}}$ defined by $\sqrt{\mathcal{D}}(n) = \frac{1}{1+\sqrt{2}} \frac{1}{2^{\frac{n+1}{2}}}$. For the normalisation factor, note that $\sum_{n\geq 0} \frac{1}{2^{\frac{n+1}{2}}} = 1 + \sqrt{2}$.

  For $n \geq 0$, the value of $\mathbb{E}\left[\inf\{n' : A_3(n') = n\}\right]$ is $(1+\sqrt{2})2^{\frac{n+1}{2}}$: this is the expectation of the geometric distribution with parameter $\frac{1}{1+\sqrt{2}} \frac{1}{2^{\frac{n+1}{2}}}$. Then

$$\begin{aligned}(A_3, \mathcal{D}) &= \sum_{n\geq 0} \frac{(1+\sqrt{2})2^{\frac{n+1}{2}}}{2^{n+1}} \\ &= (1+\sqrt{2}) \sum_{n\geq 0} \frac{1}{2^{\frac{n+1}{2}}} \\ &= (1+\sqrt{2})^2 \approx 5.83.\end{aligned}$$

  As we will prove in a more general statement next chapter (Chapter 4 Theorem 4.1), the algorithm A_3 is loss optimal among sampling algorithms. Surprisingly it is not much worse than the loss optimal algorithm, yet offers many advantages: it is much easier to implement, and requires no memory at all. Last but not least in the case of PCFG it can be implemented using a new probabilistic labelling of the PCFG inducing $\mathcal{D}$.

A number of enumerative methods have been investigated in previous works [MTG+13, BGB+17, LHAN18, ZW18]. They proceed in a top-down fashion, and can be understood as ways of exploring the tree of leftmost derivations of the grammar as illustrated in Figure 3.1.

We will present three bottom-up loss optimal search algorithms. Heap Search was our initial contribution in [FLM+22], then Bee Search was proposed in [AL23] by Levis Leli and Saqib Ameen and then we proposed Eco Search merging the advantages of both Heap Search and Bee Search. An overview is available on Figure 3.2 which gives a very high level overview of the different algorithms. A key element in the design of such enumeration algorithms is how they consider the frontier, that is the set of programs which maybe the next program to enumerate: that is the next closest in terms of cost. This process of choosing from the set of candidates is what is basically responsible for the delay of such algorithms. Ideally the smaller it is, the better, because that means less

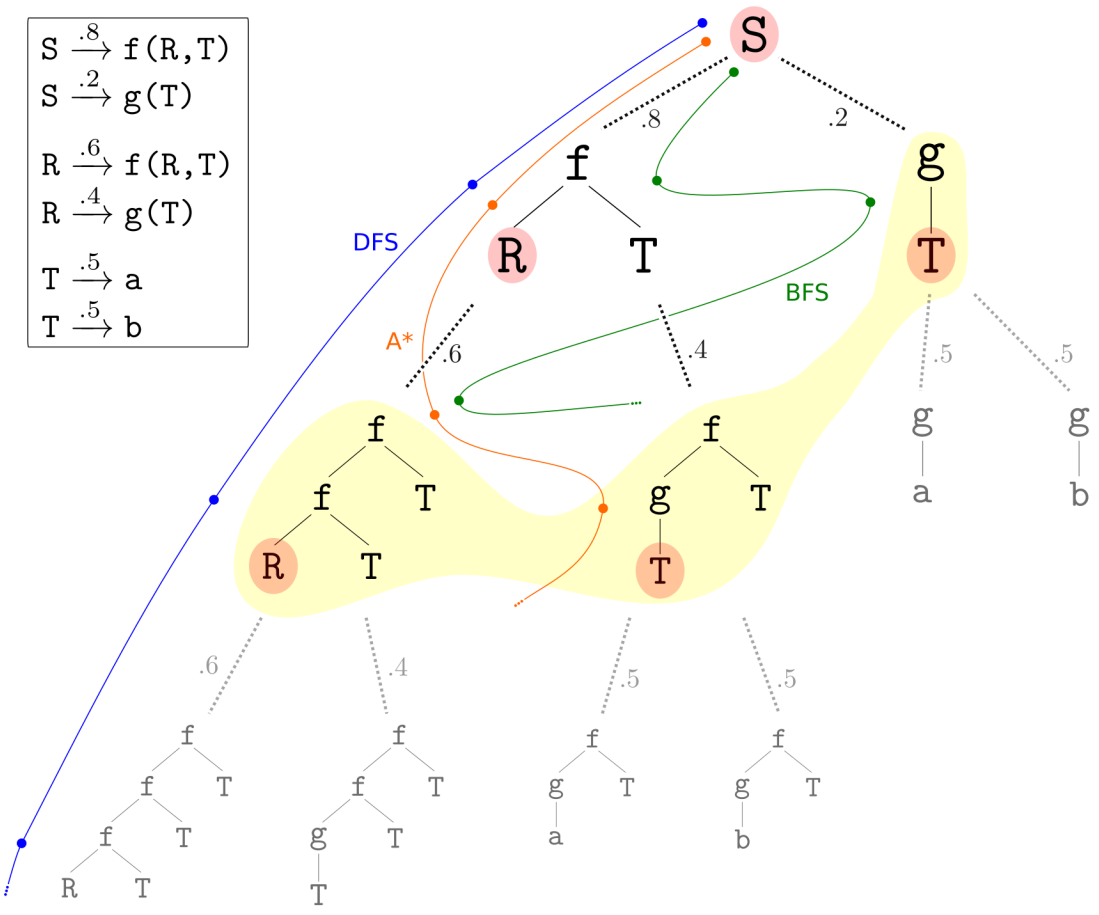
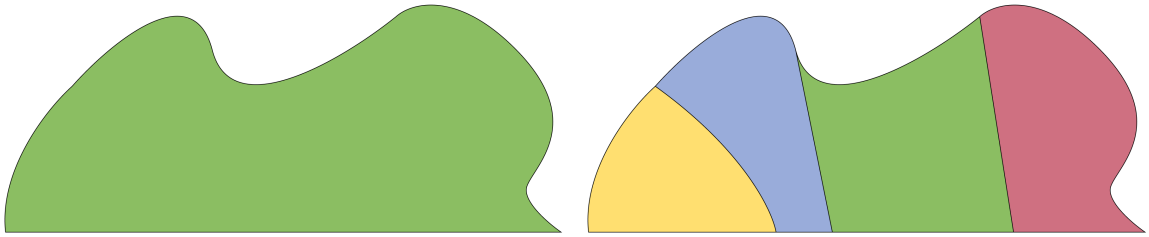
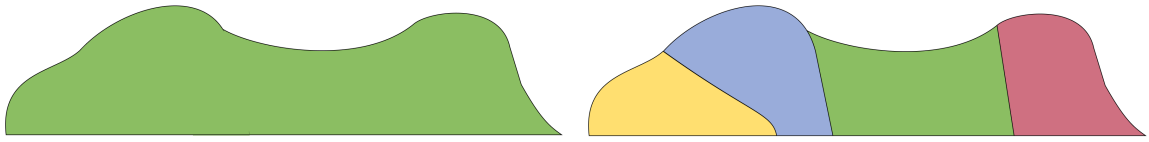


Figure 3.1: Illustration of the tree of leftmost derivations.

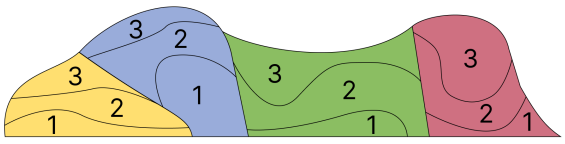
candidates to check. The logarithmic delay is quite natural since this process is close to sorting, because at each step we will add new candidates and get the minimum in terms of cost. Indeed, over n steps we have sorted through a larger set and have found the n smallest elements. This is what we had first with A^*, then with Heap Search in order to speed up the process, the sorting was made separate for each color, they represent different nonterminals of the grammar. While this does not change the complexity this led to a practical speed-up since the sets of candidates are smaller for each nonterminal. Then Bee Search did not distinguish between nonterminals however it did find a way to dramatically reduce the number of candidates while still being correct, again this does not change the delay. In fact splitting again with the nonterminals, which correspond to Eco Search without buckets still give us the natural logarithmic delay. In order to beat this logarithmic delay, we figured that actually you can put these candidates in buckets of candidate programs of equivalent costs. So you can just work with buckets instead of programs. The buckets are denoted by the numbers 1, 2 and 3 on the figure. In practice, there may be millions of buckets, but the number of buckets is a property of the grammar: it is a constant that is fixed as the number of program grows. This is the key insight that will lead us to constant delay with Eco Search.



A^* with delay $O(\log n)$ Heap Search with delay $O(\log n)$



Bee Search with delay $O(\log n)$ Eco Search w/o buckets with delay $O(\log n)$



Eco Search with delay $O(1)$

Figure 3.2: Overview of the frontiers as seen by different enumeration algorithms. n is the number of programs enumerated.

Bottom-up search starts with the smallest programs and iteratively generates larger programs by combining the smaller ones generated by the algorithm. Historically, only A^* from [**?**] was a loss optimal search algorithm.

r_1 :	`str`	$\to$	'Hello'	cost: 1.1
r_2 :	`str`	$\to$	'World'	cost: 2.0
r_3 :	`str`	$\to$	`cast(int)`	cost: 4.4
r_4 :	`str`	$\to$	`concat(str, str)`	cost: 5.3
r_5 :	`int`	$\to$	`var`	cost: 1.8
r_6 :	`int`	$\to$	`1`	cost: 3.3
r_7 :	`int`	$\to$	`add(int, int)`	cost: 5.3

Figure 3.3: The DSL we use as running example (rules are numbered for future references, and costs are explained below).

Let us write $\mathcal{D}$ for the distribution induced by a PCFG. For a program x, we say that x' is the 'successor of x' if it is the most likely program after x, meaning $\mathcal{D}(x) > \mathcal{D}(x')$ and there are no programs x'' such that $\mathcal{D}(x) > \mathcal{D}(x'') > \mathcal{D}(x')$. For a non-terminal T in the grammar, the 'successor of x from T' is the most likely program after x among those generated from T. We define 'predecessor of x' and 'predecessor of x from T' in a similar way.

To make things concrete, let us consider a small example. Our DSL manipulates strings and integers, hence it uses two types: `string` and `int`. It has three primitives:

```
cast: int -> string
concat: string -> string -> string
add: int -> int -> int
```

Let us add some constants `'Hello'`, `'World'`: `string` and `1: int`. We also add a variable `var: int`. The class of programs of type `int -> string` is generated by the CFG given in Figure 3.3, which uses two non-terminals, `string` and `int`, with the former being initial. Here in this grammar `'Hello'` is a the program of minimal cost for the nonterminal `str` then its successor would be `'World'`. For the nonterminal int, the first few programs are in ascending cost order: `var`, `1`, `add(var, var)` and then both `add(1, var)` and `add(var, 1)` have similar cost so any of them can be enumerated first but the other has to be the successor of the first.

For readability in our examples, we will use some abbreviations: $S =$ `string`, $I =$ `int`, $H =$ `'Hello'`, and $W =$ `'World'`.

Pre-generation cost functions As we will see, our algorithms will be able to work with general cost functions. However, in many cases cost functions are of a special nature: they are computed recursively alongside the grammar and induced by defining $\textsc{cost}(r)$ for each derivation rule r (see Figure 3.3 for an example). Note that $\textsc{cost}(r)$ can be any positive real number. Consider a program $\textsc{p} = f(\textsc{p}_1, \ldots, \textsc{p}_k)$ generated by the derivation rule $r : X \to f(X_1, \ldots, X_k)$, then

$$\textsc{cost}(\textsc{p}) = \textsc{cost}(r) + \sum_{i=1}^{k} \textsc{cost}(\textsc{p}_i).$$

What makes pre-generation cost functions special is that they do not depend on executions of the programs, in fact they do not even require holding the whole program in memory since they are naturally computed recursively.

1 Computing programs of minimal costs

As a warm-up, we need a procedure to compute for each non-terminal X a program of minimal cost. Note that this is well defined because costs are positive, and that we do not require to compute all minimal programs, just a single one. The pseudocode is given in Algorithm 2. The algorithm simply propagates the minimal programs and costs found across derivation rules, and repeats the propagation as long as it updates values. A simple analysis shows that the number of iterations of the **while** loop (line 9) is bounded by the number of non-terminals in the grammar, so the algorithm always terminate. In practice the number of iterations is often much smaller.

Algorithm 2 Computing programs of minimal costs

```
1: for X non-terminal do
2:     MinCost(X) ← ∞
3:     MinProg(X) ← ⊥
4: end for
5: for r : X → a derivation rule do
6:     if cost(r) < MinCost(X) then
7:         MinCost(X) ← cost(r)
8:         MinProg(X) ← a
9:     end if
10: end for

11: updated ← True
12: while updated do
13:     updated ← False
14:     for r : X → f(X_1, ..., X_k) derivation rule do
15:         c ← cost(r) + Σ_{i=1}^{k} MinCost(X_i)
16:         if c < MinCost(X) then
17:             MinCost(X) ← c
18:             MinProg(X) ← f(MinProg(X_1), ..., MinProg(X_k))
19:             updated ← True
20:         end if
21:     end for
22: end while
```

2 Heap Search

The Heap Search algorithm was the first bottom-up best-first search algorithm with logarithmic delay. The only previous algorithm was A^* [LHAN18] with logarithmic delay but it was top-down. A key idea behind Heap Search is also to introduce structures for each nonterminal instead of shared structures. Another massive advantage of Heap Search contrarily to others is that it is bottom-up, this allow to leverage this structure to introduce practical optimisations in the evaluation or in the elimination of branches of programs.

The Heap Search algorithm maintains three objects:

- SEEN: stores all programs seen so far. Note that *seen* is not the same as *generated*, as we discuss below.

- for each non-terminal X, HEAP_X is a heap of programs, using as value the costs of the programs. Programs in HEAP_X are *seen* but are yet to be *generated*.

- for each non-terminal X, $\text{SUCCESSORPROGRAM}_X$ stores the successors of programs, that we define now. Concretely, it is a mapping from programs to programs.

Let us explain the difference between *seen* and *generated*. The programs that are yield line 4 of Algorithm 4 are generated. When a program is inserted (using the function INSERT), it is seen. It is sitting in some heap waiting for its turn to be generated.

Let us fix a non-terminal X, and P, P' two programs generated by X. We remember that P' is a successor of P if $\text{COST}(\text{P}) < \text{COST}(\text{P}')$ and there does not exist P'' generated by X such that $\text{COST}(\text{P}) < \text{COST}(\text{P}'') < \text{COST}(\text{P}')$. In other words, P' has minimal cost among programs of higher cost than P generated by X.

The main function is COMPUTESUCCESSOR in Algorithm 4: given a program P generated by X, it computes a successor of P. It works as follows: either a successor was already computed (therefore stored in $\text{SUCCESSORPROGRAM}_X$), in which case it is simply returned, or it was not. In the second case, as we will argue, the minimal element of HEAP_X is a successor, so we return it, let us call it P'. The goal of the lines 10–16 is to update the data structures, adding potential successors of P'. What the invariant of the algorithm shows is that the successor of P' falls in one of two categories:

- it is already in HEAP_X,

- it is obtained from P' by replacing one of its argument by its successor (for the corresponding non-terminal).

Algorithm 3 Heap Search: initialisation

compute $\textsc{MinProg}(X)$ a program of minimal cost from X for each non-terminal X
$\textsc{Seen}$: set of programs
for X non-terminal **do**
    $\textsc{Heap}_X$: heap of programs
    $\textsc{SuccessorProgram}_X$: mapping from programs to programs
end for

function $\textsc{insert}(\textsc{p}, X)$:
    add $\textsc{p}$ to $\textsc{Heap}_X$ with value $\textsc{cost}(\textsc{p})$
    add $\textsc{p}$ to $\textsc{Seen}$
end function

for $r : X \to f(X_1, \ldots, X_k)$ derivation rule **do**
    $\textsc{p} \leftarrow f(\textsc{MinProg}(X_1), \ldots, \textsc{MinProg}(X_k))$
    $\textsc{insert}(\textsc{p}, X)$
end for

for X non-terminal **do**
    $\textsc{ComputeSuccessor}(\bot, X)$ ▷ $\bot$ is a dummy program
end for

Algorithm 4 Heap Search: main loop

$\textsc{p} \leftarrow \bot$ ▷ $\bot$ is a dummy programme
while True **do**
    $\textsc{p} \leftarrow \textsc{ComputeSuccessor}(\textsc{p}, S)$ ▷ S is the initial non-terminal
    yield $\textsc{p}$
end while

function $\textsc{ComputeSuccessor}(\textsc{p}, X)$:
    if $\textsc{SuccessorProgram}_X(\textsc{p})$ is defined **then**
        return $\textsc{SuccessorProgram}_X(\textsc{p})$
    else
        $\textsc{p}' \leftarrow \textsc{pop}(\textsc{Heap}_X)$
        $\textsc{SuccessorProgram}_X(\textsc{p}) \leftarrow \textsc{p}'$
        $\textsc{p}' = f(\textsc{p}_1, \ldots, \textsc{p}_k)$ ▷ $\textsc{p}'$ is generated by $X \to f(X_1, \ldots, X_k)$
        for i from 1 to k **do**
            $\textsc{p}'_i \leftarrow \textsc{ComputeSuccessor}(\textsc{p}_i, X_i)$
            $\textsc{p}''_i \leftarrow f(\textsc{p}_1, \ldots, \textsc{p}'_i, \ldots, \textsc{p}_k)$
            if $\textsc{p}''_i$ not in $\textsc{Seen}$ **then**
                $\textsc{insert}(\textsc{p}''_i, X)$
            end if
        end for
        return $\textsc{p}'$
    end if
end function

Example 3.3 ▶ Heap Search example

We consider the grammar and associated costs defined in Figure 3.3. In a single iteration, Algorithm 2 finds $\textsc{MinProg}(S) = H, \textsc{MinCost}(S) = 1.1$ and $\textsc{MinProg}(I) = \text{var}, \textsc{MinCost}(I) = 1.8$. During initialisation, we perform insertions of the following programs:

$$H, W, \text{concat}(H, H), \text{cast}(\text{var}), \text{var}, 1, \text{add}(\text{var}, \text{var}),$$

and then run $\textsc{ComputeSuccessor}(\bot, S)$ and $\textsc{ComputeSuccessor}(\bot, I)$. At this point, the data structures are as follows, with costs indicated below programs:

$$\textsc{SuccessorProgram}_S(\bot) = H, \ \textsc{SuccessorProgram}_I(\bot) = \text{var}$$

$$\begin{aligned}
\textsc{Heap}_S &= \{\underbrace{W}_{2.0}, \underbrace{\text{cast}(\text{var})}_{6.2}, \underbrace{\text{concat}(H, H)}_{7.5}\} \\
\textsc{Heap}_I &= \{\underbrace{1}_{3.3}, \underbrace{\text{add}(\text{var}, \text{var})}_{8.9}\} \\
\textsc{Seen} &= \{H, W, \text{concat}(H, H), \text{cast}(\text{var}), \text{var}, 1, \text{add}(\text{var}, \text{var})\}
\end{aligned}$$

Let us analyse the first four calls:

1. $\textsc{ComputeSuccessor}(\bot, S)$ returns H, already computed during initialisation.

2. $\textsc{ComputeSuccessor}(H, S)$: we pop W from $\textsc{Heap}_S$, set $\textsc{SuccessorProgram}_S(H) = W$, and return W.

3. $\textsc{ComputeSuccessor}(W, I)$: we pop cast(var) from $\textsc{Heap}_I$, let us call it $\textsc{p}'$ and set $\textsc{SuccessorProgram}_I(W) = \textsc{p}'$. Before returning $\textsc{p}'$, we need to update the data structures, lines 12 to 16. We run $\textsc{ComputeSuccessor}(\text{var}, I)$, which pops 1 from $\textsc{Heap}_I$, sets $\textsc{SuccessorProgram}_I(\text{var}) = 1$, and returns 1. We consider cast(1), currently not in $\textsc{Seen}$, so it is inserted. After this update the heaps are as follows:

$$\begin{aligned}
\textsc{Heap}_S &= \{\underbrace{\text{concat}(H, H)}_{7.5}, \underbrace{\text{cast}(1)}_{7.7}\} \\
\textsc{Heap}_I &= \{\underbrace{\text{add}(\text{var}, \text{var})}_{8.9}\}
\end{aligned}$$

4. $\textsc{ComputeSuccessor}(\text{cast}(\text{var}), S)$: we pop concat$(H, H)$ from $\textsc{Heap}_S$, let us call it $\textsc{p}'$ and set $\textsc{SuccessorProgram}_S(\text{cast}(\text{var})) = \textsc{p}'$. Before returning $\textsc{p}'$, we need to update the data structures, lines 12 to 16. We run $\textsc{ComputeSuccessor}(H, I)$, which itself calls $\textsc{ComputeSuccessor}(\text{var}, S)$. The latter returns 1, and the former add(var, var), after inserting add$(1, \text{var})$ and add$(\text{var}, 1)$ (to $\textsc{Heap}_I$ and $\textsc{Seen}$). We consider concat(add(var, var), H) and concat$(H, \text{add}(\text{var}, \text{var}))$, and insert them both.

Theoretical Aspects The following lemma gives the correctness of the Heap Search algorithm.

Lemma 3.4 ▶ Heap Search is loss optimal

For any non-terminal X and program P' generated from X, if we have already run $\textsc{ComputeSuccessor}(\mathrm{P}', X)$ for any program P' preceding P (among those generated from X) then $\textsc{ComputeSuccessor}(\mathrm{P}, X)$ outputs the successor of P from X

Proof. Suppose the lemma is not true and consider the first time where it fails, on say, input (P, X). At line 7, (P, X) was not computed before and therefore the algorithm branches out to line 9. Saying that the algorithm fails is equivalent to say that line 10 fails, meaning that $\mathrm{P}' = \mathit{pop}(\textsc{Heap}_X)$ is not the successor of *program* from X. Let us denote by y the correct successor. There are two cases which can lead the algorithm the failure:

- **First case:** P' is a program with higher probability than y. In this case, it means that P' was already the output of a previous query to $\textsc{ComputeSuccessor}$ (because (P, X) is the first query for which the algorithm fails, so programs with higher probability than y have already been output before). Thus the program P' was pop at least two times from $\textsc{Heap}_X$, one time when P' was the correct output of a query, and another time when the program failed: this is not possible since we push programs only once into the heaps thanks to the condition at line 16.
- **Second case:** P' has a smaller probability than the probability of y. In this case, it means that y was not in $\textsc{Heap}_X$ when the algorithm performed a pop at line 10 (otherwise y would have pop since y has higher probability than P'). Suppose $y = f(\mathrm{P}_1, \ldots, \mathrm{P}_k)$, generated by the derivation rule $X \to f(X_1, \ldots, X_k)$. If all P_i are maximal (meaning that they don't have a predecessor from X_i) then $f(\mathrm{P}_1, \ldots, \mathrm{P}_k)$ is pushed in $\textsc{Heap}_X$ during the initialization procedure (line 13 of Algorithm 3) so the algorithm cannot fail because of this case. Therefore there is at least one P_i, say w.l.o.g P_1, which has a predecessor P'_1 from X_1. Consider the program $f(\mathrm{P}'_1, \mathrm{P}_2, \ldots, \mathrm{P}_k)$; this program has higher probability than y and therefore has been seen before. Thus $f(\textsc{ComputeSuccessor}(\mathrm{P}'_1, X_1), \mathrm{P}_2, \ldots, \mathrm{P}_k)$ was added to $\textsc{Heap}_X$ because of line 17. To conclude, observe that $f(\textsc{ComputeSuccessor}(\mathrm{P}'_1, X_1), \mathrm{P}_2, \ldots, \mathrm{P}_k) = f(\mathrm{P}_1, \mathrm{P}_2, \ldots, \mathrm{P}_k)$ so y has previously been added to $\textsc{Heap}_X$.

□

Lemma 3.5 ▶ Heap Search has delay $O(\log i)$ for computing the i^{th} the program from the $i-1^{\text{th}}$ program.

Fix any non-terminal X. Suppose that we have already generated the first i programs generated from X (meaning that if P is the j-th program for $j < i$, then P is a key of the hash table $\textsc{SuccessorProgram}_X$ and $\textsc{SuccessorProgram}_X(x)$ is the $(j+1)$-th program generated from X). Then computing the successor of the i-th program has a running time of $O(\log i)$.

Proof. First observe that ComputeSuccessor can call recursively several others successor computations. However, for any non-terminal symbol X there is at most one call of the form $ComputeSuccessor(\text{P}, X)$ which leads the algorithm to branch out to line 9 and thus to possibly rise other queries. Indeed, this case happens only when the successor of P has not been computed yet (otherwise the query stops at line 8); this can happen for at most one program for any fixed symbol X: the last program from X already seen in any execution of the algorithm. Forgetting about recursive queries, the running time of a query going through line 9 is given by the pop and push operations (line 10 and 17). The number of pops and pushs is at most $m+1$ where m is the maximal arity of a function in the grammar. Moreover, each such operation costs a running time of $O(\log |\text{Heap}_T|)$, so the total time for the query is $O(\log |\text{Heap}_T|)$. Overall, the total running time is bounded by

$$\sum_X O(\log |\text{Heap}_X|).$$

To conclude, observe that when we query the successor of the i-th program generated from X, the size of Heap_X is bounded by $m \cdot i$ since we push at most m programs during any ComputeSuccessor not already computed before. □

Limitations of Heap Search

There are two limitations of Heap Search:

- The first is the structure of recursive calls when updating the data structures, which insert a lot of programs. More precisely, the issue is that these programs are added to the data structures although they are not generated yet, because they may have much larger costs. In other words, when generating a program of cost c, Heap Search needs to consider many programs that have costs potentially much larger than c. This makes the algorithm very memory hungry.

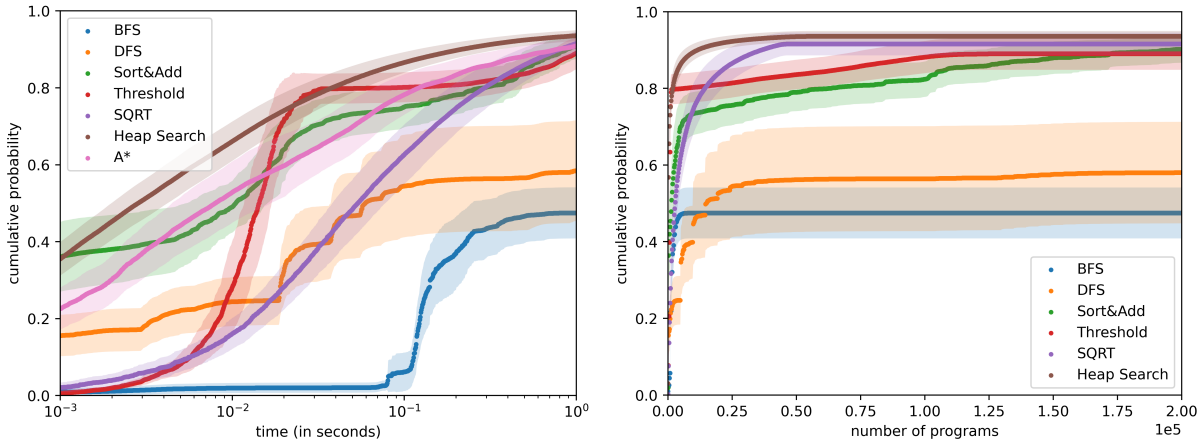
- The second is that it needs to explicit build all the programs it considers, again very heavy on memory consumption.

3 Preliminary Experiments

We report results on DSLs from DeepCoder [BGB+17] and DreamCoder [EWN+21]. Both target the classic program synthesis challenge of integer list processing programs, but with different properties. DeepCoder's DSL is larger and more specialized, with around 40 high-level primitives, and does not use polymorphic types, while DreamCoder's is smaller and more generic, with basic functional programming primitives such as map, fold, unfold, car, cons, and cdr, etc., for a total of around 20 primitives. Both DSLs are compiled into finite depth CFGs with minimal syntactic constraints generating programs of depth up to 6.

The search algorithms under consideration are:

- Threshold from [MTG+13]: iterative-deepening-search, where the threshold that is iteratively deepened is a bound on program description length (i.e. negative log probability),



(a) Cumulative probability against time in log-scale

(b) Cumulative probability against number of programs output in log-scale

Figure 3.4: Comparing all search algorithms on random PCFGs

- Sort and add from [BGB+17]: an inner loop of depth-first-search, with an outer loop that sorts productions by probability and runs depth-first-search with the top k productions for increasing values of k,
- A^* from [LHAN18]: best-first-search on the graph of (log probabilities of) tree derivations,
- Beam search from [ZRF+18]: breadth-first-search with bounded width that is iteratively increased.

As well as our new algorithms: Heap Search and SQRT Sampling (which we will discuss later in Chapter 4).

All algorithms have been reimplemented and optimised in the codebase to provide a fair and uniform comparison.

Random PCFGs

In this first set of experiments we run all search algorithms on random PCFGs until a timeout, and compare the number of programs they output and the cumulative probability of all programs output.

To obtain random PCFGs from the CFGs we sample a probabilistic labeling with an exponential decrease (this is justified by the fact that machine-learned PCFGs feature exponential decrease in transition probabilities). In this experiment the initialization cost of each algorithm is ignored. The results presented here are averaged over 50 samples of random PCFGs, the solid lines represent the average and a lighter color indicates the standard deviation.

Figure 3.4 shows the results for all algorithms in a non-parallel implementation. On the lhs we see that Heap Search (almost always) has the highest cumulative probability against both time and number of distinct programs. Note that since A^* and Heap Search enumerate the same programs in the same order they produce the same curve in the right hand side of Figure 3.4 so we did not include A^*.

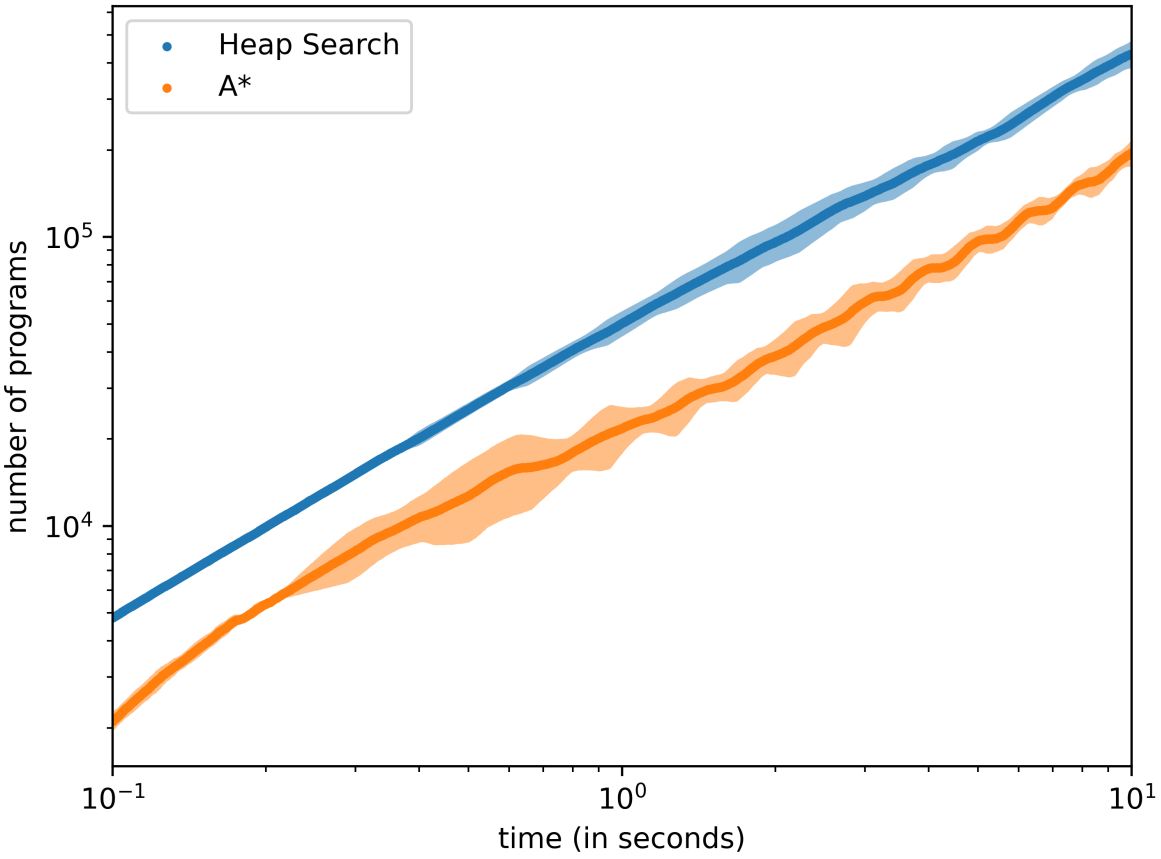


Figure 3.5: Comparing Heap Search and A^*

To compare A^* and Heap Search we refer to Figure 3.5, showing that Heap Search generates 2.35 times more programs than A^*, consistently over time. The larger variations for A^* are due to the manipulation of a single heap of growing size, requiring frequent memory reallocations. The difference in performance can be explained by the fact that A^* uses a single heap for storing past computations, while Heap Search distributes this information in a family of connected heaps and hash tables.

Machine-learned PCFGs

In this second set of experiments we consider the benchmark suites of hand-picked problems and sets of I/O. We extracted 218 problems from DreamCoder's dataset [EWN+21].

We train a neural network to make predictions from a set of I/O. Our neural network is composed of a one layer GRU [CvMG+14] and a 3-layer MLP with sigmoid activation functions, and trained on synthetic data generated from the DSL. The details of the architecture and the training can be found in Section 2.A. Our network architecture induces some restrictions, for instance the types of the programs must be `int list -> int list`; we removed tasks that did not fit our restrictions and obtained a filtered dataset of 137 tasks. For each task we run every search algorithm on the PCFG induced by the neural predictions with a timeout of 100s and a maximum of $1M$ programs. Unlike in the previous experiments the initialization costs of algorithms are not ignored.

Figure 4.1 shows the number of tasks solved within a time budget. Heap Search solves the largest number of tasks for any time budget, and in particular 97 tasks out

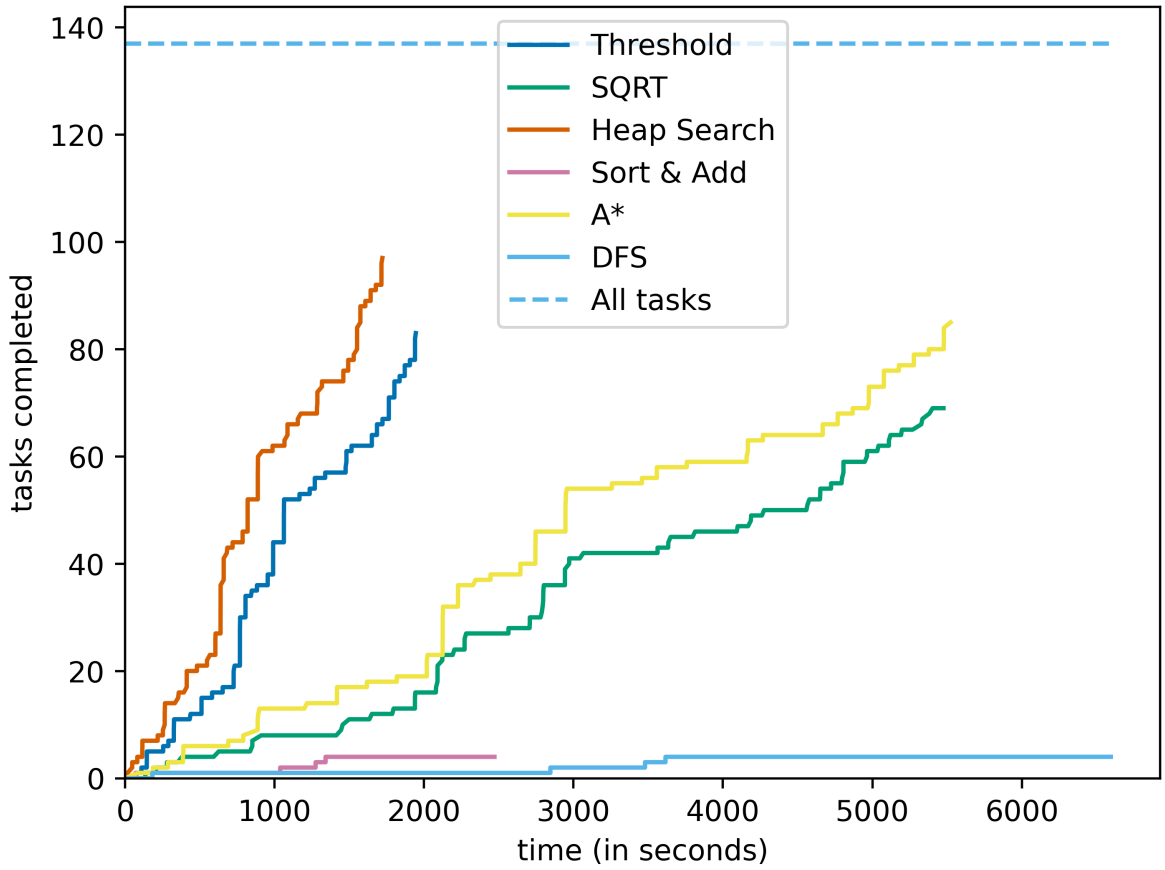


Figure 3.6: Comparing all search algorithms on the DreamCoder reduced dataset with machine-learned PCFGs

of 137 before timeout. The comparison between Threshold and A^* is insightful: A^* solves a bit more tasks than Threshold (85 vs 83) but in twice the time.

Table 3.1 shows for each algorithm how many programs were generated per second. Recall that Heap Search and A^* generate the same programs in the same order. Overall in these experiments Heap Search is 6 times faster than A^*.

Since Heap Search follows a bottom-up approach we save on program evaluations in two ways: partial programs are evaluated along the search, and the results are cached. On the other hand A^* is a top-down enumeration method so every new program has to be evaluated from scratch.

Finally, we want to see how Heap Search improves with the quality of the pre-

Table 3.1: Number of programs generated

Algorithm	Number of programs generated
Heap Search	38735 prog/s
Threshold	25381 prog/s
DFS	20281 prog/s
SQRT Sampling	14020 prog/s
A^*	6071 prog/s

dictions. According to the properties of Heap Search, we expect that the better the predictions the faster tasks are solved since Heap Search is loss optimal. In order to show this, we ran Heap Search on the reduced DreamCoder dataset with a timeout of 5 minutes and kept the tasks where a solution was found since some tasks cannot be solved with our DSL. Then we trained a neural network on this new reduced set with the solutions found. At different epochs of the training we checked how fast and how many tasks Heap Search could solve with the predictions of the network with a timeout of 30 seconds and a limit of 1M programs. We plot on Figure 3.7 the number of tasks solved with respect to total time used by Heap Search after different number of training epochs with the uniform PCFG as a baseline. We clearly observe the expected outcome, as the number of training epochs grows and the neural networks learn to better predict the solutions, Heap Search dramatically decreases the time required to solve the tasks. While this may be true for every algorithm, loss optimal algorithms like Heap Search benefit the most from it.

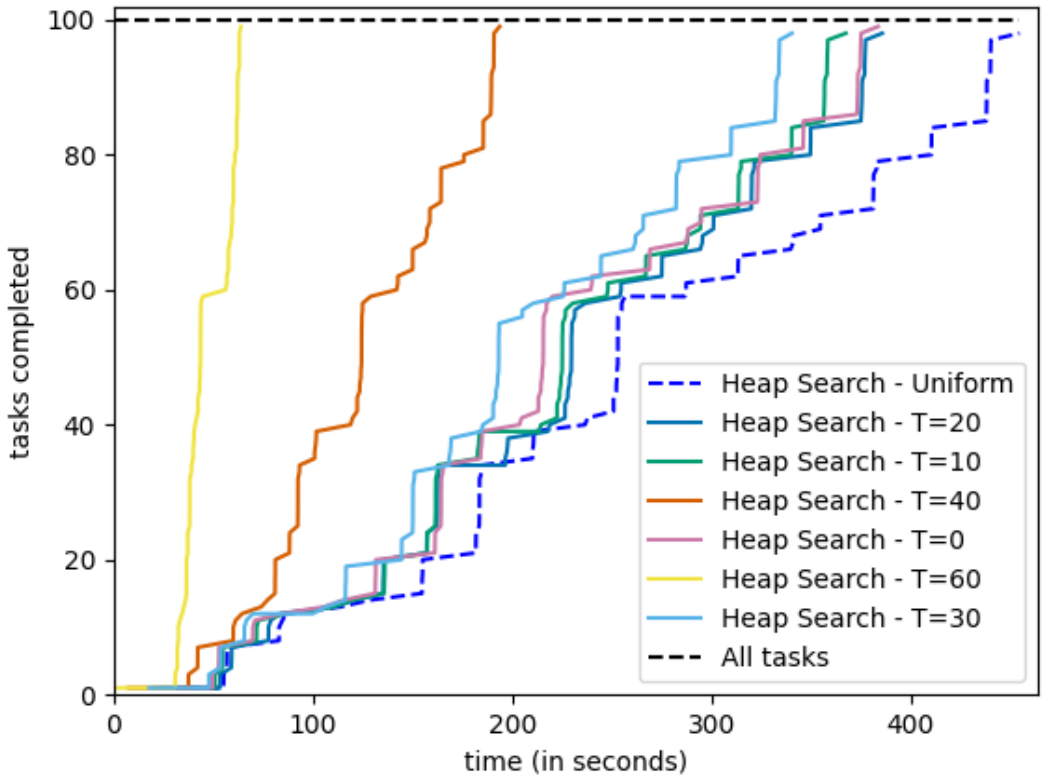


Figure 3.7: Evolution of time and tasks solved by Heap Search at different epochs of the training of a machine-learning PCFG predictor on a reduced DreamCoder dataset

4 Bee Search

Bee Search [AL23] is the work of Saqib Ameen and Levis Leli. It also has logarithmic delay and is also bottom-up, its structure offers better advantages than Heap Search for techniques such as observational equivalence. The key insight behind Bee Search

is the cost tuple representation which we will explain later. To get a complete picture of Eco Search, our no-delay best-first search algorithm, it is relevant to understand Bee Search. The algorithm maintains three objects:

- Generated: stores the set of programs generated so far, organised by costs. Concretely, it is a mapping from costs to sets of programs: Generated$[c]$ is the set of generated programs of cost c.
- Index2Cost: a list of the costs of the generated programs. Let us write Index2Cost $= [c_1, \dots, c_\ell]$, then $c_1 < \cdots < c_\ell$ and Generated$[c_i]$ is defined.
- Queue: stores information about which programs to generate next. Concretely, it is a priority queue of *cost tuples* ordered by costs, that we define now.

Definition 3.6 ▶ Cost Tuple

A cost tuple is a pair consisting of a derivation rule $r : X \to f(X_1, \dots, X_k)$ and a tuple $n = (n_1, \dots, n_k) \in \mathbb{N}^k$. For derivation rules $r : X \to a$, cost tuples are of the form $(r, \emptyset)$. A cost tuple represents a set of programs: (r, n) represents all programs generated by the rule r where the i^{th} argument is any program in Generated[Index2Cost$[n_i]$]. The cost of a cost tuple $t = (r, n)$ is defined as

$$\text{cost}(t) = \text{cost}(r) + \sum_{i=1}^{k} \text{Index2Cost}[n_i].$$

Cost tuples are an efficient way of representing sets of programs. A *cost tuple* is a pair consisting of a derivation rule $r : X \to f(X_1, \dots, X_k)$ and a tuple $n = (n_1, \dots, n_k) \in \mathbb{N}^k$. For derivation rules $r : X \to a$, cost tuples are of the form $(r, \emptyset)$. A cost tuple represents a set of programs: (r, n) represents all programs generated by the rule r where the i^{th} argument is any program in Generated[Index2Cost$[n_i]$]. The cost of a cost tuple $t = (r, n)$ is defined as

$$\text{cost}(t) = \text{cost}(r) + \sum_{i=1}^{k} \text{Index2Cost}[n_i].$$

The main function of Bee Search is output: a single call to output generates **all** programs represented by the cost tuple t found by popping Queue. Let us consider the case of a cost tuple $t = (r : X \to f(X_1, \dots, X_k), n : (n_1, \dots, n_k))$, the pseudocode addressing this case is given in Algorithm 5. The idea is to fetch all programs P_i in Generated[Index2Cost$[n_i]$] and to form the programs $f(\text{P}_1, \dots, \text{P}_k)$. The issue is here that not all such programs are derived from the grammar: we additionally need to check whether each P_i was generated from X_i so we can apply the rule $r : X \to f(X_1, \dots, X_k)$. This means that many programs are discarded at this step, and it may even happen that no program is generated by a call to output.

The original authors prove that Bee Search is also loss optimal.

Algorithm 5 The generation part of Bee Search

```
1: function OUTPUT():
2:     (skip part of the code)
3:     t = (r : X → f(X_1, ..., X_k), n : (n_1, ..., n_k))
4:     for P_1, ..., P_k in ∏_{i=1}^{k} GENERATED[INDEX2COST[n_i]] do
5:         if for all i ∈ {1, ..., k}, P_i is generated by X_i then
6:             P ← f(P_1, ..., P_k)
7:             add P to GENERATED[c]
8:             yield P
9:         end if
10:    end for
11: end function
```

Algorithm 6 Bee Search: initialisation

```
1: GENERATED: mapping from costs to sets of programs
2: INDEX2COST: list of costs
3: QUEUE: priority queue of cost tuples ordered by costs

4: c ← minimal cost of a program
5: add c to INDEX2COST

6: for r : X → f(X_1, ..., X_k) derivation rule do
7:     t ← (r, (0, ..., 0))   [k times]
8:     COST(t) ← COST(r) + k × c
9:     add t to QUEUE with value COST(t)
10: end for
```

Algorithm 7 Bee Search: main loop

```
1: while True do
2:     OUTPUT()
3: end while

4: function OUTPUT():
5:     t ← POP(QUEUE)                          ▷ generates all programs represented by t
6:     if t = (r : X → a, ∅) then
7:         c ← COST(r)                                              ▷ computes COST(t)
8:         if c ≠ INDEX2COST[−1] then
9:             add c to INDEX2COST    ▷ the last generated program did not have cost c
10:            GENERATED[c] ← ∅
11:        end if
12:        add a to GENERATED[c]
13:        return a

14:    else t = (r : X → f(X_1, ..., X_k), n : (n_1, ..., n_k))
15:        c ← COST(r) + Σ_{i=1}^{k} INDEX2COST[n_i]                    ▷ computes COST(t)
16:        if c ≠ INDEX2COST[−1] then
17:            add c to INDEX2COST    ▷ the last generated program did not have cost c
18:            GENERATED[c] ← ∅
19:        end if
20:        for P_1, ..., P_k in ∏_{i=1}^{k} GENERATED[INDEX2COST[n_i]] do    ▷ generates
   programs
21:            if for all i ∈ {1, ..., k}, P_i is generated by X_i then
22:                P ← f(P_1, ..., P_k)
23:                add P to GENERATED[c]
24:                yield P
25:            end if
26:        end for
27:        for i from 1 to k do                          ▷ updates the data structure
28:            n' ← n
29:            n'_i ← n_i + 1
30:            if t' = (r, n') not in QUEUE then
31:                COST(t') ← c + INDEX2COST[n'_i] − INDEX2COST[n_i]    ▷ efficient
   computation of COST(t')
32:                add t' to QUEUE with value COST(t')
33:            end if
34:        end for
35:    end if
36: end function
```

Example 3.7 ▶ Bee Search example

We consider the grammar and associated costs defined in Figure 3.3. The minimal cost of a program is 1.1, so we set INDEX2COST = $\{1.1\}$. During initialisation, we add the following cost tuples:

$$(r_1, \emptyset), (r_2, \emptyset), (r_3, (0)), (r_4, (0, 0)), (r_5, \emptyset), (r_6, \emptyset), (r_7, (0, 0)).$$

At this point, the queue is as follows, with costs indicated below cost tuples:

$$\text{QUEUE} = \{\underbrace{(r_1, \emptyset)}_{1.1}, \underbrace{(r_5, \emptyset)}_{1.8}, \underbrace{(r_2, \emptyset)}_{2.0}, \underbrace{(r_6, \emptyset)}_{3.3}, \underbrace{(r_3, (0))}_{5.5}, \underbrace{(r_4, (0, 0))}_{7.5}, \underbrace{(r_7, (0, 0))}_{7.5}\}$$

Let us analyse the first calls to OUTPUT:

1. We pop $(r_1, \emptyset)$ and add H to GENERATED[1.1].
2. We pop $(r_5, \emptyset)$, add 1.8 to INDEX2COST and var to GENERATED[1.8].
3. We pop $(r_2, \emptyset)$, add 2.0 to INDEX2COST and W to GENERATED[2.0].
4. We pop $(r_6, \emptyset)$, add 3.3 to INDEX2COST and 1 to GENERATED[3.3]. At this point we have INDEX2COST = $\{1.1,\ 1.8,\ 2.0,\ 3.3\}$.
5. We pop $(r_3, (0))$, of cost COST(r_3) + INDEX2COST[0] = $4.4 + 1.1 = 5.5$. We try generating programs: GENERATED[INDEX2COST[0]] = $\{H\}$. Since H is not generated by I, the rule r_3 does not apply, and the algorithm does not generate programs at this step. We then update the data structure, adding $(r_3, (1))$ to QUEUE with cost $5.5 + 1.8 - 1.1 = 6.2$. At this point, the queue is as follows, with costs indicated below cost tuples:
$$\text{QUEUE} = \{\underbrace{(r_3, (1))}_{6.2}, \underbrace{(r_4, (0, 0))}_{7.5}, \underbrace{(r_7, (0, 0))}_{7.5}\}$$
6. We pop $(r_3, (1))$, of cost COST(r_3)+INDEX2COST[1] = $4.4+1.8 = 6.2$. We try generating programs: GENERATED[INDEX2COST[1]] = {var}. The program var is generated by S, so the algorithm generates cast(var). We then update the data structure, adding $(r_3, (2))$ to QUEUE with cost $6.2 + 2.0 - 1.8 = 6.4$. At this point, the queue is as follows, with costs indicated below cost tuples:
$$\text{QUEUE} = \{\underbrace{(r_3, (2))}_{6.4}, \underbrace{(r_4, (0, 0))}_{7.5}, \underbrace{(r_7, (0, 0))}_{7.5}\}$$

Limitations of Bee Search

The main limitation of BEE SEARCH is that there may be calls to OUTPUT where the algorithm does not generate any program, as in the fifth iteration in our example.

Implementation Details Line 30 was cleverly written another way by the original authors as a way to generate exactly once each successor in the cost tuple space, with

the following condition: `if n'[i] > 1: break` which avoids any check and is sound.

Practically both the original authors implementation and our implementation must round the costs in order not to have too many instances of cost tuples that do not correspond to any program.

5 Eco Search

Eco Search is the first no-delay best-first search algorithm and follows a similar structure as the two previous algorithms. We present the four key ideas behind Eco Search: cost tuple representation, per non-terminal data structure, frugal expansion, and bucketing. The full pseudocode is given in Algorithm 11.

Eco Search makes use of the cost tuple representation introduced by Bee Search but combines it with the second key idea introduced by Heap Search: *per non-terminal data structure*. Let us apply this philosophy and define the data structures for Eco Search. It maintains three objects for each non-terminal X:

- $\textsc{Generated}_X$: stores the set of programs generated from X so far, organised by costs. Concretely, it is a mapping from costs to sets of programs: $\textsc{Generated}_X[c]$ is the set of generated programs of cost c.
- $\textsc{Index2Cost}_X$: a list of the costs of the generated programs from X. Let us write $\textsc{Index2Cost}_X = [c_1, \dots, c_\ell]$, then $c_1 < \cdots < c_\ell$ and $\textsc{Generated}_X[c_i]$ is defined.
- $\textsc{Queue}_X$: stores information about which programs to generate next. Concretely, it is a priority queue of *cost tuples* ordered by costs, that we define now.

We naturally adapt the definition as follows. A cost tuple represents a set of programs: (r, n) represents all programs generated by the rule r where the i^{th} argument is any program in $\textsc{Generated}_X[\textsc{Index2Cost}_{X_i}[n_i]]$.

This makes the generation part in Eco Search very efficient, solving the limitation discussed above in Bee Search. In Algorithm 8 we spell out part of the function output, which takes as input a non-terminal X and a natural number ℓ (and becomes recursive). To formulate its specification let us write for a non-terminal X the set of costs $c_1 < c_2 < c_3 < \dots$ of all programs generated from X, we say that c_ℓ is the ℓ-smallest cost for X. The output of $\textsc{output}(X, \ell)$ is the set of all programs generated from X with ℓ-smallest cost.

Algorithm 8 The generation part in Eco Search

```
1: function output(X, ℓ):
2:     (skip part of the code)
3:     t = (r : X → f(X_1, ..., X_k), n : (n_1, ..., n_k))
4:     for p_1, ..., p_k in ∏_{i=1}^{k} output(X_i, n_i) do
5:         p ← f(p_1, ..., p_k)
6:         add p to Generated_X[c]
7:         yield p
8:     end for
9: end function
```

5.A Frugal expansion

We have presented the data structures of Eco Search, the way it generates programs, and the specification of its main function output. We now focus on the update part of output. The third key idea is frugal expansion, which addresses the main issue with Heap Search: the number of recursive calls to output. Indeed, to maintain the invariants on the data structures, we need to add cost tuples to the queue. As fleshed out in Algorithm 9, for a tuple $n : (n_1, \dots, n_k)$ we consider the k tuples obtained by adding 1 to each index: $n' : (n_1, \dots, n_i + 1, \dots, n_k)$ for each $i \in [1, k]$.

The issue is that this happens recursively as written in Algorithm 8, leading to many recursive calls. Two things can happen for a call to $\text{output}(X, \ell)$:

- Either the result was already computed (if $\text{Index2Cost}_X[\ell]$ is defined) and its answer is read off the data structure;
- Or it was not, and we perform some recursive calls as described in Algorithm 9.

The key property of frugal expansion is that when calling $\text{output}(X, \ell)$, for each non-terminal Y, at most one recursive call $\text{output}(Y, \_)$ falls in the second case.

Algorithm 9 Update part in Eco Search

```
1: function output(X, ℓ):
2:     (skip part of the code)
3:     t = (r : X → f(X_1, ..., X_k), n : (n_1, ..., n_k))
4:     (skip generation part of the code)
5:     for i from 1 to k do
6:         n' ← n
7:         n'_i ← n_i + 1
8:         if t' = (r, n') not in Queue_X then
9:             if n'_i not in Index2Cost_{X_i} then
10:                c' ← cost(peek(Queue_{X_i}))
11:                Index2Cost_{X_i}[n'_i] ← c'
12:            end if
13:            add t' to Queue_X with value cost(t')
14:        end if
15:    end for
16: end function
```

At this point we have a simplified version of Eco Search: as we will see in the experiments, it already outperforms Heap Search and Bee Search, but it does not yet have constant delay. We will later refer to this algorithm as "Eco Search without bucketing".

5.B Bucketing

To introduce our main innovation, we need to state and prove some theoretical properties on the costs of programs induced by pre-generation cost functions. First some terminology: let us fix a non-terminal X, and p, p' two programs generated by X. Remember, that p' is a successor of p if $\text{cost}(\text{p}) < \text{cost}(\text{p}')$ and there does not exist p'' generated by X such that $\text{cost}(\text{p}) < \text{cost}(\text{p}'') < \text{cost}(\text{p}')$. In other words, p' has minimal cost among programs of higher cost than p generated by X. Note that a program may have

many successors, but they all have the same costs. We write COST-SUCC(P) for the cost of any successor of P.

We first prove that successors in the cost tuple spaces are close in the cost space.

Lemma 3.8 ▶ The difference in cost between successors is bounded

There exists a constant $M \geq 0$ such that, for any non-terminal X and for any program P and its successor P′, both generated by X, we have $\text{COST}(\text{P}') - \text{COST}(\text{P}) \leq M$.

Proof. Let $\mathcal{F}$ be the set of all programs generated by some non-terminal, with the following properties:

(*) Any non-terminal appears at most once along any path from the root to a leaf in the derivation tree of the program,

(**) The programs in $\mathcal{F}$ have a successor.

First, observe that $\mathcal{F}$ is a finite set since there is a finite number of programs satisfying property (*). Therefore we can define the constant

$$M = \max_{\text{P} \in \mathcal{F}} \{\text{COST-SUCC}(\text{P}) - \text{COST}(\text{P})\}.$$

Consider any node n of the derivation tree of P for which the subprogram P_n rooted at n is in the set $\mathcal{F}$. It is always possible to find such a node n because P has a successor: starting from the root, we can always choose a child which has at least one successor until condition (*) is satisfied. Note that as long as condition (*) is not satisfied, there is always a child with a successor because there is a duplicated non-terminal on some path, ensuring that the process is sound.

We now show that the cost difference between P and its successor P′ can be bounded by M. Since P_n belongs to $\mathcal{F}$, there exists a successor subprogram P'_n of P_n such that $\text{COST}(\text{P}'_n) - \text{COST}(\text{P}_n) \leq M$.

The overall program P can be thought of as being composed of two parts: the part above n and the subtree rooted at n. When P_n is replaced by P'_n, we obtain a program P″ for which the cost is an upper bound on the cost of the successor P′ of P. Therefore, we have:

$$\begin{aligned} \text{COST}(\text{P}') - \text{COST}(\text{P}) &\leq \text{COST}(\text{P}'') - \text{COST}(\text{P}) \\ &= \text{COST}(\text{P}'_n) - \text{COST}(\text{P}_n) \\ &\leq M. \end{aligned}$$

□

Note that it is trivial to see that M is tight by contradiction if it is not, there exists $m < M$ such that the lemma holds however M is defined as

$$M = \max_{\text{P} \in \mathcal{F}} \{\text{COST-SUCC}(\text{P}) - \text{COST}(\text{P})\}.$$

therefore it must be $m = M$ which is a contradiction therefore M is tight. Furthermore, M depends on the starting nonterminal, this implies that there are different bounds for different starting nonterminals, intuitively if S consumes nonterminal K then $M_S \geq M_K$.

A consequence of Lemma 3.8 is a similar bound, this time applying to the queue in Eco Search.

Lemma 3.9 ▶ Cost tuples in Queue_X have bounded cost

There exists a bound $M \geq 0$ such that in Eco Search at a any given time, for any non-terminal X, all programs p in the queue Queue_X satisfy:

$$\text{cost}(\text{p}) - \min_{\text{p}' \in \text{Queue}_X} \text{cost}(\text{p}') \leq M.$$

Proof. We prove the property by induction. First observe that it holds at the beginning of the algorithm. To see that it is maintained when a program p is popped from the queue and its successors are added, we make two observations.

- by Lemma 3.8, the cost difference between the program and its successors is bounded by M, and
- p has minimal cost in the queue.

□

Let us make a simplifying assumption: the cost function takes integer values, meaning $w : P \to \mathbb{N}_{>0}$. Let us analyse the time complexity of $\text{output}(X, \_)$. As discussed above frugal expansion implies that for each non-terminal Y, at most one call to $\text{output}(Y, \_)$ yields to recursive calls. Hence the total number of recursive calls is bounded by the number of non-terminals, and we are left with analysing the time complexity of a single call. It is bounded by the time needed to pop and push a constant number (bounded by the maximum arity in the CFG) of cost tuples from a queue. If the queues are implemented as priority queues, the time complexity of these operations is $O(\log N)$, where N is the number of elements in the queue.

However, thanks to Lemma 3.9, there are at most M possible costs in the queue at any given time. Therefore, we can implement the queues as "bucket queues" (a classical data structure, see for instance [Tho00]). Concretely, a bucket queue is an array of M lists, each containing cost tuples with the same cost. We keep track of the index j of the list that contains programs of minimal cost. To pop a cost tuple, we iterate over the j^{th} list. To push an element that has a cost k plus from the current minimal cost, we simply add it to the list at index $(j + k) \mod M$. If the list at index j is empty, we increment $j \mod M$ until we find a non-empty list. The time complexity of popping and pushing an element in this implementation is constant.

Theorem 3.10 ▶ Constant Delay

Assuming integer costs, Eco Search has constant delay: the amount of compute between generating two programs is constant over time.

5.C Complete Algorithm

Algorithm 10 and Algorithm 11 fully describes the algorithm.

Algorithm 10 Eco Search: initialisation

compute $\textsc{MinProg}(X)$ and $\textsc{MinCost}(X)$ a program of minimal cost from X and its cost, for each non-terminal X
for X non-terminal **do**
    $\textsc{Generated}_X$: mapping from costs to sets of programs
    $\textsc{Index2Cost}_X$: list of costs
    $\textsc{Queue}_X$: priority queue of cost tuples ordered by costs
end for

for $r : X \to f(X_1, \ldots, X_k)$ derivation rule **do**
    $c \leftarrow \textsc{cost}(r) + \sum_{i=1}^{k} \textsc{MinCost}(X_i)$ ▷ computes the cost of the (implicit) program $f(\textsc{MinProg}(X_1), \ldots, \textsc{MinProg}(X_k))$
    add $(r, (0, \ldots, 0))$ to $\textsc{Queue}_X$ with value c
end for

Algorithm 11 Eco Search: main loop

$\ell \leftarrow 0$
while True **do**
    $\textsc{Output}(S, \ell)$
    $\ell \leftarrow \ell + 1$
end while

function $\textsc{output}(X, \ell)$: ▷ generates all programs from X with ℓ-smallest cost
    if $\textsc{Index2Cost}_X[\ell]$ is defined **then** ▷ the result was already computed and can be read off from the data structure
        return $\textsc{Generated}_X[\textsc{Index2Cost}_X[\ell]]$
    end if
    $t \leftarrow \textsc{peek}(\textsc{Queue}_X)$ ▷ returns the minimal cost tuple in $\textsc{Queue}_X$ without popping it
    $c \leftarrow \textsc{cost}(t)$ ▷ $\textsc{cost}(t)$ is stored together with the cost tuple t
    if $\textsc{Index2Cost}_X$ is empty or $c \neq \textsc{Index2Cost}_X[-1]$ **then**
        add c to $\textsc{Index2Cost}_X$ ▷ the last generated program from X did not have cost c
        $\textsc{Generated}_X[c] \leftarrow \emptyset$
    end if
    while $\textsc{cost}(t) = c$ **do** ▷ we iterate as long as we find cost tuples with cost c in $\textsc{Queue}_X$
        $\textsc{pop}(\textsc{Queue}_X)$
        if $t = (r : X \to a, \emptyset)$ **then**
            add a to $\textsc{Generated}_X[c]$
            yield a
        else $t = (r : X \to f(X_1, \ldots, X_k), n : (n_1, \ldots, n_k))$
            for $\mathrm{P}_1, \ldots, \mathrm{P}_k$ in $\prod_{i=1}^{k} \textsc{Output}(X_i, n_i)$ **do** ▷ generates programs
                $\mathrm{P} \leftarrow f(\mathrm{P}_1, \ldots, \mathrm{P}_k)$
                add P to $\textsc{Generated}_X[c]$
                yield P
            end for
            for i from 1 to k **do** ▷ updates the data structure
                $n' \leftarrow n$
                $n'_i \leftarrow n_i + 1$
                if $t' = (r, n')$ not in $\textsc{Queue}_X$ **then**
                    if n'_i not in $\textsc{Index2Cost}_{X_i}$ **then**
                        $\textsc{Index2Cost}_{X_i}[n'_i] \leftarrow \textsc{cost}(\textsc{peek}(\textsc{Queue}_{X_i}))$
                    end if
                    $\textsc{cost}(t') \leftarrow \textsc{cost}(t) + \textsc{Index2Cost}_{X_i}[n'_i] - \textsc{Index2Cost}_{X_i}[n_i]$
                    add t' to $\textsc{Queue}_X$ with value $\textsc{cost}(t')$
                end if
            end for
        end if
        $t \leftarrow \textsc{peek}(\textsc{Queue}_X)$
    end while
end function

Example 3.11 ▶ Eco Search example

We consider the grammar and associated costs defined in Figure 3.3. Algorithm 2 finds $\textsc{MinProg}(S) = H, \textsc{MinCost}(S) = 1.1$ and $\textsc{MinProg}(I) = \text{var}, \textsc{MinCost}(I) = 2.0$. During initialisation, we add the following cost tuples:

$$(r_1, \emptyset), (r_2, \emptyset), (r_3, (0)), (r_4, (0,0)), (r_5, \emptyset), (r_6, \emptyset), (r_7, (0,0))$$

At this point, the queues are as follows, with costs indicated below cost tuples:

$$\textsc{Queue}_S = \{\underbrace{(r_1, \emptyset)}_{1.1}, \underbrace{(r_2, \emptyset)}_{2.0}, \underbrace{(r_3, (0))}_{6.4}, \underbrace{(r_4, (0,0))}_{7.5}\}$$

$$\textsc{Queue}_I = \{\underbrace{(r_5, \emptyset)}_{1.8}, \underbrace{(r_6, \emptyset)}_{3.3}, \underbrace{(r_7, (0,0))}_{9.3}\}$$

Let us analyse the first calls:

1. $\textsc{generate}(S, 0)$: We pop $(r_1, \emptyset)$ from $\textsc{Queue}_S$, add 1.1 to $\textsc{Index2Cost}_S$, and add H to $\textsc{Generated}_S[0]$.

2. $\textsc{generate}(S, 1)$: We pop $(r_2, \emptyset)$ from $\textsc{Queue}_S$, add 2.0 to $\textsc{Index2Cost}_S$, and add W to $\textsc{Generated}_S[1]$.

3. $\textsc{generate}(S, 2)$: We pop $(r_3, (0))$ from $\textsc{Queue}_S$ and add 6.4 to $\textsc{Index2Cost}_S$. Line 19 triggers a call to $\textsc{generate}(I, 0)$. During this call, we pop $(r_5, \emptyset)$ from $\textsc{Queue}_I$, add 1.8 to $\textsc{Index2Cost}_I$, and add var to $\textsc{Generated}_I[0]$. After the call we have $\textsc{Generated}_I[0] = \{\text{var}\}$. We now generate programs: we add cast(var) to $\textsc{Generated}_S[2]$.

   We then update the data structure. We consider $(r_3, (1))$. Since $\textsc{Index2Cost}_I[1]$ does not exist yet we compute it: it is $\textsc{cost}((r_6, \emptyset)) = 3.3$, so we add $(r_3, (1))$ to $\textsc{Queue}_S$ with cost $6.4 + 3.3 - 1.8 = 7.5$.

4. $\textsc{generate}(S, 3)$: We pop $(r_4, (0,0))$ from $\textsc{Queue}_S$ and add 7.5 to $\textsc{Index2Cost}_S$. Line 19 triggers a call to $\textsc{generate}(S, 0)$, already computed: $\textsc{Generated}_S[0] = \{H\}$. We add $\text{concat}(H, H)$ to $\textsc{Generated}_S[3]$. We then update the data structure. We consider $(r_4, (1,0))$ and $(r_4, (0,1))$. Here $\textsc{Index2Cost}_S[1]$ already exists (iteration 2.). We add $(r_4, (1,0))$ and $(r_4, (0,1))$ to $\textsc{Queue}_S$ both with cost $7.5 + 2.0 - 1.1 = 8.4$.

   We do not yet exit the **while** loop (line 13): the next cost tuple in $\textsc{Queue}_S$ has the same cost $c = 7.5$, so we also pop $(r_3, (1))$. Line 19 triggers a call to $\textsc{generate}(I, 1)$. During this call, we pop $(r_6, \emptyset)$ from $\textsc{Queue}_I$, add 3.3 to $\textsc{Index2Cost}_I$, and add 1 to $\textsc{Generated}_I[1]$. After the call we have $\textsc{Generated}_I[1] = \{1\}$. We now generate programs: we add cast(1) to $\textsc{Generated}_S[3]$. We then update the data structure. We consider $(r_3, (2))$. Since $\textsc{Index2Cost}_I[2]$ does not exist yet we compute it: it is $\textsc{cost}((r_7, (0,0))) = 9.3$, so we add $(r_3, (2))$ to $\textsc{Queue}_S$ with cost $6.4 + 9.3 - 3.3 = 12.4$.

Implementation Details We use the same trick as the Bee Search implementation details to generate each cost tuple exactly once.

In practice costs are rarely integers, therefore they need to be discretized. The discretisation depends on the case. In practice, we use for a nonterminal S, the tight bound M_S for that nonterminal to know the size of the bucket queue needed, however it may be very large: in the billions. Therefore we can use the bucket factor C which implements buckets of size C as a tree structure to describe the M_S different buckets, this gives a constant cost for pop and push of $O(\log C)$. In practice, we chose arbitrarily that if $M_S \leq 1000$ we directly use M_S otherwise we use $C = 20$ unless we specify the bucket size. Better results could be obtained if we optimised there parameters of the algorithm.

6 Experiments

To investigate whether the theoretical properties of Eco Search bear fruits we ask the following questions:

Q1: Does Eco Search improve the performance of enumerative approaches on program synthesis tasks?

Q2: How does the performance of these algorithms scale with the complexity of the grammar?

Datasets. We consider two classic domains: string manipulations and integer list manipulations. For string manipulations we use the same setting as in Bee Search [AL23]: FlashFill's 205 tasks from SyGuS. The DSL has 3 non-terminals, one per type. For integer list manipulation we use the DeepCoder [BGB+17] dataset comprised of 366 tasks. The DSL has 2 non-terminals, again one per type. We set a timeout of one hour or more.

The cost functions used are the same for all algorithms, following [FLM+22]. Predictions are obtained with the help of a neural network outputting probability for each derivation rule. The neural networks are trained on the same synthetic dataset (one for each domain). All of the details are the same as in Chapter 2.

Implementation. All algorithms are re-implemented in Python. We will open-source the implementation upon publication. The code contains the seeds used, the cost functions and all other additional minor experimental details. All experiments were run on a 16 GB RAM machine with an Intel Xeon(R) W-1270 CPU running at up to 3.40GHz, running Ubuntu Jellyfish (no GPUs were used). They were run on at least five different seeds and we report the mean performance along with the 95% confidence interval.

Algorithms. We compare Eco Search against the two state of the art best-first search algorithms: Heap Search and Bee Search. Since they are all bottom-up algorithms they all use observational equivalence (pruning programs with same outputs on all input examples). None of them have hyperparameters except for the rounding off procedure for costs. For Bee Search we follow the original implementation and round off cost values to 10^{-2} in log space (since our cost function are probabilities). For Eco Search we need to discretize costs, as follows. We discretize probabilities in log space up to 10^{-5}, meaning that in these experiments two probabilitites whose ratio is larger than 1 but less than $e^{10^{-5}}$ are the same and cannot be distinguished. By default we use

bucket size 20, but we also experiment with other values, and for comparison consider also Eco Search without bucketing and in all case if the constant M needed for exact representation, that it no bucket splitting, is less than 1000 then we instead use that constant, those parameters were not tuned, we chosed these constant as a naive tradeoff.

Does Eco Search improve the performance of enumerative approaches on program synthesis tasks?

We run all best-first search algorithms on our benchmarks. The timeout per task is five minutes ($300s$). We plot the mean cumulative time used and the 95% confidence interval with respect to the number of tasks completed successfully on Figure 3.8 for string manipulations and Figure 3.9 for integer list manipulations.

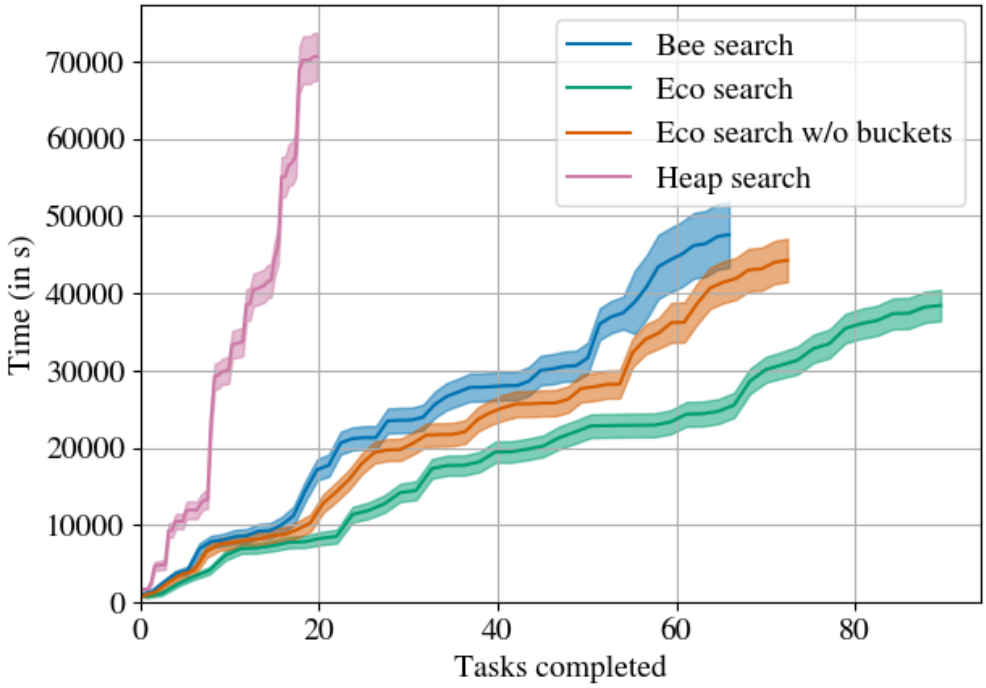


Figure 3.8: Main results: cumulative time for completing tasks for String manipulations from SyGuS using FlashFill's DSL

First, for string manipulation, we observe that Heap Search is far outperformed by other algorithms with Eco Search achieving the same score in 14% of the time it took Heap Search. This is why we did not include Heap Search in integer list manipulation because it times out on most tasks.

Second, Eco Search without buckets outperfoms Bee Search. The increase in performance is small on string manipulation with a bit less than 10 more tasks solved but larger on integer list manipulation as it solves more than 20 more tasks compared to Bee Search. To explain why the gap in performance is different in the two domains, we will see in the next experiment that Bee Search scales poorly with the number of non-terminals in the DSL, which is larger for string manipulation.

Finally, Eco Search outperforms all other algorithms by a large margin, solving 13 more tasks on integer list manipulations and 20 more tasks than its variant without

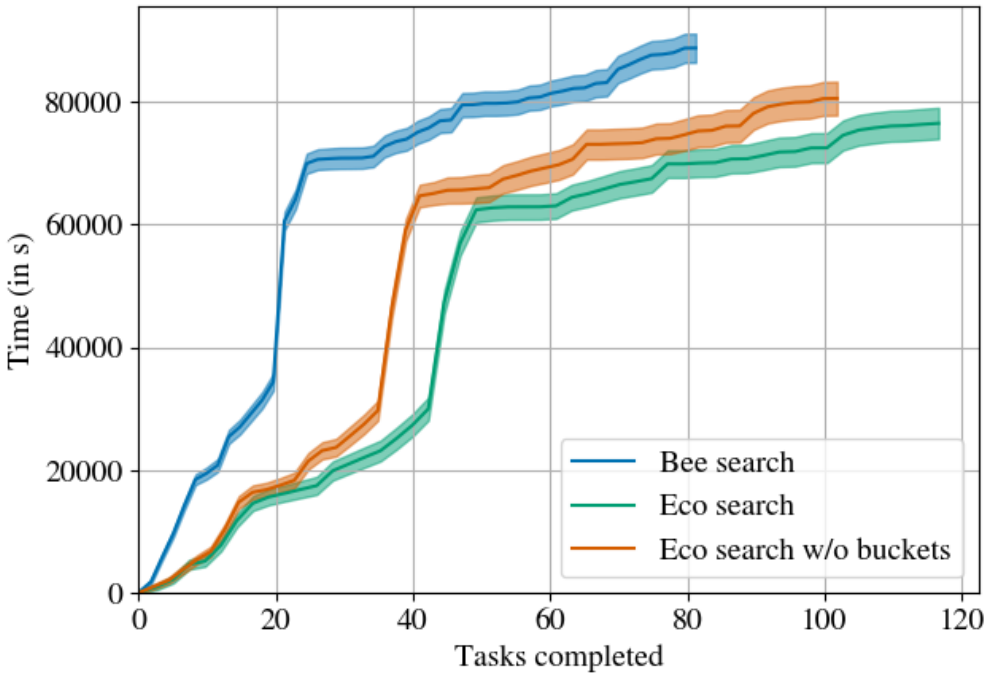


Figure 3.9: Main results: cumulative time for completing tasks for Integer List Manipulation using DeepCoder's DSL

bucketing. Comparing to Bee Search, it reaches the same number of tasks solved in slightly more than half the time for string manipulation and 78% of the time for integer list manipulation, while solving at least 20 new tasks compared to Bee Search on both datasets.

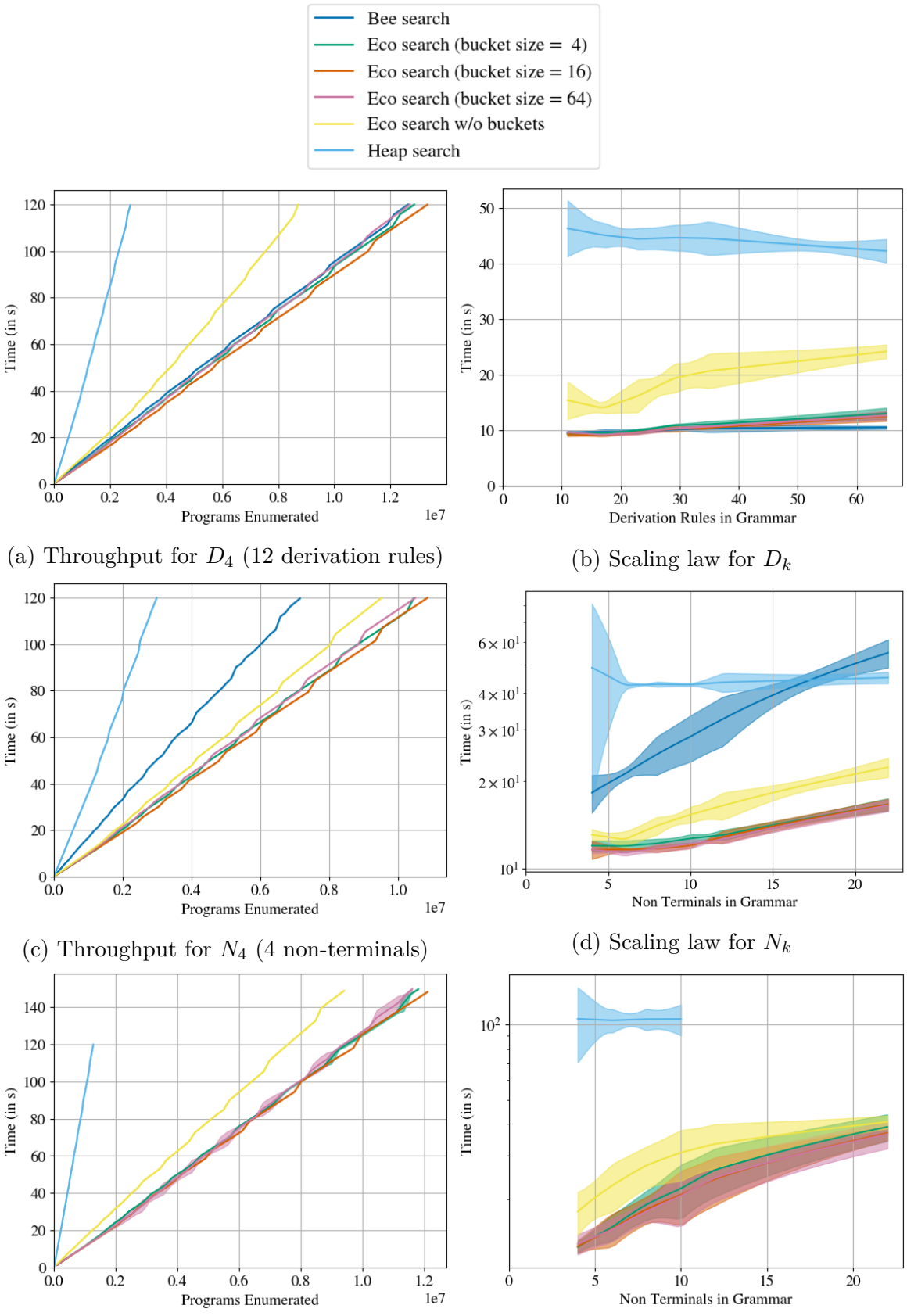
Summary

> Eco Search outperfoms all other algorithms including its variant without bucketing, reaching the same number of tasks solved in 66% of the time and solving 30% more tasks in total.

How does the performances of these algorithms scale with the complexity of the grammar?

The goal of these experiments is to understand how well our algorithms perform on more complicated grammars. However there is no agreed upon definition of "grammar complexity" as different measures of complexity can be used. A bad proxy for grammar complexity is the number of programs it generates: it is in most cases infinite, and as a function of depth it grows extremely fast hence cannot be accurately compared. We identify three parameters:

- The number of derivation rules;
- The number of non-terminals;



(a) Throughput for D_4 (12 derivation rules)

(b) Scaling law for D_k

(c) Throughput for N_4 (4 non-terminals)

(d) Scaling law for N_k

(e) Throughput for R_4

(f) Scaling law for R_k

Figure 3.10: Scaling against the three parameters: throughput and scaling laws.

- The maximal distance from a non-terminal to the start non-terminal, meaning the number of derivation rules required to reach the non-terminal.

In our experiments we measure the performance of our algorithms for pure enumeration: the programs are not evaluated on input examples, enumeration continues for a fixed amount of time. For each parameter, we created parametric grammars:

- The grammar D_k has $3k$ derivation rules. It uses a single non-terminal S. The primitives are: k primitives f_i of arity 2, k primitives g_i of arity 1, and k constants h_i (arity 0). The derivation rules are, for each $i \in [1, k]$:

$$S \to f_i(S, S) \quad ; \quad S \to g_i(S) \quad ; \quad S \to h_i$$

- The grammar N_k has k non-terminals, called $S_1, \ldots, S_k$, with S_1 initial. The primitives are: $2k$ primitives f_i, g_i of arity 1 and k constants h_i (arity 0). The derivation rules are, for each $i \in [1, k]$:

$$S_1 \to f_i(S_i) \quad ; \quad S_i \to g_i(S_1) \quad ; \quad S_i \to h_i$$

- The grammar R_k has k non-terminals, called $S_1, \ldots, S_k$, with S_1 initial. The primitives are: k primitives f_i of arity 3, k primitives g_i of arity 2, k primitives h_i of arity 1, and k constants k_i (arity 0). The derivation rules are, for each $i \in [1, k]$:

$$\begin{aligned} S_i \to f_i(S_{i-1}, S_i, S_{i+1}) \quad ; \qquad & S_i \to g_i(S_1, S_i) \\ S_1 \to h_i(S_i) \quad ; \qquad & S_i \to k_i \end{aligned}$$

For each of the three parameters, we consider two scenarios:

1. *Throughput*: For a fixed grammar, how many programs are enumerated as a function of time.
2. *Scaling law*: For a range of values of the parameter, how long does it take to enumerate one million programs.

We plot the results for the three parameters and both scenarios on Figure 3.10.

First, we look at the evolution with the number of derivation rules. Eco Search and Bee Search perform equally well, almost irrespective of the bucket size. However, removing bucketing makes Eco Search much slower. When we look at the scaling law, the same result is observed, and more generally it seems there is little to no influence of the number of derivation rules.

Second, looking at the number of non-terminals, for the throughput scenario the results are the same as for the main experiments: Heap Search < Bee Search < Eco Search without bucketing < Eco Search. On the scaling law, we however observe that Bee Search is outperformed by Heap Search for grammars with more than 15 non-terminals. The same growth is observed for all variants of Eco Search albeit at a slower pace. This suggests that Bee Search scales badly with the number of non-terminals: increasing the number of non-terminals 4x, Bee Search takes 3x more time, while Eco Search takes only 2x more time.

Finally, looking at the distance to the starting non-terminal. Bee Search is missing since we failed to enumerate $10K$ programs within the timeout, even for R_4. Simiarly, Heap Search was not plotted for larger parameters because it failed to enumerate 1M programs. For the throughput, except for the disappearance of Bee Search the results are as expected. For the scaling law, we observe that the distance has a significant impact: Eco Search takes 6x more time for R_{22} compared to R_4.

Summary

Eco Search scales better than alternatives in terms of number of non-terminals and distance to starting non-terminal, and equally well as Bee Search for the number of derivation rules. Also, Eco Search is relatively robust to the choice of bucket size.

7 Related Work

Combinatorial search for program synthesis has been an active area [ASFS18], and a powerful tool in combination with neural approaches [CEP+21]. In particular, cost-guided combinatorial search provides a natural way of combining statistical or neural predictions with search [MTG+13, BGB+17].

By exploring the space in the exact order induced by the cost function, best-first search algorithms form a natural family of algorithms. The first best-first search algorithm constructed in the context of cost-guided combinatorial search was an A^* algorithm [LHAN18]. Eco Search can be thought of as the unification of Heap Search [FLM+22] and Bee Search [AL23], both best-first search bottom-up algorithms. Best-first search algorithms were also developed for Inductive Logic Programming [CD20].

Many recent works targeting cost-guided combinatorial search introduce a trade-off: enumerating faster by deviating from the cost function. The first algorithm in this direction is the Threshold algorithm [MTG+13], and the second the Sort & Add algorithm [BGB+17], both are improved variants of a simple depth-first search approach.

Importantly, Eco Search follows the bottom-up paradigm, where larger programs are obtained by composing smaller ones [URD+13]. Bottom-up algorithms have been successfully combined with machine learning approaches, for instance the PC-Coder [ZW18], Probe [BPP20], TF-Coder [SBS22b], and DreamCoder [EWN+21]. In these works, machine learning is used to improve combinatorial search, while Bustle [OSB+21] and Execution-Guided Synthesis [CLS18] use neural models to guide the search process itself. Alternatively, CrossBeam [SDES22] and LambdaBeam [SDL+24] leverage Reinforcement Learning for this purpose.

Interestingly, LambdaBeam can solve many tasks that LLMs cannot solve thanks to its ability to perform high-level reasoning and composition of programs. Together with recent approaches using LLMs for guiding combinatorial search [LPP24, LE24], this motivates developing better and better algorithms for cost-guided combinatorial search.

8 Futures

With Eco Search we have finished the theoretical development of enumeration algorithm for cost grammars, it is not possible to do better than no-delay. However, it is still possible to improve the performance of enumeration algorithms. Our experiments reveal an important gap: combinatorial search algorithms suffer large drops in performance when increasing the complexity of the grammar. In many cases the grammar remains small and this limitation is not drastic. However, recent applications of program synthesis use large or even very large grammars, for instance [Hod24] constructs a very large DSL towards solving the Abstraction Reasoning Corpus [Cho19]. We leave as an open question to construct best-first search algorithms that can operate on such large DSLs.

Another direction for future work is memory. Currently, Eco Search and its predecessors need to remember all previously enumerated programs. For example, on a 11th generation i7 running Eco Search on the grammars used in the scaling law experiments, 2 minutes suffice to fill 16GB of memory. For program synthesis, the evaluation of programs takes most of time which in most tasks makes the problem less crucial, but that is not the case for all tasks and eventually memory becomes an issue on all tasks with large enough timeouts. This memory issue also appears with the bottom-up evaluation which needs to remember previously evaluated programs. My conjecture is that these enumerations algorithms can be adapted to use a fixed amount of memory in the sense that they need to remember a fixed number of programs, but the programs could be arbitrarily large. Perhaps this could also be used for bottom-up evaluation. The end goal being constant delay and constant number of programs but we have doubt that it is possible for generic PCFGs. A large issue is the size of the frontier of possibly next-to-be-generated programs which grows indefinitely and that we would have to limit. We believe this could be achieved for more structured PCFGs.

Finally, all these algorithms are nor parallelised, it is not trivial to parallelise them, and if they could be the improvements would be great. Of great interest is to parallelise the evaluation since this is most likely the bottleneck but at some point, enumeration will come back to being the bottleneck. So how can we use multiple processors to parallelise these approaches?

Chapter 4

Parallel Search

TL;DR 4

The future of computing is a parallel one. However, as described previously enumerative algorithms are complex and rely on advanced data structures. An alternative are sampling algorithms, they boast an impressive property: they are memoryless making them the perfect fit for parallel search. We introduce SQRT Sampling a sampling algorithm that is optimal in the best first search sense defined last chapter. Despite this facility to parallelise it is not a very effective search approach in practice. Another direction we propose is splitting the grammar into sub grammars and enumerating them independently. Splitting the grammar enables speeding up the search proportionally to the number of processors used. We describe an algorithm that works on finite grammars describing a finite number of programs.

First, in Section 1 we explore the space of sampling algorithms which have natural advantages in a parallel context. Then in Section 2 we tackle the idea of splitting the grammar to divide the search space into multiple subspaces making the parallelisation easy. This section is an extract of our work [FLM+22].

Our contributions:

- SQRT sampling the optimal best-first search algorithm among sampling algorithm.
- The grammar splitter an algorithm that enables splitting a grammar into sub grammars as way to easily parallelize the search independently of the algorithm used.

Note In this chapter we will only consider probabilistic grammars but cost grammars can be transformed into probabilistic grammar.

1 Sampling

A *sampling algorithm* takes random samples from a distribution $\mathcal{D}'$; what is both a strength and a weakness is that a sampling algorithm is memoryless: a weakness because the algorithm does not remember the previous draws, which means that it may draw them again, but also a strength because it uses very little space and can be very easily implemented.

In the case of sampling algorithms, we identify algorithms with distributions. The following theorem shows a dichotomy: either there exists a loss optimal sampling algorithm among sampling algorithms, and then it is characterised as the 'square root' of the distribution, or all sampling algorithms have infinite loss.

Theorem 4.1 ▶ Loss Optimal Sampling

Let $\mathcal{D}$ a distribution over a set X. If $\sum_{x\in X}\sqrt{\mathcal{D}(x)} < \infty$, the distribution $\sqrt{\mathcal{D}}$ defined by

$$\sqrt{\mathcal{D}}(x) = \frac{\sqrt{\mathcal{D}(x)}}{\sum_{y\in X}\sqrt{\mathcal{D}(y)}}$$

is loss optimal among all sampling algorithms. If $\sum_{x\in X}\sqrt{\mathcal{D}(x)} = \infty$, for all sampling algorithms $\mathcal{D}'$ we have $(\mathcal{D}', \mathcal{D}) = \infty$.

Proof. Let $\mathcal{D}'$ be a distribution. For an element x, the expectation of the number of tries for $\mathcal{D}'$ to draw x is $\frac{1}{\mathcal{D}'(x)}$: this is the expectation of success for the geometric distribution with parameter $\mathcal{D}'(x)$. It follows that

$$(\mathcal{D}', \mathcal{D}) = \mathbb{E}_{x\sim D}\left[\frac{1}{\mathcal{D}'(x)}\right] = \sum_{x\in X}\frac{\mathcal{D}(x)}{\mathcal{D}'(x)}.$$

Let us assume that $\sum_{x\in X}\sqrt{\mathcal{D}(x)} < \infty$. Thanks to Cauchy-Schwarz inequality we have:

$$\begin{aligned}\left(\sum_{x\in X}\sqrt{\mathcal{D}(x)}\right)^2 &= \left(\sum_{x\in X}\sqrt{\tfrac{\mathcal{D}(x)}{\mathcal{D}'(x)}}\sqrt{\mathcal{D}'(x)}\right)^2 \\ &\leq \left(\sum_{x\in X}\tfrac{\mathcal{D}(x)}{\mathcal{D}'(x)}\right)\cdot\underbrace{\left(\sum_{x\in X}\mathcal{D}'(x)\right)}_{=1} \\ &= \sum_{x\in X}\tfrac{\mathcal{D}(x)}{\mathcal{D}'(x)}.\end{aligned}$$

We note that $(\sqrt{\mathcal{D}}, \mathcal{D}) = \left(\sum_{x\in X}\sqrt{\mathcal{D}(x)}\right)^2$, so the previous inequality reads $(\mathcal{D}', \mathcal{D}) \geq (\sqrt{\mathcal{D}}, \mathcal{D})$. Thus $\sqrt{\mathcal{D}}$ is loss optimal among sampling algorithms, and if it is not defined, then for any $\mathcal{D}'$ we have $(\mathcal{D}', \mathcal{D}) = \infty$. □

Theorem 4.1 characterises the loss optimal sampling algorithm, but does not explain how to implement it. The following result answers that question.

Theorem 4.2 ▶ Effective Construction of the Square Root Distribution

If $\mathcal{D}$ is defined by a PCFG and $\sqrt{\mathcal{D}}$ is well defined, then we can effectively construct a PCFG defining $\sqrt{\mathcal{D}}$.

The PCFG for $\sqrt{\mathcal{D}}$ is obtained from the PCFG for $\mathcal{D}$ by taking the square root of each transition probability, and then globally renormalising.

1.A Implementation details

The construction follows two steps: first construct a weighted CFG that recognises $\sqrt{\mathcal{D}}$ and then normalise it into a PCFG using [Chi99]. The normalisation requires computing

the partition function Z defined by

$$Z(S) = \sum_{P \text{ generated from } S} \mathcal{D}(P).$$

In general the partition function can be computed by solving a system of polynomial equations. This is easier in our case since we restrict ourselves to acyclic PCFGs. We did not implement the general case.

Our implementation of SQRT Sampling uses the Alias method [Wal77], which is an efficient data structure for sampling from a categorical distribution. We associate to each non-terminal an Alias table, reducing the task of sampling a derivation rule with n choices to sampling uniformly in $[1, n]$ and in $[0, 1]$. Furthermore, we have optimised the implementation using Cython.

1.B Experiments

This section takes the same experiment as the one described in Section 3, however we focus solely on SQRT here, as such we do not duplicate the figures. We report results on DSLs from DeepCoder [BGB+17] and DreamCoder [EWN+21]. Both target the classic program synthesis challenge of integer list processing programs, but with different properties. DeepCoder's DSL is larger and more specialized, with around 40 high-level primitives, and does not use polymorphic types, while DreamCoder's is smaller and more generic, with basic functional programming primitives such as map, fold, unfold, car, cons, and cdr, etc., for a total of around 20 primitives. Both DSLs are compiled into finite depth CFGs with minimal syntactic constraints generating programs of depth up to 6.

The search algorithms under consideration are:

- Threshold from [MTG+13]: iterative-deepening-search, where the threshold that is iteratively deepened is a bound on program description length (i.e. negative log probability),
- Sort and add from [BGB+17]: an inner loop of depth-first-search, with an outer loop that sorts productions by probability and runs depth-first-search with the top k productions for increasing values of k,
- A^* from [FMBD18]: best-first-search on the graph of (log probabilities of) tree derivations,
- Beam search from [ZRF+18]: breadth-first-search with bounded width that is iteratively increased.

As well as our new algorithms: Heap Search (from Chapter 3) and SQRT Sampling.

All algorithms have been reimplemented and optimised in the codebase to provide a fair and uniform comparison.

Random PCFGs

In this first set of experiments we run all search algorithms on random PCFGs until a timeout, and compare the number of programs they output and the cumulative probability of all programs output.

To obtain random PCFGs from the CFGs we sample a probabilistic labeling with an exponential decrease (this is justified by the fact that machine-learned PCFGs feature exponential decrease in transition probabilities). In this experiment the initialization cost of each algorithm is ignored. The results presented here are averaged over 50 samples of random PCFGs, the solid lines represent the average and a lighter color indicates the standard deviation.

Figure 3.4 shows the results for all algorithms in a non-parallel implementation. On the lhs we see that Heap Search (almost always) has the highest cumulative probability against both time and number of distinct programs. Note that since A^* and Heap Search enumerate the same programs in the same order they produce the same curve in the right hand side of Figure 3.4 so we did not include A^*.

The results show that SQRT is clearly not the bets algorithm but despite being the only sampling algorithm there its performance seems relatively close to that of Heap Search.

Machine-learned PCFGs

In this second set of experiments we consider the benchmark suites of hand-picked problems and sets of I/O. We extracted 218 problems from DreamCoder's dataset [EWN+21].

We train a neural network to make predictions from a set of I/O. Our neural network is composed of a one layer GRU [CvMG+14] and a 3-layer MLP with sigmoid activation functions, and trained on synthetic data generated from the DSL. The details of the architecture and the training can be found in Chapter 2 Section 2.A. Our network architecture induces some restrictions, for instance the types of the programs must be `int list -> int list`; we removed tasks that did not fit our restrictions and obtained a filtered dataset of 137 tasks. For each task we run every search algorithm on the PCFG induced by the neural predictions with a timeout of 100s and a maximum of $1M$ programs. Unlike in the previous experiments the initialization costs of algorithms are not ignored.

Figure 4.1 shows the number of tasks solved within a time budget. Heap Search solves the largest number of tasks for any time budget, and in particular 97 tasks out of 137 before timeout. The comparison between Threshold and A^* is insightful: A^* solves a bit more tasks than Threshold (85 vs 83) but in twice the time. SQRT Sampling performs just a bit worse than A^* despite being a sampling algorithm, however it is still far from other algorithms.

Table 4.1 shows for each algorithm how many programs were generated per second. It is interesting to compare the rates of SQRT Sampling and A^*: although SQRT Sampling generates over two times more programs, their overall performances are similar. This can be explained in two ways: SQRT Sampling may sample the same programs many times, while A^* enumerates each program once and starts with the most promising ones according to the predictions.

1.C Futures

The comparison of SQRT was done with Heap Search since then enumerative algorithm have been made a lot faster putting SQRT out of the picture. Despite the advantage of SQRT being easily parallelisable, this is not sufficient for large grammars, indeed a large issue it also that it is memoryless and sample often programs already enumerated.

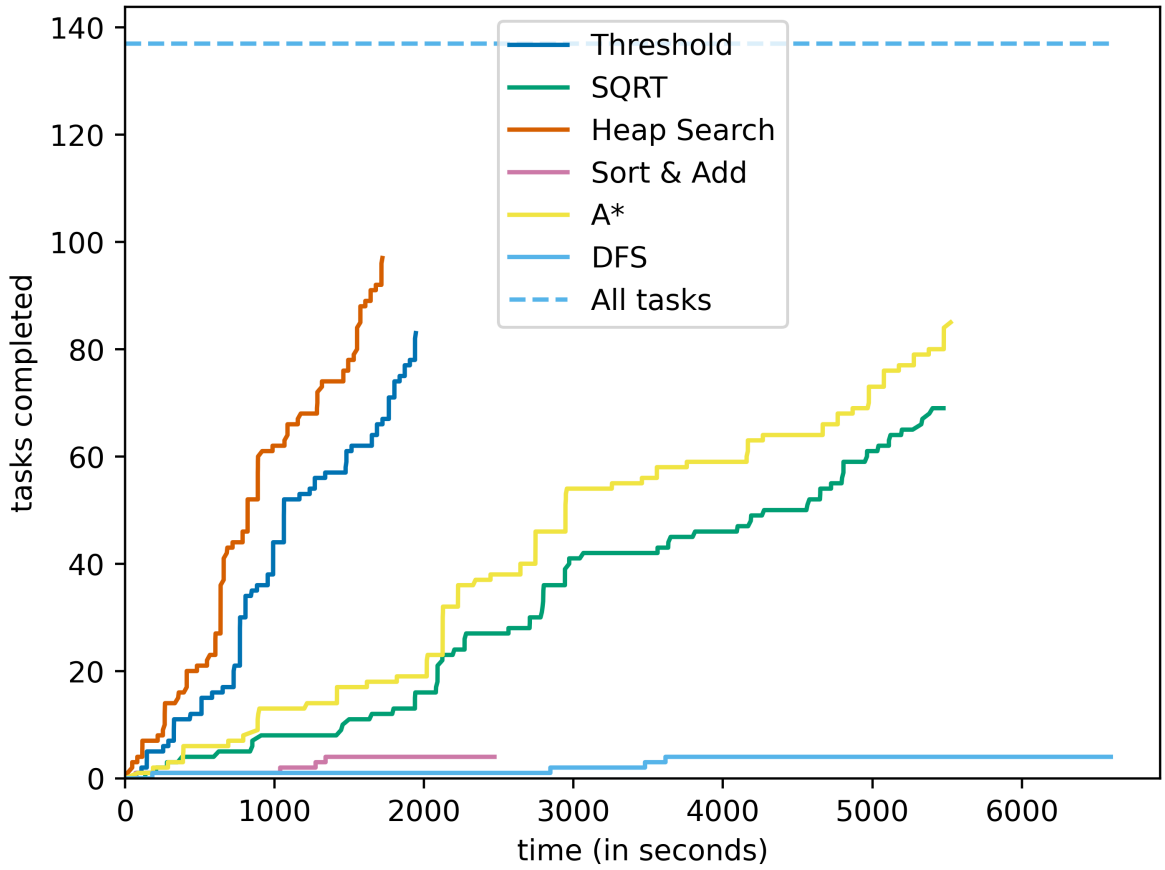


Figure 4.1: Comparing all search algorithms on the DreamCoder reduced dataset with machine-learned PCFGs

We tried combining it with unique randomizer [SBS20b] in order to not enumerate twice the same program but we failed to make it better than the base SQRT algorithm. Future directions, include speeding-up the sampling to catch up with enumerative approach and finding a way to avoid sampling similar programs too many times.

2 Grammar Splitting

As explained previously, program synthesis is equivalent to combinatorial search which bottlenecks on compute, this is why harnessing parallel compute environments is the key. Accordingly, we have taken care to design and evaluate extensions of our algorithms

Table 4.1: Number of programs generated

Algorithm	Number of programs generated
Heap Search	38735 prog/s
Threshold	25381 prog/s
DFS	20281 prog/s
SQRT Sampling	14020 prog/s
A^*	6071 prog/s

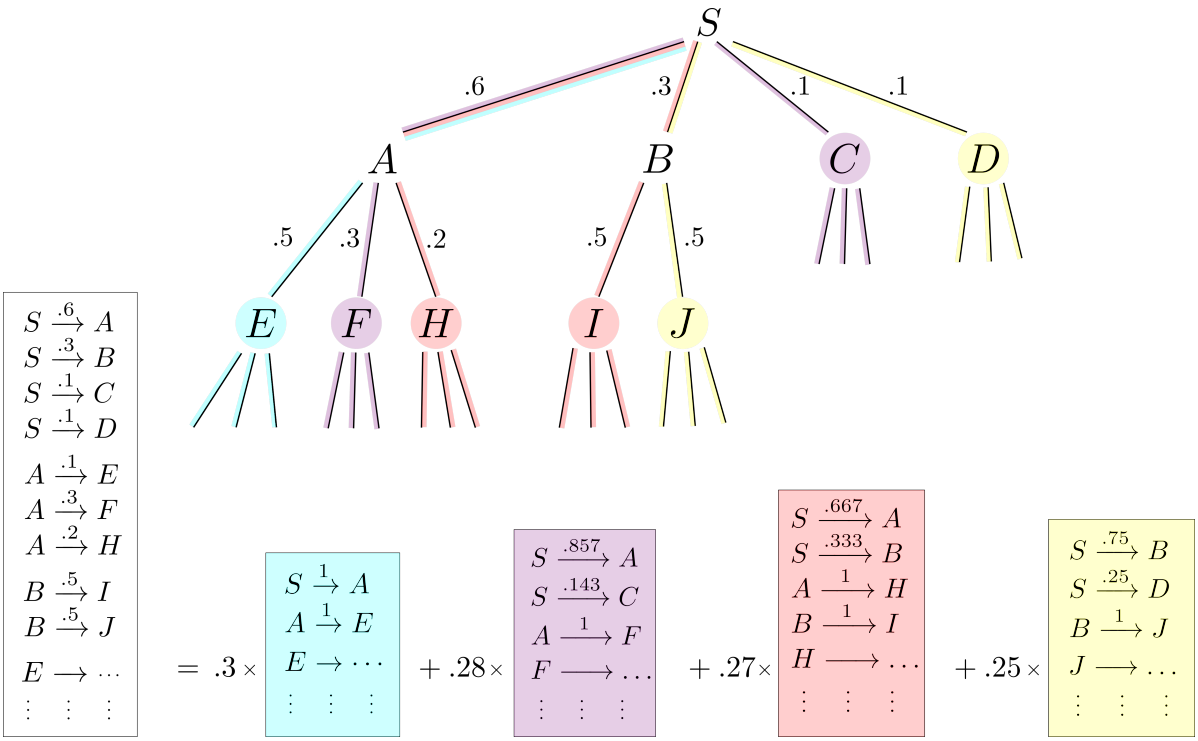


Figure 4.2: The grammar splitter: a balanced partition with quality $\alpha = \frac{.3}{.25} = 1.2$.

which can metabolize these compute resources through multiprocessing.

We introduce a new algorithm called the *grammar splitter*, which partitions a finite PCFG into a balanced family of k sub-PCFGs. We say that a PCFG is *finite* when it contains a finite number of programs. Each of the k sub-PCFG can be searched independently making it perfect for a parallel approach. In fact, the scaling is thus linear with number of sub-PCFG assuming we have a computing unit per sub-PCFG. Furthermore, this approach works with any search algorithm at no additional cost but a preprocessing step. The output of the grammar splitter is illustrated in Figure 4.2: the white PCFG is split into 4 sub-PCFGs. The nodes represent nonterminals and the numbers on the edges represent the probabilities of the derivation rules.

There are two key properties:

- *non redundancy:* the space of programs is partitioned into k subspaces. This implies that the different processors do not carry out redundant work and that all programs are generated,
- *balance:* the k program subspaces are balanced, meaning that their mass probabilities are (approximately) equal. This implies that all processors contribute equally to the search effort.

A split is a collection of partial programs, for instance `map (* 2) HOLE` and `fold + HOLE HOLE`, it induces a PCFG. A set of k incomparable splits yields a partition of the PCFG. Let us write α for the quality of a partition, defined as the ratio between the maximum and the minimum probability mass of a split. We are looking for a balanced partition, *i.e.* one for which the quality α is close to 1.

Our algorithm finds a balanced partition through a hill climbing process: at each point the algorithm either looks for an improving swap or a refinement. In the first case,

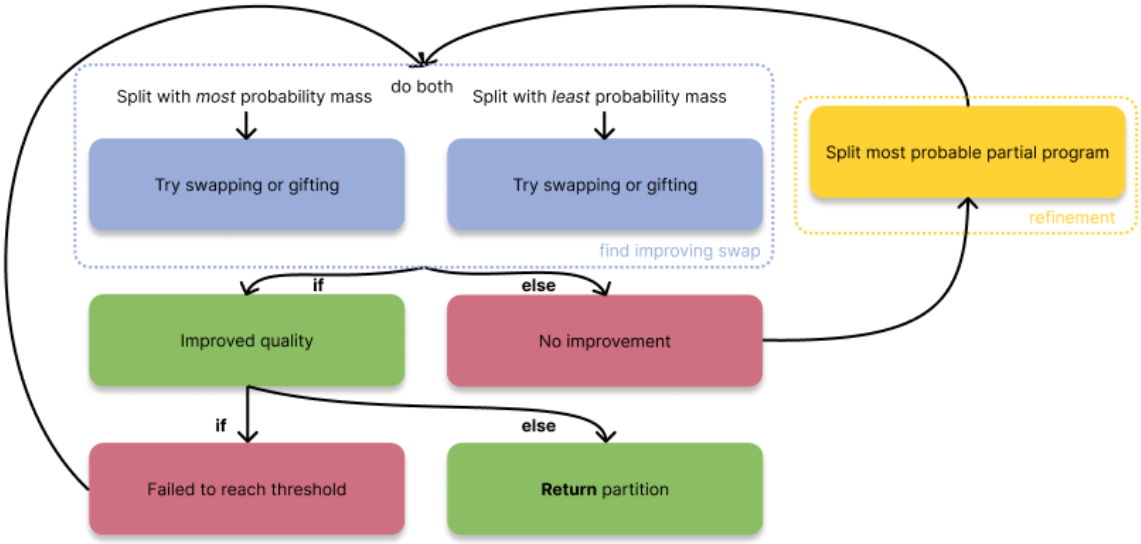


Figure 4.3: Simplified overview of the grammar splitter algorithm

the action of an improving swap is to transfer a partial program from one split to another, and its goal is to lower the quality coefficient. In the second case, we consider the partial program with maximal probability in a split and refine it: for example `map (* 2) HOLE` could be replaced by `map (* 2) var0` and `map (* 2) (filter HOLE HOLE)` enabling us to make more precise adjustments to the splits.

The pseudocode of the grammar splitter is given in Algorithm 12 using two procedures: split and find improving swap. The split procedure describes at a high level how our grammar splitter works. The find improving swap procedure is here to provide a clear method of finding an improving swap or refinement. The pseudocode is quite extensive so Figure 4.3 gives a high level overview of the loop in the algorithm.

Algorithm 12 Grammar splitter

procedure SPLIT(G: PCFG, *nsplits*: number of splits, $\alpha_{desired}$: target quality)
    Create an initial splitting SPLITS
    $\alpha \leftarrow \frac{\max_{sG \in \text{SPLITS}} \text{probability mass}(sG)}{\min_{sG \in \text{SPLITS}} \text{probability mass}(sG)}$
    while $\alpha > \alpha_{desired}$ **do**
        if an improving swap exists **then**
            Update SPLITS with the improving swap
            $\alpha \leftarrow \frac{\max_{sG \in \text{SPLITS}} \text{probability mass}(sG)}{\min_{sG \in \text{SPLITS}} \text{probability mass}(sG)}$
        else
            Find the partial program with largest probability P
            Replace P in its split with its children
        end if
    end while
    return SPLITS
end procedure

Algorithm 13 Grammar splitter: find improving swap

procedure FIND IMPROVING SWAP(G: PCFG, SPLITS)
  $\alpha \leftarrow \frac{\max_{sG \in \text{SPLITS}} \text{probability mass}(sG)}{\min_{sG \in \text{SPLITS}} \text{probability mass}(sG)}$
  $\alpha^* \leftarrow \alpha$ ▷ best improving swap α
  $s \leftarrow$ None ▷ best improving swap
  $L \leftarrow \text{ARGMAX}_{sG \in \text{SPLITS}}$ probability mass(sG)
  Sort SPLITS by increasing probability mass
  for all $sG \in \text{SPLITS} \setminus \{L\}$ **do**
    for all $P' \in L$ **do**
      for all $P \in G$ **do**
        $\beta \leftarrow$ Compute new α with SWAP(L, sG, P, P')
        if $\beta < \alpha^*$ **then**
          $\alpha^* \leftarrow \beta$
          $s \leftarrow \text{SWAP}(L, sG, P, P')$
        end if
      end for
      $\beta \leftarrow$ Compute new α with GIFT(L, sG, P')
      if $\beta < \alpha^*$ **then**
        $\alpha^* \leftarrow \beta$
        $s \leftarrow \text{GIFT}(L, sG, P')$
      end if
    end for
  end for
  $l \leftarrow \text{ARGMIN}_{sG \in \text{SPLITS}}$ probability mass(sG)
  Sort SPLITS by decreasing probability mass
  for all $sG \in \text{SPLITS} \setminus \{L, l\}$ **do**
    for all $P \in sG$ **do**
      for all $P' \in l$ **do**
        $\beta \leftarrow$ Compute new α with SWAP(l, sG, P, P')
        if $\beta < \alpha^*$ **then**
          $\alpha^* \leftarrow \beta$
          $s \leftarrow \text{SWAP}(l, sG, P, P')$
        end if
      end for
      $\beta \leftarrow$ Compute new α with GIFT(sG, l, P)
      if $\beta < \alpha^*$ **then**
        $\alpha^* \leftarrow \beta$
        $s \leftarrow \text{GIFT}(sG, l, P)$
      end if
    end for
  end for
  return s
end procedure

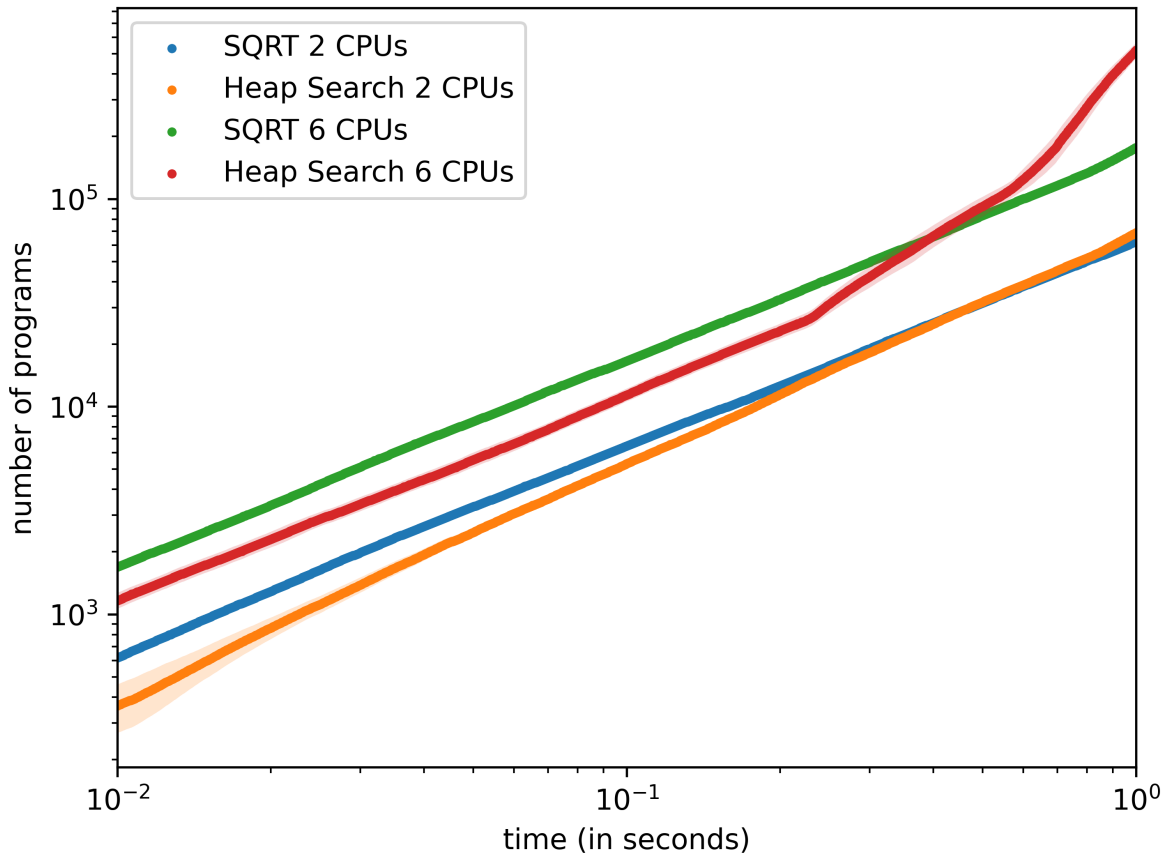


Figure 4.4: Number of programs enumerated with respect to time sued in a log-log scale

2.A Experiment

In our experiment, we initialized the splitting as follows: we split the node with highest probability until the total number of nodes is greater than the number of splits required, then assign one node to each split, and the remaining nodes to the last split. We also limited the search of an improving swap or refinement to the most probable split and the least probable split unlike in the find improving swap procedure where all splits are considered. Finally, in all of our experiments $\alpha_{desired} = 1.05$.

We take five random PCFGs and measure the speed-ups obtained with the grammar splitter. We do not report on running SQRT Sampling in parallel which would simply sample using the same PCFG on multiple CPUs. Indeed this naive approach performs poorly in comparison, since thanks to the grammar splitter two CPUs cannot generate the same program. We report the results on Figure 4.4 and observe the linear speed-up obtained is the same as the one claimed. The results are shown in Figure 6, where we count programs with repetitions. We see that for SQRT Sampling the scale-up is linear, and it is mostly linear for Heap Search with an acceleration from the 0.2s mark. This acceleration can be explained in two ways: first, each sub-PCFG is shallower since it is split thus it is faster to enumerate program from it, second, once all the successors have been computed Heap Search is a simple lookup table. At the end of the experiment, SQRT Sampling has generated 2.8 times more programs with 6 CPUs than with 2 CPUs, whereas Heap Search has generated 7.6 times more programs with 6 CPUs than with 2 CPUs.

2.B Futures

The main issue with this approach is that this does not apply to grammars which are not finite. Indeed, in these grammars, you can loop to the same nonterminal, which is the source of the infinite number of programs. Perhaps there are reasonable adaptions that sacrifice part of the speed-up but still enable to parallelise part of the process. Inspirations from work on regular languages [AM23] in which they split regular languages into disjoint languages is a good start point. This is an interesting direction of future work.

Chapter 5

Runtime Filtering

TL;DR 5

Most grammars contain semantically redundant but syntactically different programs, while these programs are eliminated thanks to observational equivalence techniques. These programs are enumerated and evaluated for each task whereas most of these redundancies are due to properties of the DSL not of the task. We propose checking semantic equivalences of the programs of a DSL, this enables us to build a tree automaton that will be able to filter these programs. We introduce a process to generate automatically these equivalent programs and then compile them into a tree automaton. Finally, this tree automaton is transformed into an unambiguous CFG which is enumerated. This leads us to propose a pre-processing step for a grammar, that can be done once and for all, that enables at runtime to speed up the search by only enumerating non redundant programs without evaluating them.

Facing an application domain, the first task in designing a program synthesis approach is to define a programming language. In most cases, a domain specific language (DSL) is preferable over a generic one such as Python or C. Indeed, by further specifying what constructs can be used, we make the program synthesis problem easier: the space of programs is smaller, hence more manageable. This observation led some years ago to the celebrated notions of program sketching [Sol08, Sol13] and templates [SGF13], which revolutionised modern program synthesis: the idea is to restrict the program by specifying its syntactic structure and leaving holes to be filled. This is task specific: each task requires writing a dedicated sketch or template. The goal of this paper is to push this idea further: *runtime filtering works at the DSL level in order to reduce the space of all programs*. This work will be submitted soon.

Our contributions:

- A method to automatically generate equations between syntactically different but semantically equivalent program from a DSL. It relies on techniques and insights from rewriting theory.
- A method that compiles equations of a DSL to filter syntactically redundant programs at runtime in constant time. It is rooted in regular tree languages and tree automata models.

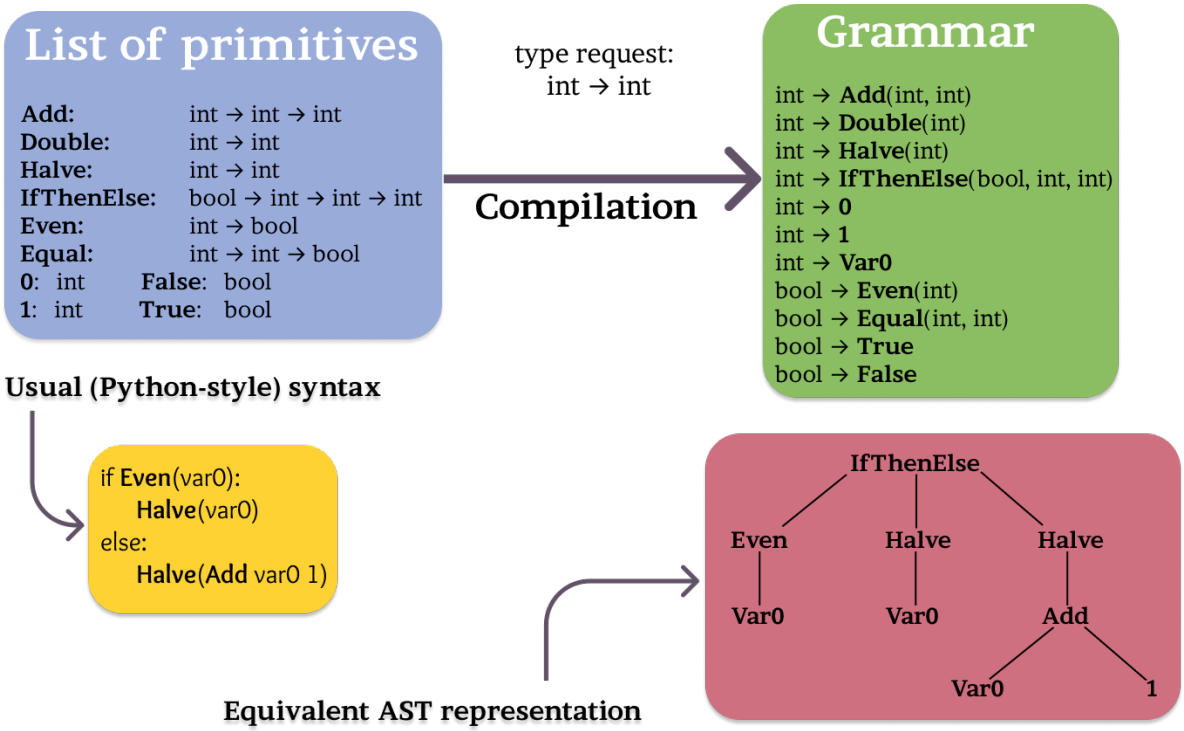


Figure 5.1: A simple grammar for representing a DSL. In this simplified example, the two non-terminals of the grammar are the atomic types 'int' and 'bool'.

- Combining both methods enables us to have a pre-processing algorithm that can prune a large part of observational equivalent programs without the need for evaluation
- We perform an experimental evaluation. The first part of the experiments focus on the gain of runtime filtering in isolation and the second integrates these methods in a program synthesis pipeline evaluated in two application domains.

To define a DSL the user specifies a set of primitives to be used. We refer to Figure 5.1 for an illustration: there are 10 primitives (4 are constants) for manipulating integers and boolean values, spelled out on the left hand side. Let us further specify that we are interested in programs of type `int` → `int`; to represent all such programs the long-adopted approach is to use a context-free grammar (CFG), on the right hand side. An example program is given in the bottom of the figure, on the left in a standard syntax, and on the right as a tree generated by the grammar (called Abstract Syntax Tree, AST). We call *compilation* the process of constructing the grammar representing all programs. Although rather transparent on the example above because the grammar only verifies that programs type correctly, it can become much more involved as we will see.

Runtime filtering.

As illustrated above, the first property ensured by the grammar is that it only generates well-typed programs. However, the grammar also generates a lot of useless or redundant programs. Let us illustrate this with some examples, where we use `P`,`Q`,`R` to represent generic programs:

1. The program `0+1` is useless: it does not use the input variable `Var0`;

2. The program `Double(Halve(P))` is redundant: it is equivalent to `P`;

3. The programs `Add(Add(P,Q),R)` and `Add(P, Add(Q,R))` are equivalent, so one of them could be disposed of;

4. The programs `Add(P,Q)` and `Add(Q,P)` are equivalent.

The starting point of this work is the (experimental) observation that the redundancy in the space of programs significantly slows down program synthesis tools. This observation was made in several places in the program synthesis literature, see e.g. [BHD+18, CLS19, SA19] and the related works section. To address this, let us define *runtime filtering* as the task of improving a DSL by reducing as much as possible the number of useless and redundant programs. Runtime filtering is meant as a preprocessing step, to be performed once for each DSL. Our goal in this paper is to define methods for runtime filtering and evaluate them experimentally.

Rules. We illustrate our rules-based approach on the examples above.

1. Requiring that the input variable `Var0` must be used at least once rules out the program `0+1`;

2. Forbidding the pattern `Double(Halve)` rules out `Double(Halve(P))`;

3. Forbidding the pattern `Add(_,Add)` rules out programs associating addition to the right;

4. Choosing between `Add(P,Q)` and `Add(Q,P)` would imply ordering all programs, which context-free grammars cannot do. As we will see, our techniques cannot handle this redundancy.

In this paper, we consider two scenarios for specifying rules: either they are input by a user together with the set of primitives, or they are automatically generated. We will make use of the notions of CFG in this work, we invite the reader to freshen up on that if there is an issue in comprehension.

1 A Motivating Example

Let us consider a DSL consisting of Boolean formulas. The primitives are the binary operators `And`, `Or` and the unary `Not`, and the Boolean variables are `Var0`,`Var1`. Let us fix bool $\to$ bool as type request, then the class of programs of this type is generated by the following grammar with a single non-terminal bool:

$$\text{bool} \Longrightarrow \texttt{And}(\text{bool}, \text{bool}) \mid \texttt{Or}(\text{bool}, \text{bool}) \mid \texttt{Not}(\text{bool}) \mid \texttt{Var0} \mid \texttt{Var1}$$

This grammar generates a lot of redundant programs: one of many examples is $\texttt{And}(\texttt{Var0}, \texttt{Var0})$, which is equivalent to `Var0`. As a first step towards simplifying this grammar, we would like to enforce that all formulas are in Conjunctive Normal Form (CNF). It would not be

too hard to directly write a CFG generating such formulas, but our approach is instead to write rules:

$$\mathcal{G}\neg(\texttt{Or And \_}) \quad ; \quad \mathcal{G}\neg(\texttt{Or \_ And}) \quad ; \quad \mathcal{G}\neg(\texttt{Not } \{\texttt{ And}, \texttt{Or }\})$$

The first two rules specify that (globally, meaning anywhere in the program) `And` cannot be an argument of `Or` and the third one that `And` and `Or` cannot be arguments of `Not`. The output of a compilation algorithm using these two rules could be the following grammar, which generates formulas in CNF.

$$\begin{array}{lcl} \text{bool}_1 & \Longrightarrow & \texttt{And}(\text{bool}_1, \text{bool}_1) \mid \texttt{Or}(\text{bool}_2, \text{bool}_2) \mid \texttt{Not}(\text{bool}_3) \mid \texttt{Var0} \mid \texttt{Var1} \\ \text{bool}_2 & \Longrightarrow & \texttt{Or}(\text{bool}_2, \text{bool}_2) \mid \texttt{Not}(\text{bool}_3) \mid \texttt{Var0} \mid \texttt{Var1} \\ \text{bool}_3 & \Longrightarrow & \texttt{Not}(\text{bool}_3) \mid \texttt{Var0} \mid \texttt{Var1} \end{array}$$

There are still a lot of equivalent programs generated by this grammar. One could consider the following rule further reducing symmetries:

$$\mathcal{G}\neg(\texttt{And And \_})$$

It specifies that (globally) the first argument of `And` cannot be `And`, in other words that `And` is associated to the right: the formula $\texttt{And}(\texttt{And}(\varphi_1, \varphi_2), \varphi_3)$ is replaced by $\texttt{And}(\varphi_1, \texttt{And}(\varphi_2, \varphi_3))$. Many other properties can be ensured, for instance that `Or` associates to the right, that `Not` is not composed twice, and so on.

The first point of this example is to show that simple rules can significantly reduce the number of redundant programs found in a DSL. Although one could directly write a CFG ensuring some simple properties such as CNF and associativity of `And` to the right, it becomes very quickly tedious from a user's perspective, especially when combining many properties on large CFGs. Instead, we believe that it is relatively easy to write rules. Our compilation algorithm (see Section 4) constructs a minimal CFG from the set of primitives and the set of rules. Better still: some rules can be automatically generated, as we will do in Section 3.

This example also illustrates that rules can be of two different types. The first set of rules aim at enforcing a normal form (here CNF), which is a design choice of the user. The last rule is derived from an equation: $\texttt{And}(\texttt{And}(\varphi_1, \varphi_2), \varphi_3) \equiv \texttt{And}(\varphi_1, \texttt{And}(\varphi_2, \varphi_3))$, which holds for all formulas. We will automate the generation of the second type of rules.

2 Syntax

The syntax we propose is conceptually similar to sketches [Sol08,Sol13] and templates [SGF13], with some differences that we discuss below. Let us write Γ for the set of primitives and variables. Rules are manipulating sets of primitives:

$$\text{NSet} \quad \doteq \quad \{f_1, \ldots, f_k\} \quad \mid \quad \_$$

where $f_1, \ldots, f_k$ are from Γ and $\_$ represents any symbol. As a shorthand we write f instead of $\{f\}$. Rules are defined using the following grammar:

$$\begin{array}{rcl} \text{Rules} & \doteq & (\text{NSet} \quad \text{Rules}_1 \ldots \text{Rules}_k) \\ & \mid & \mathcal{G}\ \text{Rules} \\ & \mid & \neg\text{Rules} \\ & \mid & \#[\text{Rules}]_{\geq N} \\ & \mid & \#[\text{Rules}]_{\leq N} \end{array}$$

where k and N are constant integers. For the special case where $k = 0$ we remove the parenthesis: $\{f, g\}$ instead of $(\{f, g\})$.

Let us give intuitive definitions first. The rule $(f\ \{g, h\}\ \_)$ specifies that the program starts with f and that the first argument of that particular f is g or h, not constraining the second argument. Adding $\mathcal{G}$ makes the rule *global*: the rule $\mathcal{G}\neg(f\ \{g, h\}\ \_)$ specifies that for each occurrence of f, its first argument cannot be g nor h. The rule $\#[\texttt{Var0}]_{\geq 1}$ specifies that `Var0` appears at least once. The rule $\#[\_]_{\leq 10}$ says that the whole program has size at most 10.

Formally, the semantics of a rule R is a set of programs $[\![R]\!]$, defined inductively. For a program P, we write $P' \sqsubseteq P$ if P' is a subprogram of P.

For a rule $R = FR_1 \dots R_k$:
$[\![R]\!] = \{P = f(P_1, \dots, P_k) : f \in F \text{ and } P_i \in [\![R_i]\!]\}$.
For a rule $R = \neg R'$:
$[\![R]\!] = \{P : P \notin [\![R']\!]\}$.
For a rule $R = \mathcal{G}\ R'$:
$[\![R]\!] = \{P : \forall P' \sqsubseteq P, P' \in [\![R']\!]\}$.
For a rule $R = \#[R']_{\geq N}$,
$[\![R]\!] = \{P : |\{P' \sqsubseteq P : P' \in [\![R']\!]\}| \leq N\}$.
For a rule $R = \#[R']_{\leq N}$,
$[\![R]\!] = \{P : |\{P' \sqsubseteq P : P' \in [\![R']\!]\}| \geq N\}$.

The design of our syntax was guided by simplicity; although expressive enough for most use cases, it could be extended. As a counterexample: the (arguably not very useful) rule that some primitive f appears an even number of time cannot be written in our syntax. To put this remark in a wider perspective: as we will show in Section 4, all rules defined in our syntax induce regular tree languages, but conversely some regular tree languages cannot be defined in our syntax. For completeness, in our implementation we also allow the user to input rules directly as CFGs, so that rules can express all regular tree languages.

Let us compare our syntax to sketches and templates. They all follow the same principles: a partial program is specified, and the holes can be filled by programs drawn from a given regular expression. In sketches, the regular expressions are limited: they only use concatenation, choice, and optional, but no Kleene star nor counting. Our syntax has a higher expressivity thanks to the global semantics and counting capabilities, and the natural interleaving between global and local rules. However, let us emphasise that the main difference between our work and program sketching and templates is not in the exact syntax, but rather in their applications: sketches and templates are task-specific, while the aim of our rules are to work at the DSL level to reduce the space of all programs, as a preprocessing step.

3 Generating Valid Equations

Some rules are design choices to be made by the user, for instance the rule $\#[\texttt{Var0}]_{\geq 1}$ specifying that `Var0` appears at least once, or the rule $\#[\_]_{\leq 10}$ limiting the size of the whole program to 10. However, a great number of rules can be automatically generated:

those induced by *equations*. Formally, an equation is a pair of equivalent programs, for instance `Double(Halve(Var0))` $\equiv$ `Var0`. Extrapolating, we obtain that for all programs `P` of output type bool, we have that `Double(Halve(P))` $\equiv$ `P`. Here it is clear that it `P` is always preferable to `Double(Halve(P))`. To materialise this, we create a rule rejecting the latter but not the former, which here is naturally:

$$\mathcal{G}\neg(\texttt{Double Halve})$$

However not all equations can be oriented in this way: for instance the equation `Add(Var0,Var1)` $\equiv$ `Add(Var1,Var0)` says that addition is commutative. One can show that orienting this equation would imply defining a total order over all programs. More generally, turning equations into a terminating and confluent rewriting system is one of the oldest and most fundamental problem in rewriting theory and still actively investigated nowadays: we refer to the excellent textbook for references and classical algorithms [BN98]. Here we do not tackle the general problem as we will not be able to turn all equations into rules, our focus is to give a practical solution in our specific framework.

Let us first describe our approach at a high level. The first step is to define a partial well order on all programs, meaning an order which does not contain infinite descending chains. Intuitively, rules forbid programs which have smaller equivalent programs, hence the absence of infinite descending chains imply termination. Our algorithm follows three steps:

1. We enumerate all programs where each variable appears at most once, up to some fixed depth and some fixed number of variables;
2. We check for program equivalence amongst all generated programs;
3. For each equation found where one program is larger than the other one, we add a rule to forbid the larger program;

We now give more technical details.

A partial well order on programs. We need an order on programs in order to decide whenever presented with an equation $P \equiv Q$ which of P or Q we want to forbid. Let us introduce an order on Γ the set of primitives and variables. First, we consider an arbitrary total order on primitives, and augment it by stipulating that variables are minimal elements. Importantly, all variables are incomparable. Writing `P1,P2,`$\dots$`,Pn` the set of primitives, the order on Γ is:

$$\left.\begin{array}{l}\texttt{Var0}\\ \texttt{Var1}\\ \dots\end{array}\right\} < \texttt{P1} < \texttt{P2} < \cdots < \texttt{Pn}$$

We then define a partial order on programs lexicographically, where $|P|$ is the size of a program, meaning its number of primitives and variables:

$$P < P' \text{ if } \begin{cases} |P| < |P'|, \text{ or} \\ |P| = |P'|, \text{ let us write } P = f(P_1, \dots, P_k), P' = g(P'_1, \dots, P'_\ell) \\ f < g, \text{ or} \\ f = g, P_1 = P'_1, \dots, P_{i-1} = P'_{i-1}, P_i < P'_i \end{cases}$$

Lemma 5.1

The partial order $<$ is well-founded and context-preserving.

- **Well-founded**. There are no infinite descending chains of programs with respect to $<$, meaning $P_1 > P_2 > P_3 > \dots$.
- **Context-preserving**. For all programs $P < P'$ and Q a program with a single hole, then $Q[P] < Q[P']$, where $Q[P]$ is the program obtained by replacing the hole in Q by P, and similarly for P'.

Proof. We prove well-foundedness. Towards contradiction, consider an infinite descending chain. First look at the sequence of sizes: it is a non-increasing sequence of natural numbers, hence eventually constant. Thus without loss of generality we can assume that all programs in the infinite descending chain has the same size. However the lexicographic order induced by a partial order is a well partial order over trees of fixed size, a contradiction.
The context-preserving property is clear. □

Enumeration of small programs. The exact task at hand is the following: we fix ourselves a target size and a target number of variables, and set to generate all programs where each variable appears at most once of size and number of variables under the respective targets. We use off-the-shelves enumeration algorithms for deterministic grammars, and since the goal is to generate a few hundred or thousand programs we do not need advanced data structures or algorithms.

Equivalence checking. Once we have a set of programs, we check whether any of them are equivalent. Program equivalence may be hard to solve, but it can be efficiently approximated, or at least it is in practice easy to determine that two programs are *not* equivalent: we evaluate the programs on a set of inputs that is either sampled or scrapped from a dataset. If two programs agree on all inputs, depending on the DSL we can use either an SMT solver or a theorem prover to determine whether the programs are indeed equivalent. Note that false positives, meaning pairs of non-equivalent programs declared equivalent, would lead to unsound equations. Each pair of (properly) equivalent programs yields a valid equation.

Turning equations into rules. To turn the equation $P \equiv P'$ into a rule, we need to determine whether $P < P'$ or $P' < P$. Note that it may be the case that neither of these hold: for instance, in the equation `Add(Var0,Var1)` $\equiv$ `Add(Var1,Var0)` there are no orders between `Var0` and `Var1`. If $P < P'$, we create a rule of the form $\mathcal{G}\neg P'$. On an example: start from the equation, `Add(Var0,Add(Var1,Var2))` $\equiv$ `Add(Add(Var0,Var1),Var2)` the lefthand term is smaller than the righthand term (because `Var0 < Add`), so this induces the rule $\mathcal{G}\neg$(`Add Add _`).

We emphasise here a subtle technical point: we only generate programs where each variable appears once because equations such as `Var0-Var0` $\equiv$ `0` cannot be encoded in CFGs. Indeed, this equation requires checking that the two arguments of `-` are equal, much beyond the power of CFGs (more precisely: the corresponding set of trees is not a regular tree language, see the next section for definitions).

Theorem 5.2

The algorithm is sound and semantic-preserving. Let $\mathcal{C}$ the class of programs over the set of primitives and variables Γ. We let $\mathcal{E}$ denote the class of equations produced by the algorithm, and $\mathcal{R}$ the class of rules induced by the equations. We write $[\mathcal{C}]_{\sim R} = \{P \in \mathcal{C} : \forall R \in \mathcal{R}, P \in [\![R]\!]\}$ the class of programs modulo the rules in $\mathcal{R}$.

- **Sound**. Every equation of $\mathcal{E}$ is valid in $\mathcal{C}$.
- **Semantic-preserving**. For every program $P \in \mathcal{C}$, there exists an equivalent program $P' \in [\mathcal{C}]_{\sim R}$.

However, the algorithm is not complete, in the sense that it does not remove all equations: they may exist $P, P' \in [\mathcal{C}]_{\sim R}$ with $P \equiv P'$.

Proof. Soundness is by construction of the algorithm. To prove the semantic-preserving property, we define a rewriting system: for each equation $P \equiv P' \in \mathcal{E}$ where $P < P'$, we define the rewrite rule $P' \hookrightarrow P$. This rewriting system preserve equivalence of programs, and it is terminating because the order $<$ is well founded and context-preserving. For each program $P \in \mathcal{C}$, let us write $[P]$ for the normal form of P with respect to this rewriting system. Then P is equivalent to $[P]$ and $[P] \in [\mathcal{C}]_{\sim R}$. □

4 Compilation Algorithm based on Tree Automata

We start this section by a discussion on what the compilation algorithm is expected to achieve. As we will see, this will condition the class of CFGs we will use. We have defined above (see Chapter 1) two grammar models: deterministic and non-deterministic, abbreviated det-CFG and ND-CFG. Let us introduce a third one: we say that a CFG is unambiguous (*U-CFG*) if it is non-deterministic but satisfies that all trees have at most one derivation, that it is to say, as explained previously it can locally ambiguous despite being non ambiguous. Clearly, a deterministic CFG is also unambiguous, but the converse does not hold: unambiguity makes CFGs much more expressive. For instance, there is no det-CFG generating all trees of size at most 15, but there is a U-CFG doing that. Let us list the properties we require from a grammar model to be useful in our compilation algorithm.

Expressiveness. We require the existence of an algorithm for compiling the rules defined in our syntax into grammars. We have

$$\text{det-CFG} \subsetneq \text{U-CFG} = \text{ND-CFG}.$$

Further, U-CFG and ND-CFG recognise the well studied and robust class of regular tree languages. In that sense, det-CFGs are strictly less expressive than the other two models, which are equivalent. A concrete example witnessing the strict inclusion: det-CFGs cannot express 'the set of trees of size at most 15'. This is a strong blow on the expressivity requirement, and it disqualifies deterministic grammars for our compilation algorithm, since we cannot compile the rule $\#[\_]_{\leq 15}$. Note that a finer analysis would include succinctness: indeed all tree languages realised by a ND-CFG can be realised by a U-CFG, but they may be exponentially larger.

Model	Expressive	Minimisation	Enumeration
det-CFG	✗	✓	✓
U-CFG	✓	✗	✓
ND-CFG	✓	✓	✗

Minimisation. We require the existence of an algorithm transforming a grammar into an equivalent minimal one. This is necessary for compiling a large number of rules and avoiding combinatorial explosion, furthermore as seen in Chapter 3, enumerative approaches struggle with more complex grammars. Det-CFGs can be efficiently minimised with textbook algorithms, but no minimisation algorithm is known for U-CFGs and an efficient algorithm is unlikely to exist: minimisation of unambiguous automata over words is already NP-complete [BM08].

Enumeration. We require the existence of efficient search algorithms applicable to program synthesis. Typical approaches include enumeration, constraint solving, and sampling. Most existing algorithms [MTG+13, BGB+17, FMBD18, ZRF+18, FLM+22] assume grammars to be either deterministic or unambiguous, and become very inefficient for non-deterministic CFGs. Intuitively, this is because they all enumerate derivation trees. For both deterministic and unambiguous grammars, derivation trees are in one-to-one correspondence with programs, but for non-deterministic grammars many derivation trees can correspond to the same program, inducing a lot of redundant work.

The table above summarises the situation. The situation seems desperate, but it is not: our key insight is to use a different model. Deterministic bottom-up tree automata (DBTA), will be used to compile rules. Once all rules are compiled into a DBTA, we transform it into a U-CFG which we will enumerate. HeapSearch was defined for det-CFGs. Its extension to U-CFGs is not very complicated, although computationally slightly more expensive. Let us quickly define a DBTA.

Definition 5.3 ▶ Deterministic Bottom-up Tree Automaton (DBTA)

A *deterministic bottom up tree automaton* A on the alphabet Σ is a tuple (S, Δ, F) where:

- S is the set of states
- $F \subset S$ is the set of final states
- Δ is a set of transition functions for each arity of the letters in Σ. For $\delta_k \in \Delta$ of arity k, we have $\delta_k : \Sigma \times S_1 \times \cdots \times S_k \to S$ as the transition function.

A tree is accepted if by using rewriting rules of appropriate arity from Δ in a bottom-up manner it can be rewritten as a state $s \in F$. Since we consider here deterministic automata, each tree has at most one run.

Example 5.4 ▶ DBTA example

Let us look build a DBTA from a CFG. Here is the grammar for boolean expressions of three variables:

$$S \to \wedge\ S\ S \qquad S \to \vee\ S\ S$$
$$S \to \neg\ S \qquad S \to x_1$$
$$S \to x_2 \qquad S \to x_3$$

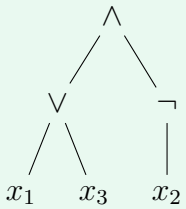
Building the DBTA that recognises this grammar is quite easy, here are the transitions:

$$S \leftarrow \wedge\ S\ S \qquad S \leftarrow \vee\ S\ S$$
$$S \leftarrow \neg\ S \qquad S \leftarrow x_1$$
$$S \leftarrow x_2 \qquad S \leftarrow x_3$$

We have one state S that is final.

Example 5.5 ▶ DBTA parsing

Let us take the DBTA from Example 5.4 and parse the following expression that does belong to our grammar:



First, we start from the leaves and apply the transition function:

then we apply transitions depth by depth according to the state in the leaves:

which leaves us with the single state S since S is final, this tree is recognised by the automaton.

We construct a compilation algorithm using Deterministic Bottom-up Tree Automata (DBTA). The technical properties of DBTA that we will rely on are the following:

- DBTA are expressive enough to express all rules.
- DBTA can be effectively minimised.
- Reversing a DBTA yields an equivalent (finite family of) U-CFG, see Lemma 4.
- There are efficient enumeration algorithms for U-CFGs.

The theory and practice of tree automata has reached a mature level, in particular the following are by now standard results (we refer to [CDG+08] for an excellent textbook).

Theorem 5.6 ▶ Essential Properties of DBTA [CDG+08]

DBTA recognise the class of regular tree languages, in particular they are closed under all Boolean operations. They can be efficiently minimised.

More concretely, our compilation routine is based on the following lemma.

Lemma 5.7

Let Γ be a set of primitives and variables, and τ a type request.

- There exists a DBTA A such that $L(A)$ is the set of all programs using Γ of type τ.
- For every rule R, there exists a DBTA A such that $L(A) = [\![R]\!]$.

The proof of the lemma, consisting in encoding rules in DBTAs, is a routine work that we do not expand further on here. Rules are compiled one by one as illustrated in Algorithm 14, interleaved with the minimisation algorithm to keep the automaton small. Let us show why DBTAs are as expressive as U-CFGs by showing how to go from one to

Algorithm 14 Compilation algorithm

```
procedure Compilation(Γ, τ, R)
    A ← DBTA representing all programs using Γ of type τ
    for R ∈ R do
        A_R ← DBTA representing [[R]]
        A ← A ∧ A_R
        Minimise A
    end for
    return A
end procedure
```

the other.

Lemma 5.8 ▶ Equivalence between DBTA and U-CFGs

Given a DBTA, we can compute a finite family of U-CFGs such that the languages of the U-CFGs are pairwise disjoint and their union is the language of the DBTA.

The idea is to 'reverse' the DBTA: a DBTA processes the tree from leaves to the root, by reverting the arrows we obtain a non-deterministic process generating trees from the root to the leaves. The fact that the bottom-up process is deterministic implies that each

tree has at most one derivation, hence the non-deterministic top-down process is actually unambiguous.

> *Proof.* For each accepting state of the DBTA, we construct a U-CFG as follows. The set of non-terminals is the set of states of the DBTA, the initial non-terminal is the accepting state under consideration, and the rules are:
>
> $$q \to f(q_1, \ldots, q_k) \quad \text{when} \quad \delta(q_1, \ldots, q_k, f) = q.$$
>
> Derivations of the constructed U-CFGs are in one-to-one correspondence with runs of the DBTA. □ □

We therefore obtain a compilation algorithm satisfying all three requirements discussed earlier:

- The compilation of rules are performed on DBTAs.
- Minimisation are performed on DBTAs.
- The DBTA is 'reversed' into a finite family of U-CFGs
- Enumeration is performed on these U-CFGs.

5 Experiments

We perform experiments to answer the following questions:

(Q1) How efficient is runtime filtering in removing redundant and useless programs from a DSL?

(Q2) How much automatically generated rules contribute to the efficiency of runtime filtering?

(Q3) Does runtime filtering enable better performance in a program synthesis pipeline?

Environment. We use the same setup as previously, see Chapter 2, using Heap Search as the enumeration algorithm.

List programming. We re-implemented DreamCoder's list manipulation DSL [EWN+21], it has 33 primitives. It is inspired by DeepCoder's DSL [BGB+17], a classic in program synthesis. The original dataset contains 217 tasks, we keep a subset of 126 tasks that can be solved with our DSL by programs of depth at most 5.

Tower Building. We re-implemented DreamCoder's tower building DSL [EWN+21], it has 24 primitives. Our DSL is quite different, we illustrate it on an example program:

```
IfY (True):
    IfX (x >= y):
        Block(1x3)
```

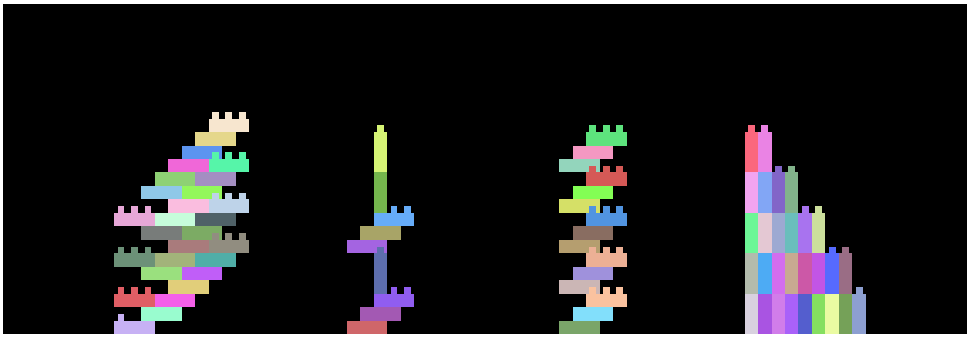


Figure 5.2: Illustration for the Towers DSL: four randomly generated towers.

This program does not follow the exact syntax used in the code but it shows the general structure. We impose that programs start with `IfY`, followed by `IfX`. To evaluate a program, we iterative run the program on all X values, and then on all Y values, yielding the following program (n is a parameter for the size of the tower):

```
For(y in range(n)):
    If (True):
        For(x in range(n)):
            If (x >= y):
                Block(1x3)
```

This program builds a tower similar to the right most one in Figure 5.2.

In all experiments we compare two compilation algorithms: the first is a baseline without rules, that we call det-CFG, and the second uses rules and the compilation algorithm through DBTA, we refer to it as U-CFG since it has the same expressivity despite being different objects conceptually they are the same here.

5.A Runtime filtering in Isolation

Towards answering (Q1), we compare the number of programs in both DSLs. To better appreciate scalability issues we perform this experiment for different values of maximum depth of the programs. Towards answering (Q2) and understanding the contribution of automatically generated equations, we perform an ablation study. The results are presented in Table 5.1 and Table 5.2, more statistics are available in the appendix.

Table 5.1: Number of programs and respective proportions (prop.) with respect to maximum depth in the List Programming DSL with type 'int list → int list'.

depth	det-CFG (no rules)	U-CFG (prop., no equations)	U-CFG (prop.)
3	8.77e+04	1	0.83
4	3.34e+16	0.99	0.51
5	9.20e+52	0.98	0.13
6	4.79e+165	0.93	0.0015
7	4.14e+510	0.78	10^{-9}

Table 5.2: Number of programs and respective proportions (prop.) with respect to maximum depth in the Tower Building DSL.

depth	det-CFG (no rules)	U-CFG (prop., no equations)	U-CFG (prop.)
3	9.12e+03	0.047	0.047
4	6.20e+09	0.017	0.0078
5	7.80e+21	10^{-5}	10^{-6}
6	2.81e+46	10^{-8}	10^{-12}
7	7.24e+95	10^{-16}	10^{-25}

Interpretation The simplification power of runtime filtering increases with depth. This is explained by the fact that larger programs are more likely to be simplified by rules. For the List Programming DSL in Table 5.1, there are only equations. However for the Tower Building DSL in Table 5.2, most of the reduction is driven by user-defined rules since we have strong requirements on the structure of the program. The contrast is drastic between the Tower Building DSL where rules dictate a very strict structure for programs enabling powerful runtime filtering and the List Programming DSL where there is no structure and only equations implying a smaller impact of runtime filtering.

5.B Runtime filtering for Program Synthesis

Including runtime filtering in the program synthesis pipeline reveals the multi-dimensionality of (Q1): removing redundant program comes at the price of constructing larger (in the number of nonterminals) grammars, hence potentially slower enumeration. The adaptation of Heap Search to U-CFGs is quite straightforward but requires memorising derivations used and therefore is slower.

In the Programming by Example scenario, a task is a set of pairs of one input and one output, and the goal is to find a program satisfying all pairs. To train our prediction model we randomly generate a dataset of 2,500 tasks. We then train one prediction model. We try solving the 126 selected tasks (all tasks that can be solved on the List Programming DSL with maximum depth 5) with a timeout of 60 seconds for each task.

Results and interpretation. On Figure 5.3, we plot the number of tasks solved by our models with respect to cumulative time. Out of 126 tasks, only 106 are solved by either of the two models. While it takes the U-CFG model close to 450s to solves 104 tasks it takes more than 800s for the det-CFG model to solve 97 tasks. So not only does **runtime filtering enables solving more tasks, but it also enables solving them faster.** We observed that the det-CFG enumerates about 450,000 programs per minute, whereas this number drops to 360,000 programs per minute for the U-CFG. This slower enumeration speed can be explained by the fact that the U-CFG grammar is larger and that Heap Search is slower over U-CFGs than over det-CFGs. Despite a slower enumeration speed, removing redundant programs pays off for wallclock comparisons.

Figure 5.4 further explains the large gap between the two models. Although the enumeration speed of the U-CFG model is 80% that of the det-CFG (450k vs 360k), it enumerates only 4 times less programs. Thus despite enumeration being slower for U-CFGs there is a clear speed-up when solving tasks using the U-CFG grammar. We note that this observation already holds at depth 5, suggesting that it would be even more

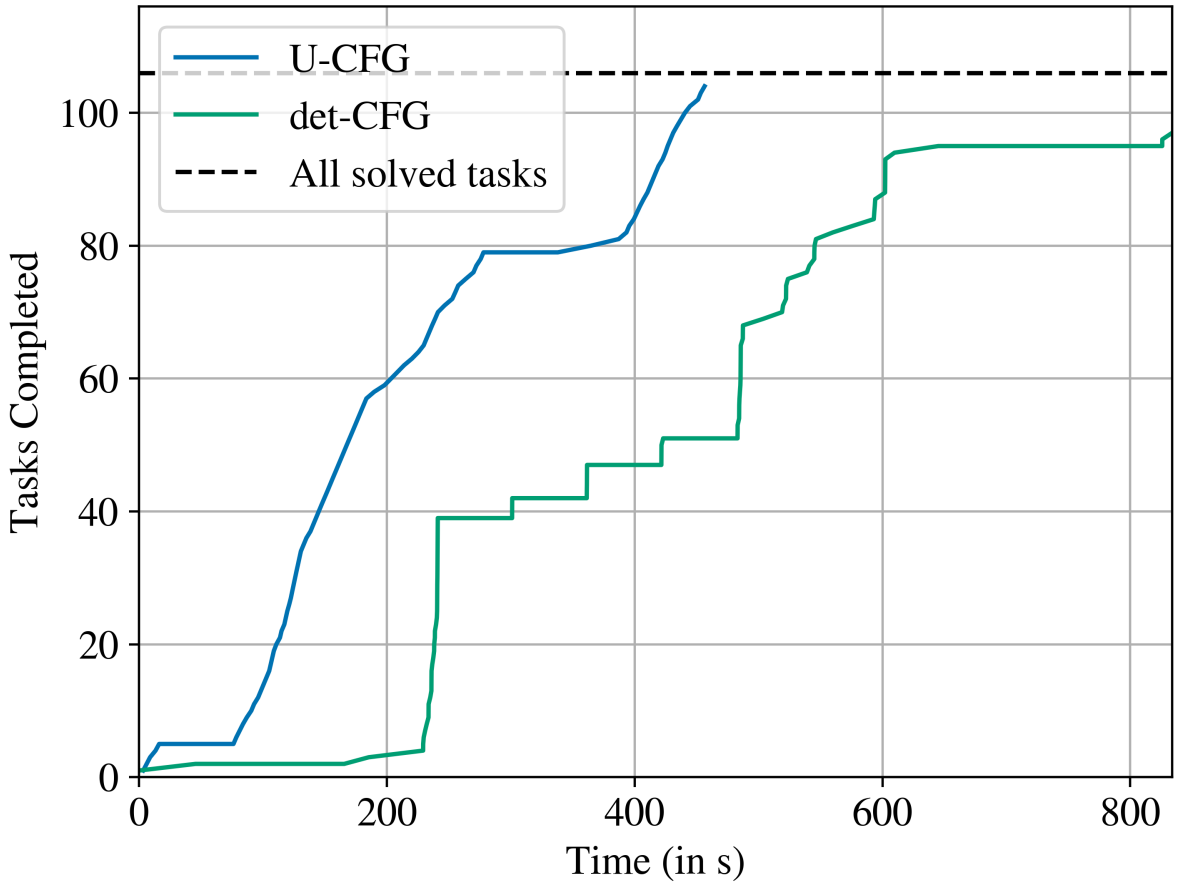


Figure 5.3: Number of tasks solved with respect to cumulative time on the set of all solved List Programming tasks

important for larger depths (reachable using larger timeouts).

6 Related work

The main inspiration for this work is the idea of program sketching [Sol08] and templates [SGF13], which brought to program synthesis the crucial idea of putting syntactic restrictions on the program to be found. Whereas these techniques are task-specific, our approach works at the level of the DSL and as a preprocessing step: the goal is to construct a reduced DSL with less redundant and useless programs. In that sense, it is closer to DSL learning, which is the task of learning a DSL from basic principles: this very ambitious goal was spearheaded by DreamCoder [EWN+21]. But runtime filtering does not invent new primitives in the way that DSL learning does: it rather inspects the interactions of the existing primitives in order to optimise their compositions. Other works have considered automatically generating equations satisfied by programs [SFA17, CSH10]. The specificity of our approach is to consider rules that can be compiled in grammars. Smith and Albarghouthi [SA19] make use of much more complicated equations directly within the search algorithm. Nye et al [NHTSL19] have explored learning to infer program sketches instead of equations, using a seq2seq model.

Another close line of work to ours is pruning techniques for program synthesis, which aim at improving search by ruling out entire subtrees. For instance, Wang, Dillig, and

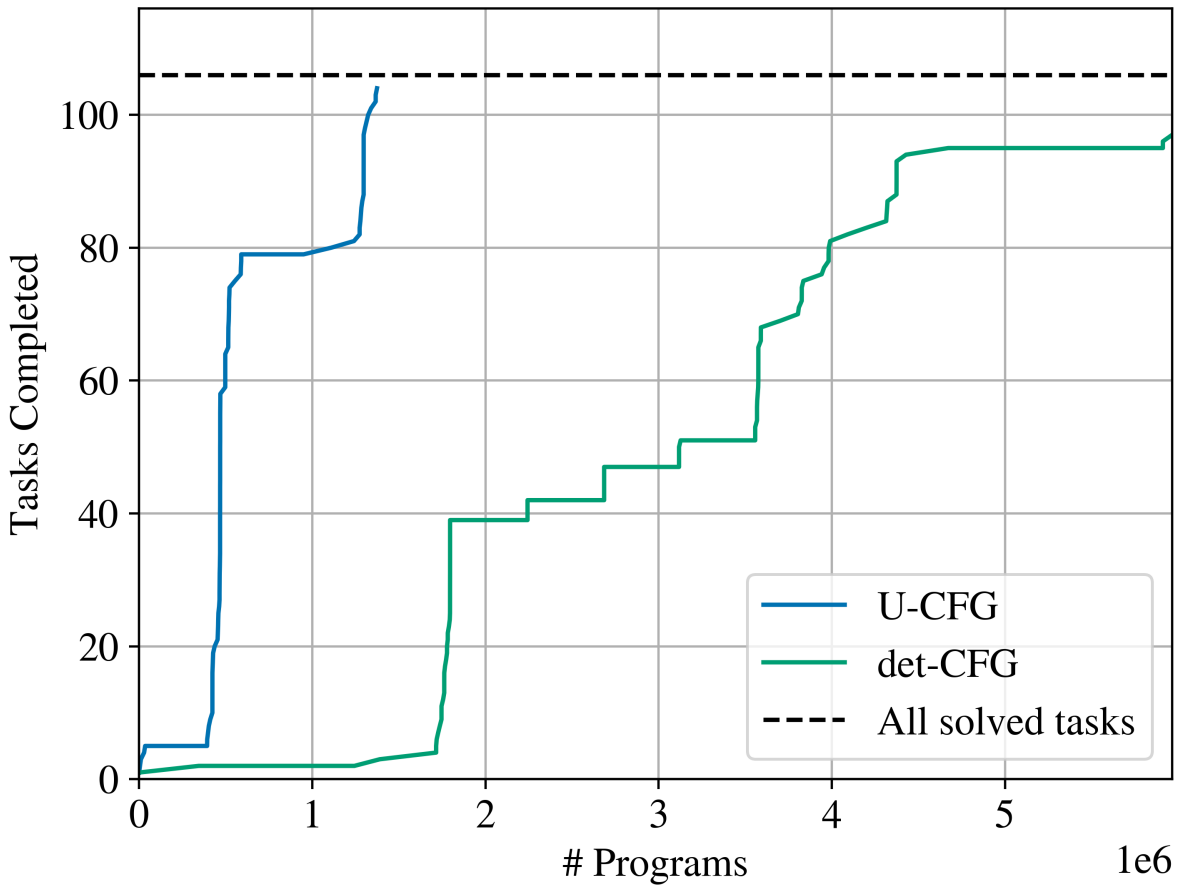


Figure 5.4: Number of tasks solved with respect to cumulative number of programs enumerated on the set of all solved List Programming tasks

Singh [WDS18] use abstraction refinement techniques to describe larger and larger DSLs throughout the search. Interestingly, their approach is also based on Bottom Up Tree Automata. There are key differences: first, runtime filtering is a preprocessing step, whereas pruning operates and integrates with the search. Note that this means that these two techniques can be easily combined. Second, the use of tree automata is for equating programs having the same outputs on particular inputs, which is orthogonal to our use of tree automata for compiling rules. Allowing the users to specify syntactic restrictions on the programs has been investigated in [PGIY20], whose goal is to lift the Read-Eval-Print Loop model to Read-Eval-Synth Loop: through interactions, the programmer further specifies its intent. Again the key difference is that the specifications are given at runtime, while the focus of runtime filtering is to provide a processing step for reducing DSLs before solving any synthesis task.

7 Futures

The two main paradigms for reducing the search space for program synthesis are pruning and sketching. We introduced runtime filtering, where the goal is to reduce a DSL through rules, as a preprocessing step. Our technical contributions are a compilation algorithm together with an algorithm for automatically generating rules. Through evaluations in two application domains we have demonstrated the effectiveness of runtime filtering in a

program synthesis pipeline. We believe that runtime filtering can have an even broader impact on program synthesis through the developments of more powerful compilation algorithms. The potential of learning equations (instead of automatically generating them) is very appealing and could further simplify DSLs. We hope that this work triggers new ideas for runtime filtering. This paper focuses on the theoretical aspects of runtime filtering. A more user-based approach would be useful to understand the kind of rules programmers would like to use for simplifying DSLs. We leave that for future work.

Chapter 6

Knowledge Powered Programs

TL;DR 6

Some program synthesis require external knowledge, that is not found neither in the task nor in the DSL, we call this knowledge-powered program synthesis. We describe three different levels of complexity of such tasks. We also explain why LLM fail on these kind of tasks. We propose a solution: WikiCoder that solves the first level of difficulty. By leveraging a knowledge graph containing the relevant information, it splits a task into subtasks and try to solve each subtask in parallel with both the classic program synthesis pipeline and queries to the knowledge graph.

This chapter tackles the task of adding external knowledge to programs and is mainly an extract from one of our published work [MFM23]. *External knowledge* is knowledge that is not included in the DSL. Furthermore, this knowledge is often related to the current example. Let us consider as an example the following task:

```
f("17", "United States") = "17 USD"
f("42", "France") = "42 EUR"
```

Solving this task requires understanding that the second inputs are countries and mapping them to their currencies. This piece of information is not present in the examples nor the DSL and therefore no program synthesis tool can solve that task without relying on external information. In other words, a solution program must be *knowledge-powered*!

An example knowledge-powered program yielding a solution to the task above is given below in a Python-like syntax:

```
def f(x,y):
    return x + " " + CurrencyOf(y)
```

It uses a function `CurrencyOf` obtained from an external source of knowledge.

First, we motivate the need for external knowledge in Section 1, then we develop different milestones for knowledge-powered program synthesis in Section 1.A. We discuss the failures of LLMs in Section 2 and then propose a solution that solves part of the problem in Section 3 which we evaluate in Section 4 and finally we discuss outlooks in Section 5.

Knowledge-powered program synthesis extends classical program synthesis by targeting knowledge-powered programs. The challenge of combining syntactical manipulations performed in program synthesis with semantic information was recently set out

by [VLG21a]. They discuss a number of applications: string manipulations, code refactoring, and string profiling, and construct an algorithm based on very large language models (see Section 5).

Our approach is different: the methodology we develop relies on knowledge graphs, which are very large structured databases organising knowledge using ontologies to allow for efficient and precise browsing and reasoning. There is a growing number of publicly available knowledge graphs, for instance Wikidata.org, which includes and structures Wikipedia data, and Yago [TWS20, HSBW13], based on Wikidata and schema.org. We refer to [HBC+21] for a recent textbook and to [Hog22] for an excellent survey on knowledge graphs and their applications, and to Figure 6.1 for an illustration.

The recent successes of both program synthesis and knowledge graphs suggest that the time is ripe to combine them into the knowledge-powered program synthesis research objective.

Our contributions:

- We introduce knowledge-powered program synthesis, which extends program synthesis by allowing programs to refer to external information collected from a knowledge graph.
- We identify a number of milestones for knowledge-powered program synthesis and propose a human-generated and publicly available dataset of 46 tasks to evaluate future progress on this research objective.
- We construct an algorithm combining state of the art machine learned program synthesizers with queries to a knowledge graph, which can be deployed on any knowledge graph.
- We implement a general-purpose knowledge program synthesis tool WikiCoder and evaluate it on various domains. WikiCoder solves tasks previously unsolvable by any program synthesis tool, while still operating at scale by integrating state of the art techniques from program synthesis.

1 The Need for External Knowledge

When considering knowledge-powered programming by examples, we do not change the problem, only the solution: instead of classical programs performing syntactic manipulations, we include knowledge-powered programs. To illustrate the difference, let us consider the following two tasks.

```
f("Paris") = "I love P"
f("Berlin") = "I love B"

g("Donald Knuth") = "DK is American"
g("Ada Lovelace") = "AL is English"
```

The first is a classical program synthesis task in the sense that it is purely syntactical, it can be solved with a two-line program concatenating “I love ” with the first letter of the input. On the other hand, the second requires some knowledge about the input individuals, here their nationality: one needs a knowledge-powered program to solve this task. Since almost all program synthesis tools perform only syntactical manipulations of

the examples (we refer to Section 5 for an in-depth discussion), they cannot solve the second task.

Knowledge-powered programming by examples goes much beyond query answering: the goal is not to answer a particular query, but to produce a program able to answer that query for any input. This is computationally and conceptually a much more difficult problem.

For concreteness we introduce some terminology about knowledge graphs, and refer to Figure 6.1 for an illustration. Nodes are called entities, and edges are labelled by a relation. Entities are arranged into classes: "E. Macron" belongs to the class of people, and "France" to the class of countries. The classes and relations are constrained by ontologies, which define which relations can hold between entities. The de facto standard for querying knowledge graphs is through `SPARQL` queries, which is a very powerful and versatile query language.

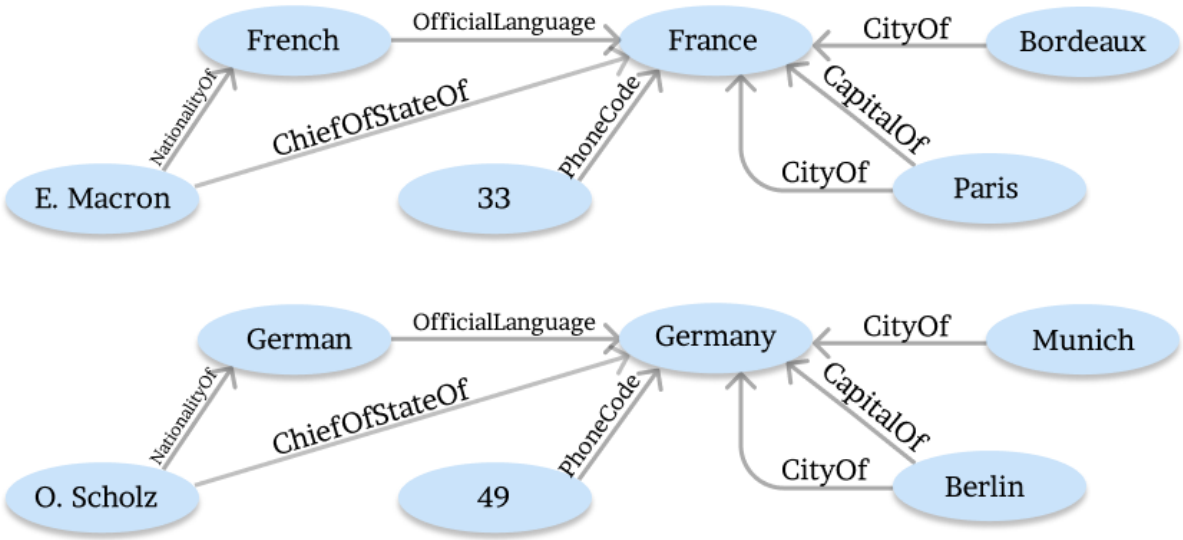


Figure 6.1: Illustration of part of a knowledge graph.

1.A Milestones

We identify three independent ways in which semantic information can be used. They correspond to different stages for solving a task:

- *preprocessing:* the first step is to extract entities from the examples;
- *relating:* the second step is to relate entities in the knowledge graph;
- *postprocessing:* the third step is to process the information found in the knowledge graph.

How much preprocessing to extract entities? Let us consider two tasks.

```
f("Aix, Paris, Bordeaux") = "Paris"
f("Hamburg, Berlin, Munich") = "Berlin"

g("President Obama") = "Obama"
g("Prime Minister de Pfeffel Johnson") = "de Pfeffel Johnson"
```

In the function `f` the goal is to extract the second word as separated by commas: this is a purely syntactic operation. In the function `g` we need to remove the job title from the input: this requires semantic knowledge, for instance it is not enough to use neither the second nor the last word.

How complicated is the relationship between entities? We examine two more tasks.

```
f("Paris") = "France is beautiful"
f("Berlin") = "Germany is beautiful"

g("Paris") = "Phone country code: 33"
g("Berlin") = "Phone country code: 49"
```

The function `f` relates two entities: a city and a country. One can expect that the knowledge graph includes the relation `CapitalOf`, which induces a labelled edge between "Paris" and "France" as well as "Berlin" and "Germany" (as in Figure 6.1). Note that it could also be the relation `CityOf`. More complex, the function `g` requires crossing information: indeed to connect a city to its country code, it is probably required to compose two relations: `CapitalOf` and `PhoneCode`. In other words, the entities are related by a path of length 2 in the knowledge graph, that we write `CapitalOf-PhoneCode`. More generally, the length of the path relating the entities is a measure of complexity of a task.

How much postprocessing on external knowledge? Let us look again at two tasks.

```
f("Paris") = "Country's first letter: F"
f("Berlin") = "Country's first letter: G"

g("President Obama") = "B@r@ck"
g("Prime Minister Johnson") = "B0r1s"
```

For the function `f`, the difficulty lies in finding out that the external knowledge to be search for relates "Paris" to "France" and "Berlin" to "Germany", and then to return only the first letter of the result. Similarly, for `g`, before applying a leet translation we need to retrieve as external knowledge the first name. In both cases the difficulty is that the intermediate knowledge is not present in the examples.

The three steps can be necessary, the most complex tasks involve at the same time subtle preprocessing to extract entities, complicated relationships between entities, and significant postprocessing on external knowledge.

1.B Motivating Examples

We illustrate the disruptive power of knowledge-powered program synthesis in three application domains.

General knowledge The first example, which we use throughout the paper for illustrations, in the dataset and in the experiments, is to use the knowledge graph to obtain general facts about the world, such as geography, movies, people. Wikidata and Yago are natural knowledge graphs candidates for this setting. This domain is heavily used for

query answering: combining it with program synthesis brings it to another level, since programs generalize to any input.

Grammar exercises The second example, inspired from [VLG21b], is about language learning: tasks are grammar exercises, where the goal is to write a grammatically correct sentence. Here the knowledge graph includes grammatical forms and their connections, such as verbs and their different conjugated forms, pronouns, adjectives, and so on. Generating programs for solving exercises opens several perspectives, including generating new exercises as well as solving them automatically.

Advanced database queries In the third example knowledge-powered program synthesis becomes a powerful querying engine. This scenario has been heavily investigated for SQL queries [WCB17, ZBCW22], but only at a syntactic level. Being able to rely on the semantic properties of the data opens a number of possibilities, let us illustrate them on an example scenario. The knowledge graph is owned and built by a company, it contains immutable data about products. The database contains customer data. Crossing semantic information between the database being queried and the knowledge graph allows the user to generate complex queries to extract more information from the database including for instance complex statistics.

2 The Failures of LLMs

This section is not in our original paper, it was added due to the fact that LLMs have seen a tremendous improvements since the experiments were conducted, however we still give basic arguments againts LLMs that reveal to still be true despite the recent advances. LLMs fail on knowledge powered tasks for multiple reasons that are particular to these tasks.

First, they fail because first and foremost they need to be able to do program synthesis which means generating code. Generating the missing answers is easier for them than generating code. Generating code is important in most production use, because it enables users to ensure that the answer is related to their intent. That is code enables trust while answers despite easier to generate for LLMs are black boxes.

Second, continuing on trust, LLMs suffer from hallucinations [RSD23], hallucinations are lies in the sense that they are incoherent with provided data or generated data that is not present. One such example of incorrect data is: '*Yuri Gagarin was the first person to land on the Moon.*' which is wrong since it is Neil Armstrong. An example of created data is: '*Unicorns were documented to have roamed the plains of Atlantis around 10,000 BC, where they were considered sacred creatures and were often associated with royalty.*' which is hallucinated since Atlantis did not exist. Third, they are not consistent with relations of objects. They also fail in the paradigm of relating objects in both directions: A is related to B but B is also related to A. This is called the reversal curse [BTK+23]. One such example is that LLMs can answer correctly the question: 'Who is Tom Cruise's mother?' but asking the reversed question that is 'Who is the son of Mary Lee South?', LLMs fail to answer that her son is Tom Cruise.

Finally, their knowledge is fixed at a given time, while when thinking about practical use cases, the most likely one is spreadsheets à la Flashfill where users want to talk about local knowledge relevant to their local entity such as employees, clients, *etc.* These data

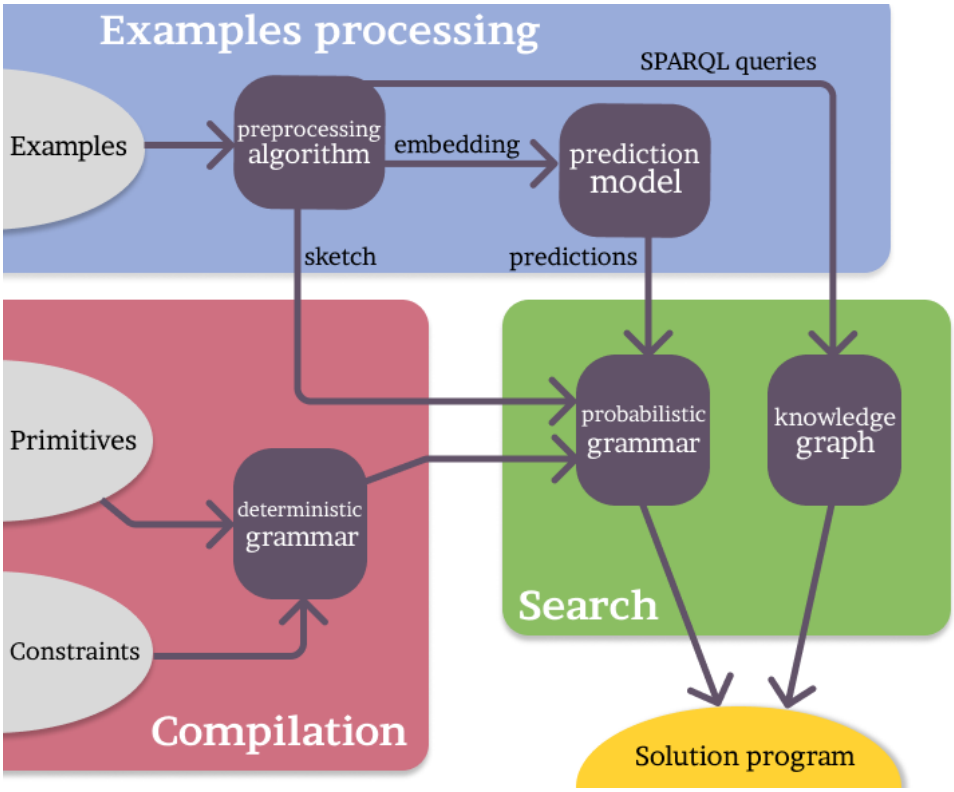


Figure 6.2: WikiCoder pipeline.

are volatile in the sense that they may change rapidly from one day to another. Using techniques for fetching data from trusted sources such as retrieval augmented generation (RAG) [LPP+20] for LLMs do not enable to bridge the gap since LLMs fail to be factual.

3 WikiCoder

WikiCoder is a general-purpose grammar-based program synthesis tool, it was developed in Python integrating state of the art program synthesis techniques. It is publicly available on GitHub as part of our code on the branch wikicoder. The main functionality of WikiCoder is to solve programming by examples tasks: the user inputs a few examples and WikiCoder synthesizes a knowledge-powered program satisfying the examples. The programming language is specified as a domain specific language (DSL), designed to solve a common set of tasks. WikiCoder supports a number of classical DSLs, including towers building [EWN+21], list integer manipulations [BGB+17], regular expressions, and string manipulation tasks à la FlashFill [Gul11b]: our experiments report on the latter DSL. It follows almost the same pipeline described in Chapter 2, WikiCoder is divided into three components: compilation, examples processing, and search, as illustrated in Figure 6.2 and discussed in the next three sections. This is very different from LLM architectures, which is based on auto-regressive very large models directly generating code.

3.A Compilation

As explained previously, the DSL is specified as a list of primitives together with their types and semantics.

The DSL we use in our experiments is tailored for string manipulation tasks à la Flashfill [Gul11a]. For the sake of presentation we slightly simplify it. We use two primitive types: `STRING` and `REGEXP`, and one type constructor `->` to create functions. We list the primitives below, they have the expected semantics.

```
$         : REGEXP                    # end of string
.         : REGEXP                    # all
[^_]+     : STRING -> REGEXP          # all except X
[^_]+$    : STRING -> REGEXP          # all except X at the end
compose   : REGEXP -> REGEXP -> REGEXP

concat    : STRING -> STRING -> STRING
match     : STRING -> REGEXP -> STRING

# concat_if: concat if the second argument (constant)
#            is not present in the first argument
concat_if     : STRING -> STRING -> STRING
# split_fst: split using regexp, returns first result
split_fst   : STRING -> REGEXP -> STRING
# split_snd: split using regexp, returns second result
split_snd : STRING -> REGEXP -> STRING
```

In the implementation we use two more primitive types: `CONSTANT_IN` and `CONSTANT_OUT`, which correspond to constants in the inputs and in the output. This is only for improving performances, it does not increase expressivity. Some primitives are duplicated to use the two new primitive types.

3.B Examples processing

A preprocessing algorithm produces from the examples three pieces of information: a sketch, which is a decomposition of the current task into subtasks, a set of `SPARQL` queries for the knowledge graph, and an embedding of the examples for the prediction model.

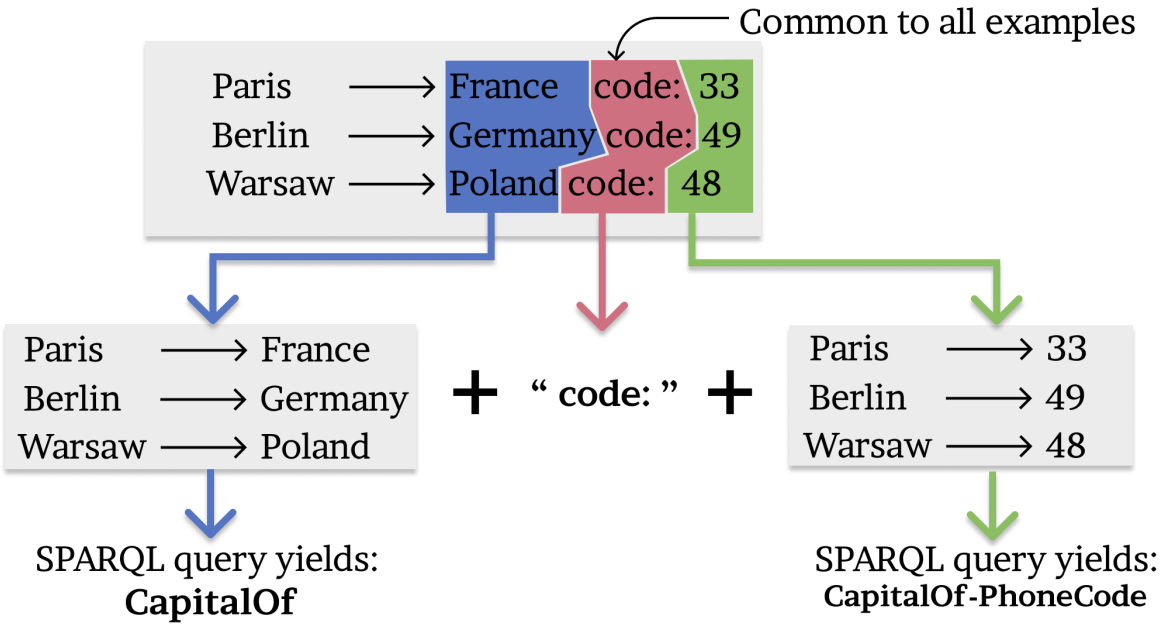


Figure 6.3: Example of the preprocessing algorithm.

Preprocessing algorithm Figure 6.3 illustrates the preprocessing algorithm in action. The high level idea is that the algorithm is looking for a shared pattern across the different examples. In the task above, " code: " is shared by all examples, hence it is extracted out. A naive implementation is to look for the largest common factor between the strings, and proceed recursively on the left and on the right. The process is illustrated with Algorithm 15, which produces a list of constants from a list of strings. This procedure is applied to both the inputs and outputs independently to create sketches. Thanks to these constants the task is split into subtasks as illustrated in Figure 6.3 where we have split the task into two subtasks: `CapitalOf` and `CapitalOf-PhoneCode`. To solve each subtask we query the knowledge graph with the inputs and outputs. If no path is found, we run a regular – syntactical – program synthesis algorithm.

Algorithm 15 Constant extraction

procedure GETCONSTANTS($S = (S_k)_{k\in[1,n]}$: strings)
  if there is an empty string in S **then**
    return empty list
  end if
  factor $\leftarrow$ longest common factor among all strings in S
  if len(factor) ≤ 2 **then**
    return empty list
  else
    $S_{\text{left}} \leftarrow$ prefix of factor in S
    $L_{\text{left}} \leftarrow$ GETCONSTANTS(S_{left})
    $S_{\text{right}} \leftarrow$ suffix of factor in S
    $L_{\text{right}} \leftarrow$ GETCONSTANTS(S_{right})
    return L_{left} + factor + L_{right}
  end if
end procedure

Generated `SPARQL` queries The `SPARQL` queries are generated from the examples after preprocessing. Once we have the constants with Algorithm 15 of the inputs and outputs, we can split the inputs and outputs in constant parts and non-constant parts, only the non-constant parts are relevant. For each non-constant part in the outputs, we generate queries from the non constant part of the inputs which should map to this non-constant part of the output. Since relations may be complex, that is "Paris" is at distance 1 from "France" but "33" is two relations away from "Paris", we generate `SPARQL` queries for increasing distances up to a fixed upper bound. Here is the query at distance 2, that we execute for the example in Figure 6.3 with `CapitalOf-PhoneCode`:

```
PREFIX w: <https://en.wikipedia.org/wiki/>
SELECT ?p0 ?p1 WHERE {
   w:Paris ?p0 ?o_1_0 .
   ?o_1_0 ?p1 w:33 .
   w:Berlin ?p0 ?o_2_0 .
   ?o_2_0 ?p1 w:49 .
   w:Warsaw ?p0 ?o_3_0 .
   ?o_3_0 ?p1 w:48 .
```

```
}
```

Notice that intermediary entities make an apparition in order to accommodate for longer path lengths. The output of the above query would consist of two paths: `CityOf-PhoneCode` and `CapitalOf-PhoneCode`.

As disambiguation strategy (inspired by [ZSC+22]) we choose the path with the least number of hits across all examples, called the least ambiguous path. In this example there is no preferred path since both paths lead to a single entity for each example. As an example of this disambiguation strategy, let us consider paths from "33" to "Paris": there are two paths, `PhoneCodeOf-Capital` and `PhoneCodeOf-City`. The least ambiguous path is `PhoneCodeOf-Capital` since `PhoneCodeOf-City` leads to all cities of the country.

To find the least ambiguous path, we need to count the number of hits, which is done using more `SPARQL` queries. Here is a sample query to get all entities at the end of the path `CapitalOf-PhoneCode` from the starting entity "Paris":

```
PREFIX w: <https://en.wikipedia.org/wiki/>
SELECT ?dst WHERE {
   w:Paris w:CapitalOf ?e0 .
   ?e0 w:PhoneCode ?dst .
}
```

We count the number of results for all examples to get the number of hits for a path.

3.C Search

As presented in Chapter 2, the prediction model assigns to each candidate program a likelihood of being a solution to the task at hand. WikiCoder uses Heap Search (see Chapter 3). The candidate programs use the results of the `SPARQL` queries on the knowledge graph to be run on the examples.

4 Evaluation

We perform experiments to answer the following questions:

(Q1) Which milestones as described in Section 1.A can be achieved with our algorithm?

(Q2) How does WikiCoder compare to GPT-3 and Codex, the algorithm powering Copilot based on very large language models?

(Q3) Can a knowledge-powered program synthesis tool operate at scale for classical purely syntactic tasks?

4.A Environment

The dataset along with the code for the experiments are available publicly on GitHub on the same repository. We favoured simplicity over performance as well as clear separation into independent components as described above.

Benchmark suite Since there are no existing datasets to test the new aspects introduced in Section 1.A, we created one ourselves. The dataset is comprised of 46 tasks in the spirit of the FlashFill dataset [Gul11b]. Some of the tasks are inspired or extracted from [VLG21b], they are tagged as such in our code. Each task requires external knowledge to be solved and is labelled with 3 metadata:

- preprocessing for entity extraction: 0 if the inputs are already the sought entities, 1 if a syntactical program is enough to extract them, and 2 otherwise;
- complexity of the relationships between entities : 0 when no relation from the knowledge graph is needed, 1 for simple (single edge in the knowledge graph), and 2 for composite;
- postprocessing on external knowledge: 0 if the knowledge is used without postprocessing, and 1 otherwise.

This induces 8 categories, we provide at least 4 tasks for each category.

Knowledge graph For simplicity and reproducibility, we use a custom-made small knowledge graph for the experiments. We provide `SPARQL` queries to construct the knowledge graph.

Experimental setup All our experiments were performed on a consumer laptop (MSI GF65 Thin 9SE) with an intel i7-9750H CPU working up to 2.60GHz on a single thread. The operating system is Manjaro 21.3 with linux kernel 5.17.5-1. The code is written in Python 3.8.10. The framework used for the `SPARQL` database is BlazeGraph 2.1.6 The code made available includes all the details to reproduce the experiments, it also includes the exact version of the Python libraries used.

Prediction Model The details are similar to those described in Chapter 2 Section 2.A. Our prediction model is close to the one in Chapter 2 Figure 2.4 except the LSTM was replaced with a classic RNN, this change however does not impact performances.

The prediction model is trained on a dataset of 2500 tasks for 2 epochs with a batch size of 16 tasks. The tasks were generated with the following process: a program was sampled randomly from a uniform probabilistic context-free grammar, then inputs are randomly generated and run on the program. We used the Adam optimiser with a cross entropy loss to maximise the probability of the solution program being generated from the grammar.

4.B Results on the new dataset

WikiCoder solves 18 out of 46 tasks with a timeout of 60 seconds. Let us make more sense of this number:

- Since our algorithm does not perform postprocessing on knowledge, none of the 16 tasks involving knowledge postprocessing were solved. Thus only 30 tasks are within reach of our algorithm.
- Among the 30 tasks, for 19 of them the inputs are directly the sought entities. WikiCoder solves 12 out of these 19 tasks, most failed tasked are not solvable with our DSL.

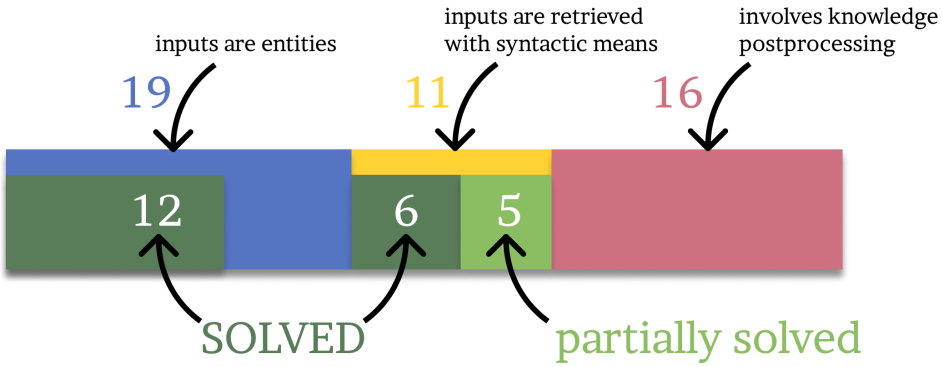


Figure 6.4: WikiCoder performance on our new dataset

- In the remaining 11 tasks, the entities can be retrieved using syntactic manipulations. WikiCoder solves 6 out of these 11 tasks.
- Digging deeper for the last case, the fault lies in all remaining 5 cases with the preprocessing algorithm, which fails to correctly decompose the task and formulate the appropriate `SPARQL` queries. However, when provided with the right decomposition, WikiCoder solves all 5 tasks.

The results can be visualised on Figure 6.4. The takeaway message is: WikiCoder solves almost all tasks which does not involve knowledge postprocessing, and when it does not the main issue is entities extraction with the preprocessing algorithm.

4.C Comparison with Codex and GPT-3

At the time the experiments were conducted only GPT-3 was available, ChatGPT just had been released during the review process. It is not easy to perform a fair comparison with Codex and GPT-3 as they solve different problems. There are two ways these models can solve programming by examples tasks. The experiments were performed on OpenAI's beta using the Da Vinci models (the most capable) for GPT-3 and Codex, with default parameters.

Using Codex: as *code completion*, provided with examples as docstring. This makes the problem very hard for Codex for two reasons: first, it was trained to parse specification in natural language, possibly enriched with examples. Providing only examples is a very partial specification which Codex is not used to. Second, Codex does not use a domain specific language, but rather general purpose programming languages such as Python and C. This implies that the program might be wrong in different ways: incorrect syntax, typing issues, or compilation errors. For the reasons above, in this unfavourable comparison Codex solves only the 5 easiest tasks.

There are two ways of using GPT-3: *query answering* or *code completion*. For query answering, we feed GPT-3 with all but the last examples, and ask it to give the output for the input of the last example:

```
Query:
    f("France") = "I live in France"
    f("Germany") = "I live in Germany"
    f("Poland") = "I live in Poland"
```

```
    f("New Zealand") = ?
Output:
    f("New Zealand") = "I live in New Zealand"
```

The weakness of this scenario is that GPT-3 does not output a program: it only answers queries. One consequence is that the correctness test is very weak: getting the right answer on a single query does not guarantee that it would on any input. Worse, not outputting a program means that the whole process acts as a black-box, giving up on the advantages of our framework (see related work section) and program synthesis in general.

Using GPT-3/ChatGPT as *code completion*: the prompt is to generate a Python function that satisfies the following examples. This fixes the weakness of not outputting a program with query answering. However, on tasks that require a knowledge graph, the answer is either a succession of if statements or using a dictionary which is semantically equivalent to the if statements. In some sense, the system having only partial knowledge, it does not attempt to generalise beyond the given examples.

GPT-3 performs very well in the query answering setting, solving 31 tasks out of 46. Only 2 tasks were solved by WikiCoder and not by GPT-3, and conversely 11 by GPT-3 and not by WikiCoder. GPT-3 only solves 3 tasks involving knowledge postprocessing: in this sense, WikiCoder and GPT-3 suffer from the same limitations, they both struggle with knowledge postprocessing.

4.D Results on FlashFill

To show that WikiCoder operates at scale on classical program synthesis tasks, we test it against the classical and established FlashFill dataset. The results are shown on Figure 6.5 (displaying cumulated time), WikiCoder solves 70 out of 101 tasks with a timeout of 60 seconds per task, on par with state of the art general purpose program synthesis tools. Only 85 tasks can be solved with our DSL.

4.E Limitations

To show the limitations of our approach, we discuss some counterexamples.

Entities extraction Our preprocessing algorithm looks for the longest shared pattern of length at least 2 in the examples. Let us consider the following example:

```
f("France") = "I live in France"
f("Germany") = "I live in Germany"
f("Poland") = "I live in Poland"
```

The longest pattern is naturally “I live in ”, however another pattern occurs, unexpectedly: “an” appears in each country’s name. When using our preprocessing algorithm this leads to a wrong sketch. Adding a fourth example would remove this ambiguity if the country name does not include “an”. A related example would be if all examples include year dates in the same millennium, say all of the form “20XX”: then “20” appears in each example, but it should not be separated from XX.

Knowledge postprocessing None of the tasks involving knowledge postprocessing were solved. Indeed, if the entity to be used is not present in the example, it is very hard to guess which one it is. Natural candidates are entities in the neighbourhood of the starting entity. Our approaches to this challenge have proved inefficient.

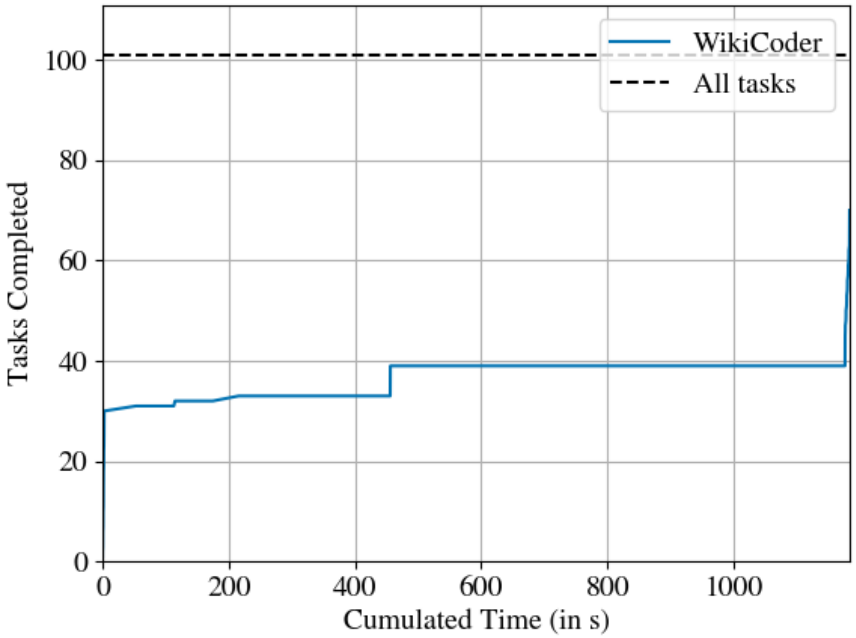


Figure 6.5: WikiCoder solves 70% of FlashFill tasks

4.F Examples of Programs

To show that the programs generated are clear and interpretable by humans, we show a few of them translated into Python equivalents. Here is the first example where two relations are needed:

```
// Examples:
    f("France") = "French, capital:Paris"
    f("Germany") = "German, capital:Berlin"
    f("China") = "Chinese, capital:Beijing"
    f("New Zealand") = "New Zealander, capital:Wellington"

// Generated program:
def f(x: str) -> str:
    a = label(follow_edges_from(x, "demonym"))
    b = ", capital:"
    c = label(follow_edges_from(x, "isCapitalOf"))
    return a + b + c
```

We present another example where a relation at distance 2 is needed:

```
// Examples:
    f("Paris") = "The phone country code is 33"
    f("Berlin") = "The phone country code is 49"
    f("Detroit") = "The phone country code is 1"
    f("Chihuahua") = "The phone country code is 52"

// Generated program:
def f(x: str) -> str:
    a = "The phone country code is "
    b = label(follow_edges_from(x, "CityOf", "phoneCode"))
    return a + b
```

5 Discussion

5.A Related work

The closest related work is [VLG21b]: the tool FlashGPT3 solves knowledge-powered program synthesis with a completely different approach: instead of querying a knowledge graph, FlashGPT3 uses a large language model (GPT3 [BMR+20b]) to get external knowledge. FlashGPT3 is shown superior to both program synthesis tools and GPT3 taken in isolation, and achieves impressive results in different domains. There are three advantages of using a knowledge graph over a large language model:

- **Reliability**: the synthesized program is only as reliable as the knowledge source that it uses. To illustrate this point, let us consider the task

```
f("Paris") = "France"
f("Berlin") = "Germany"
```

  GPT3 needs very few examples to predict that `f("Washington")= "United States "` but it might fail on more exotic inputs. On the other hand, querying a knowledge graph results in setting `f = CountryOf` implying that the program will correctly reproduce the knowledge from the graph. Although GPT3 has achieved extraordinary results in query answering, it is still more satisfactory to rely on an established and potentially certified source of knowledge such as Wikipedia.

- **Explainability**: the constructed program has good explainability properties: the exact knowledge source it uses can be traced back to the knowledge graph and therefore effectively verified.

- **Adaptability**: knowledge-powered program synthesis can be deployed with any knowledge graph, possibly collecting specialised or private information which a large language model would not know about.

Other knowledge-powered program synthesis tools Most program synthesis tools work at a purely syntactical level. However, some included limited level of semantic capabilities for domain-specific tasks: for instance Transform-data-by-example (TDE) uses functions from code bases and web forms to allow semantic operations in inductive synthesis [HCG+18], and APIs were used for data transformations [BSMK17].

Knowledge graphs A growing research community focuses on creating, maintaining, and querying knowledge graphs. The most related problems to our setting are query by example, where the goal is either to construct a query from a set of examples [JKL+16, MSS17], and entity set expansion, aiming at expanding a small set of examples into a more complete set of entities having common traits [ZSC+22].

5.B Contributions and Outlook

We have introduced knowledge-powered program synthesis, extending program synthesis by allowing programs to rely on knowledge graphs. We described a number of milestones for this exciting and widely unexplored research objective, and proposed a dataset to evaluate the progress in this direction. We constructed an algorithm and implemented a

general-purpose knowledge-powered program synthesis tool WikiCoder that solves tasks previously unsolvable. Our tool can only address about one third of the dataset; we believe that solving the whole dataset would be an important step forward towards deploying program synthesis in the real world.

The most natural continuation of this work is to use very large language models for knowledge-powered program synthesis. As discussed above, how can we retain the properties of our framework with knowledge graphs: reliability, explainability, and adaptability, while leveraging the power of very large language models?

Chapter 7

Futures of Program Synthesis

In this chapter, in Section 1 we summarise what has been tackled in this manuscript, then we discuss future directions in Section 2.

1 This thesis

First with Chapter 1, we have presented the program synthesis problem and the main specifications framework: logic, natural language and examples. This thesis focuses on programming by examples, that is the specification are given as examples of input and outputs pairs and the space of programs is described by a grammar giving us functional programs. We restrict ourselves to context-free grammars. We also introduced the necessary notions of different CFGs needed.

In Chapter 2, we introduced the concept of domain specific language, but argued for the development of domain agnostic techniques for program synthesis. Then we introduced our domain agnostic framework based on feeding the examples into a prediction model which enables us to produce a probabilistic grammar. This grammar gives us a total order over programs, this order is used to guide the search, this is what we call cost-guided search. We have shown that in practice this prediction model is usually a neural network, and that they are a lot of issues in training them notably due to the lack of data, this is why current endeavours look at using encoding parts of LLM for program synthesis in order to solve these problems.

Chapter 3 starts from this grammar and describes best-first search algorithms: algorithms that enumerate programs in order of increasing cost. We introduced Heap Search which was the first bottom-up logarithmic delay best-first search algorithm. The delay translates the complexity of enumeration between enumerating the i^{th} program and the next. Heap Search follows a bottom-up approach which enables it to fully leverage bottom-up evaluation techniques. Then Bee Search [AL23] was developed by other authors which improved upon Heap Search and introduced the idea of cost tuple representation. Then we investigated whether we can build a constant delay, or no-delay, best-first search algorithm. We successfully built Eco Search the first no-delay best-first search algorithm. It combines ideas of Heap Search and Bee Search to achieve better performances and with the help of a non trivial theoretical analysis we showed that using bucket priority queues enables us to achieve constant delay. We have also shown that practically Eco Search outperforms its competitors by a good margin. Looking back before this thesis the best best-first search algorithm was A^* [LHAN18] with logarithmic delay and now the chapter is closed with a no-delay best-first search

algorithm Eco Search, it is not possible to do better than constant delay, however it is still possible to make faster algorithms.

Chapter 4 investigates approaches to parallelise the search process. First, we looked at sampling algorithms which have the nice property of being memoryless thus making the parallelisation trivial. We introduced SQRT Sampling a sampling algorithm that is best among the the sampling algorithm in order to minimise the expected number of samples to draw before finding a solution. Despite such advantages it fails to be competitive with Heap Search. We also looked into the idea of splitting the grammar into disjoints grammars: we would simply have to search one disjoint grammar per processor. The advantage of such a method is that it is agnostic to the algorithm used. We introduced the *grammar splitter* algorithm which can take a finite grammar and split it into multiple disjoint grammars and showed it works in practice.

In Chapter 5, we tackled compiling language semantic properties into filters. More precisely, pruning such as observational equivalence eliminates a lot of redundant programs at runtime after evaluation. However, the evaluation is costly, and a large part of these redundancies are due to semantic properties of the DSL not of the task at hand. Therefore we developed a method to automatically generate these redundant programs; and then we showed how to compile them in a DBTA then transform it into a U-CFG, these redundant programs can be pruned without being evaluated at minimal cost. In fact, we show in our experiments that despite slower enumeration speed due to the more expressive U-CFGs, it improves solving time and the number of tasks solved.

Chapter 6 tackled the idea of introducing knowledge in programs. This is what we defined as knowledge powered program synthesis. This knowledge is of course not present in the DSL and we view as a relational knowledge. We show that knowledge powered program synthesis has three major difficulty levels. With the help of a knowledge graph containing the necessary knowledge, we described a tool-agnostic approach in order to solve the first difficulty level.

2 Futures directions

We will discuss more general directions for program synthesis than what has been covered in the chapters. On a larger scale, program synthesis has seen a growing interest by the research community over the last years and we believe this will continue to grow. Our conjecture is that in a few years, program synthesis methods will be drastically more powerful than current state of the art methods, and our methods will look like child's play.

One synthesiser to synthesise them all The dream of domain agnostic efficient program synthesis seems even more out of reach than before in the sense that even with Eco Search it is still unreachable to enumerate all programs of depth 5 in minutes for a simple DSL like DeepCoder. An idea that has emerged within the community and that we share is the idea of making a meta program synthesiser. To have a generic program synthesis approach, the meta program synthesiser would generate a program synthesis pipeline specific to the domain. Albeit no one knows how to do this, perhaps a first step is looking at a language of search procedures as a first step towards this goal.

LLM Combining classic algorithms with LLMs correctly is a hot topic of interest. It seems that everyone is working on a version of that, yet it seems as if no one managed to have resounding success. LambdaBeam [SDL+24] and CrossBeam [SDES22] are approaches that adapt very well to LLMs with strong oracles choosing the next step. Other works such as [LPP24] look at replacing the prediction model with a LLM. Ideas have merged to design the DSL or to augment the DSL with the help of a LLM. There are many relevant directions and most of them are unexplored. Specifically, LLMs are good at solving known tasks but struggle to invent new algorithms whereas classic program synthesis approach do not suffer an increase in difficulty for generating novel programs, a better metric of difficulty would be program size. This naturally look as if combining both would yield tremendous advantages.

Evaluation A key issue in program synthesis is the evaluation, the more advanced the domains are the more expensive it usually becomes. Despite techniques such as observational equivalence, sketching, runtime filtering, it is way faster to enumerate than to test programs. It remains an open question whether we can develop other domain agnostic techniques or at least quite generic techniques that enable us to dramatically speed up the evaluation. Perhaps another interesting direction is to look into the automated synthesis of more efficient evaluators.

Reward-Guided program synthesis The main approach when tackling program synthesis tasks where we want to optimise a reward of the program has been evolutionary search [RLSL20]. This is a more general case of program synthesis that has not been explored much. This is also very interesting in the case of reinforcement learning, can we solve reinforcement learning by working only with programs as controllers? As of now, works have either focused on extracting a program from a neural network [VMS+18] or with co-learning with a neural network [VLYC19] in order to ensure the neural network is not too far from a program.

Other works

During my PhD, I had the chance to work on various other topics, some of which led to publications, some others currently under review. With the objective of having one cohesive unit for my thesis, these were not included in the manuscript. Below, is a brief summary of these projects.

Statistical Comparison of Algorithm Performance Through Instance Selection This work has led to two published work [MAF+21, AMH22]. Empirical performance evaluations, in competitions and scientific publications, play a major role in improving the state of the art in solving many automated reasoning problems, including SAT, CSP and Bayesian network structure learning (BNSL). To empirically demonstrate the merit of a new solver usually requires extensive experiments, with computational costs of CPU years. This not only makes it difficult for researchers with limited access to computational resources to test their ideas and publish their work, but also consumes large amounts of energy. We propose an approach for comparing the performance of two algorithms: by performing runs on carefully chosen instances, we obtain a probabilistic statement on which algorithm performs best, trading off between the computational cost of running algorithms and the confidence in the result. We describe a set of methods for this purpose and evaluate their efficacy on diverse datasets from SAT, CSP and BNSL. On all these datasets, most of our approaches were able to choose the correct algorithm with about 95% accuracy, while using less than a third of the CPU time required for a full comparison; the best methods reach this level of accuracy within less than 15% of the CPU time for a full comparison.

Benchmark Minimization for Efficient Software Variant Ranking This work with Mathieu Acher is under review. Benchmarking is a common practice in software engineering to assess the qualities and performance of software variants, coming from multiple competing systems or from configurations of the same system. Benchmarks are notably used to compare and understand performance of variants, to fine-tune software, to detect regressions, or to design new software systems. The execution of benchmarks to get a complete picture of software variants is highly costly in terms of computational resources and time. In this paper, we propose a method to reduce the computational cost of benchmarks while still effectively ranking software variants. Our method strategically retains the most critical tests and applies bisection sampling together with a divide-and-conquer algorithm to efficiently sample among relevant remaining tests. We experiment with datasets and use cases from LLM leaderboards, SAT competitions, and configurable systems for performance modeling. Our results show that our method outperforms baselines based on random or greedy search, even when operating over a subset of variants or with an authorized ranking error. By employing this approach, we can reduce the computational cost of benchmarks by 20% with no loss and up to 99% with small losses.

Theoretical foundations for programmatic reinforcement learning This work is under review [SFM24]. The field of Reinforcement Learning (RL) is concerned with algorithms for learning optimal policies in unknown stochastic environments. Programmatic RL studies representations of policies as programs, meaning involving higher order constructs such as control loops. Despite attracting a lot of attention at the intersection of the machine learning and formal methods communities, very little is known on the theoretical front about programmatic RL: what are good classes of programmatic policies? How large are optimal programmatic policies? How can we learn them? The goal of this paper is to give first answers to these questions, initiating a theoretical study of programmatic RL.

Re-evaluating Metamorphic Testing of Chess Engines This joint work with Axel Martin, Djamel Eddine Khelladi and Mathieu Acher is under review. This study aims to confirm, replicate and extend the findings of a previous article entitled *'Metamorphic Testing of Chess Engines'* that reported inconsistencies in the analyses provided by *Stockfish*, the most widely used chess engine, for transformed chess positions that are fundamentally identical. Initial findings, under conditions strictly identical to those of the original study, corroborate the reported inconsistencies. However, the original article considers a specific dataset (including randomly generated chess positions, end-games, or checkmate problems) and very low analysis depth (10 plies, a ply refers to a single turn taken by one player in a game. Two plies, one from each player, together constitute a complete move, corresponding to 5 moves). These decisions pose threats that limit generalizability of the results, but also their practical usefulness both for chess players and maintainers of Stockfish. Thus, we replicate the original study considering this time (1) positions derived from actual chess games, (2) analyses at appropriate and larger depths, and (3) different versions of Stockfish. The replication results show that the Stockfish chess engines demonstrate significantly greater consistency in its evaluations. The metamorphic relations are not as effective as in the original article, especially on realistic chess positions. We also demonstrate that, for any given position, there exists a depth threshold beyond which further increases in depth do not result in any evaluation differences. We perform an in-depth analysis to identify and clarify the implementation reasons behind Stockfish's inconsistencies when dealing with transformed positions. A first concrete result is thus that metamorphic testing of chess engines is not yet an effective technique for finding faults of Stockfish. Another result is the lessons learned through this replication effort: metamorphic relations must be verified in the context of the domain's specificities; without such contextual validation, they may lead to misleading or irrelevant conclusions; changes in parameters and input dataset can drastically alter the effectiveness of a testing method.

Bibliography

[AAA+23] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[ABJ+13] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis*. IEEE, 2013.

[ABLY19] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *International Conference on Learning Representations*, 2019.

[AFSSL16a] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Results and analysis of sygus-comp'15. *arXiv preprint arXiv:1602.01170*, 2016.

[AFSSL16b] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2016: Results and analysis. *arXiv preprint arXiv:1611.07627*, 2016.

[AFSSL17] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2017: Results and analysis. *arXiv preprint arXiv:1711.11438*, 2017.

[AL23] Saqib Ameen and Levi HS Lelis. Program synthesis with best-first bottom-up search. *Journal of Artificial Intelligence Research*, 77:1275–1310, 2023.

[AM23] Antoine Amarilli and Mikaël Monet. Enumerating regular languages with bounded delay. In *40th International Symposium on Theoretical Aspects of Computer Science (STACS 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[AMH22] Marie Anastacio, Théo Matricon, and Holger Hoos. Challenges of acquiring compositional inductive biases via meta-learning. In Pavel Brazdil, Jan N. van Rijn, Henry Gouk, and Felix Mohr, editors, *ECMLPKDD Workshop on Meta-Knowledge Transfer*, volume 191 of *Proceedings of Machine Learning Research*, pages 11–23. PMLR, 23 Sep 2022. URL: https://proceedings.mlr.press/v191/anastacio22a.html.

[AON+21] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc

Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[ARU17] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 319–336. Springer, 2017.

[ASFS18] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. Search-based program synthesis. *Communications of the ACM*, 61(12), 2018. URL: https://doi.org/10.1145/3208071.

[BCD+11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. cvc4. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 171–177. Springer, 2011.

[BCJ18] Roderick Bloem, Krishnendu Chatterjee, and Barbara Jobstmann. *Graph Games and Reactive Synthesis*, pages 921–962. Springer International Publishing, Cham, 2018. doi:10.1007/978-3-319-10575-8_27.

[BGB+17] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations, ICLR*, 2017. URL: https://openreview.net/forum?id=ByldLrqlx.

[BGK+24] Shraddha Barke, Emmanuel Anaya Gonzalez, Saketh Ram Kasibatla, Taylor Berg-Kirkpatrick, and Nadia Polikarpova. Hysynth: Context-free llm approximation for guiding program synthesis. *arXiv preprint arXiv:2405.15880*, 2024.

[BHD+18] Rudy Bunel, Matthew J. Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations, ICLR*, 2018. URL: https://openreview.net/forum?id=H1Xw62kRZ.

[Bie78] Alan W. Biermann. The inference of regular lisp programs from examples. *IEEE Transactions on Systems, Man, and Cybernetics*, 8(8):585–600, 1978. doi:10.1109/TSMC.1978.4310035.

[BL69] J. Richard Buchi and Lawrence H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969. URL: http://www.jstor.org/stable/1994916.

[BM08] Henrik Björklund and Wim Martens. The tractability frontier for NFA minimization. In *International Colloquium on Automata, Languages and Programming, ICALP*, volume 5126 of *Lecture Notes in Computer Science*, pages 27–38. Springer, 2008. doi:10.1007/978-3-540-70583-3_3.

[BMR+20a] 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.

[BMR+20b] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Annual Conference on Neural Information Processing Systems, NeurIPS*, 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. doi:10.1017/CBO9781139172752.

[BPP20] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2020.

[BSL08] Rastislav Bodík and Armando Solar-Lezama. Program synthesis by sketching. 2008. URL: https://api.semanticscholar.org/CorpusID:8149812.

[BSMK17] Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Deep API programmer: Learning to program with APIs. *CoRR*, abs/1704.04327, 2017. URL: http://arxiv.org/abs/1704.04327, arXiv:1704.04327.

[BTK+23] Lukas Berglund, Meg Tong, Max Kaufmann, Mikita Balesni, Asa Cooper Stickland, Tomasz Korbak, and Owain Evans. The reversal curse: Llms trained on 'a is b' fail to learn 'b is a'. *arXiv preprint arXiv:2309.12288*, 2023.

[Bun88] Alan Bundy. The use of explicit plans to guide inductive proofs. In *9th International Conference on Automated Deduction: Argonne, Illinois, USA, May 23–26, 1988 Proceedings 9*, pages 111–120. Springer, 1988.

[Bun99] Alan Bundy. The automation of proof by mathematical induction. Technical report, 1999.

[CD20] Andrew Cropper and Sebastijan Dumancic. Learning large logic programs by going beyond entailment. In *International Joint Conference on Artificial Intelligence, IJCAI*, pages 2073–2079. ijcai.org, 2020. URL: https://doi.org/10.24963/ijcai.2020/287, doi:10.24963/IJCAI.2020/287.

[CDG+08] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. INRIA, 2008. URL: https://hal.inria.fr/hal-03367725.

[CEP+21] Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, and Yisong Yue. Neurosymbolic programming. *Foundations and Trends in Programming Languages*, 7(3):158–243, 2021. doi:10.1561/2500000049.

[CGL+23] José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. Flashfill++: Scaling programming by example by cutting to the chase. *Proceedings of the ACM on Programming Languages*, 7(POPL):952–981, 2023.

[CH85] Thierry Coquand and Gerard Huet. Constructions: A higher order proof system for mechanizing mathematics. In *European Conference on Computer Algebra*, pages 151–184. Springer, 1985.

[CHhH02] Murray Campbell, A.Joseph Hoane, and Feng hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1):57–83, 2002. URL: https://www.sciencedirect.com/science/article/pii/S0004370201001291, doi:https://doi.org/10.1016/S0004-3702(01)00129-1.

[Chi99] Zhiyi Chi. Statistical properties of probabilistic context-free grammars. *Computational Linguistics*, 25(1), 1999. URL: https://www.aclweb.org/anthology/J99-1004.

[Cho56] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956. doi:10.1109/TIT.1956.1056813.

[Cho57] N. Chomsky. Syntactic structures. *The Hague: Mouton*, 1957.

[Cho19] François Chollet. On the measure of intelligence. *CoRR*, abs/1911.01547, 2019. URL: http://arxiv.org/abs/1911.01547, arXiv:1911.01547.

[Chu57] A. Church. Applications of recursive arithmetic to the problem of circuit synthesis. In *Summaries of the Summer Institute of Symbolic Logic.*, 1957.

[CKPR73] Alain Colmerauer, Henri Kanoui, Robert Pasero, and Philippe Roussel. Un systeme de communication homme-machine en francais. *Rapport preliminaire, Groupe de Res. en Intell. Artif*, 1973.

[CLS18] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations, ICLR*, 2018.

[CLS19] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations, ICLR*. OpenReview.net, 2019. URL: https://openreview.net/forum?id=H1gfOiAqYm.

[CMF+20] Judith Clymo, Haik Manukian, Nathanaël Fijalkow, Adrià Gascón, and Brooks Paige. Data generation for neural programming by example. In *International Conference on Artificial Intelligence and Statistics*, Proceedings of Machine Learning Research, pages 3450–3459. PMLR, 2020. URL: http://proceedings.mlr.press/v108/clymo20a.html.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151158, New York, NY, USA, 1971. Association for Computing Machinery. doi:10.1145/800157.805047.

[CPS20] Nicolas Chan, Elizabeth Polgreen, and Sanjit A Seshia. Gradient descent over metagrammars for syntax-guided synthesis. *arXiv preprint arXiv:2007.06677*, 2020.

[CSH10] Koen Claessen, Nicholas Smallbone, and John Hughes. Quickspec: Guessing formal specifications using testing. In *International Conference on Tests and Proofs, TAP@TOOLS*, 2010. doi:10.1007/978-3-642-13977-2_3.

[CST21] Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis beyond domain-specific languages. *Advances in Neural Information Processing Systems*, 34:22196–22208, 2021.

[Cur34] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934. doi:10.1073/pnas.20.11.584.

[CvMG+14] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Conference on Empirical Methods in Natural Language Processing, EMNLP*, 2014. URL: https://aclanthology.org/D14-1179.

[CvMG+14] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoderdecoder for statistical machine translation. In *Conference on Empirical Methods in Natural Language Processing*, 2014. URL: https://api.semanticscholar.org/CorpusID:5590763.

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics*, 2019. URL: https://api.semanticscholar.org/CorpusID:52967399.

[DDM06] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2):1–2, 2006.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DRJDLM10] Luc De Raedt, Manfred Jaeger, Sau Dan Lee, and Heikki Mannila. A theory of inductive query answering. In *Inductive databases and constraint-based data mining*, pages 79–103. Springer, 2010.

[DUB+17] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdelrahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 990–998. PMLR, 2017. URL: http://proceedings.mlr.press/v70/devlin17a.html.

[ERSLT18] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. *Advances in neural information processing systems*, 31, 2018.

[EWN+21] Kevin Ellis, Catherine Wong, Maxwell I. Nye, Mathias Sablé-Meyer, Lucas Morales, Luke B. Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *International Conference on Programming Language Design and Implementation, PLDI*, 2021. URL: https://doi.org/10.1145/3453483.3454080.

[FBH+22] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

[FLM+22] Nathanaël Fijalkow, Guillaume Lagarde, Théo Matricon, Kevin Ellis, Pierre Ohlmann, and Akarsh Potta. Scaling neural program synthesis with distribution-based search. In *International Conference on Artificial Intelligence, AAAI*, 2022. URL: https://arxiv.org/abs/2110.12485.

[FMBD18] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2018. URL: https://doi.org/10.1145/3192366.3192382.

[Fuk69] Kunihiko Fukushima. Visual feature extraction by a multilayered network of analog threshold elements. *IEEE Transactions on Systems Science and Cybernetics*, 5(4):322–333, 1969. doi:10.1109/TSSC.1969.300225.

[GGGS86] Giovanni Guida, Marco Guida, Sergio Gusmeroli, and Marco Somalvico. Design and experimentation of an expert system for programming in-the-large. 1986.

[GHS12] Sumit Gulwani, William R Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

[GJJ+19] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. Program synthesis by type-guided abstraction refinement. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28, 2019.

[Gon98] Jie Gong. Specsyn: An environment supporting the specify-explore-refine paradigm for hardware/software system design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6(1):84–100, 1998.

[Goo21] Google. Use smartfill to help automate data entry in google sheets, 2021. Accessed: 2021-5-26. URL: https://workspaceupdates.googleblog.com/2020/10/smart-fill-google-sheets-automate-data-entry.html.

[Gul11a] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

[Gul11b] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, 2011. URL: https://doi.org/10.1145/1926385.1926423.

[GZY+24] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

[H+80] William A Howard et al. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[HBC+21] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutiérrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. *Knowledge Graphs*. Synthesis Lectures on Data, Semantics, and Knowledge. Morgan & Claypool Publishers, 2021. doi:10.2200/S01125ED1V01Y202109DSK022.

[HCG+18] Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek R. Narasayya, and Surajit Chaudhuri. Transform-data-by-example (TDE): an extensible search engine for data transformations. *Proceedings of the VLDB Endowment*, 11(10):1165–1177, 2018. URL: http://www.vldb.org/pvldb/vol11/p1165-he.pdf, doi:10.14778/3231751.3231766.

[HML75] Peter Hancock and Per Martin-Löf. Syntax and semantics of the language of primitive recursive functions. *preprint*, (3), 1975.

[Hod24] Michael Hodel. Addressing the abstraction and reasoning corpus via procedural example generation, 2024. URL: https://arxiv.org/abs/2404.07353, arXiv:2404.07353.

[Hog22] Aidan Hogan. Knowledge graphs: A guided tour (invited paper). In Camille Bourgaux, Ana Ozaki, and Rafael Peñaloza, editors, *International Research School in Artificial Intelligence in Bergen, AIB 2022, June 7-11, 2022, University of Bergen, Norway*, volume 99 of *OASIcs*, pages 1:1–1:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/OASIcs.AIB.2022.1.

[HS97] Sepp Hochreiter and Jorgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997. arXiv:https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco1997.9.8.1735.pdf, doi:10.1162/neco.1997.9.8.1735.

[HSBW13] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. YAGO2: A spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, 194:28–61, 2013. doi:10.1016/j.artint.2012.06.001.

[HU79] J. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 1st edition, 1979.

[IB96] Andrew Ireland and Alan Bundy. *Productive use of failure in inductive proof*. Springer, 1996.

[IKCZ16] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In Katrin Erk and Noah A. Smith, editors, *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany, August 2016. Association for Computational Linguistics. URL: https://aclanthology.org/P16-1195, doi:10.18653/v1/P16-1195.

[JKL+16] Nandish Jayaram, Arijit Khan, Chengkai Li, Xifeng Yan, and Ramez Elmasri. Querying knowledge graphs by example entity tuples. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 1494–1495. IEEE Computer Society, 2016. doi:10.1109/ICDE.2016.7498391.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[KUIKZ24] Samia Kabir, David N Udo-Imeh, Bonan Kou, and Tianyi Zhang. Is stack overflow obsolete? an empirical study of the characteristics of chatgpt answers to stack overflow questions. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 1–17, 2024.

[LBCO04] Tessa Lau, Lawrence Bergman, Vittorio Castelli, and Daniel Oblinger. Sheepdog: learning procedures for technical support. In *Proceedings of the 9th International Conference on Intelligent User Interfaces*, IUI '04, page 109116, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/964442.964464.

[LE24] Wen-Ding Li and Kevin Ellis. Is programming by example solved by llms? *CoRR*, abs/2406.08316, 2024. URL: https://doi.org/10.48550/arXiv.2406.08316, arXiv:2406.08316, doi:10.48550/ARXIV.2406.08316.

[LHAN18] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2018. URL: https://doi.org/10.1145/3211992.

[LLA+24] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

[LLS+24] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, and Li Zhang. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*, 2024.

[LMPS23] Yixuan Li, Federico Mora, Elizabeth Polgreen, and Sanjit A Seshia. Genetic algorithms for searching a matrix of metagrammars for synthesis. *arXiv preprint arXiv:2306.00521*, 2023.

[LPP+20] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[LPP24] Yixuan Li, Julian Parsert, and Elizabeth Polgreen. Guiding enumerative program synthesis with large language models. In *36th International Conference on Computer Aided Verification*. Springer, 2024.

[LWDW03] Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53:111–156, 2003.

[LXWZ23] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL: https://openreview.net/forum?id=1qvx610Cu7.

[LYB+19] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. Aroma: code recommendation via structural code search. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019. doi:10.1145/3360578.

[MAF+21] Théo Matricon, Marie Anastacio, Nathanaël Fijalkow, Laurent Simon, and Holger H. Hoos. Statistical Comparison of Algorithm Performance Through Instance Selection. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 43:1–43:21, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2021.43, doi:10.4230/LIPIcs.CP.2021.43.

[MDK+18] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. *Advances in neural information processing systems*, 31, 2018.

[MFM23] Théo Matricon, Nathanaël Fijalkow, and Gaëtan Margueritte. Wikicoder: Learning to write knowledge-powered code. In Georgiana Caltais and Christian Schilling, editors, *Model Checking Software*, pages 123–140. Springer Nature Switzerland, 2023.

[ML82] Per Martin-Löf. Constructive mathematics and computer programming. In *Studies in Logic and the Foundations of Mathematics*, volume 104, pages 153–175. Elsevier, 1982.

[MSS17] Steffen Metzger, Ralf Schenkel, and Marcin Sydow. QBEES: query-by-example entity search in semantic knowledge graphs based on maximal aspects, diversity-awareness and relaxation. *J. Intell. Inf. Syst.*, 49(3):333–366, 2017. doi:10.1007/s10844-017-0443-x.

[MTG+13] Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. A machine learning framework for programming by example. In *International Conference on Machine Learning, ICML*, 2013. URL: http://proceedings.mlr.press/v28/menon13.html.

[Mug91] Stephen Muggleton. Inductive logic programming. *New generation computing*, 8:295–318, 1991.

[MW80] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90121, jan 1980. doi:10.1145/357084.357090.

[NHTSL19] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. In *Proceedings of the International Conference on Machine Learning, ICML*, volume 97 of *Proceedings of Machine Learning Research*. PMLR, 2019. URL: https://proceedings.mlr.press/v97/nye19a.html.

[OSB+21] Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. BUSTLE: Bottom-up program synthesis through learning-guided exploration. In *International Conference on Learning Representations (ICLR)*, 2021.

[Pau94] Lawrence C Paulson. *Isabelle: A generic theorem prover*. Springer, 1994.

[PGIY20] Hila Peleg, Roi Gabay, Shachar Itzhaky, and Eran Yahav. Programming with a read-eval-synth loop. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):159:1–159:30, 2020. doi:10.1145/3428227.

[Qui90] J. Ross Quinlan. Learning logical definitions from relations. *Machine learning*, 5:239–266, 1990.

[RBN+19] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. cvc 4 sy: smart and fast term enumeration for syntax-guided synthesis. In *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II 31*, pages 74–83. Springer, 2019.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986. URL: https://api.semanticscholar.org/CorpusID:205001834.

[RLSL20] Esteban Real, Chen Liang, David So, and Quoc Le. AutoML-zero: Evolving machine learning algorithms from scratch. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 8007–8019. PMLR, 13–18 Jul 2020. URL: https://proceedings.mlr.press/v119/real20a.html.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):2341, jan 1965. doi:10.1145/321250.321253.

[RSD23] Vipula Rawte, Amit Sheth, and Amitava Das. A survey of hallucination in large foundation models. *arXiv preprint arXiv:2309.05922*, 2023.

[RWC+19] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[SA19] Calvin Smith and Aws Albarghouthi. Program synthesis with equivalence reduction. In *Verification, Model Checking, and Abstract Interpretation, VMCAI*, 2019. doi:10.1007/978-3-030-11245-5_2.

[SBS20a] Kensen Shi, David Bieber, and Rishabh Singh. TF-coder: Program synthesis for tensor manipulations. In *Workshop on Computer-Assisted Programming, CAP*, 2020. URL: https://openreview.net/forum?id=nJ5Ij53umw2.

[SBS20b] Kensen Shi, David Bieber, and Charles Sutton. Incremental sampling without replacement for sequence models. In *International Conference on Machine Learning*, pages 8785–8795. PMLR, 2020.

[SBS22a] Kensen Shi, David Bieber, and Rishabh Singh. Tf-coder: Program synthesis for tensor manipulations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44(2):1–36, 2022.

[SBS22b] Kensen Shi, David Bieber, and Rishabh Singh. Tf-coder: Program synthesis for tensor manipulations. *ACM Transactions on Programming Languages and Systems*, 44(2):10:1–10:36, 2022. doi:10.1145/3517034.

[SDES22] Kensen Shi, Hanjun Dai, Kevin Ellis, and Charles Sutton. Crossbeam: Learning to search in bottom-up program synthesis. *arXiv preprint arXiv:2203.10452*, 2022.

[SDL+24] Kensen Shi, Hanjun Dai, Wen-Ding Li, Kevin Ellis, and Charles Sutton. Lambdabeam: Neural program search with higher-order functions and lambdas. *Advances in Neural Information Processing Systems*, 36, 2024.

[SEM24] Leopoldo Sarra, Kevin Ellis, and Florian Marquardt. Discovering quantum circuit components with program synthesis. *Machine Learning: Science and Technology*, 5(2):025029, 2024.

[SFA17] Calvin Smith, Gabriel Ferns, and Aws Albarghouthi. Discovering relational specifications. In *Proceedings of the Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, pages 616–626. ACM, 2017. doi:10.1145/3106237.3106279.

[SFM24] Guruprerana Shabadi, Nathanaël Fijalkow, and Théo Matricon. Theoretical foundations for programmatic reinforcement learning, 2024. URL: https://arxiv.org/abs/2402.11650, arXiv:2402.11650.

[SG12] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings 24*, pages 634–651. Springer, 2012.

[SG16] Rishabh Singh and Sumit Gulwani. Transforming spreadsheet data types using examples. *SIGPLAN Not.*, 51(1):343356, jan 2016. doi:10.1145/2914770.2837668.

[SGF13] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):497–518, 2013. doi:10.1007/s10009-012-0223-4.

[Sha82] Ehud Yehuda Shapiro. *Algorithmic program debugging*. Yale University, 1982.

[SL08] Armando Solar-Lezama. *Program synthesis by sketching*. University of California, Berkeley, 2008.

[Sol08] Armando Solar-Lezama. *Program Synthesis By Sketching*. PhD thesis, University of California, Berkeley, 2008.

[Sol13] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5-6):475–495, 2013. doi:10.1007/s10009-012-0249-7.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[TAB+23] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[Tea24] The Coq Development Team. The coq proof assistant, June 2024. URL: https://doi.org/10.5281/zenodo.11551307%7D, doi:10.5281/zenodo.11551307.

[Tho00] Mikkel Thorup. On ram priority queues. *SIAM Journal on Computing*, 30(1):86–109, 2000. doi:10.1137/S0097539795288246.

[TLI+23] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[Tra84] B.A. Trakhtenbrot. A survey of russian approaches to perebor (brute-force searches) algorithms. *Annals of the History of Computing*, 6(4):384–400, 1984. doi:10.1109/MAHC.1984.10036.

[TWS20] Thomas Pellissier Tanon, Gerhard Weikum, and Fabian M. Suchanek. YAGO 4: A reason-able knowledge base. In *The Semantic Web - 17th International Conference, ESWC*, volume 12123 of *Lecture Notes in Computer Science*, pages 583–596. Springer, 2020. doi:10.1007/978-3-030-49461-2_34.

[URD+13] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo M K Martin, and Rajeev Alur. TRANSIT: Specifying protocols with concolic snippets. In *Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[UT20] Silviu-Marian Udrescu and Max Tegmark. Ai feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16):eaay2631, 2020.

[VLG21a] Gust Verbruggen, Vu Le, and Sumit Gulwani. Semantic programming by example with pre-trained models. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–25, 2021. doi:10.1145/3485477.

[VLG21b] Gust Verbruggen, Vu Le, and Sumit Gulwani. Semantic programming by example with pre-trained models. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021. doi:10.1145/3485477.

[VLYC19] Abhinav Verma, Hoang Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[VMS+18] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, pages 5045–5054. PMLR, 2018.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[Wal77] A. J. Walker. An efficient method for generating discrete random variables with general distributions. *ACM Transactions on Mathematical Software*, 3(3), 1977. URL: https://doi.org/10.1145/355744.355749.

[WB89] Martin Wirsing and Manfred Broy. A modular framework for specification and implementation. In *Colloquium on Trees in Algebra and Programming*, pages 42–73. Springer, 1989.

[WCB17] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. Synthesizing highly expressive SQL queries from input-output examples. In Albert Cohen and Martin T. Vechev, editors, *Conference on Programming Language Design and Implementation, PLDI*, pages 452–466. ACM, 2017. doi:10.1145/3062341.3062365.

[WDS17] Xinyu Wang, Isil Dillig, and Rishabh Singh. Synthesis of data completion scripts using finite tree automata. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–26, 2017.

[WDS18] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages*, 2:63:1–63:30, 2018. doi:10.1145/3158151.

[ZBCW22] Xiangyu Zhou, Rastislav Bodík, Alvin Cheung, and Chenglong Wang. Synthesizing analytical SQL queries from computation demonstration. In Ranjit Jhala and Isil Dillig, editors, *International Conference on Programming Language Design and Implementation, PLDI*, pages 168–182. ACM, 2022. doi:10.1145/3519939.3523712.

[ZRF+18] Lisa Zhang, Gregory Rosenblatt, Ethan Fetaya, Renjie Liao, William E. Byrd, Raquel Urtasun, and Richard S. Zemel. Leveraging constraint logic programming for neural guided program synthesis. In *International Conference on Learning Representations, ICLR*, 2018. URL: https://openreview.net/forum?id=HJIHtIJvz.

[ZSC+22] Yuyan Zheng, Chuan Shi, Xiaohuan Cao, Xiaoli Li, and Bin Wu. A meta path based method for entity set expansion in knowledge graph. *IEEE Transactions on Big Data*, 8(3):616–629, 2022. doi:10.1109/TBDATA.2018.2805366.

[ZW18] A. Zohar and L. Wolf. Automatic program synthesis of long programs with a learned garbage collector. In *Neural Information Processing Systems, NeurIPS*, 2018. URL: https://proceedings.neurips.cc/paper/2018/hash/390e982518a50e280d8e2b535462ec1f-Abstract.html.

Index