THE UNIVERSITY *of* EDINBURGH

A foundation for synthesising programming language semantics

Sándor Bartha

Doctor of Philosophy

Laboratory for Foundations of Computer Science

CDT Data Science

School of Informatics

The University of Edinburgh

2024

Abstract

Programming or scripting languages used in real-world systems are seldom designed with a formal semantics in mind from the outset. Therefore, the first step for developing well-founded analysis tools for these systems is to reverse-engineer a formal semantics. This can take months or years of effort.

Could we automate this process, at least partially? Though desirable, automatically reverse-engineering semantics rules from an implementation is very challenging, as found by Krishnamurthi, Lerner and Elberty. They propose automatically learning desugaring translation rules, mapping the language whose semantics we seek to a simplified, core version, whose semantics are much easier to write. The present thesis contains an analysis of their challenge, as well as the first steps towards a solution.

Scaling methods with the size of the language is very difficult due to state space explosion, so this thesis proposes an incremental approach to learning the translation rules. I present a formalisation that both clarifies the informal description of the challenge by Krishnamurthi et al, and re-formulates the problem, shifting the focus to the conditions for incremental learning. The central definition of the new formalisation is the desugaring extension problem, i.e. extending a set of established translation rules by synthesising new ones.

In a synthesis algorithm, the choice of search space is important and non-trivial, as it needs to strike a good balance between expressiveness and efficiency. The rest of the thesis focuses on defining search spaces for translation rules via typing rules. Two prerequisites are required for comparing search spaces. The first is a series of benchmarks, a set of source and target languages equipped with intended translation rules between them. The second is an enumerative synthesis algorithm for efficiently enumerating typed programs. I show how algebraic enumeration techniques can be applied to enumerating well-typed translation rules, and discuss the properties expected from a type system for ensuring that typed programs be efficiently enumerable.

The thesis presents and empirically evaluates two search spaces. A baseline search space yields the first practical solution to the challenge. The second search space is based on a natural heuristic for translation rules, limiting the usage of variables so that they are used exactly once. I present a linear type system designed to efficiently enumerate translation rules, where this heuristic is enforced. Through informal analysis and empirical comparison to the baseline, I then show that using linear types can speed up the synthesis of translation rules by an order of magnitude.

Lay summary

Computers are curious machines – they do not work according to a fixed plan, but rather the plan itself can be changed. This plan is called a program. Processors nowadays can execute billions of instructions in every second, and the plans they follow in doing so can be exceedingly complicated. It is way beyond human capabilities to fine control these plans, to write the programs by instructions the machine understands.

Fortunately, computers can help manage the very problem their enormous capabilities cause. Programmers usually do not use the instructions built into the machine. Instead, they use programming languages to give a high level description of the plan. The computer itself interprets these high level plans through a program: an interpreter, that can translate the high level description to instructions the machine can understand.

Interpreters are very complicated. All that are in use contain bugs – they do not work exactly as we wish. They often behave in unexpected ways, differently how the programmer expects it, their behaviour can surprise even their creators. Moreover, for a different kind of computer with different built-in instruction set, we need a different interpreter to run the same programs. Since interpreters are so complex, it is very hard to make sure that two different version of the interpreter of a particular language interprets programs the same way – there are often subtle differences.

One way to tackle some of these problems is to create a reference version of the interpreter for the language, which does not try to efficiently execute the programs, and not tied to a particular architecture. Instead, it is defined mathematically, in a way that is both precise and (relatively) easy to understand for human experts. It is called a definitional interpreter, or operational semantics. It can be used to find bugs in existing interpreters, or to develop tools for programming that need a precise definition of the language.

Unfortunately, programming languages are rarely designed in advance with a mathematically precise meaning, and engineers need to extract the definition from an existing interpreter. This reverse engineering can be challenging and cumbersome, and few of the programming languages that are in use today has a full formal operational semantics.

The idea this thesis is built upon is that we can use computers to help us finding the operational semantics for an existing language, based on an existing interpreter. The computer can generate candidate definitional interpreters, and test them by comparing them to the existing implementation. As computers are really fast, they can test mil-

lions of potential mathematical definitions quickly, and find the one which corresponds to the actual behaviour of the interpreter.

This task, however, is still really hard. I imagined an interactive system where humans and computers work together. I investigated the assumption that the mathematical definition of a programming language can be found in small, incremental steps. I also investigated what kind of definitions are good to try out so that we have a good chance to find the one we would like. I evaluated my work on simplified examples, as presently we can not apply my methods to real world languages.

Acknowledgements

My first heartfelt thanks goes to James Cheney. I could not have wished for a better supervisor. His constant responsiveness, helpfulness, stability and reliability have been invaluable. The care and attention shown in his suggestions, criticisms, and questions, could be a course in itself. The critical feedback I received from him was always driving me forward, and never holding me back. His immense knowledge has been available to me for almost six years now, and it has enriched me and provided me with a lasting perspective into the future.

I am also hugely grateful to my second supervisor, Vaishak Belle, who has provided a perspective into wider applications of my research topic, always coming up with fresh questions and feedback, and never missing a chance to encourage me. The learning opportunities I had while working as a research assistant under his supervision were extremely useful on my PhD journey.

The present version of the thesis has been greatly improved by the insightful comments and recommendations of my two examiners, Meng Wang and Liam O'Connor. I greatly appreciate the time they took to thoroughly review the thesis, their feedback allowed me to correct many errors and strengthen the presentation.

Covid hit just halfway into the research, and Jane Hillston and Mary Cryan have been extremely helpful when I needed support.

The journey would have been lonely without my fellow researchers. I want to thank Michael P. J. Camilleri, my office mate for six years, for always being ready to listen to my problems and to cheer me up, and Weili Fu, who spent hours understanding my ideas and giving useful feedback. I would also like to thank Wilmer Ricciotti and Elizabeth Polgreen for pointing me towards research relevant to my topic.

I am obligated to my wonderful proofreader, László Vincze, who was always ready to re-read the rewritten text just one more time on short notice.

I am grateful to my family for their support throughout the PhD years. I am indebted to my parents, without whose assistance I would not have been here. This thesis is dedicated to the love of my life, my wife, Judit, without whose longstanding support and encouragement none of this would have been possible. Finally, I am grateful to my children, Sebő and Johanna, who bore with me through hectic times, and provided a turbulent and energetic work environment.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

The thesis – in particular Chapters 2, 4 and 6 – contains material presented in the following publication, of which I am first author:

- Bartha, S., Cheney, J., and Belle, V. (2021). One down, 699 to go: Or, synthesising compositional desugarings. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA)

(*Sándor Bartha, Edinburgh, Scotland, 2023)*

To Judit

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Many different languages are used to describe automated systems. In addition to the multitude of general purpose programming languages, large systems rely on various domain specific languages, configuration languages, query languages and modelling languages. The specification for libraries, frameworks or hardware components is often given by an abstract language. These languages frequently change their behaviour from version to version, further increasing their number. Every system seems to invent its own idiosyncratic notation to represent computations.

The specifications of these languages tend to be insufficiently precise. Ambiguities are inevitable if the specification is written in a natural language. Various implementations may interpret the requirements differently, making it difficult to port the programs between them. Future version changes may break properties that the creators are unaware of but that the users rely on, thus making programs unnecessarily difficult to maintain. Security risks may go unnoticed, impacting every system built with the language [Amin and Tate, 2016].

By contrast, formal semantics are a pure mathematical model of the programming language. Their purpose, like that of other mathematical models, is to help understand the system and guide our reasoning about it. Formal semantics are useful for the maintenance and analysis of programming languages. A formal semantics is a prerequisite for applying formal methods or static analysis. Moreover, the ability to reconstruct the semantics of an opaque system would make it possible to compare properties of the induced semantics with the intended design, aiding the rationalisation or redesign of the language.

The usage of formal semantics is standard in the hardware industry [Kern and Greenstreet, 1999], and there is a lot of research effort to formalise aspects of mainstream languages [Jung et al., 2017; Nienhuis et al., 2016]. However, their impact on the wider software industry is modest: the vast majority of systems in use today do not have formal semantics. This is partly because practitioners who design languages often lack the skills needed to write formal semantics. There are only a small number of languages, like Scheme [Abelson et al., 1991] or Standard ML [Milner et al., 1997], that were designed with formal semantics in mind. But more importantly, writing formal specifications is tedious, especially if we want to write the specification retrospectively for an existing system without one. To get a sense of the scale of the work involved in writing the full formal semantics of one language, we can look at some recent examples for popular programming languages, such as JavaScript [Maffeis et al., 2008; Guha et al., 2010], R [Morandat et al., 2012], Python [Politz et al., 2013], and PHP [Filaretti and Maffeis, 2014]. Each of these formal semantics is the result of months of work by research groups.

An appealing idea is to partially automate the process of reverse engineering semantics of programming languages. Krishnamurthi, Lerner and Elberty, the authors of ¨¨The Next 700 Semantics: A Research Challenge" (2019) gave an initial analysis of the problem of semi-automation, and posed it as a challenge for the research community. A full solution to their challenge would facilitate defining the formal semantics for programming languages. The title of their work alludes to Peter Landin's landmark paper "The Next 700 Programming Languages" (1966), written more than 50 years ago, which facilitated the creation of these languages in the first place.

The present thesis aims to envision a possible solution: an assistant software to ease the work of semantic engineers. This goal can perhaps best be characterised as a semi-automatic *semantics synthesis assistant* requiring additional user guidance or feedback in the course of its operation, and the present work hopefully brings this one step closer to realisation. Throughout my work, I will follow and build upon the pioneering analysis of Krishnamurthi et al., hereinafter abbreviated as KLE.

1.2 Overview of the problem

Let us step into the shoes of the semantics engineer, whose task it is to reverse engineer interpretable formal specifications for an actual programming language by observing the behaviour of its opaque or non-intelligible interpreter. For textual languages, these

formal descriptions are divided into two parts: the language's (concrete) syntax, which describes the translation of the string representation into abstract syntax trees (terms), and its semantics, which describes the behaviour of the terms. We can distinguish between two separate tasks: reverse-engineering a formal grammar for a language from its examples, and reverse-engineering formal semantics for the term language.

Reverse-engineering a formal grammar for a string language from its examples is also an interesting problem. Known as unsupervised grammar induction, or grammatical inference, it is a well-studied task with applications in pattern recognition, computational biology, and, most importantly, in natural language processing [Clark and Lappin, 2010; Muralidaran et al., 2021]. But this is a different task, and not that important to semantic engineers, as most languages already have formal syntax: parsers in interpreters and compilers are routinely generated from a grammar given in BNF or similar formats. Moreover, not all languages even have string representation: for example, there are some abstract languages used to describe libraries, and some intermediate languages which are only represented by data structures inside compilers. In the present thesis, we focus solely on term languages consisting of abstract syntax trees, and by "language" we always mean a term language.

Languages are usually riddled with many varieties of interrelated constructs. The tedious parts of producing formal specifications involve writing a lot of small example programs covering the wide range of language features, then testing their behaviour through the opaque implementation. As a first step towards achieving partial automation, KLE suggest that semantics be divided into a complex part provided by human semantic engineers, and a tedious but shallow part hopefully synthesised automatically. They formulate this division as follows: let human engineers first specify a core version of the language, capturing its essential features, reducing the potentially hundreds of language constructs to a few. Next, have them provide a definitional interpreter (e.g. a direct implementation of an operational semantics) for this core language, which is much smaller. The computer's task then is reduced to finding translation rules from the original (source) language to this core language, based on observing the behaviour of source programs. Our goal is to automatically find a program that translates source terms into core terms.

To use a familiar example, let the source language be a simple functional language with arithmetic. We give its semantics by first providing the semantics of the λ-calculus extended with the primitive types (numbers) and operations (arithmetic) of the language, then translating additional language constructs into this core language.

$$\begin{aligned} \textbf{let}\ x = 1\ \textbf{in}\ x + x &\rightsquigarrow 2 \\ \textbf{let}\ x = 2\ \textbf{in}\ 1 &\rightsquigarrow 1 \end{aligned}$$

(a) Partial input: test cases for **let** expression

$$[\![\texttt{let}\ i = e_1\ \texttt{in}\ e_2]\!] = \left(\lambda i\,.\,[\![e_2]\!]\right)[\![e_1]\!]$$

(b) Partial output: translation rule for **let** expression

Figure 1.1: Learning translation rule example

Figure 1.1a shows some potential test cases run on the opaque interpreter of the source language, and Figure 1.1b shows an example translation rule that should be generated based on these test cases. The rule expresses a **let** expression (in the source language) in terms of a λ expression and application (which are both parts of the reduced core language). This example shows the semantics of one language construct for the sake of demonstration, but in the general case, the source language may contain hundreds of constructs that need to be reduced to the core language: this reduction of the number of term constructors is the main point of the translation.

KLE also suggest a natural search space: compositional translations, which we already relied on in our example. They aim to synthesise a separate rule (or rules) for each source language term constructor.

Their formulation is insightful and practical, but at this point, the specified task is still open-ended. Without additional assumptions, the framework can apply to any compositional translation between formal languages. While this makes the formulation widely applicable, it also presents a very difficult challenge. The challenge is hard in two ways. First, it is a difficult problem computationally:

- The search space is enormous, even larger than the search spaces of common, already notoriously hard problems in program synthesis.
- The learning framework is non-standard, which prevents us from applying most existing learning methods out-of-the-box.

Second, KLE's the informal description leaves many important questions open:

- What is a shallow translation rule?

- What class of languages do we target: can it be non-deterministic? What side effects can it have?
- Is the core interpreter opaque?

Accordingly, the design space of the open-ended problem is very complex, as it allows several different problem formulations. The overall difficulty of the challenge is well illustrated by the fact that KLE investigated four natural-seeming approaches based on existing learning or synthesis algorithms, but found that none were able to fully solve the problem they set.

1.3 Contributions of this thesis

As suggested by the above, my goal was to set up a research path that enables gradual progress. This involved looking for simplifying assumptions and developing an empirically testable baseline solution for the simplified problem in tandem. Chapter 2 presents the research context, contains a critical analysis of the open-ended challenge of KLE, and details our two main simplifying assumptions corresponding to these joint tasks:

- The main simplifying assumption is that it is possible to learn the semantics incrementally, in small steps. In the present thesis we work under the assumption that the user provides an appropriate decomposition of the language, and the automated solution is only required to find a portion of the full compositional translation, while at the same time it can rely on the part of the translation which is already established.
- The main assumption behind the empirically testable solution is that it is useful to limit our attention to only one aspect of program synthesis—the search space—and employ enumerative synthesis as the search technique.

Accordingly, the main contribution of the thesis can be summarised in two points:

1. Establishing a formal framework in Chapter 4 that both clarifies the informal description of KLE and re-formulates the problem based on my own critical analysis. The formalisation highlights that there are important choices left unattended in the informal description. The reformulation shifts the focus of the framework to the conditions for incremental learning, which not only allows us to set up

much simpler and easier-to-solve subtasks, but also allows a continuous transition from trivial tasks to a full solution. Accordingly, at the heart of the chapter lies the definition of the *desugaring extension problem*, a central problem for our research, which poses the task of automatically extending a set of established translation rules with a small number of new rules covering new language features. The framework also contains the simplifications needed to ensure direct testability, and clarifies the concept of compositional translations to enable the definition of search spaces for them.

2. Defining search spaces by typing rules to find a good balance between expressiveness and efficiency, and evaluating them on a set of benchmarks. This involved a series of steps:

   (a) Chapter 3 defines a series of benchmarks which I implemented in Haskell. That is, the chapter presents simplified source and target (core) languages, and intended translation rules between them. The first benchmark is the pair of languages KLE used to demonstrate the potential complications of modelling desugarings. Dubbing these the Pidgin source and core, I implemented them and used them to evaluate the expressiveness of search spaces. I also implemented two other benchmarks, one of which was based on translating list comprehension syntax into monadic operations, inspired by the (more complex) definition of list comprehensions in Haskell, and a second which was a translation of a try/catch/finally construct into try/catch constructs.

   (b) Chapter 5 presents a general method for enumerating typed program fragments, based on algebraic enumerations of type derivations. The enumeration library is also implemented in Haskell, and the enumerative synthesis algorithm employed in later chapters is based on it. Reviewing algebraic enumerations that were originally applied to enumerating algebraic data types, I show that they can be applied to the more general task of enumerating typed term languages with a context of variables, based on the deductive (typing) rules. I also demonstrate the possibility of applying it to more expressive languages, like fragments of the simply typed λ-calculus. I highlight the conditions required for a (relatively) efficient enumeration. Note that the content of this chapter is based on what I found efficient in practice, and lacks formal analysis or exhaustive comparison to other meth-

ods. The evaluation of the enumeration method is through the case studies presented in later chapters. Note that our paper [Bartha et al., 2021] relied on this enumeration algorithm for evaluation, but it did not go into discussing it.

(c) Chapter 6 creates a baseline search space for the re-formulated problem, and evaluates it on the set of benchmarks using enumerative synthesis. This can serve both as a baseline for future improvements to compare to, and as a building block of future synthesis algorithms employing more sophisticated search techniques. The successful synthesis of the intended translations of the benchmarks requires not only a user-defined decomposition of the full benchmarks into a series of steps, but additional user guidance as well, which is carefully recorded. The combined run-times obtained for each full benchmark (all which are solved incrementally, in multiple steps) are summarised in the following table:

Benchmark	Time
Pidgin	40min 32s
List comprehensions	25min 30s
Try/catch/finally	2h 5m 49s

(d) Chapter 7 creates a refined, linear search space, based on the observation that in most translation rules, variables are used exactly once. I gave an informal comparison of the asymptotic growth of the base and linear search spaces, and evaluated the linear search space on a subset of the benchmarks, demonstrating that these heuristics can lead to faster learning by an order of magnitude:

Benchmark	Time	Baseline	speed-up
Pidgin	2min 25s	40min 32s	$\times$16.8
List comprehensions	1m 49s	25min 30 s	$\times$14.0

The open-endedness of the problem allows for many different approaches, and we were necessarily obliged to limit the scope of the current thesis. Hoping that this work will lay a helpful foundation for this fascinating new area of research, it is important to highlight its boundaries. These can be summarised in four points:

- Some simplifying assumptions allow for directly testing correctness in a relatively straightforward way, but they also limit the solution to a restricted subset of the general problem domain.

- The enumerative algorithm requires user guidance: this is expected, as mirrored in the envisioned assistant software rather than a fully automated learning algorithm.
- The enumerative algorithm does not scale to larger programs.
- The set of benchmarks used for evaluation are limited.

I view these not as fatal flaws, but rather possible new research directions, which I summarise in the concluding Chapter 8. I believe that this thesis could serve as the foundation for future works in this area by providing a formal definition of the problem and by defining search spaces that could be used in other synthesis algorithms.

1.4 Thesis structure

The content of the thesis is structured as follows:

Chapter 2 presents the research context and a critical analysis of the open-ended challenge of KLE, and details our two main simplifying assumptions. We assume that incremental learning is possible, and delineate the scope of the present work by focusing on a single aspect of synthesis: the search space.

As the challenge presented by KLE is open-ended and hard, it is premature to aim for a full solution. Rather, we need to content ourselves with gradual progress, along with a way to quantify said progress. Chapter 3 introduces benchmarks, which I rely on to both demonstrate the problem, and to empirically evaluate algorithms that synthesise translation rules.

Chapter 4 specifies a formal framework for incremental learning, defining the *desugaring extension problem*, a central problem for our research.

Chapter 5 describes a general enumeration algorithm that can be used for both synthesis and testing, and which also provides the basis for later chapters.

Chapter 6 explores the design space of the open problem, proposes search spaces, and evaluates my research assumptions and the enumeration algorithm on our case studies.

Chapter 7 contains a description of a linear type system, allowing for the fast enumeration of translation rules, and accelerating the process of synthesis by at least an order of magnitude.

Finally, Chapter 8 highlights the limitations of my approach, and identifies possible future directions for research.

Chapter 2

Problem analysis

In this chapter, I present the research challenge of KLE and my own views of the problem. Thus, the chapter is divided into two distinct parts. The first section reviews past work: the challenge and its research context, as well as all past solution attempts I am aware of. I also included a brief review of some of the research underpinning my own analysis. The second section is a critical analysis of the problem, followed by my own approach building on it, listing my assumptions and reasoning. The contributions of the present thesis are built upon this analysis.

2.1 The challenge and its research context

As already emphasised in the Introduction, the immediate context for the present thesis is the vision of "The Next 700 Semantics: A Research Challenge" by Krishnamurthi, Lerner and Elberty (2019, abbreviated as KLE). The authors highlight the need for and lack of formal semantics of many languages in use, and point out that the reverse engineering process required to retrospectively write formal semantics for an already existing language is tedious. They put forth the need for semi-automation: the task is to automatically synthesise parts of the semantics of a source term language. This challenge, and the research in the present thesis, lies at the intersection of two fields: formal semantics and program synthesis. We start with a detailed review of the challenge, as well as a brief summary of both fields in their relation to the challenge.

2.1.1 KLE's challenge

KLE suggest searching for the semantics of a programming language in a specific format. This allows sharing the work between a human semantic engineer and the semantics synthesis assistant program, but they also argue that this format is useful on its own merits.

The main idea of KLE is to divide the work between human and computer as depicted by Figure 2.1, quoted from their paper.

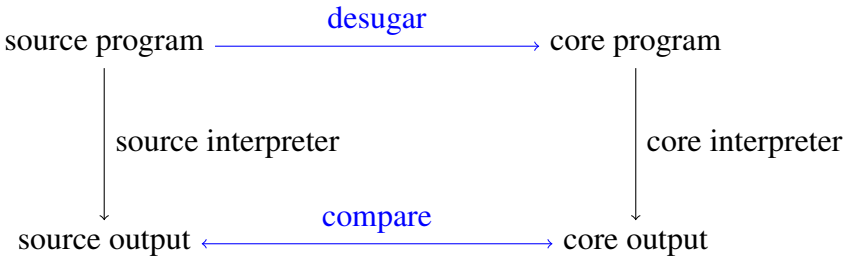

Figure 2.1: Testing strategy for desugarings (cf. Krishnamurthi et al. [2019])

They trust the human semantic engineer to write a reduced, *core* version of the source language, together with an operational semantics (a definitional interpreter) for this core language. Note that both languages are term languages. They expect the computer to automatically find a *desugaring* translation from the original source language to this reduced core version, based on the testing strategy shown in Figure 2.1. We can execute source programs with the source interpreter, or desugar the same source programs with a candidate translation into core programs, and execute them with the core interpreter. Comparing the results we can test the candidate translation. The comparison may need to take into account the candidate translation itself, as the outputs may use different representation for the source and core languages.

KLE choose the term "desugaring" (as opposed to more general words like compilation or translation) to emphasise the requirements that the translation be "shallow", non-expressive. They also expect the translation to be compositional, consisting of separate translation rules for every language construct in the source language. We will refer to this general and informally specified challenge as the *desugaring problem*, or alternatively as KLE's challenge.

2.1.2 The challenge's background in formal semantics

KLE's desugaring problem is based on an ongoing body of research in the area of formal semantics of programming languages.

We can summarise the main point of KLE's desugaring problem as follows:

1. We seek the semantics in the form of a *translation* between term languages, from the source language (whose semantics we seek) to the core language (whose semantics we assume to be known).

2. We look for the translation in a *compositional* form: we assume that it can be decomposed into separate rules according to the abstract syntax of the source language.

3. The translation we seek is assumed to be a shallow syntactic translation: a *desugaring*.

4. The semantics is *operational*, and thus executable and *testable*: through the operational semantics of the core language, we can compare the original implementation of the source language to the induced semantics.

The appeal of this formulation is that KLE build on standard practices in the area of programming language semantics.

Semantics as a translation The problem of defining semantics is as old as philosophy itself: we cannot create meaning from nothing. The early pioneers of formal (or mathematical) semantics of programming languages were well aware of this. Objecting to the criticism that formal semantics was necessarily circular—since the language in which we express the semantics should have formal semantics itself—John McCarthy famously retorted "Nothing can be explained to a stone" [McCarthy, 1966]. At a high level, meaning is often expressed as a translation from the language whose semantics we are interested in to a language whose meaning we presume to understand.

Compositional translation Defining the semantics structurally, following the abstract syntax of the source language, has a long tradition, going back at least to Burstall's "structural induction" [Burstall, 1969] and his student, Gordon Plotkin's structural operational semantics (SOS) [Plotkin, 1981]. The latter is also one of the most influential

frameworks for operational semantics. Many modern frameworks for operational semantics follow in its footsteps.

But unlike SOS semantics, KLE expects the semantics to be expressed by a translation. We expect this translation to be built up from separate translation rules for each source language construct in a structured, compositional manner. Thus, the semantics we seek is not only structurally defined, but also compositional.

Desugarings But, as KLE point out, not every semantics is equal in value: we wish to gain insight into the essence of the language. This motivates the division of the full semantics of the source language into an "essential" core language and a "shallow" translation into it. The distinction of expressive and shallow features goes back a long way, originally proposed by Peter Landin, whose seminal work, "The Next 700 Programming Languages" (1966) KLE alluded to. Landin introduced the expression "syntactic sugar" (the source of the technical term "desugaring"), describing how various concepts in programming languages can be explained as being a syntactic variant of λ-terms [Landin, 1964], essentially reducing a programming language to the λ-calculus. (The translation shown previously in Figure 1.1 is an example of syntactic sugar, demonstrating how a (non-recursive) **let** expression could be viewed as a syntactic variant of a λ-application.) Landin had many followers, most notably Reynolds [1997]. KLE themselves referred to the formal theory of expressiveness of Felleisen [1991], who summarises the various precursors of this distinction, and defined "syntactic abstractions" as an attempt to capture the same notion that KLE called *desugaring*. Felleisen lists many names for the notion across various areas, like "macros" in Lisp [Kohlbecker, 1986]. This theory of expressiveness is an important starting point as the first formal definition of what we consider "shallow", but, as is apparent from KLE's investigations, the exact distinction can be further refined, and in the present thesis we will consider a more broad definition of shallow translations. Nevertheless, even if the distinction between shallow and essential is unsettled, KLE argue that presenting the semantics this way has a long tradition and many benefits on its own, mainly, that it gives insight into the language.

KLE observes that a translation defined by a separate macro expansion for every source construct can be modelled by a tree transducer [Comon et al., 2007]. They highlight that this model is limited, however, and introduce a pair of simplified languages (a source and a core version) to demonstrate potential problems with the tree transducer model (hence using simple macro expansions as desugaring rules). We rely

on their example languages throughout this thesis, and dubbed them "Pidgin source" and "Pidgin core". These languages are part of our benchmark suite, and we quote them in Figure 3.1a.

KLE also argues that this distinction is also a good way of dividing the work between humans and machines. They recall their experience on two works presenting the full semantics of a language divided this way into an essential core and a desugaring translation: the semantics of Javascript [Guha et al., 2010] and the semantics of Python [Politz et al., 2013]. They argue that finding the core of a language requires insight and domain knowledge, so it is appropriate to entrust a human semantic engineer with this task. They also report that writing the semantics for this core language is usually straightforward, while writing the desugaring translation that covers every language feature can be very tedious, as it requires more repetitive work, further motivating the division of work between humans and machines.

Tested semantics If the semantics is operational, it can itself be implemented, serving as a mathematically defined reference implementation of the language (a definitional interpreter, as explained by Reynolds). This allows semantic engineers, who attempt to describe programming languages that already have an implementation but no semantics, to test their induced semantics against the original implementations of the language. Tested semantics have become a recurring theme in the last two decades, and KLE underscored the need for tested semantics, as it is important for established languages to have the formal semantics conform to existing implementations, rather than merely to ideals in the head of a researcher. KLE lists a long line of work in this area [Bodin et al., 2014, 2018; Bogdanas and Roşu, 2015; Charguéraud et al., 2018; Dasgupta et al., 2019; Ellison and Roşu, 2012; Filaretti and Maffeis, 2014; Guha et al., 2010; Hathhorn et al., 2015; Hildenbrandt et al., 2018; Kheradmand and Roşu, 2018; Park et al., 2015; Politz et al., 2012, 2013], which demonstrates the need for tested semantics.

Setting up the problem as finding a testable compositional desugaring translation between languages makes it widely applicable. This formulation is insightful and practical, but the challenge remains very hard. KLE's earnest efforts in publishing their four failed attempts is both illustrative of the difficulty of the problem, and provides significant guidance.

In the following section, we start with a brief review of program synthesis, followed

by a more detailed analysis of past solution attempts, focusing on KLE's findings.

2.1.3 A brief and high-level overview of program synthesis

The desugaring problem is a problem of program synthesis, but rather a unique one. It is relatively new, with not even any partially successful solutions for me to build on. The specification significantly differs from those used in major synthesis frameworks or algorithms.

This section aims to give a high level overview of program synthesis, providing context for the rest of the chapter. This will establish the background for past solution attempts detailed in Section 2.1.4. It serves as the basis of comparison for the differences between existing methods and the challenge in Section 2.2.1. And finally, it underpins my arguments for the direction I chose, as described in Section 2.2.2. The bulk of the thesis (the following chapters) does not directly build on the material presented in this section; rather, this section serves as the context and basis for my approach.

Program synthesis is a vast research area, with many methods and applications; listing them all would be beyond the scope of the present thesis. We refer the reader to Gulwani et al. [2017] for a recent survey of the field, acknowledging that this area is developing very quickly, with many newer results not included there. This section will summarise selected parts of the survey.

Program synthesis is a second-order search problem: the task is to find a program that satisfies a specification on all inputs. Search algorithms can be classified by three properties: the specification format, the set in which we look for a solution, and the search method. Gulwani et al. [2017] classifies program synthesis methods around these three dimensions: *user intent* (the specification), *search space*, and *search technique*. When we develop a new synthesis method, the user intent is largely dictated by the task at hand, while the search technique can be freely chosen. Search space selection falls somewhere in between these two extremes.

We start with a list of various forms of user intents and classes of search spaces. Next, we move on to briefly cover the most popular search methods.

User intent

User intent is the input of a synthesis algorithm, and can take various forms based on the application domain. As our aim is to highlight the difference between problem

specifications in program synthesis and the desugaring problem, we briefly list the most typical specification formats.

Examples Input-output examples are easy to obtain in some domains, but can be ambiguous. Hence the synthesised program may not generalise well. Example-based learning is known as inductive synthesis, or programming by examples, and is one of the most popular forms of input for synthesis algorithms.

Trace A trace is a detailed, step-by-step behaviour on certain inputs, and contains more information than simple input-output examples, but could be more taxing to provide from the user's perspective.

Logical specification A logical specification is a formal relation between inputs and outputs. It is precise and succinct, and can be very efficiently leveraged by constraint based systems. But it can be quite tricky to write, and many problems are not trivial to encode as logical constraints.

Templates Program synthesis methods often rely on cooperating with the user, by allowing them to write part of the solution. A template is a high-level program written by the user, with some "holes" in it that the synthesis algorithm is supposed to fill. A more limited version of templates is a sketch [Solar-Lezama et al., 2008], a template where the holes are over a bounded domain.

Hints The user can also specify program fragments that the solution should (attempt to) use.

Programs The specification itself can be a program, where the task is to find a more optimised version, or an observably equivalent program in a different language.

Static semantics Some systems expect the user to provide the type of the program in an expressive type system.

Natural language Many modern systems use natural language as the input, as it can be easier for a non-technical user to produce. Recent years saw enormous improvements in this area, with even domain-agnostic large language models capable of generating complex programs.

Search space

As we have already noted in the Introduction, one goal of the present thesis is to find good search spaces for KLE's challenge.

A crucial aspect of setting up such a search is finding a representation of programs—a search space—that can "strike a good balance between expressiveness and efficiency" (Gulwani et al. [2017], section 1.3.2). It can be a domain specific language or a subset of an existing language. It can be an imperative language, a functional language, or a logic language.

Sometimes the search space has limited expressivity, such as regular expressions or tree transducers, but often it is a subset of a Turing-complete language. Some synthesis methods target expressions of a first-order theory, like linear arithmetic.

The most common way to specify a search space is through a grammar, usually a context-free grammar: SyGus (syntax-guided synthesis) [Alur et al., 2013] is a popular standard for specifying synthesis problems using context-free grammars.

A grammar usually allows for the creation of a wide range of programs, and errs on the side of expressiveness. To alleviate this, the high level control structure is often limited, e.g. to sequential programs. Another way to set up a limited search space is by defining a parametric search space: sketches define a parametric search space over a bounded domain. In some statistical learning methods, the usual search space is a high-dimensional function space over real (floating-point) parameters.

Note that there is some overlap between the user intent and the search space. Some methods to provide user intent, like templates or types for the program, limit the search space, while others, like examples or natural language, do not affect it directly.

Search technique

Gulwani et al. [2017] highlights four important areas of search techniques: enumerative search, constraint solving, stochastic search and programming by examples.

Enumerative synthesis We will be testing search spaces by enumerative synthesis in this thesis, so it is worth covering this area in more detail. Note, however, that we do not build on common enumerative techniques, therefore the following only serves as a context for the research, not a foundation.

Enumerative search based syntheses are conceptually simple: we enumerate all programs generated by a grammar up to a given size, and check for a condition. Despite

being simple, they "have proven to be one of the most effective techniques for synthesising small programs in rich complex hypothesis spaces" (Gulwani et al. [2017], p. 57). The reason is that while enumerative methods are very sensitive to space explosion, excellent techniques have been developed for pruning and accelerating program synthesis. Thus, the main focus of enumerative methods is on pruning and, to a lesser extent, ordering the search space. Most implementations are limited to context-free grammars, but the basic pruning techniques do not depend on the grammar, and can be generalised. We start our overview by briefly listing the main techniques for pruning and acceleration.

There are two main basic enumerative search algorithms, both of which allow for different pruning techniques. Top-down methods, as their name implies, generate the terms of the grammar starting from the top of the abstract syntax tree, maintaining a set of partial derivations. They usually check for subsumption to prune the search space. Note that the subsumption relation depends on term equality, which, in synthesis algorithms, takes into account the properties of operators, such as commutativity.

Bottom-up methods start from the leaves (terminals) of the abstract grammar, and maintain a set of small programs from which to build larger ones. As the maintained set contains complete programs, we can execute them. Bottom-up methods often check for program equivalence to prune the search space.

Many award-winning synthesis methods are based on enumerative synthesis, including MagicHaskeller [Katayama, 2013], Unagi [Akiba et al., 2013], and EUSOLVER [Alur et al., 2017]. These usually employ additional methods for faster synthesis. Two such main techniques are offline exhaustive enumeration and unification. Offline exhaustive enumeration involves evaluating all programs up to a given size on a large set of predefined inputs, thereby obtaining a mapping between programs and input-output pairs, and using the map during the active phase to retrieve programs corresponding to examples. Unification in this context refers to a divide-and-conquer method, where we synthesise smaller programs on part of the specification, which are then unified for a full solution.

Another technique is to build a probabilistic model in advance, based on training data. The model attempts to predict program attributes, and can be used to augment various search techniques, including enumerative synthesis. While original approaches required pre-existing training data and learning the model before the synthesis phase, there is one enumerative synthesis algorithm that has been scaled by learning a probabilistic model during the synthesis phase, without relying on training data [Barke et al.,

2020].

Enumerations are also the main building block of more complex synthesis systems, where they are combined with other search techniques. CVC4SY [Reynolds et al., 2019] is an enumerative synthesiser that is part of a constraint-based synthesis framework. The Duet system [Lee, 2021] combines bottom-up enumerative synthesis with top-down propagation. Top-down propagation is a common divide-and-conquer search method in inductive synthesis, where the input-output examples are propagated downwards to smaller sub-problems.

Stochastic search Stochastic search techniques involve first learning a distribution over the search space, where the distribution is based on the specification, then sampling this distribution for programs. Three popular classes of techniques are the Metropolis-Hastings algorithm, genetic programming, and machine learning.

Metropolis-Hastings is a popular Markov Chain Monte Carlo (MCMC) algorithm. These methods define a distribution on programs using a score function, which measures how well the program satisfies the specification. Starting with a random program in the search space, we do a random walk, attempting to increase the score function. The distribution modelled by the score function is usually high-dimensional and highly irregular, where the Metropolis-Hastings algorithm is known to perform well compared to other MCMC algorithms.

Genetic programming is a different kind of stochastic search algorithm, based on genetic algorithms inspired by the biological process of natural selection. Genetic algorithms attempt to find a solution to a search problem through gradual optimisation. They keep track of a set of candidate solutions (a population), which are randomly modified by a set of operations, commonly including crossover (combining candidates) and mutation (small changes in a candidate). Candidates that perform better according to a metric will multiply, while those that perform worse are removed from the population. Genetic programming applies this general search method to programs.

Instead of specifying a distribution by a score function, we can try to learn it from the specification using machine learning. An example of this method is Menon et al. [2013], which learns the distribution from input-output examples. With the recent success of deep learning methods, it is no surprise that neural networks also have been applied to the task of program synthesis.

Constraint-based synthesis Constraint solving refers to the task of finding an instantiation of free variables in a logic formula that makes the formula true. In program synthesis, the entire specification (including syntactic restrictions) is encoded in a single formula, in the form of $\exists P \forall i \Phi(P, i)$, where P is the free variables determining the program, i is the free variables determining the input, and Φ is a logic formula expressing the specification. The two main methods to find an instantiation for P is through a SAT/SMT solver in solver-aided methods, or through inductive logic programming as a logic program. While the two areas were developed independently and are quite different, we may note that many inductive logic programming frameworks are based on answer set programming, where common implementations also rely on SAT/SMT solvers.

Encoding the specification in a logic formula is cumbersome and time-consuming. SKETCH [Solar-Lezama et al., 2008] was the first meta-synthesis framework for solver-aided synthesis, which could encode the specification in a host language, then solve synthesis problems through a built-in solver. SKETCH is named after sketches, high-level programs written by the user, with some "holes" over a bounded domain that the synthesis algorithm is supposed to fill. The holes are mapped to the variables of a logic formula, following a formal semantic framework for the SKETCH language. SKETCH is a highly influential framework with many applications and followers.

In inductive logic programming (ILP), the task is to generalise positive and negative examples to logic programs. The speciality of this technique is that the examples, the syntactic and semantic bias (templates for logic programs and their specification) are all represented as logic programs. As the outputs are also logic programs, ILP is also capable of learning non-deterministic programs, which most other methods do not target.

Programming by examples While the above three general methods can target synthesis problems given by various user intents, deductive search targets one very common specification format: input-output examples. The area is called inductive synthesis, or programming by examples. Deductive search is a divide-and-conquer algorithm, where we decompose the synthesis problem into smaller sub-problems by pushing the input-output examples through the grammar top-down. The most influential framework is called Prose (originally FlashMeta) [Polozov and Gulwani, 2015], which is based on inverse semantics for the operators, while Myth [Osera and Zdancewic, 2015] and Synquid [Polikarpova et al., 2016] are based on a type-theoretic interpretation of

synthesis problems, and guide the search by the type of the program in an expressive type system.

2.1.4 Past solution attempts

The present section provides an overview of other attempts that address the desugaring problem, following KLE and my own past work. KLE discuss four attempts to solve the desugaring problem: they report three attempts of their own, and analyse a fourth method from the research literature that addresses a similar problem. Although their summaries do not contain the details of the algorithms or their evaluation criterion, essential research insights can be gained from the accounts of their partial attempts. As these past attempts only share the goal but not the tools with the present thesis, the bulk of the thesis does not build on this section. However, the critical analysis in Section 2.2.2 refers to these research insights as well as to the summary of my own past work. A further goal of this section is to discuss related works, to give a more complete picture of the challenge and its history.

The first two attempts analysed by KLE

Their first two attempts, based on a naive tree matching algorithm and Gibbs sampling, were inspired by the solutions of a similar problem in natural language translation: learning tree transducers between two term languages. As KLE pointed out, there is a crucial difference between the desugaring problem and learning natural language translations: in the latter, we assume the existence of a corpus of translations between the source and target languages, but there is no ground truth to test a potential translation against. In their learning framework (Figure 2.1) the situation is reversed. We do not have example translations (in other words, the desugaring problem is not an inductive learning problem), but we do have ground truth: we can test a potential translation by comparing the two outputs obtained by the source and core interpreters. To accommodate the algorithms, KLE assumed the existence of such translation examples (i.e. pairs consisting of a source program and its corresponding intended translation in the core language), which would be unlikely to be much easier to produce than the rules themselves. KLE also highlights that the tree transducer model is limited, and can not express many naturally occurring translation rules even in theory. With their Pidgin languages (see benchmark 3.1), they highlight that many common intended translation rules in this domain cannot be expressed with tree transducers. While extended tree

transducers may handle these cases, the inference problem for extended tree transducers is even harder. Thus, the first two attempts required additional user guidance and relied on a limited search space.

The search method used in the first attempt was a naive search, which is better viewed as a program specification in terms of a naive algorithm than a serious attempt at a solution. The second method, based on Gibbs sampling, was developed building on this specification. A stochastic search algorithm based on Gibbs sampling was used to learn tree transducers. Gibbs sampling is a Markov Chain Monte Carlo algorithm, closely related to the Metropolis-Hastings algorithm, which we briefly reviewed in Section 2.1.3.

For the desugaring problem, we can set up a metric on translations using the testing framework depicted in Figure 2.1. We can generate source programs, then test whether the two outputs (through the source interpreter, and, via the candidate translation, through the core interpreter) agree for a given translation. A metric can be based on the percentage of source programs upon which a candidate translation passes this test. But this property is still too discrete, and KLE noted that the convergence of the algorithm is troublesome.

Third attempt: genetic programming

The third method was based on genetic programming, which we briefly reviewed in Section 2.1.3. In the case of the desugaring problem, we can use the same metric as we introduced for Gibbs sampling, the number of source programs upon which a candidate translation passes the test shown in Figure 2.1.

In theory, this method could work with an arbitrary search space, and is not limited to tree transducers. It also does not necessarily need additional user guidance. However, they found that convergence is still troublesome when using genetic programming, and KLE noted that they could not scale it to even the simplified Pidgin languages. We can conclude that it is not straightforward to apply stochastic search algorithms to the desugaring problem, because the score function we have at hand is too discrete.

Fourth attempt: constraint-based program synthesis

Their fourth method was a constraint-based program synthesis framework named SYNTREC [Inala et al., 2017]. Here, KLE's analysis is based on SYNTREC's reported

abilities and benchmarks, rather than their first-hand experience on applying it to the desugaring problem. The benchmark suite used in evaluating SYNTREC had some tasks to synthesise simple, structurally recursive translations between term languages.

SYNTREC is built on the top of SKETCH, extending it with an ability to automatically generate (part of) the sketches following structural recursion on an algebraic datatype. Thus, it allows a different form of user intent: instead of a sketch, we can define the high-level structure by algebraic data types (ADTs), and the synthesis algorithm generates structurally recursive translations from one ADT to another.

KLE noted that some benchmarks solved by SYNTREC can be seen as a simplified version of the desugaring problem, as these benchmarks synthesise mappings between algebraic data types that can represent languages. There are differences, however. Applying SYNTREC on the desugaring problem relies on additional assumptions, in order to accommodate the learning framework to constraint-based program synthesis. First, the problem needs to be simplified by assuming a shared output space for the two interpreters, allowing the comparison of the outputs directly, without relying on the translation to be synthesised. This avoids using the unknown translation (what we aim to learn) twice in the problem specification and is crucially relied upon in the constraint solver. Moreover, SYNTREC does not allow desugarings to deviate from a fixed structurally recursive structure; essentially, it can only synthesise tree homomorphisms. While this allows learning simple desugarings very fast, the method can not be applied to the more complicated desugarings of the Pidgin languages, which we will discuss in detail in the next chapter. KLE shows that the intended desugaring for the SFor source constructor, see Figure 3.1, cannot be expressed with this restriction, as it requires a deep re-arranging of children, which is common in desugarings. This limitation is shared between the search space of SYNTREC and three transducers used in the first two solution attempts.

Finally, SYNTREC also requires a deep embedding of the core interpreter into the framework: the core interpreter must be implemented in the framework's language, the SKETCH language. But, as noted by KLE, the formal semantics of even a relatively small language is too large for program synthesis methods to handle, and expanding these semantics in the framework results in huge constraints. The authors of SYNTREC reported an exponential explosion of the search space for certain target languages: when the target language has state, the SKETCH framework translates the implementation of the target language into huge constraints. In the benchmarks successfully solved by SYNTREC, the target languages are much simpler than what we

would expect from an "essence" of a programming language. A further problem with this approach is that producing such an embedding is already non-trivial.

In summary, KLE listed four problems with constraint-based methods:

1. the requirement of a shared output space for the source and core languages,
2. the search space excludes desugarings that re-arrange children,
3. the requirement of a deep embedding of the core interpreter into the framework, which could be hard to produce, and
4. the exponential explosion of the search space for core languages with state.

My own previous attempt: inductive logic programming

The only other work I am aware of in the context of synthesising semantics of a programming language is my own [Bartha and Cheney, 2020]. Note, however, that this work did not address KLE's challenge, instead focusing on a closely related but different problem. The following brief overview summarises my results and the obstacles I encountered through this line of research, and compares the task with the desugaring problem.

In the generic context of reverse engineering the semantics of an existing system, input-output examples are naturally available, as we can execute an existing program on arbitrary inputs. We can even use active learning, executing the interpreter inside the learning process. Inductive logic programming [Muggleton and de Raedt, 1994] aims to learn logical rules, generalising individual examples to logic programs. In my Master's thesis and the publication based on it [Bartha and Cheney, 2020], we used inductive logic programming to reverse engineer the semantics of a simple interpreter. In particular, we used meta-interpretive learning [Muggleton et al., 2014] and its implementation, the METAGOL framework [Cropper and Muggleton, 2016].

We extended the METAGOL framework, adding the ability to learn new rules for unobserved and partially specified predicates, relaxing the strict separation between specified and learnt predicates and the need for examples of the learnt predicate. Unobserved predicates do not have direct input-output examples; rather, the examples are transformed by predicates already specified by the user. My extensions were added to the official METAGOL implementation.

Based on these extensions, we incrementally learnt the SOS rules of a simple language from input-output examples. In SOS, the operational semantics of a language

is defined as a state transition system, where the states are composed of the program term and a configuration. The translation rules, that is, the SOS rules, syntactically transform the program term and move from configuration to configuration. In the simplified language we did not use any configuration, and the syntactic transformations were elementary.

The user intent of meta-interpretive learning is inductive examples and meta-rules. Meta-rules are templates for logic rules (Horn clauses), in which we can replace certain predicate or function symbols with template variables, which are filled by the synthesis algorithm. The main limitation of this approach was that it did not generalise well. It was not clear how to obtain templates without knowing the exact translation rules in advance for general SOS rules, i.e. how to set up a generic search space for SOS rules.

Note that this problem has two sides. One is connected to the algorithm itself, which expects user defined meta-rules, requiring the user to specify the high level structure of the logic program. This only allows for learning which individual predicates we use in this structure. This problem has been alleviated since: recent advances show a way to set up more general search spaces in meta-interpretive learning [Patsantzis and Muggleton, 2022]. The flip side of this problem, however, is connected to the task rather than the method: I never found a balanced search space that would cover a wide variety of SOS rules while still being efficient. This problem remains unsolved, and prevented me from extending my work to a wider range of languages. Future research could build on Patsantzis' work and try to answer this problem.

My own past work did not address the desugaring problem, instead focusing on the related task of learning SOS rules directly for the source language. We can summarise the advantages and disadvantages of the two approaches as follows. KLE's challenge, the desugaring problem, contains the idea of learning shallow translation rules, providing the basis for a balanced search space. The desugaring problem also nicely divides the work between human and computer by tasking the human engineer with the creation of the core language. Furthermore, in KLE's formulation the semantics is compositional, which may have benefits for the learning process. Learning SOS rules lack these advantages. However, learning SOS rules is an inductive learning problem, allowing us to rely on the generated input-output examples directly, which may open the path to classic inductive learning algorithms. In the present thesis, we follow KLE's approach, focusing solely on the desugaring problem.

The next section is an analysis of the challenging aspects of the desugaring problem in light of KLE's report, followed by putting forth my own approach in Section 2.2.2.

2.2 Critical analysis of the challenge

2.2.1 Challenges

Following the critical analysis of KLE, we can pinpoint two challenging aspects of the problem, around which all four solution attempts revolve: the unusual learning framework and the vast search space. Due to the unusual learning framework, three of the four attempts required assumptions that made them unlikely to be a viable improvement over manually writing the desugaring rules. In addition, the enormous search space resulted in all four attempts failing, even for a simplified example.

Unusual learning framework Let us look once again at Figure 2.1, quoted from KLE, that specifies the search task at a high level. It depicts a rather unusual learning framework, where most kinds of synthesis algorithms are not straightforward to apply.

One way to analyse the situation is to consider what kind of user intent is available, since it can narrow down the list of usable search methods. The setup is similar to an active inductive synthesis problem [Jha et al., 2010; Gulwani et al., 2017]: no logical specification is given, but we can produce input/output examples from evaluations of programs by the interpreters. But there is one crucial difference: the function we are trying to learn is not one we can directly test. Following KLE, in our problem statement we also assumed the existence of two languages, the source language and the core language, and their respective interpreters. We can evaluate the source programs with the source interpreter, and the core programs with the core interpreter — but we can only relate them to each other through the very translation we aim to learn. Writing input-output examples for the desugaring by hand will likely take more work than writing the desugaring itself. This rules out the direct application of inductive program synthesis methods, like the PROSE framework or inductive logic programming. KLE's first and second attempts encountered this very problem. The present thesis, on the other hand, does not assume the availability of any such examples. This, of course, also excludes traces from potential user intents.

Many program synthesis methods rely on formal constraints and derive solutions from a logical specification. We know from KLE's analysis of SYNTREC that encoding the desugaring problem as a logical constraint requires a deep embedding of the core interpreter into the constraint-generator framework. For example, for SYNTREC, we would need to implement the core interpreter in the programming language of the SKETCH framework. This highlights that there are two slightly different problems at

hand, depending on whether the core interpreter is treated as opaque (similarly to the source), or whether it is required to have an implementation accessible to the synthesis algorithm. Since we assumed that the semantic engineer produces a formal semantics of the core language, using it in the learning process sounds reasonable. But we have seen that producing the embedding is not trivial in the first place, and a deep embedding could lead to huge constraints.

In the present thesis, we assume that the core interpreter is opaque. On one hand, this makes the challenge harder: for example, it excludes constraint based methods. On the other hand, it simplifies the open problem considerably. The question of how to make use of an arbitrary core interpreter in the learning process, e.g. how to turn it into solvable constraints, is a hard problem in itself. I believe that the results presented in the thesis can be useful for synthesis methods that do make use of a transparent core interpreter, but in the present thesis, we will simply not rely on any such information.

When looking at the rest of the user intents listed in Section 2.1.3, we can see that a specification as a program or a natural language specification is not relevant for our use case. This leaves us with templates, hints, and types. The synthesis methods employed in the present thesis will make use all of these.

We may also consider what class of algorithms we can expect to work well. We can exclude inductive synthesis or constraint based synthesis, based on the lack of appropriate user intent. This still leaves us with two large group of synthesis algorithms that can be applied to diverse specification: stochastic search based synthesis and enumerative synthesis.

Two of the past solution attempts analysed by KLE are based on stochastic search: one is based on Gibbs sampling, and the other on genetic programming. KLE highlighted a common problem in stochastic search applied to program synthesis: the score function is too discrete. In my own opinion, while stochastic methods could play a major role in defeating the enormous search space, they are easier to apply to problems where we have a working non-stochastic approach to build on. In the present thesis, we do not work on stochastic methods. Instead, we look for a good search space, which may serve as a foundation for future research in stochastic search for the desugaring problem.

Enumerative synthesis techniques are simple, but should not be underestimated: they can be quite fast with clever pruning techniques. However, applying the usual pruning techniques to the desugaring problem is also not straightforward. The reason is that most common pruning techniques depend on the specific properties of the search

space, such as program equivalence (in bottom-up methods) or symmetries of operators (e.g. commutativity of addition, in top-down methods). Our expressions are parametric in the core language, therefore we do not have universal equivalences or symmetries. But even if we equip the core language with a computable equivalence relation or some similar information that can be used in pruning, equivalences or symmetries on the core language often do not translate to equivalences or symmetries of translation rules. When synthesising arithmetic expressions, we can treat addition as commutative, and prune the search space accordingly. However, when synthesising translation rules, addition is usually no longer commutative, as the core language most likely has some side effect which makes evaluation order matter.

Vast search space The second problem is shared with program synthesis in general: the astronomical size of the search space. In addition, the challenge extends the search space of the already notoriously hard program synthesis problem along a new dimension: the number of source term constructors, with which we intend to scale. For a compositional translation, we need to synthesise a separate "program" for each such constructor. This results in a high number of program synthesis problems, some of which are interdependent while others can be solved independently. Not only does the search space grow exponentially by the number of term constructors, but the term constructors' translation rules may depend on each other, which makes it difficult to test translation rules independently. The main point of the translation is to reduce the number of source term constructors, by translating the complex source language into a simple core language. Thus, we need an approach that scales well along this dimension.

The first two methods KLE analysed are based on a reduced notion of compositional translations: tree transducers. Their third and fourth attempts, based on genetic programming and constraint-based program synthesis, avoid this restriction and can, in principle, express arbitrary computable translation rules. But the application of general program synthesis methods such as these seems even harder: the search space for arbitrary computable translations is much larger than for tree transducers.

None of the methods succeeded in scaling up to the full Pidgin language (which, notably, is still not the size of a real world language). KLE highlighted the need for a reduced search space, but as they point out, tree transducers are too limiting. The partial progress reported by KLE highlights that this problem is very difficult. To make progress, we therefore make additional assumptions and reformulate the problem in a

way that hopefully helps with both challenging aspects: the learning framework and the extra dimension of the search space.

2.2.2 My approach

My approach rests on two ideas. One is a simplifying assumption for the problem: that incremental learning is possible. The other idea is to first focus on finding a good search space as a necessary step towards better solutions, leaving the other aspects of the synthesis algorithm (e.g. the search technique) for later. Thus, we limit our investigations to enumerative synthesis, whose success directly depends on the search space, and allows us to empirically test and compare search spaces.

Incremental learning When teaching students how to program, we typically do not immediately burden them with all of the language features at once, as this would most likely overwhelm or confuse them. Instead, our first lecture might start with "hello world", followed by gradually introducing related features in small groups. Indeed, Felleisen et al. [2001] explicitly adopted this approach by providing language "levels": self-contained sublanguages that intentionally exclude complex features for pedagogical purposes.

We follow KLE in their rational reconstruction of the problem, while exploring a simplifying assumption. I hypothesise that the compositional translation can be learned incrementally, only learning the translation rules of a few term constructors at a time. For this strategy to work, we must assume that starting from the rules of a few term constructors given initially, we can iteratively find small groups of term constructors (i.e. language "levels") whose translation rules can be found by testing them only on that part of the language where the translation has already been established. This idea originated in my Master's thesis, reviewed in Section 2.1.4.

An interesting question in incremental learning is whether sequential decomposition could be automated. In this thesis, we assume that the semantic engineer provides the sequential decomposition of the learning problem.

Search space The design of every successful program synthesis algorithm has three separate aspects: the search space, the search technique, and the user intent (see Gulwani et al. [2017] p. 7). KLE proposed a search space (tree transducers), but also proved that it was insufficiently expressive – so the choice of search space remains open. They also investigated four search techniques, among them two stochastic algorithms (Gibbs

sampling and genetic programming) and one constraint-based method (SYNTREC), but neither showed much promise. They also introduced the testing framework to (partially) specify the user intent, but three of the investigated algorithms made further assumptions not contained in the framework.

To limit the scope of the present thesis, let us focus our attention on the first aspect: the search space. Looking for a good search technique seems premature, and the search space is a necessary first step, a building block towards a successful synthesis algorithm.

We can test the search space by limiting ourselves to the brute force method of enumerative synthesis, as the success of an enumerative synthesis algorithm directly depends on the search space. Our primary objective is to find a good search space (crucial for any successful program synthesis algorithm); we can leave finding a good search method over that search space or the integration of the synthesis algorithm into a semantic engineering framework (that is, deciding how best to provide user guidance) to future research. This does not mean that we do not try to optimise our enumerative algorithm, or that we do not experiment with additional user guidance: these are required for beating even the simplest benchmarks. But we do commit ourselves to an enumerative synthesis algorithm, and let this choice dictate the rest of the investigations.

It is necessary to limit the scope of the investigated synthesis algorithms, as there are a wide variety of possible candidates (for a recent survey, once again see Gulwani et al. [2017]), and testing all of them is beyond the scope of the present research. Beyond the ease of comparing search spaces, there are a number of additional arguments favouring enumerative synthesis:

- Enumerative synthesis, being conceptually simple, is straightforward to apply to quite general sub-tasks, so the unusual learning framework does not pose a problem.

- It is not necessarily slow compared to other synthesis methods: enumerative synthesis has proven to be a very efficient technique in domains where the search space is rich and complex, but the size of programs is small (see Alur et al. [2013]). This description somewhat fits our sub-tasks, giving us reason to believe that it will serve as a good baseline.

- Enumerative synthesis serves as the building block for more sophisticated synthesis algorithms, some of which have fared well in competitions. See e.g. Alur

et al. [2017] or Reynolds et al. [2019] for recent competition-winning synthesis algorithms using an enumerative method as part of the algorithm.

For defining the search space, we focus on the meta-language in which we express the translation rules. We need to find a good balance between expressiveness and efficiency (see again Gulwani et al. [2017], p. 9), but we do not know how to exploit program equivalence on translation rules or symmetries like commutativity, which are the most common source of pruning. Instead, we employ a typed meta-language with a simple type system. These ideas will be elaborated in Chapter 4, which gives the conditions for incremental learning and provides a high level definition for search spaces, and Chapters 6 and 7 which explore search spaces defined as a fragment of a typed meta-language. But first, let us begin by introducing benchmarks, which will allow us to measure our progress, as well as to illustrate our definitions and methods.

Chapter 3

Benchmarks

Program synthesis is hard. When attempting to apply program synthesis to a new field, it is often not possible to immediately test the algorithms on real-world problems, as they are simply too complex. To measure progress, program synthesis research relies on benchmarks suites, collections of simple problems, often drawn from exercises for humans learning programming. We need such a benchmark suite for synthesising translation rules.

The current chapter presents a set of source and core languages, connected by shallow transformations. We can check whether a given algorithm can solve the problem in theory, and whether it can solve it within a reasonable time. The benchmarks will be used to demonstrate the problems and solutions (algorithms) presented, as well as to compare various implementations.

3.1 Pidgin languages

KLE presented a case study highlighting the challenge of learning desugarings from examples. They introduced two languages, a source and a core, and intended translations between them. Their main goal was to demonstrate translations that they would like to treat as shallow, but which cannot be expressed by tree transducers (they are not macro expansions). I will use their languages as my first benchmark: although they did not intend their example as a benchmark, these languages are useful for the comparison of search spaces because the special translation rules cover many features. For convenience, let us give a name to the languages they proposed: Pidgin. Figure 3.1 recapitulates the syntax of Pidgin's source and core languages, and quotes the intended translation of Pidgin source (red) to pidgin core (blue). Except for minor notational

differences, we repeat their definitions verbatim; some features, such as list "cons", are omitted but easy to add.

The syntax of the languages (as shown by Figure 3.1a) and their respective interpreters are used as the input of the learning process. The intended output is the set of translation rules described in Figure 3.1b.

KLE did not explicitly define the semantics of the core language. I wrote an implementation in Haskell, and choose a precise meaning of the languages. The following is a brief list of examples where clarification was needed:

- KLE did not specify evaluation order. We assume the core language to be a standard call-by-value lambda-calculus with arguments evaluated left-to-right.
- KLE did not specify which variables can appear on the left side of an assignment. We treat all variables, whether introduced by `CLam`, `SLet` or `CLetRec`, as assignable references, and we do not allow assignment for undeclared variables, which result in an error.
- KLE used multiple parameters in `CLam`, but did not specify whether an empty parameter list is allowed. We assume that `CLam` can have zero parameters (empty identifier list) and an application can have zero arguments (empty list).

KLE also relied on an informal notation for the desugaring rules. The desugaring rules used are intuitive—but not formally defined—notations for fresh names and list parameters, paraphrased in Figure 3.1b. In the rule for `SBetween`, the notations $\%i_1, \%i_2, \%i_3$ stand for freshly generated identifiers. In rules such as `SLam`, `SApp`, and `SList`, notations $[t, \ldots]$ and $[[\![t]\!], \ldots]$ stand for the (possibly empty) lists of sub-terms t and their translations. Finally, in the rule for `SFor`, the notation $[\texttt{SFBind}(i, t_3), \ldots]$ stands for a (possibly empty) list of bindings of identifiers i to expressions t_3, and on the right-hand side the notations $[i, \ldots]$ and $[[\![t_3]\!], \ldots]$ stand for the lists of the first and (translated) second components of the bindings, respectively (that is, the results of unzipping the list considering the bindings as pairs).

We describe reference implementations of the core language and the desugaring rules in Appendix A.

KLE introduced the Pidgin source and core languages to illustrate several potential complications in the modelling of translations: the presence of primitive types and operations, the unrestricted number of children, and the special role of names. Handling argument lists may require expressive translation rules that can re-arrange them (`SFor`)

$$
\begin{array}{rcl}
b \in \mathit{Bool} & ::= & \{\text{TRUE}, \text{FALSE}\} \qquad i \in \mathit{Id} \qquad n \in \mathit{Number} \qquad s \in \mathit{String} \\
o \in \mathit{Op} & ::= & \{0-, \textbf{not}, +, -, \wedge, \vee, <, >\} \\
t, t_1, t_2, t_3 \in \mathit{STerm} & ::= & \mathtt{STrue} \mid \mathtt{SFalse} \mid \mathtt{SNum}(n) \mid \mathtt{SVar}(i) \mid \mathtt{SStr}(s) \mid \mathtt{SBetween}(t_1, t_2, t_3) \\
& \mid & \mathtt{SPrim}(o, [t, \ldots]) \mid \mathtt{SIf}(t_1, t_2, t_3) \mid \mathtt{SLam}([i, \ldots], t) \mid \mathtt{SApp}(t_1, [t_2, \ldots]) \mid \\
& \mid & \mathtt{SLet}(i, t_1, t_2) \mid \mathtt{SLetRec}(i, t_1, t_2) \mid \mathtt{SAssign}(i, t) \mid \mathtt{SList}([t, \ldots]) \mid \\
& \mid & \mathtt{SListCase}(t_1, t_2, t_3) \mid \mathtt{SFor}(t_1, [f, \ldots], t_2) \\
f \in \mathit{SForBind} & ::= & \mathtt{SFBind}(i, t) \\
e, e_1, e_2, e_3 \in \mathit{CTerm} & ::= & \mathtt{CBool}(b) \mid \mathtt{CNum}(n) \mid \mathtt{CVar}(i) \mid \mathtt{CStr}(s) \mid \mathtt{CPrim1}(o, e) \mid \mathtt{CPrim2}(o, e_1, e_2) \\
& \mid & \mathtt{CIf}(e_1, e_2, e_3) \mid \mathtt{CLam}([i, \ldots], e) \mid \mathtt{CApp}(e_1, [e_2, \ldots]) \mid \mathtt{CLet}(i, e_1, e_2) \\
& \mid & \mathtt{CLetRec}(i, e_1, e_2) \mid \mathtt{CAssign}(i, e) \mid \mathtt{CList}([e, \ldots]) \mid \mathtt{CListCase}(e_1, e_2, e_3)
\end{array}
$$

(a) Syntax of Pidgin source (red) and core (blue) languages

$$
\begin{array}{rcl}
[\![\mathtt{STrue}]\!] & = & \mathtt{CBool}(\text{TRUE}) \\
[\![\mathtt{SFalse}]\!] & = & \mathtt{CBool}(\text{FALSE}) \\
[\![\mathtt{SNum}(n)]\!] & = & \mathtt{CNum}(n) \\
[\![\mathtt{SVar}(i)]\!] & = & \mathtt{CVar}(i) \\
[\![\mathtt{SStr}(s)]\!] & = & \mathtt{CStr}(s) \\
[\![\mathtt{SBetween}(t_1, t_2, t_3)]\!] & = & \mathtt{CLet}(\%i_1, [\![t_1]\!], \mathtt{CLet}(\%i_2, [\![t_2]\!], \mathtt{CLet}(\%i_3, [\![t_3]\!], \\
& & \qquad \mathtt{CPrim2}(\wedge, \mathtt{CPrim2}(<, \%i_1, \%i_2), \mathtt{CPrim2}(<, \%i_2, \%i_3))))) \\
[\![\mathtt{SPrim}(o, [t_1])]\!] & = & \mathtt{CPrim1}(o, [\![t]\!]) \\
[\![\mathtt{SPrim}(o, [t_1, t_2])]\!] & = & \mathtt{CPrim2}(o, [\![t_1]\!], [\![t_2]\!]) \\
[\![\mathtt{SIf}(t_1, t_2, t_3)]\!] & = & \mathtt{CIf}([\![t_1]\!], [\![t_2]\!], [\![t_3]\!]) \\
[\![\mathtt{SLam}([i, \ldots], t)]\!] & = & \mathtt{CLam}([i, \ldots], [\![t]\!]) \\
[\![\mathtt{SApp}(t_1, [t_2, \ldots])]\!] & = & \mathtt{CApp}([\![t_1]\!], [[\![t_2]\!], \ldots]) \\
[\![\mathtt{SLet}(i, t_1, t_2)]\!] & = & \mathtt{CLet}(i, [\![t_1]\!], [\![t_2]\!]) \\
[\![\mathtt{SLetRec}(i, t_1, t_2)]\!] & = & \mathtt{CLetRec}(i, [\![t_1]\!], [\![t_2]\!]) \\
[\![\mathtt{SAssign}(i, t)]\!] & = & \mathtt{CAssign}(i, [\![t]\!]) \\
[\![\mathtt{SList}([t, \ldots])]\!] & = & \mathtt{CList}([[\![t]\!], \ldots]) \\
[\![\mathtt{SListCase}(t_1, t_2, t_3)]\!] & = & \mathtt{CListCase}([\![t_1]\!], [\![t_2]\!], [\![t_3]\!]) \\
[\![\mathtt{SFor}(t_1, [\mathtt{SFBind}(i, t_3), \ldots], t_2)]\!] & = & \mathtt{CApp}([\![t_1]\!], [\mathtt{CLam}([i, \ldots], [\![t_2]\!]), \mathtt{CList}([[\![t_3]\!], \ldots])])
\end{array}
$$

(b) Intended translation of the Pidgin source language into the core language

Figure 3.1: Pidgin source language, core language, and desugaring

or can pattern match on the list (`SPrim`). Fresh name generation may be needed to control the order of evaluation of arguments (`SBetween`). KLE also highlighted that their proposed model, tree transducers, is unable to express the re-arrangement of arguments, or to generate fresh names.

The basic Pidgin languages will be the primary example used to illustrate our methods, as they conveniently contain all of these complications. We will also use them for evaluation. In more realistic examples, the source language could be much larger than the core language, since the point of the translation is to reduce the size of the language.

3.2 Pidgin extensions

We present our next benchmarks as extensions of the Pidgin languages (both source and core) with some common constructs.

3.2.1 Basic list comprehensions

For our first extension of the Pidgin languages we consider basic list comprehensions a la Wadler [1992] with the following syntax:

$$t \quad ::= \quad \cdots \mid [t \mid q] \qquad q \quad ::= \quad \varepsilon \mid x \leftarrow t, q \mid t, q \mid \text{let } x = t, q$$

This is a simplification over standard Haskell list comprehensions, which support patterns in bindings and more general definitions in `let`. They can be expressed by desugaring qualifiers to functions $[\![q]\!]$ mapping core terms to core terms, and desugaring a comprehension by applying the function to $[\![t]\!]$. The functions are expressed with λ-abstraction and application (substitution) over the core terms, both implemented in the meta-language. We use the familiar syntax $\lambda x.T$ for a λ-abstraction in the meta-language, and $\mathsf{app}(T_1, T_1)$ for application in the meta-language. We will use $\%u$ as a meta-variable, and implement application with substitution.

Figure 3.2a shows the extensions of the syntax of the Pidgin languages required for this case study, while Figure 3.2b shows the intended translation of the new source constructs into the extended core language.

The `Return` and `Bind` primitives correspond to the following standard Haskell functions:

```
return :: a -> [a]    (>>=)  :: [a] -> (a -> [b]) -> [b]
```

$$
\begin{array}{rcl}
t \in \mathit{STerm} & ::= & \ldots \mid \texttt{SListComp}(t,q) \\
q \in \mathit{SQual} & ::= & \texttt{QEmpty} \mid \texttt{QBind}(i,t,q) \mid \texttt{QGuard}(t,q) \mid \texttt{QLet}(i,t,q) \\
e,e_1,e_2 \in \mathit{CTerm} & ::= & \ldots \mid \texttt{Return}(e) \mid \texttt{Bind}(e_1,e_2)
\end{array}
$$

(a) Extending Pidgin with basic list comprehensions – syntax

$$
\begin{array}{rcl}
[\![\texttt{SListComp}(t,q)]\!] & = & \mathsf{app}([\![q]\!],[\![t]\!]) \\
[\![\texttt{QEmpty}]\!] & = & \lambda \%u.\ \%u \\
[\![\texttt{QBind}(x,t,q)]\!] & = & \lambda \%u.\ \texttt{Bind}([\![t]\!],\texttt{CLam}([x],\mathsf{app}([\![q]\!],\%u))) \\
[\![\texttt{QGuard}(t,q)]\!] & = & \lambda \%u.\ \texttt{CIf}([\![t]\!],\mathsf{app}([\![q]\!],\%u),\texttt{CList}([])) \\
[\![\texttt{QLet}(x,t,q)]\!] & = & \lambda \%u.\ \texttt{CLet}(x,[\![t]\!],\mathsf{app}([\![q]\!],\%u))
\end{array}
$$

(b) Extending Pidgin with basic list comprehensions – intended translation

Figure 3.2: Extending Pidgin with basic list comprehensions

The desugaring of comprehensions is described as a two-argument function $D[t|q]$ in the GHC documentation[1].

The generic comprehension desugaring rules used in GHC actually allow for arbitrary monads, not just lists. In contrast, the new core language primitives `Return` and `Bind` are monomorphic, only operating on lists. `Return` wraps an element into a one-element list, `Bind` can be expressed as the composition of a map operation and concatenation. We kept the generic names to highlight the correspondence with the Haskell desugarings. Also note that a desugaring relying on mapping and concatenation is much larger in our core language, where there is no function composition operator.

In GHC, guards are desugared to sequential compositions $(>>)$ and a guard operation for the monad. We omitted these constructs from the core language extension, since for lists, they are directly definable.

3.2.2 Try-catch-finally

Our third benchmark extends the Pidgin languages with exceptions. The source Pidgin is extended with **try**/**catch**/**finally**, which desugars to the core language extended with just **try**/**catch**.

[1] `https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/monad_comprehensions.html`

$$
\begin{array}{rcl}
t, t_1, t_2, t_3 \in \mathit{STerm} & ::= & \ldots \mid \texttt{STryCatchFinally}(t_1, i, t_2, t_3) \mid \texttt{SThrow}(t) \\
e, e_1, e_2, e_3 \in \mathit{CTerm} & ::= & \ldots \mid \texttt{CTryCatch}(e_1, i, e_2) \mid \texttt{CThrow}(e)
\end{array}
$$

Figure 3.3: Extending Pidgin with try-catch-finally – syntax

In a $\texttt{CTryCatchFinally}(t_1, i, t_2, t_3)$ expression, first t_1 is executed. If it terminates normally with value v, then t_3 is executed with default outcome v. If it raises an exception $\texttt{ex}$, then t_2 is executed with i bound to $\texttt{ex}$. If executing t_2 terminates normally with value v, then t_3 is executed with default outcome v. Otherwise, if t_2 also raises an exception $\texttt{ex}'$, then t_3 is executed with default outcome of raising $\texttt{ex}'$. (In particular, the absence of an exception handler can be emulated by having the handler be $\texttt{Throw}(i)$.) When the **finally** expression t_3 is executed, if it terminates normally with some result value, that value is ignored, and the default outcome is performed instead. If t_3 also raises an exception, then this exception shall be the result, and the default outcome is ignored.

To express this expected semantics in the core language we need to rely on thunking, otherwise the finally block would be inserted twice, resulting in code bloat. Thunking means that we can save an expression without evaluating it into a λ expression, and trigger evaluation by application.

Our intended desugaring is the following:

$$
\begin{array}{rcl}
[\![\texttt{STryCatchFinally}(t_1, i, t_2, t_3)]\!] & = & \texttt{CLet}(\%f, \texttt{CLam}([], [\![t_3]\!]), \\
& & \quad \texttt{CLet}(\%v, \texttt{CTryCatch}(\texttt{CTryCatch}([\![t_1]\!], i, [\![t_2]\!]), \\
& & \quad \%e, \texttt{CLet}(\%\_, \texttt{CApp}(\%f, []), \texttt{CThrow}(\texttt{CVar}(\%e)))), \\
& & \quad \texttt{CLet}(\%\_, \texttt{CApp}(\%f, []), \texttt{CVar}(\%v))) \\
[\![\texttt{CThrow}(t)]\!] & = & \texttt{CThrow}([\![t]\!])
\end{array}
$$

Figure 3.4: Extending Pidgin with try-catch-finally – intended translation

As usual, $\%f$, $\%v$, $\%e$ stands for fresh identifiers, while %_ stands for a fresh identifier that is never used anywhere (a dummy variable). We also give a more readable version of the intended desugaring of $\texttt{STryCatchFinally}$, in pseudo-code instead of the core language constructs:

```
let f = λ_.t3
in let v =
      try
          try t1 catch(i) t2 end
      catch(e)
          f() ; throw e
      end
in f() ; return v
```

However, in the core language there are no observable difference between the thunked version or a version which inserts the finally block twice. This means that learning the thunked version is not always possible.

Chapter 4

Formal framework

This chapter formalises the desugaring learning problem of KLE, and reformulates it as the *desugaring extension problem*, the central problem of my investigations. I define what it means to structure a full desugaring as a series of extension learning problems. While it is not clear whether partitioning into a sequence of desugaring extension problems is always possible, in later chapters I present empirical results showing that it is possible at least for the case studies.

I also focus on the necessity to find a good search space. My definitions aim to clarify the notion of "shallow compositional syntactic translation" (which I call a desugaring, following KLE).

The purpose of this chapter is thus twofold. On the one hand, it introduces the background, listing notations and concepts used in the rest of the thesis. On the other hand, it also presents a contribution of its own: a first attempt at formalising this new task. The definitions are the result of the rational reconstruction of the informal challenge, and I will elaborate on the motivation behind them. My hope is that these definitions, together with the search spaces presented in Chapter 6 and Chapter 7, will serve as a foundation for future research.

My definitions can be summarised as follows:

1. Modelling the abstract syntax, using multi-sorted term languages.

2. Using a resource limit in interpreters to ensure decidable termination.

3. Limiting the side-effects of interpreters to errors and non-termination.

4. Modelling interpreters and translations with the error monad (also called the exception monad).

5. Defining the compositional translations as structurally recursive functions.
6. Using monads to allow side effects in the translation rules.
7. Defining typed translation rules w.r.t. a user-defined sort-mapping from source sorts to core sorts.
8. Defining the correctness of translations by requiring that the mappings in the testing framework (Figure 2.1) commute.
9. Defining adequacy of translation rules by requiring that the translation is injective on values.
10. Defining sublanguages as subsets generated by a subset of term constructors that are closed w.r.t. the evaluation function, and extensions in terms of sublanguages.
11. Finally, defining the desugaring extension problem, and sequential learning as a series of desugaring extension problems.

I conclude the chapter with a series of questions raised by the framework.

4.1 Background - monads

Within our framework, we regard the interpreters of the source and core languages as opaque computations, and the translations as structurally recursive computations. Computations differ from mathematical functions; these differences are usually called effects.

Moggi [1991] described a uniform way to model computations with effects (he called these "notions of computations"), using category theory and monads. Wadler [1992] showed how Moggi's monads can be used to systematically structure pure functional programs that have effects, such that the structure provides modularity. Modelling computations as monads is commonplace, and I will be relying on them occasionally in my own definitions.

The two notions of monad, one in category theory and one in functional programming, are slightly different. The thesis does not require an understanding of category theory, and we refer interested readers to standard textbooks on the subject such as the classic from MacLane [1971], as well as Awodey [2006], which is more accessible to non-mathematicians.

Note that the effects that we rely on are very limited. We need to model errors (exception monad) and local effects in the translation, such as fresh name generation (which can be implemented with a state monad, for example). Both of these are easy to understand without the notion of monads. The bulk of our thesis will not rely on a formal treatment of these effects. As the present chapter serves as formal background, for completeness, we include the definition of a monad used by functional programmers, in the "Kleisli triple" form (see also Moggi [1991], Def. 1.2).

Definition 4.1.1 (Monad)**.** A *monad* is a triple:

1. A type constructor with a single polymorphic argument $\mathcal{M}$ (i.e. for any type τ the type constructor $\mathcal{M}$ creates a new type $\mathcal{M}\ \tau$). The type $\mathcal{M}\ \tau$ can be thought of as "computations of type τ with effects from $\mathcal{M}$".

2. A function $\mathbf{return} : \tau \to \mathcal{M}\ \tau$, polymorphic in τ, that embeds any computation of type τ into the effectful computations of type $\mathcal{M}\ \tau$, equipped with trivial effects.

3. A function $\mathbf{bind} : \mathcal{M}\ \tau_1 \times (\tau_1 \to \mathcal{M}\ \tau_2) \to \mathcal{M}\ \tau_2$, polymorphic in τ_1 and τ_2, that can compose effectful computations.

Such a triple is a monad iff it satisfies the three monad-laws:

1. The **return** function is left identity for **bind**:

$$\forall x : \tau_1, f : \tau_1 \to \mathcal{M}\ \tau_2. \quad \mathbf{bind}(\mathbf{return}(x), f) = f(x)$$

2. The **return** function is right identity for **bind**:

$$\forall m : \mathcal{M}\ \tau_1. \quad \mathbf{bind}(m, \mathbf{return}) = m$$

3. The **bind** function is "associative":

$$\forall m : \mathcal{M}\ \tau_1,\ \ f : \tau_1 \to \mathcal{M}\ \tau_2,\ \ g : \tau_2 \to \mathcal{M}\ \tau_3.$$
$$\mathbf{bind}(m,\ \lambda x.\mathbf{bind}(f(x), g)) = \mathbf{bind}(\mathbf{bind}(m, f), g)$$

Writing out monadic computations in terms of the **bind** function can be fiddly. A syntactic sugar used by the Haskell language is the **do** notation, which we will use on one occasion, where the **do** notation is not only much more readable, but will also provide a syntax more familiar to programmers less versed in functional languages. Note that the **do** notation is a generalisation of the list comprehension desugaring from Section 3.2.1.

Definition 4.1.2 (Do notation)**.** Let $m_1 : \mathcal{M}\ \tau_1, m_2 : \mathcal{M}\ \tau_2, \ldots, m_n : \mathcal{M}\ \tau_n$ be computations, and $f : \tau_1 \times \tau_2 \times \cdots \times \tau_n \to \mathcal{M}\ \tau$ be a parametric computation.

Then

$$\begin{aligned} \textbf{do}\ \ & x_1 \leftarrow m_1 \\ & x_2 \leftarrow m_2 \\ & \ldots \\ & x_n \leftarrow m_n \\ & f(x_1, x_2, \ldots, x_n) \end{aligned}$$

is notational shorthand for

$$\textbf{bind}(m_1, \lambda x_1.\textbf{bind}(m_2, \lambda x_2 \ldots \lambda x_n.\textbf{bind}(m_n, f(x_1, x_2, \ldots, x_n)) \ldots))$$

Note that in Haskell the notation covers many more cases, but we will only use it in this limited form.

4.2 Syntax

We employ multi-sorted term languages (see Meinke and Tucker [1993], section 2.1.4, page 201) as a standard model of syntax. The definition of the problem would work with single-sorted languages as well, but I chose multi-sorted terms for the following reasons:

- It is reasonable to assume the existence of a multi-sorted representation. We have assumed that the source and core languages have formal abstract syntax, which most likely employ multiple syntactic classes. The sorts correspond to these syntactic classes, with additional auxiliary sorts for lists and tuples if needed.
- This multi-sorted representation contains useful information. It allows us to limit translations to sort-preserving (typed) ones, which automatically excludes many ill-formed translation rules from the search.
- The usage of multi-sorted terms also highlights that in practice, we might not evaluate every term, only those that belong to the sort (syntactic class) of programs. There could be multiple syntactic categories that can be evaluated (like expressions, programs, modules, etc.). We ignore this potential complication, assuming that there is one distinct set of terms, called programs, that can be evaluated.

Definition 4.2.1 (Signature)**.** A *signature* Σ consists of

- A non-empty finite set of *sorts* S_Σ, with a designated sort $\sigma^p_\Sigma \in \Sigma$ for programs.
- A finite set of *term constructors* (or function symbols) $\mathcal{F}_\Sigma$.
- The *signature function* $sig : \mathcal{F}_\Sigma \to S^* \times S$, where S^* is a (possibly empty) finite sequence of sorts. We will write $f : \sigma_1 \times \cdots \times \sigma_n \to \sigma$ (where $f \in \mathcal{F}_\Sigma$ and $\sigma_1, \ldots, \sigma_n, \sigma \in S_\Sigma$) when $sig_\Sigma(f) = (\sigma_1, \ldots, \sigma_n; \sigma)$. The number n can be 0, in which case we will write $f : \sigma$, and will call f a *constant* (or nullary symbol).

For a term constructor $f : \sigma_1 \times \cdots \times \sigma_n \to \sigma$ we will call $\sigma_1 \times \cdots \times \sigma_n$ the domain, and σ the codomain of the term constructor. Note that term constructors are not functions and these domains and codomains are not sets, merely a shorthand reference to sorts in the signature.

Σ and Ω will stand for signatures $(S_\Sigma, \sigma^p_\Sigma, \mathcal{F}_\Sigma, sig_\Sigma)$ and $(S_\Omega, \sigma^p_\Omega, \mathcal{F}_\Omega, sig_\Omega)$, respectively. We will use Σ for the signature of the source language and Ω for the signature of the core language. We will also omit the Σ or Ω subscripts if they are unambiguous from the context.

Remark 1. Note that for simplicity, we do not use term constructors with an arbitrary number of arguments, therefore we need to model lists of arguments with an additional sort for the list and additional term constructors NIL and CONS. We do not have sort polymorphism, so we need to add a separate sort for each type of list, and add separate term constructors as well.

Remark 2. Also for simplicity, our model does not include an infinite number of constants. Literals such as numbers and strings therefore have their own term constructors. In our benchmarks in Chapter 3, we left out literals in the simplified grammar presented. For example, the simplified grammar of the Pidgin languages in Figure 3.1a does not contain the grammar rules of numbers or strings.

We could have distinguished between sorts that have literals and those that do not, with the possible intention that literals are shared between the source and target languages, and are preserved by translations. However, this makes no practical difference in our framework, as it does not matter whether the translation preserves literals or the term constructors of the literals. Therefore, we can omit this complication from the definition of signatures and languages.

Example 4.2.1. *In the Pidgin source language (see Figure 3.1a) the sorts are Id, Number, String, Op, STerm, SForBind,* $\texttt{List}_{Id}$*,* $\texttt{List}_{STerm}$ *and* $\texttt{List}_{SForBind}$*. The sorts of*

the core language are Bool, Id, Number, String, Op, CTerm, $\texttt{List}_{Id}$, and $\texttt{List}_{CTerm}$. We also consider each list sort to be automatically equipped with the suitable term constructors $\texttt{nil}_s : \texttt{List}_s$ and $\texttt{cons}_s : s \times \texttt{List}_s \to \texttt{List}_s$, with the sorts of identifiers, numbers and strings all having their own term constructors.

Definition 4.2.2 (Abstract language)**.** An *abstract language* (or language for short) over signature Σ is the set of terms defined inductively with the term constructors in the signature. We extend the signature function *sig* to terms, (ab)using the same notation for the extended function, and we will use T_Σ^σ for terms that belong to the sort σ.

The sets T_Σ^σ for all $\sigma \in S_\Sigma$ are defined inductively with the following rules:

1. If $c \in \mathcal{F}_\Sigma$ is a constant with signature $c : \sigma$, then $c \in T_\Sigma^\sigma$.
2. If $f \in \mathcal{F}_\Sigma$ is a term constructor with signature $f : \sigma_1 \times \cdots \times \sigma_n \to \sigma$, and $t_1 \in T_\Sigma^{\sigma_1}, \ldots, t_n \in T_\Sigma^{\sigma_n}$ are terms, then $f(t_1, \ldots, t_n) \in T_\Sigma^\sigma$.

We write $T_\Sigma = \bigcup_{\sigma \in S_\Sigma} T_\Sigma^\sigma$ for all of the terms belonging to any sort. The *size* of a term is the number of term constructors in it.

We call $T_\Sigma^{\sigma^p}$ the set of *program terms*, or programs for short, and will abbreviate it as T_Σ^p.

In functional programming, the terms of a language can be represented using a finite family of mutually recursive algebraic data types (defining a separate type for each sort in the signature). We will call such a set of types a *data system*. Note that we only allow finite structures, not infinite trees. In other words, we use well-founded semantics for algebraic data types.

In universal algebra terminology, an abstract language is the free multi-sorted algebra over the signature Σ.

4.3 Semantics

As explained earlier, we view both the source and core languages as opaque interpreters, which might seem natural to model as mathematical functions from terms to result values. However, as well-known, this approach is too simplistic, since there are several complications that may arise:

1. Some computations may not terminate. We may model the output of diverging computations with a special value ($\bot$), and model the interpreter as a total

function. But for Turing-complete languages this evaluation function is not computable.

2. The output space may not be part of the (input) language: it may contain opaque functions, locations, or other non-observable values.

3. Similarly, we may get errors because the source language is not defined on all terms. Some terms may be allowed by the abstract grammar, but still result in a syntax error or type error. For example, the abstract grammar of the Pidgin source language permits nonsensical expressions like `SPrim`$(<, [])$.

4. The observable behaviour may not be reduced to an input-output pair: there could be other side effects like I/O operations.

5. The interpreter may not be deterministic.

We rely on monads modelling the interpreters. Looking at our learning setting (see Figure 2.1), we can see that we need to be able to compose the source interpreter, the core interpreter, the translation between them, and a comparison operation: that is, we want all of them to live in the same monad. Note that this monad, being our model in the learning framework, is limited to observable effects. We do not need to model internal effects like state, which only reside in one of the interpreters. We only deal with effects where the program interacts with its context, the learning framework.

Modelling with an abstract monad is not enough, as we need to make sense of the comparison of outputs in the learning framework (see Figure 2.1), and the learning algorithm may depend on this comparison. For example, if the interpreters are not deterministic this comparison can only be approximated. Right now, I can only offer individual solutions for the different complications, leaving generalisations for later. In this section, I list the simplifications that I employed to deal with each of the aforementioned complications in turn, finally arriving at modelling with the error monad.

Non-termination Modelling non-termination is a central problem in denotational semantics. The classic solution uses domain theory, which models how non-terminating behaviour arises as a fixpoint of recursive equations. For an introduction of the subject and a summary of its history, see Abramsky et al. [1995]. In our model, we do not need to consider how non-termination arises; it is enough for us to simply acknowledge

that the source and core languages could be Turing-complete, and may not terminate. Therefore it suffices to treat non-termination as an error that can not be handled.

All of our languages in our case studies have non-terminating constructs. Due to the halting problem, it is impossible to decide whether a program in these languages terminates or not, or whether two such programs (the source and the core program) agree on termination. The comparison of the source and core behaviour can only be approximated. Our model assumes that the interpreter can be run with bounded resources (including time, CPU cycles, and memory), the evaluation function is computable, and all programs where evaluation exhausts resources evaluate to a specific error ($\perp$).

This means that the evaluation function has a parameter r (the *resource limit*) controlling the bounds on resources (the maximum time/memory available). The parametric evaluation function is only an approximation of the interpreter's behaviour, and all that we can guarantee is that for any particular program there exists a sufficiently large r such that the evaluation function behaves correctly on that program if the bound is at least r. For smaller values, the comparison of the source and core behaviour of the program may be incorrect.

This error could be in either direction. The comparison operation determines whether the source interpreter agrees with the composition of the translation and the core interpreter for a particular input. With a small resource limit, it is possible that the comparison operation reports that the two compared computations agree (or disagree), while the reality is the opposite. For a given bound, it could be that both the source and the core evaluation function returns $\perp$ (therefore agreeing), when with a larger bound they would return two different values, or one of them returns a value while the other diverges. For a given bound, it is also possible that one of the evaluation functions returns $\perp$ and the other does not (therefore disagreeing), while with a larger bound they would return the same value.

Remark 3. While this approximation makes the evaluation function computable, it may result in non-determinism. Even in a deterministic language, resource usage (such as time) may not be deterministic, and replacing the output with $\perp$ when the computation takes too much time reflects the running time in the observable output. However, we ignore this source of non-determinism. In our case studies, we implemented the "opaque" source interpreters ourselves, and allowed for limiting the number of steps in the evaluation. With a realistic interpreter, deterministic resource limitation may not be easy to implement.

Output space The output of a program could be opaque. For example, it can be an opaque closure, in which case we can only approximately compare such outputs by testing. We simplify our setting by assuming that output values (of a successfully terminating program) are part of the input language, such that we can apply translations on the outputs as well, and compare the resulting core terms.

Errors Interpreters may return errors. We extend the co-domain of evaluation functions with a set of errors, and require that the set of errors is shared between the source and core languages to allow for comparing the outputs.

Side effects We simplify our setting by not allowing any other observable behaviour but the output (which can be a value, an error, or $\bot$ when the computation does not terminate).

Non-determinism If the interpreter is not deterministic, then again the comparison of outputs can only be approximated. Not only are we unable to prove whether a candidate desugaring is correct on all possible inputs (this limitation is there for any inductive learning without a full specification, as we can merely test on a finite set), we may also not refute a candidate reliably, since a disagreement may be due to the non-determinism. The given opaque source- and core implementations would likely not provide any reliable way to explore the search space, making enumerative synthesis problematic. While the non-deterministic interpreters could be modelled in various ways, but specifically in the learning process we most likely need to employ statistical methods. As far as I know, all approaches to tested semantics rely on repeatable (i.e. deterministic) tests.

The present thesis assumes deterministic interpreters.

What we get through these simplifications is an interpreter that maps terms to either values (i.e. fully-evaluated terms) or errors. In other words, our computations take place in the error monad over the category of total computable functions. We give a simplified definition that suffices for our purposes:

Definition 4.3.1 (Error monad)**.** Let E be a fixed, finite set of errors that contains $\bot$ and any errors the interpreter may return (that are not part of the input language), such as **syntax_error**. We assume that E is disjoint from any set of terms in any languages, and let $\uplus$ mean the disjoint union of two disjoint sets. Our interpreters and translations

will be functions of the form $f : A \to B \uplus E$. We abuse notation by defining a notion of composition for such functions, propagating the error as follows: Let $f : A \to B \uplus E$ and $g : B \to C \uplus E$. Then

$$\forall a \in A, g(f(a)) = \begin{cases} g(b) & \text{if } f(a) = b \in B \\ e & \text{if } f(a) = e \in E \end{cases}$$

Definition 4.3.2 (Semantics)**.** Let R be a totally ordered set, the set of possible resource limits, and E the set of errors with at least one element $\bot \in E$. A (bounded) *evaluation function* (or *interpreter*) of a language T_Σ is a total computable function $\Phi : R \to T_\Sigma^p \to T_\Sigma^p \uplus E$. We treat the first argument as a parameter and we will use the notation $\Phi[r](t)$, visually distinguishing the resource limit parameter from the program term argument.

1. $\forall r \in R, \forall t \in T_\Sigma^p, \Phi[r](\Phi[r](t)) = \Phi[r](t)$ (idempotence)
2. $\forall r_1, r_2 \in R, r_1 < r_2, \forall t \in T_\Sigma^p, \Phi[r_2](t) = \bot \implies \Phi[r_1](t) = \bot$ (monotonicity)
3. $\forall r_1, r_2 \in R, r_1 < r_2, \forall t \in T_\Sigma^p, \Phi[r_1](t) \neq \bot \implies \Phi[r_2](t) = \Phi[r_1](t)$

The elements of the set

$$\left\{ t \in T_\Sigma^p \;\middle|\; \exists t' \in T_\Sigma^p, \exists r \in R, \Phi[r](t') = t \right\}$$

(the image of Φ apart from the errors E) are called the *values* of the language, denoted by $V(T_\Sigma)$.

If Φ is a bounded evaluation function, we will call the unbounded evaluation function Φ^∞ which maps from T_Σ^p to $T_\Sigma^p \uplus E$. The unbounded evaluation function is not computable in general, but can be defined as the limit of the bounded evaluation functions:

$$\Phi^\infty(t) = \begin{cases} \bot & \text{if } \forall r \in R, \Phi[r] = \bot \\ v & \text{if } \exists r \in R, \Phi[r] = v, v \neq \bot \end{cases}$$

Note that $\Phi^\infty(t)$ is the limit which always exists: in the second case, where there is a resource limit r where $\Phi[r]$ terminates (does not return $\bot$), due to condition 3, the bounded evaluation function can not change its return value for higher resource limits. This means that the value v is unique, and it is the limit of the bounded evaluation functions.

Remark 4. A value evaluates to itself (cf. Remark 7), that is: $\forall v \in V(T_\Sigma), \Phi^\infty(v) = v$.

4.4 Translations

The definition of permissible translations is the basis for defining search spaces. The requirements are partially identified by KLE, namely: 1) that the translation needs to be compositional and 2) it needs to be able to express the re-arrangement of children and 3) it needs to be able to generate fresh names. I also added two new requirements: 4) the translation needs to be able to return errors, and 5) the translation must be a typed translation between two sets of mutually recursive algebraic data types. In the present section, I proceed step-by-step to refine definitions to incorporate all of these requirements.

Typed translation By a typed translation, we mean one that conforms to the abstract grammars of the source and core languages. We restrict the potential translations to ones that preserves syntactic categories, which in our model are represented by the sorts.

In general, the sorts of the source and the core language are different, and indeed, in the Pidgin language the core language has Booleans but the source does not, while the source language has a sort for *SForBind* and the added $\texttt{List}_{\mathit{SForBind}}$, and neither has a corresponding sort in the core language. Preserving the sorts can be defined w.r.t. a mapping from the source sorts to the core sorts.

Definition 4.4.1 (Sort mapping)**.** Let Σ and Ω be two signatures. A *sort mapping* from signature Σ to signature Ω is a partial function $s : S_\Sigma \rightharpoonup S_\Omega$ from Σ-sorts to Ω-sorts which preserves the sort of programs: $s(\sigma^p_\Sigma) = \sigma^p_\Omega$. We will write $dom(s)$ for the domain of the sort mapping, which at least contains σ^p_Σ. A *total sort mapping* is a sort mapping where $dom(s) = S_\Sigma$.

Errors Translations take place in the same error monad as interpreters: this lets us compose them together with the interpreters. Translations can naturally return errors. If evaluating a source term returns a syntax error, then we may not want to map it onto the core language, we may simply want to translate it directly into the syntax error. For example, our intended translation of the Pidgin source language in Figure 3.1 does not give a mapping for a term with constructor *SPrim* and zero arguments. We may imagine the source interpreter raising a syntax error in this case, and it is more natural for the translation to raise the same syntax error directly, since the core language contains no corresponding constructs that result in a syntax error in the core

interpreter.

Definition 4.4.2 (Translations)**.** Let Σ and Ω be two signatures and s a sort mapping from signature Σ to signature Ω. Let E be the set of errors.

We will refer to a (total) computable function $\delta : T_\Sigma \to T_\Omega \uplus E$ as a *translation w.r.t. sort mapping* $s : S_\Sigma \rightharpoonup S_\Omega$ if it conforms to the sort mapping:

$$\forall \sigma \in dom(s), t \in T_\Sigma^\sigma \implies \delta(t) \in T_\Omega^{s(\sigma)}$$

We treat the sort mapping as a fixed part of the entire desugaring learning problem, thus we often talk about translations without explicitly mentioning the sort mapping, and the sort mapping should be understood from the context. Note that we interpret translations on the whole source language, not just on programs, because our aim is to define a special kind of translations, compositional translations.

Compositionality A straightforward way to define compositional translations is to treat them as structurally recursive functions. While this definition is too simple, we can use this as a starting point to introduce compositional translations, and extend it later as needed.

Definition 4.4.3 (Interpretation)**.** Let Σ be the source and Ω the core signature, and s be a total sort mapping from Σ to Ω. Let $f \in \mathcal{F}_\Sigma$ be a term constructor with signature $f : \sigma_1 \times \cdots \times \sigma_n \to \sigma$. Then, we will call members of the function space (of computable functions) $T_\Omega^{s(\sigma_1)} \times \cdots \times T_\Omega^{s(\sigma_n)} \to T_\Omega^{s(\sigma)}$ *interpretations* of f over the signature Ω. When the signature of the term constructor is $f : \sigma$ (that is, f is constant), then an interpretation is simply an element of $T_\Omega^{s(\sigma)}$.

Remark 5. We limit interpretations to total sort mappings, as the definition of a structurally recursive translation requires interpretation for all term constructors of the source language. We will extend the definition to partial sort mappings later.

Definition 4.4.4 (Structurally recursive functions)**.** Let Σ be the source signature and Ω be the core signature, and s be a total sort mapping from Σ to Ω.

An *interpretation* of the source signature Σ into the target language T_Ω is a (computable) function π assigning an interpretation to every term constructor $f \in \mathcal{F}_\Sigma$ over the signature Ω. That is,

- The domain of π is $\mathcal{F}_\Sigma$, and

- for every term constructor $f \in \mathcal{F}_\Sigma$ with signature $f : \sigma_1 \times \cdots \times \sigma_n \to \sigma$,

$$\pi[f] : T_\Omega^{s(\sigma_1)} \times \cdots \times T_\Omega^{s(\sigma_n)} \to T_\Omega^{s(\sigma)}$$

Note that since π is a function that returns functions, we use square brackets to visually distinguish the two function applications when π is applied to a term constructor f, and plain parentheses when the resulting interpretation is applied to terms.

We refer to $\pi[f]$ as the *translation rule* for f in π.

An interpretation of the source signature defines a sort-preserving function $\delta : T_\Sigma \to T_\Omega$ from source to core terms using structural recursion.

That is,

- For every $t \in T_\Sigma$ which is a constant, δ returns the same term as π:

$$\exists c \in \mathcal{F}_\Sigma, c : \sigma, t = c \implies \delta(t) = \pi[c]$$

- For every $t \in T_\Sigma$ that is a composite term, we apply δ recursively on the subterms, and use them as arguments of the interpretation returned by π:

$$\exists f \in \mathcal{F}_\Sigma, f : \sigma_1 \times \cdots \times \sigma_n \to \sigma, \exists t_1 \in T_\Sigma^{\sigma_1}, \ldots, t_n \in T_\Sigma^{\sigma_n}, t = f(t_1, \ldots, t_n) \implies \delta(t) = \pi[f](\delta(t_1), \ldots, \delta(t_n))$$

We will call such translations *structurally recursive functions*. In functional programming terms, these are "folds" over the datatype defined by Σ. In universal algebra terms, these are just multi-sorted Σ-algebras with signature Σ, where every $\sigma \in S_\sigma$ source sort has the set $T_\Omega^{s(\sigma)}$ as its carrier set.

Effects The translation rules of structurally recursive translations can not return errors or generate fresh names. For this, we need to extend the definitions from pure computable functions to computations. We will model computations as functions in a general functional programming language.

The capability to return errors is reflected in the type of translations, that is, errors are an observable, outside effect. Generating fresh names is purely internal: we only need to treat it as an effect because we want to decompose the translation into separate translation rules. The output of translation rules capable of generating fresh names depends not only on the arguments but also on the context (the other names generated). To the best of my knowledge, no standard mathematical model currently exists for

generating fresh names, but we can surely implement fresh name generation with other effects, for example via a state monad.

In general, translation rules may use any effects as long as they can be handled internally, so that the whole translation only has errors as observable effects. We need an internal monad for translations, with a specific run operation that handles the internal effects and takes us back to the error monad of the learning framework.

To change the definition of a structurally recursive function into one which allows us to work with effectful computations, we need to sequence the operations. For a composite term, we need to fix the order of evaluation of the recursive calls on the sub-terms: we evaluate arguments from the left.

Next, I will provide an informal definition relying on the familiar do-notation as the modelling language for computations over a monad.

Definition 4.4.5 (Compositional translation)**.** Let Σ be the source signature and Ω be the core signature, and s be a total sort mapping from Σ to Ω. Let $\mathcal{M}$ be the monad for our translation rules , with an operation

$$\mathbf{run} :: \mathcal{M}\, a \to E \uplus a$$

That is, **run** handles any effect that translation rules may use, except errors.

An *interpretation* of a term constructor $f \in \mathcal{F}_\Sigma$ with signature $f : \sigma_1 \times \cdots \times \sigma_n \to \sigma$ is a computation with type

$$T_\Omega^{s(\sigma_1)} \times \cdots \times T_\Omega^{s(\sigma_n)} \to \mathcal{M}\ T_\Omega^{s(\sigma)}$$

An *interpretation* of the signature Σ into computations over $\mathcal{M}\ T_\Omega$ is a computable function π that assigns an interpretation to every term constructor in $\mathcal{F}_\Sigma$. We call $\pi(f)$ the *translation rule* for f in π.

A *compositional translation* is a translation $\delta : T_\Sigma \to T_\Omega \uplus E$ that can be defined in terms of a computation $\delta' : T_\Sigma \to \mathcal{M}\ T_\Omega$:

$$\delta(t) \quad = \quad \mathbf{run}\ \delta'(t)$$

where δ' is defined on the terms of T_Σ by pattern matching on the term constructors, using the following scheme (for every term constructor $f \in \mathcal{F}_\Sigma$ with signature $f : \sigma_1 \times$

$\cdots \times \sigma_n \to \sigma$):

$$\begin{array}{lcll} \delta(f(t_1,\ldots,t_n)) & = & \textbf{do} & x_1 \leftarrow \delta(t_1) \\ & & & \ldots \\ & & & x_n \leftarrow \delta(t_n) \\ & & & \pi[f](x_1,\ldots,x_n) \end{array}$$

Note that here $\pi[f]$ is merely notation expressing which function is called. π should not be mistaken for a function in the programming language.

Re-arranging children Compositional translations can express certain re-arrangings of children, because translation rules can access the translated arguments and re-arrange the children after the recursive call.

Example 4.4.1. *Let the source and core languages both be a simple language of unlabelled trees, with arbitrary numbers of ordered children.*

$$\begin{aligned} S_\Sigma &= \{\mathit{SNode}, \mathtt{List}_{\mathit{SNode}}\} \\ \mathcal{F}_\Sigma &= \{\mathit{node} : \mathtt{List}_{\mathit{SNode}} \to \mathit{SNode}, \mathtt{nil}_{\mathit{SNode}} : \mathtt{List}_{\mathit{SNode}}, \\ &\qquad \mathtt{cons}_{\mathit{SNode}} : \mathtt{List}_{\mathit{SNode}} \times \mathit{SNode} \to \mathtt{List}_{\mathit{SNode}}\} \end{aligned}$$

$$\begin{aligned} S_\Omega &= \{\mathit{CNode}, \mathtt{List}_{\mathit{CNode}}\} \\ \mathcal{F}_\Omega &= \{\mathit{node} : \mathtt{List}_{\mathit{CNode}} \to \mathit{CNode}, \mathtt{nil}_{\mathit{CNode}} : \mathtt{List}_{\mathit{CNode}}, \\ &\qquad \mathtt{cons}_{\mathit{CNode}} : \mathtt{List}_{\mathit{CNode}} \times \mathit{CNode} \to \mathtt{List}_{\mathit{CNode}}\} \end{aligned}$$

The translation that reverses the order of the children can be expressed as a structurally recursive function with the following interpretation of the source term constructors:

$$\begin{aligned} \pi(\mathtt{nil}_{\mathit{SNode}}) &= \mathtt{nil}_{\mathit{CNode}} \\ \pi(\mathtt{cons}_{\mathit{SNode}}) &= \mathtt{cons}_{\mathit{CNode}} \\ \pi(\mathit{node})(\mathtt{children} : \mathtt{List}_{\mathit{CNode}}) &= \mathit{node}(\mathtt{reverse\,(children)}) \end{aligned}$$

This translation can re-arrange the children because the translation rules of $\mathtt{nil}_{\mathit{SNode}}$ and $\mathtt{cons}_{\mathit{SNode}}$ preserve the list structure; thus, the translation rule can re-arrange them after the recursive call. Compositional translations are limited in their ability to re-arrange children: the ability to re-arrange depends on whether the recursive call on the arguments preserves the children in the first place.

Partial sort maps In general, the sort map is not total. We extend compositional translations to partial sort maps as follows:

Definition 4.4.6 (Extension to partial sort maps)**.** Let Σ and Ω be the source and core signatures. Let $s : S_\Sigma \rightharpoonup S_\Omega$ be a partial sort mapping.

We create an extended language Ω' in the following way. The sorts of Ω' are:

$$S_{\Omega'} = S_\Omega \dot\cup (S_\Sigma \setminus dom(s))$$

We extend the s partial sort mapping into an $\bar{s}$ total sort mapping from S_Σ to S'_Ω:

$$\bar{s}(\sigma) = \begin{cases} s(\sigma), & \text{if } \sigma \in dom(s) \\ \sigma, & \text{if } \sigma \notin dom(s) \end{cases}$$

We also "copy" the term constructors belonging to the new sorts. For any $f \in \mathcal{F}_\Sigma, f : \sigma_1 \times \ldots \sigma_n \to \sigma, \sigma \notin dom(s)$, let $\bar{f}$ be a new term constructor with signature $\bar{f} : \bar{s}(\sigma_1) \times \ldots \bar{s}(\sigma_n) \to \bar{\sigma}$. We extend $\mathcal{F}_\Omega$ with these new term constructors:

$$\mathcal{F}_{\Omega'} = \mathcal{F}_\Omega \dot\cup \left\{ \bar{f} : \bar{s}(\sigma_1) \times \ldots \bar{s}(\sigma_n) \to \bar{s}(\sigma) | \forall f \in \mathcal{F}_\Sigma, f : \sigma_1 \times \ldots \sigma_n \to \sigma, \sigma \notin dom(s) \right\}$$

We will call δ a compositional translation from Σ to Ω w.r.t. the partial sort mapping s if

- It is a compositional translation from Σ to Ω' w.r.t total sort mapping $\bar{s}$.
- It maps every term constructor $f \in \mathcal{F}_\Sigma, f : \sigma_1 \times \ldots \sigma_n \to \sigma, \sigma \notin dom(s)$ to its copy $\bar{f}$. (In other words, it preserves the term constructors with which we extended the original target signature Ω)

Note that such a δ preserves the (partial) sort mapping s, making it a translation from Σ to Ω according to our definitions. In particular, it maps source programs into core programs.

Remark 6. Partial sort mappings allow some re-arrangements without the aforementioned limitations, when the arguments that we want to re-arrange belong to sorts that are not in the domain of the sort mapping.

Shallow translations In our intended translation of the Pidgin languages, `SPrim` has two rules. This poses no problem in the tree transducer model originally proposed by KLE, provided that the tree transducer is not deterministic. But in a compositional translation, we always assume one translation rule per term constructor, and the rule for `SPrim` can only be expressed by case analysis on lists.

Translation rules can analyse their arguments: this allows for case analysis or rearranging children. But this can also be too expressive: translations that can not be considered shallow in any sense can be expressed with deep analysis. We can limit this analysis by restricting the depth of analysis for arguments.

We will call a compositional translation *shallow* w.r.t. a (partial) sort mapping s if it never analyses terms belonging to sorts in S_Ω, that is, if it only performs case analysis on the "extra" sorts not in $dom(s)$.

This approach divides syntactic categories into two sets: one which contains elements we think about as "structures", such as lists or tuples, and the other we think about as "content". A shallow translation is one which can look into the structure, but not the content.

4.5 Correctness

Definition 4.5.1 (Soundness)**.** Let T_Σ and T_Ω be two languages, with two evaluation functions Φ_Σ and Φ_Ω, respectively. A translation $\delta : T_\Sigma \to T_\Omega \uplus E$ is *sound* iff (cf. Figure 2.1)

$$\forall t \in T_\Sigma^p, \Phi_\Omega^\infty(\delta(t)) = \delta(\Phi_\Sigma^\infty(t))$$

In other words, a translation is sound iff the following diagram (drawn based on the testing framework shown in Figure 2.1) commutes:

$$\begin{array}{ccc} T_\Sigma^p & \xrightarrow{\quad\delta\quad} & T_\Omega^p \uplus E \\ \Big\downarrow{\scriptstyle \Phi_\Sigma^\infty} & & \Big\downarrow{\scriptstyle \Phi_\Omega^\infty} \\ T_\Sigma^p \uplus E & \xrightarrow{\quad\delta\quad} & T_\Omega^p \uplus E \end{array}$$

Remark 7. A sound translation maps values to values or errors: if $v \in V(T_\Sigma)$, then $\Phi_\Sigma^\infty(v) = v$, and since δ is sound, $\Phi_\Omega^\infty(\delta(v)) = \delta(\Phi_\Sigma^\infty(v)) = \delta(v)$, which means that $\delta(v)$ is either a value or an error.

Note that soundness is not enough: a sound translation can still map everything to an error, or collapse all values into a single value. We would like that:

- Values are mapped to values and not errors.
- The translation is injective on values: different source values should not be mapped to the same target value.

We call these additional requirements *adequacy*:

Definition 4.5.2 (Adequacy)**.** Let T_Σ and T_Ω be two languages, with two evaluation functions Φ_Σ and Φ_Ω, respectively. A translation $\delta : T_\Sigma \to T_\Omega \uplus E$ is *adequate* iff

$$\forall v \in V(T_\Sigma), \delta(v) \in V(T_\Omega) \qquad \text{and} \qquad \forall v_1, v_2 \in V(T_\Sigma), v_1 \neq v_2 \implies \delta(v_1) \neq \delta(v_2)$$

that is, δ maps values to values and is injective when restricted to values.

Remark 8. For lack of better names we borrowed the terminology "soundness" and "adequacy" from denotational semantics. While our semantics is operational, it is defined compositionally through a translation, and we may think of the translation of the source program into the core language as its denotation. This gives further motivation for the adequacy condition, which, in denotational semantics, means that observably different programs should have distinct denotations. In our setting the observable part of a program is its output, so we interpret adequacy as the requirement that values need to have distinct translations. Note that together with soundness, this condition ensures that source programs with different outputs have different translation ("denotation").

Definition 4.5.3 (Correctness)**.** We will call a sound and adequate translation *correct*.

However, these conditions are not testable in practice, for two reasons:

1. The unbounded evaluation functions may not be computable; we can only approximate the conditions with a resource limit r.
2. The set of source programs can be infinite. We never restricted the evaluation functions in any way, so they can return anything on programs we did not test. We can only test the conditions on a selected test set of inputs.

Note that even if we fix the resource limit and assume that both evaluation functions are non-opaque, correctness is still undecidable by Rice's theorem, since it is a nontrivial semantic property of the program.

We define notions approximating correctness with a fixed resource limit and a set of source programs: a *test set*. In a general setting, we may assume that—putting non-termination aside—we can choose to evaluate arbitrary source language terms during the search. The following definition expresses a condition that is directly testable by synthesis algorithms.

Definition 4.5.4 (Correctness with respect to a test set)**.** Let T_Σ and T_Ω be two languages with evaluation functions Φ_Σ and Φ_Ω, respectively. Let $r \in R$ be a resource limit and let $I \subset T_\Sigma^p$ be a subset of programs of the source language. A translation $\delta : T_\Sigma \to T_\Omega$ is *sound with respect to the resource limit r and the test set I* iff

$$\forall t \in I, \Phi_\Omega[r](\delta(t)) = \delta(\Phi_\Sigma[r](t))$$

A translation is *adequate with respect to the resource limit r and the test set I* iff

$$\forall t \in I, \Phi_\Sigma[r](t) \notin E \implies \delta(\Phi_\Sigma[r](t)) \notin E \qquad \text{and}$$

$$\forall t_1, t_2 \in I, \Phi_\Sigma[r](t_1) \neq \Phi_\Sigma[r](t_2) \implies \delta(\Phi_\Omega[r](t_1)) \neq \delta(\Phi_\Omega[r](t_2))$$

A translation is *correct with respect to the resource limit r and test set I* iff it is sound and adequate with respect to r and I.

Remark 9. Although we assumed above that all values are representable, that is, the output space is a subset of the input language, languages with non-representable values could be handled by requiring that the example programs (members of the test set) do return a representable value.

4.6 Sublanguages

Our main idea is that compositional translations can naturally be partitioned along with the language. First, we define the notion of sublanguage and language extensions, followed by the extension of compositional translations.

Definition 4.6.1. (Sublanguage) A signature $\Sigma' = (S_{\Sigma'}, \sigma^p_{\Sigma'}, \mathcal{F}_{\Sigma'}, sig_{\Sigma'})$ is a *subsignature* of signature $\Sigma = (S, \sigma^p_\Sigma, \mathcal{F}_\Sigma, sig_\Sigma)$, denoted as $\Sigma' \subseteq \Sigma$, if

1. $S_{\Sigma'} = S_\Sigma$,
2. $\mathcal{F}_{\Sigma'} \subseteq \mathcal{F}_\Sigma$, and

3. $\forall f \in \mathcal{F}_{\Sigma'}, sig_{\Sigma'}(f) = sig_{\Sigma}(f)$.

Let Φ_Σ be the evaluation function of the language T_Σ. We will call $T_{\Sigma'}$ the *sublanguage* of T_Σ, if it is closed with respect to the evaluation function, that is:

1. $\Sigma' \subseteq \Sigma$, and

2. $\forall r \in R, t \in T_{\Sigma'}^{\sigma_p}, \Phi_\Sigma[r](t) \in T_{\Sigma'} \uplus E$

We will also call Σ an *extension* of Σ' and similarly T_Σ an extension of $T_{\Sigma'}$. We will also use Φ_Σ as the evaluation function of the sub-language $T_{\Sigma'}$ (which is a restriction of the evaluation function of the full language).

Remark 10. If Σ' is a subsignature of Σ and s is a sort mapping from Σ to Ω then s is trivially a sort mapping from Σ' to Ω as well, since a subsignature has the same set of sorts.

Definition 4.6.2 (Extension of a compositional translation)**.** Let $T_{\Sigma'}$ be a sublanguage of T_Σ, and let T_Ω be our target language. Let s be a sort mapping from Σ to Ω. Let π' be an interpretation of Σ' into T_Ω w.r.t. s.

We will call an interpretation π of Σ into T_Ω an *extension* of π' if it assigns the same interpretation to every term constructor in the sub-signature:

$$\forall f \in \mathcal{F}_{\Sigma'}, \pi'(f) = \pi(f)$$

We will also call the compositional translation δ defined by π an *extension* of δ' defined by π'.

We intend to consider different *search spaces* or *hypothesis spaces* (i.e. sets of possible desugaring rules to consider). We also might want to consider different search spaces for different language extensions.

Definition 4.6.3 (Search space)**.** Let us fix T_Σ as our source language and T_Ω as our target language. Let $T_{\Sigma'}$ be a sublanguage of T_Σ. The search space corresponding to the signatures Σ, Σ' and Ω is an enumerable set of interpretations (into T_Ω) for each term constructor $f \in \mathcal{F}_\Sigma \setminus \mathcal{F}'_\Sigma$.

We will use $\mathcal{H}$ to stand for our chosen search space, and $\mathcal{H}^f$ for the set of interpretations assigned by the search space to the term constructor $f \in \mathcal{F}_\Sigma \setminus \mathcal{F}'_\Sigma$.

We will call a compositional translation that is an extension of the translation of the sublanguage and whose interpretations belong to the search space a *desugaring*

(when the source language, its sublanguage, the target language, the interpretation of the sublanguage into the target language and the search space is determined by the context).

4.7 The desugaring extension problem

We can finally define our main problem, a sub-task of the desugaring problem: extending a desugaring from a sublanguage to a larger portion of the language.

Definition 4.7.1 (Desugaring extension problem)**.** The *desugaring extension problem* is defined as follows:

Inputs:

1. A signature Σ of the source language.
2. A signature Ω of the target language.
3. A sort mapping s from Σ to Ω.
4. A resource limit r.
5. A finite test set consisting of input programs of the source language: $I \subset T_{\Sigma}^{\sigma_p}$, and their corresponding outputs according to an evaluation function Φ_{Σ} with resource limit r.
6. An opaque evaluation function Φ_{Ω} for the target language T_{Ω}.
7. A subset signature $\Sigma' \subset \Sigma$, that defines a sublanguage $T_{\Sigma'}$
8. A search space $\mathcal{H}$ for the language T_{Σ}, sublanguage T'_{Σ}, and target language T_{Ω}.
9. A correct translation defined on the sublanguage: $\delta' : T_{\Sigma'} \to T_{\Omega}$.

Output: A desugaring $\delta : T_{\Sigma} \to T_{\Omega}$, such that:

1. δ is an extension of δ'
2. $\forall f \in \mathcal{F}_{\Sigma} \setminus \mathcal{F}_{\Sigma'}$, the translation rule of f in δ belongs to the search space $\mathcal{H}^f$
3. δ is correct with respect to the resource limit r and test set I.

This is the central problem of the present work, and we will referring to it as DEP.

The *active* version of the desugaring extension problem differs in that instead of a test set, we have access to the opaque source evaluation function.

Definition 4.7.2 (Active desugaring extension problem)**.**

Inputs: The same as for the desugaring extension problem, except for the test set:

⋮

5. An opaque evaluation function Φ_Σ for the source language T_Σ.

⋮

Output: A desugaring $\delta : T_\Sigma \to T_\Omega$, such that the requirements are the same as in the desugaring extension problem, except that we want a correct desugaring, not merely one that is correct w.r.t. the test set.

⋮

3. δ is correct.

In the case of the active DEP, the output condition is undecidable. We have different aims with the two problems: for the non-active problem, we aim for an algorithm that only returns outputs that strictly conform to the post condition, although it may or may not find a result. In the case of the active problem, the post condition is interpreted loosely, and we aim for algorithms that return desugarings fulfilling it in most practical cases.

Remark 11. The main differences between the DEP and the task informally described by KLE are:

- We assume that the desugaring rules may be partially known.
- We explicitly use multi-sorted terms, a sort-mapping between the source and core languages, and sort-preserving translations.
- We assume that the test set of source programs is given as input, and the source language interpreter may not be called on inputs outside of this set.

For the active DEP, only the first two conditions hold. The first two conditions restrict the search space, allowing us to define feasible tasks, and the last one highlights the separation of synthesis and testing.

The benchmarks defined in Chapter 3 can be seen as a DEP. The sublanguage $T_{\Sigma'}$ is restricted to literals (numbers, strings), identifiers and operation symbols that are not included in the intended translation, as their translation is fixed in advance; the extended language is the full source language. However, the solution to this problem is currently unknown, as the search space is too large. Therefore, we divide each benchmark into a series of DEPs.

4.8 Sequential learning

Let us assume that we are looking for the full desugaring of a source language T_Σ into a target language T_Ω. The user should divide the language into an expanding series of sublanguages:

$$\Sigma_0 \subset \Sigma_1 \subset \cdots \subset \Sigma_n = \Sigma$$

where, for every $n \in [1 \ldots n]$, $T_{\Sigma_{n-1}}$ is a sublanguage of T_{Σ_n}. Assume that we know the translation for Σ_0 (which typically contains literals and operations for primitive types). The user also needs to provide a search space $\mathcal{H}_i$ and a test set I_i for each sub-task.

For now, we will not be investigating how to obtain a suitable partitioning of the language, or suitable test sets; we assume they are part of the user input.

Definition 4.8.1 (Sequential desugaring learning problem)**.** The definition of the full learning task is as follows:

Inputs:

1. A signature Σ of the source language.
2. A signature Ω of the target language.
3. A sort mapping s from Σ to Ω.
4. A resource limit r.
5. A series of sub-signatures: $\Sigma_0 \subset \Sigma_1 \subset \cdots \subset \Sigma_n = \Sigma$ where, for every $k \in [1 \ldots n]$, $T_{\Sigma_{k-1}}$ is a sublanguage of T_{Σ_k}.

6. A series of finite test sets $I_k \subset T_{\Sigma_k}$ for every $k \in [1 \dots n]$, and their corresponding outputs according to an evaluation function Φ_Σ.

7. A black-box evaluation function Φ_Ω for the core language T_Ω.

8. search spaces $\mathcal{H}_1, \dots, \mathcal{H}_n$.

9. A desugaring $\delta_0 : T_{\Sigma_0} \to T_\Omega$ defined on the minimal sublanguage, that is correct on T_{Σ_0} .

Output: A desugaring $\delta : T_\Sigma \to T_\Omega$, such that

1. δ is an extension of δ_0

2. $\forall k \in [1 \dots n], \forall f \in \mathcal{F}_{\Sigma_k} \setminus \mathcal{F}_{\Sigma_{k-1}}$, the translation rule of f in δ belongs to the search space $\mathcal{H}_k^f$

3. δ is correct with respect to the full test set $\bigcup_{k \in [1 \dots n]} I_k$.

Note that the sequential desugaring learning problem is almost the same as the original desugaring extension problem, we merely added some additional inputs dividing the search space, so we partition one desugaring extension problem into a series. An active version of the sequential DEP can be defined as well.

4.9 Conditions of a solution

I conclude this chapter with a short discussion of additional research questions entailed by my approach.

Do solutions exist? Can we solve sequential problems by composing solutions one extension at a time? Some desugaring extension problems may not have a solution, since the chosen search space may not contain the intended desugaring. But even if a correct desugaring exists, it is possible that the given partial desugaring (while being correct on the sublanguage) can not be extended to a correct full desugaring. The reason is that some state or value may not be accessible in the sublanguage, which makes some globally bad translations correct when only tested on the sublanguage. For example, desugaring a conditional **if** X **then** Y **else** Z expression into Y could be correct in the sublanguage, if in the sublanguage all Boolean expressions evaluate to true. Further examples are desugaring a **try** A **catch** E **in** H expression simply to A in a

sublanguage without exceptions, or desugaring a fst selector of a pair to an error value in a sublanguage without pairs. This means that a naive, greedy strategy for solving a sequential problem might not work: we might commit to the wrong semantics for a sublanguage too early, making later extensions impossible. These examples may seem trivial, but in practice, it seems likely that a work-in-progress semantics will be in such a "stuck" state most of the time, so understanding and finding ways of mitigating this situation is a major challenge.

Are solutions unique? In program synthesis, especially in inductive (example-based) synthesis, a common problem is having multiple programs that satisfy the specification (see Gulwani et al. [2017], chapter 7.4). Our task is similar: not only could semantically different translations be correct with respect to the finite test set, but even correct translation rules can often be expressed in multiple, semantically equivalent ways. These equivalent programs can often be transformed into each other by semantics-preserving transformations, such as swapping the two branches of a conditional expression and negating the condition, etc. Our task has a special ambiguity of this kind, that not only concerns the individual translation rules but the entire translation as a whole. The reason is that languages often have (many) *symmetries*: correct, invertible translations into themselves. A composition of two correct translations is still correct, so such symmetries induce many equivalent but syntactically different desugarings. Swapping two branches of the conditional is just the special case of the well-known symmetry of the Boolean language that exchanges TRUE and FALSE. It is possible to map Boolean constants to the opposite values, switch the **and** and **or** logical operations, and switch the order of the branches of the **if** conditional, extending the transformation from an individual translation rule to the whole translation. While such symmetries sometimes break in the full language (for example, the symmetry of TRUE and FALSE may disappear if we have a way to convert Boolean values to strings), they could still easily lead to the aforementioned stuck state. It may be necessary to redefine the DEPs, and fix the meaning of some term constructors manually.

Can we devise test sets that ensure correctness? It is also possible that there are multiple, semantically inequivalent solutions that are correct with respect to the given test set, but with not all of them being correct with respect to the whole source language. A question that arises is whether there is a finite test set I such that if a translation is correct with respect to I then it is correct with respect to the whole source

language. Assuming opaque evaluation functions, such a test set may not exist. What reasonable assumptions can we make about the evaluation functions to ensure the existence of such test sets? Moreover, when do small test sets exist (where the size of a test set is the sum of the size of its elements), so that we can find a correct desugaring not just in theory, but in practice with extensive testing? In particular, since our approach currently places the burden of designing a good test set on the user, it is an interesting question whether (and how) we could also automate the process of synthesising such test sets, perhaps using counterexample-guided inductive synthesis (CEGIS) loop Solar-Lezama et al. [2005]; Jha et al. [2010].

Can desugaring learning problems be decomposed into feasible sequential extension problems? The final, and perhaps most important, question is the following: what is the exact semantic condition for the existence of a partitioning of the source language into a series of sublanguages such that each DEP (induced by the sublanguages) is feasible and collectively they lead to a solution of the full problem? For example, what condition could guarantee the existence of a partitioning with the following properties:

1. Each extension is small (contains fewer than n term constructors for some small n).

2. Extensions of the partially correct translations exist in each step.

3. Extensions can be found with a small test set.

This seems like an inherently empirical and language-dependent question. In this thesis, I present experimental results investigating this question for the benchmarks introduced in Chapter 3.

Can we decompose a desugaring learning problem in advance? Our sequential framework could be used to define multiple DEPs in advance. But as we have showed, for a single DEP step the solution may not exist: it is possible that the partial solution for the previous steps can not be extended to a correct translation. This "stuck" state might mean not only that we need to backtrack and find new solutions during the earlier steps, but also that the decomposition itself may need to be revised. Our formal framework does not model this backtracking process, but it could describe the current state of a possibly stuck tested semantics learning problem.

4.10 Concluding remarks

This Chapter has presented a formalisation of the task that we will deal with in the rest of the thesis. The definitions are not the only possible ones, and they certainly can be extended in various directions. We may handle effects other than errors in the interpreters, we may want a more precise characterisation of shallow translations, or we could include in the model the backtracking process we talked about in the previous passage. The many possible directions for extensions highlight the open-ended nature of the problem space again. I hope that my definitions clarify the complex scenario and will serve as a foundation for future research.

Chapter 5

Enumerating typed program fragments

In the previous chapter, we defined the desugaring extension problem, which poses the task of finding translation rules conforming to a non-standard learning framework. In the next two chapters, we will be looking for a good search space for translation rules, comparing them by enumerative synthesis as proposed in Chapter 2. We will define search spaces by inductive rules, achieving feasible synthesis by restricting the search space to well-typed and well-scoped translation rules. The present chapter describes a reasonable enumeration algorithm based on typing rules, which I have found to perform well in practice, and which could be useful for both synthesis and testing.

In this chapter, I will be discussing the enumeration algorithm independently of the specific problem (the desugaring extension problem) and the specific type systems that I am using it for later, as it can be useful in a more general context. The type systems we will consider later are relatively simple: they are modest extensions of algebraic data types with a context for the variables. In the present chapter, our characteristic example is a type system for β normal form terms in the simply typed λ-calculus. It is important to note that all of these systems usually have many type derivations for a given typing context and type, and our aim is not just to find one, but rather to efficiently enumerate all type derivations.

At a high level, our enumeration of typed programs is based on *proof enumeration*. Given a proof system (a set of inference rules), proof enumeration means to enumerate all proofs of a given judgement. By applying a proof enumeration algorithm to a type system, we can enumerate all type derivations for a given typing context and type. Finally, we can turn type derivations into program terms, resulting in an enumeration

of typed programs.

The enumeration algorithm expects an opaque, functional representation of the inference rules: the user is expected to write a function which, given a judgement, generates all possible sets of premise sets from which that judgement can be derived. The algorithm can enumerate the proofs of a finitely branching proof system, as we expect the set of these premise sets to be finite. Encoding inference rules as functions requires modest programming effort, and allows us to quickly prototype enumeration algorithms for various type systems.

Our implementation of proof enumeration is based on *algebraic enumerations*. Algebraic enumerations allow us to define the enumeration of structures in a generic way, following the inductive definition of the structures. An important property of algebraic enumerations is that they exploit the highly recursive search space, and provide a balanced caching mechanism.

Enumerative synthesis based on proof enumeration is a standard technique, as is the algebraic enumeration of recursive structures. The novel contribution of this chapter is the combination of the two: I show that the algebraic approach to defining enumerations, previously only applied to enumerating algebraic data types and indexed type families, can also be applied to the more generic task of proof enumeration for finitely branching proof systems, and thus to the enumeration of well-typed programs. Algebraic enumerations yield an efficient indexing function for the enumeration, which could be useful for various purposes. We observed that it actually provides a reasonably fast iteration over the enumeration as well, but further research is needed for an accurate comparison with other methods.

The chapter is structured as follows. Section 5.2 contains a review of algebraic enumerations and describes a general algorithm to turn the definition of algebraic data types into an enumeration of terms. Section 5.3 investigates what properties are required from structures to be efficiently enumerable by algebraic methods, and shows that finitely branching proof systems can be enumerated the same way as algebraic data types.

Apart from an efficient caching mechanism to exploit the recursive search space, another important aspect of an efficient enumerative synthesis algorithm is the elimination of symmetries. This applies to synthesis through proof enumeration as well, and Section 5.4 contains a review of symmetry-eliminating techniques in proof enumeration, based on the related and more common task of proof search. The design of the search spaces presented in the next two chapters relies on these considerations.

5.1 Introduction

The enumeration of program fragments is the main building block of testing and synthesis frameworks. The present chapter describes an enumeration algorithm for finitely branching search spaces defined by inductive rules.

We expect a good enumeration algorithm to have the following two properties:

1. Automatic caching of meta-information on the recursive structure. Extensive caching is essential for good performance when enumerating recursive structures, but simply caching small program terms makes the enumeration memory-bound, which in my experience can too easily hit memory limits in a highly branching search space.

2. Easy prototyping. The programmer should be able to turn typing rules into an enumeration of well-typed programs with only moderate effort.

While there is an abundance of enumeration algorithms discussed in the literature, implementation techniques with both of the above properties are not easy to find. Many high-speed enumerative synthesis frameworks like CVC4SY [Reynolds et al., 2019] use context-free grammar as input. In other words, they can enumerate the terms of an algebraic data type (ADT). This does not allow for limiting the enumerated set to program fragments that are well-typed with respect to a type system, a typing context and a type. Well-typed programs are usually only a sparse subset of all programs corresponding to an abstract grammar [David et al., 2009], therefore limiting the testing space or search space to typed programs could be highly beneficial for both validation and synthesis. Since the reduction can be very large, generating the well-typed subset directly is preferable to a generate-and-test approach, as the latter would spend most of its time generating a lot of programs, only to later reject them after testing whether they are well-typed.

Algebraic enumerations allow for separating the declarative definition of an enumeration from the implementation. Through an algebra of enumeration combinators, we can define enumerations following the inductive structure of the data type in a generic way. Algebraic enumerations focus on the index operation: an efficiently calculable bijection from the natural numbers to the set we enumerate. Iterating through the structures based on indexing may seem counter-intuitive at first, but it allows a caching mechanism that in my experience offers a good trade-off between speed and

memory usage. Naive, generic enumeration methods, such as graph-search-based enumerations, are slow due to not exploiting the recursive structure of the search space and recalculating structures multiple times, while simply caching small programs makes the algorithm memory-bound, and in my experience ends up performing worse even before hitting the memory limit. Our enumeration algorithm delegates memoisation to the algebraic enumeration library. A further benefit of algebraic enumerations is that an efficient indexing into the enumeration could be useful for implementing uniform sampling or testing.

We rely on the well-known analogy between proofs and typed programs. Type systems and proof systems are closely related; their relationship is referred to as the *Curry-Howard correspondence*, which states that there is a close analogy between the following series of concepts:

$$\begin{aligned} \text{Proof system} &\longleftrightarrow \text{Type system} \\ \text{Formula, Proposition} &\longleftrightarrow \text{Type} \\ \text{Proof} &\longleftrightarrow \text{Program} \end{aligned}$$

These analogies hold across many concepts in the two areas, such as between proof normalisation and program evaluation. In our enumeration algorithm, we enumerate proofs (type derivations) and extract programs from the proof objects.

When talking about synthesising well-typed programs, we may want to distinguish between two different kinds of specifications, based on type systems. The first kind is based on rich (e.g. polymorphic) type systems, which are frequently used in program synthesis as part of the specification [Osera and Zdancewic, 2015; Frankle et al., 2016; Polikarpova et al., 2016]. In these systems, the types correspond to (a subset of) first-order logical formulas, and finding even one program that corresponds to the given type can be challenging. Synthesis problems with a rich type system are closely related to proof search and are characterised as deductive. In enumerative synthesis, we are interested in a second kind of specification, which are typically simple systems extending the context-free rules of the abstract syntax with a context of variables. The typical type systems employed to speed up enumerative synthesis are sort systems (i.e. type systems with a finite number of types, without type constructors), or have types corresponding to formulas in propositional calculus. While the type system significantly narrows the search space, there are usually still a large number of program terms corresponding to a given type and context within a small size limit. Our aim is not just to find one, but rather to quickly enumerate every such term. The corresponding problem we solve is *proof enumeration*, and the synthesis is enumerative.

Still, proof search and proof enumeration are related tasks. Not every proof system can be efficiently searched or efficiently enumerated. The requirements are similar for both tasks. The literature on proof search is the basis of the design of proof and type systems that can be efficiently enumerated. I will discuss the design of such formal systems in Section 5.4.

Note that this chapter is based mainly on personal observations, and contains no systematic evaluation of performance. Some algebraic enumeration libraries do have favourable reports on empirical comparison to other enumeration methods, but not in the context of program synthesis. For empirical evaluation, we refer to the synthesis benchmarks presented in the following chapters. The enumeration method presented here serves as the basis of Chapter 6, which will introduce a simple sort system for the metalanguage used for the translation rules, and Chapter 7, which will use linear variables as heuristics for further pruning the search space. All reports on empirical evaluation in those chapters will be based on the enumeration algorithm presented here.

This work will not include comparisons of the algorithm to other candidates. As a smoke test, I implemented some simple enumerations in Prolog and executed them using various execution methods (breadth-first search, iterative deepening and SLD resolution). I observed that the logic program matches the formal description of type inference rules much more closely, but the performance of the Prolog-based implementations was much worse: they did not scale to the size of my benchmarks.

The enumeration algorithm is implemented in Haskell and relies on the FEAT library [Duregård et al., 2012].

5.2 Algebraic enumerations

Algebraic enumerations are based on two ideas. One is to represent enumerations functionally, as an efficiently computable bijective function from the natural numbers to the set of values. The second idea is to build functions that index the values of the data type in a datatype-generic manner. We can define enumerations for all algebraic data types at once, in terms of enumeration combinators that match the algebra of data types.

We based our implementation of proof enumeration on the FEAT Haskell library (Functional Enumeration of Algebraic Types, Duregård et al. [2012]), which originally popularised the algebraic approach to defining enumerations. As implied by its name, FEAT focuses on defining an exhaustive enumeration of all values belonging

to an algebraic data type (ADT). FEAT provides fast streaming with efficient caching, and efficient indexing into the enumeration, which is useful for implementing uniform sampling in random testing. The main target of the FEAT library is property-based testing.

5.2.1 Combinators for enumerations

An enumeration is an ordered collection of values of a set S, where S can be finite or infinite. Enumerations typically provide two main operations:

1. indexing, an efficiently computable bijective function from (an initial segment of) the natural numbers to S, and

2. an efficient iteration over the values, in other words, a stream of all values of the enumeration.

The specification of the enumeration requires the indexing function to be a bijection, but usually only one direction needs to be efficiently computable: indexing into the enumeration. There are algebraic enumeration libraries that provide other operations, such as the efficient inverse of the indexing function. As we use enumerations for synthesis, our main goal is efficient iteration, but for algebraic enumerations, the focus is on the indexing function, as iteration is defined in terms of indexing. For a type a, we use $E\,a$ as the enumeration of all values belonging to the type.

Specification The main idea is to construct all enumerations from combinators matching the algebra of data types. Enumerations are recursively built up from four combinators. The four combinators are the following:

$$
\begin{array}{lll}
\mathsf{empty} & :: & \mathtt{E}\,a \\
\mathsf{singleton} & :: & a \to \mathtt{E}\,a \\
(\otimes) & :: & \mathtt{E}\,a \to \mathtt{E}\,b \to \mathtt{E}\,(a \otimes b) \\
(\oplus) & :: & \mathtt{E}\,a \to \mathtt{E}\,a \to \mathtt{E}\,a
\end{array}
$$

The combinators of enumerations match the combinators of algebraic data types, and we overload the $\otimes$ and $\oplus$ symbols accordingly. For types, $a \otimes b$ means the product of the two types (pairs). For enumerations, the constant empty represents an empty enumeration, which contains no value. The singleton function creates an enumeration with only one element from a value. The $\oplus$ combinator on enumerations is disjoint

union: it contains the elements of both enumerations, but it requires the enumerations to be disjoint, otherwise the elements in the intersection of the two enumerations will appear twice in the resulting enumeration. The $\otimes$ combinator is the Cartesian product on enumerations: it returns an enumeration of pairs containing every combination of the values from the arguments.

FEAT also contains two additional operations for building enumerations. Since FEAT is implemented as a Haskell library, general recursion in the definition of enumerations is provided by the host language, and FEAT itself provides a guard function for recursive references: pay. FEAT also defines a map operation: biMap, which is necessary for adding the labels to sums when building an enumeration of an ADT. The name indicates that the function argument is expected to be bijective (otherwise the resulting enumeration would not be a bijection), which trivially holds for data constructors.

$$\begin{aligned} \mathsf{pay} &:: \mathtt{E}\,a \to \mathtt{E}\,a \\ \mathsf{biMap} &:: (a \to b) \to \mathtt{E}\,a \to \mathtt{E}\,b \end{aligned}$$

Equipped with the combinators, we can define enumerations with (potentially recursive) algebraic definitions:

$$\begin{aligned} \mathtt{E\,Bool} &= \mathsf{singleton}\ \textsc{True} \oplus \mathsf{singleton}\ \textsc{False} \\ \mathtt{E\,(List\,Bool)} &= \mathsf{singleton}\ \textsc{Nil} \oplus \mathsf{biMap}\ \textsc{Cons}\ \mathtt{E\,Bool} \otimes \mathsf{pay}\ (\mathtt{E\,(List\,Bool)}) \end{aligned}$$

These definitions are declarative, separating the specification and the implementation of enumerations, allowing for different implementations and implementing different operations on enumerations.

Implementation The implementation of the combinators is straightforward, once we consider the enumeration as an indexing function. For example, the sum combinator ($\oplus$) simply maps even indexes to the left enumeration and odd indexes to the right. The product combinator ($\otimes$) is somewhat more tricky, and various implementations correspond to pairing functions, that is, bijections $\pi : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, such as the Cantor pairing function. Pairing functions are known to have efficient implementations and are used in many algorithms [Regan, 1992; Rosenberg, 2002].

However, this simple view does not take into account that some enumerations are finite and some are infinite. We also need to be careful with recursive definitions.

But the biggest problem is that iterating based on this simple view of the indexing function may not be efficient. Strangely enough, enumeration libraries still define it-

eration in terms of the indexing function. The reason is that, as the indexing function itself is defined recursively, this recursive structure allows an efficient, balanced memoisation strategy. Efficient memoisation and re-using already computed structures are very beneficial and greatly speed up the enumeration of recursively defined structures, at the expense of memory usage. We must find a good balance, however, as excessive memory usage hits the memory limit too soon: this is why we wouldn't want to simply cache all small structures.

There are various implementations of a more efficient indexing function. FEAT itself divides the enumeration into finite partitions. This ensures efficient indexing (by storing the sizes of the partitions), simplifies the implementation (as we only combine the finite partitions) and helps with efficient memoisation. The creators of FEAT reported that memoising parts of the indexing function, carefully controlling what to memoise and what not to, leads to fast and efficient enumerations that perform better than caching all small terms (which hits the memory limit early) or no memoisation at all. The authors of SCIFE [Kuraj et al., 2015] also compared iteration over algebraic enumerations to different enumeration libraries, presenting experiments and empirical evidence, and found performance to be excellent. SCIFE also includes a variety of other operations, like the inverse of indexing, finding the index of a given structure.

5.2.2 Datatype-generic enumerations

Besides the algebraic combinators that work on arbitrary enumerations, FEAT can also derive enumerations for ADTs automatically. It can turn a finite family of mutually recursive ADTs (which we call a *data system*) given by a finite set of type declarations into an enumeration for each data type. FEAT can use declarative input (the definition of the data system) and implements datatype-generic enumerations. This is an example of datatype-generic programming [Gibbons, 2006].

The implementation is simply based on structural recursion: the type constructors used to build a data system (sum and product) correspond to the combinators used to build enumerations, and the algorithm simply transforms the definition of a data system into a set of enumerations by replacing type constructors with enumeration combinators. The entire purpose of defining an algebra of enumerations analogously to the algebra of data types is to automatically derive enumerations for ADTs.

FEAT itself targets the ADTs in the host language, Haskell. Our purpose is not to interact with definitions in the host language, but rather to extend the generic defini-

tion of enumerations from data systems to more generic structures. In the following section, we analyse the precondition of this structural recursive algorithm in a functional language, showing that it actually works for more general structures than just data systems.

5.3 Enumerating proofs

In this section, we create enumerations of sets of trees. By a tree, we always mean a *finite, node-labelled, ordered* tree. By a subtree of a tree T, we mean a tree formed by a node in T and all of its descendants.

In programming, we can represent such trees with a Rose tree. In Haskell notation:

$$\textbf{data} \quad \texttt{Tree}\ \mathit{label} \quad = \quad \textsc{Node}\ \mathit{label}\ [\texttt{Tree}\ \mathit{label}]$$

Note that in Haskell the lists can be infinite (generated on-the-fly by some computation), and in some systems, proofs could be represented as infinite mathematical objects. In proof theory, there is a tradition of infinitary proof systems going back to Tarski and Hilbert, where a judgement can have an infinite number of premises. The common example is an axiom of "transfinite induction", also called the ω-rule, which allows deriving a universal statement from infinitely many individual statements (for an overview of its use in various proof theories, see [Sundholm, 1983]). But these are not common in our use case, as we are interested in proof systems where the proofs themselves are finite objects. Among type systems, finitary systems are standard. We will only consider finitary proof systems, and will only enumerate (potentially infinite) sets of finite trees.

5.3.1 Enumerating data systems

The simplest sets of trees we consider are those defined by a data system.

Definition 5.3.1 (Data system)**.** We call a finite family of mutually recursive algebraic data types a *data system*. Formally, they can be defined by a set of polynomial equations between the data types, where each data type is defined as a sum of product of other types.

Let $\{T_1, \ldots, T_n\}$ be a finite set of names of the data types. For each type name T_k, let $\{C_k^1, \ldots, C_k^{m_k}\}$ be a finite set of (data) constructors. Let t range over the type names $T_1, \ldots, T_n$. Then, a data system is a set of n equations of the following form:

$$\begin{aligned} T_1 &= (C_1^1\ t \otimes \cdots \otimes t) \oplus \cdots \oplus (C_1^{m_1}\ t \otimes \cdots \otimes t) \\ &\vdots \\ T_n &= (C_n^1\ t \otimes \cdots \otimes t) \oplus \cdots \oplus (C_n^{m_n}\ t \otimes \cdots \otimes t) \end{aligned}$$

Every type name T_k has one equation where it appears on the left side, and we call this equation the type's definition.

Note that the set of constructors $\{C_k^1, \ldots, C_k^{m_k}\}$ can be empty ($m_k = 0$), in which case the type is empty. The set of types in the product can also be empty, in which case the constructor C_k^j corresponds to a single value.

For each type T_k we can assign a set of trees labelled with the constructors and built up according to the inductive definitions. The data types are interpreted with well-founded semantics: we only consider finite trees.

FEAT can be seen as a generic algorithm constructing enumerations for any data system: the input is the set of equations (a data system), and the output is a set of enumerations for each data type ($\mathtt{E}\ T_1, \ldots, \mathtt{E}\ T_n$). We will call this algorithm the algebraic enumeration algorithm.

The algorithm is structural recursion on the inductive definition. For each type name T_k we build the enumeration $\mathtt{E}\ T_k$ by enumeration combinators as follows. First, we define enumerations for each constructor (each product appearing in the definitions):

- If the product $C_k^j\ t \otimes \cdots \otimes t$ is empty, then the constructor C_k^j is a single value: $singleton\ C_k^j$.
- Otherwise, we build the product of the enumerations corresponding to the type names in the product with the product combinator, map the constructor with *biMap*, and wrap the whole enumeration with *pay* to ensure any recursive reference is guarded.

Based on the enumerations corresponding to products, we can define enumerations for the sums:

1. If the sum is empty, then the enumeration is also empty (constructed by the *empty* combinator).
2. Otherwise, we calculate the enumeration for each product as above, and build the sum of the enumerations with the sum combinator.

We can now slightly generalise this algorithm, by giving a functional representation of a data system. The representation is based on the type required by the algorithm, but it allows for more general input.

The following is a brief description of our representation, using Haskell's notation. We represent the data system by a function that assigns the type definition to the type name.

We represent our polynomial type expressions with sum and product combinators of an arbitrary number of arguments, generalising the binary type constructors $\otimes$ and $\oplus$. This better matches the definition of data systems, as we can uniquely represent them. Combinators with an arbitrary number of arguments also allow us to implement similar combinators of enumerations with multiple arguments. When we combine three or more enumerations with a binary sum combinator, the resulting enumeration is necessarily biased: the binary sum combinator on enumerations is not associative. Combinators with an arbitrary number of arguments can be fair, however, and without bias. For a definition of fairness of enumeration combinators and possible implementation, see New et al. [2017]. Using multiple arguments also allows us to only use two combinators, as empty types are represented as a sum with zero arguments, and singleton types are represented as a product with zero arguments.

To make the algorithm generic, we abstract away from the set of types, and we simply use a type parameter *typeName* for the type names $\{T_1, \ldots, T_n\}$, and another type parameter *constr* for all constructors $\bigcup_{k=1}^{n} \{C_k^1, \ldots, C_k^{m_k}\}$ for any type.

```
data ProdType typeName constr   = PROD constr [typeName]
data TypeDef typeName constr    = SUM [ProdType typeName constr]
type DataSystem typeName constr = typeName ⟶ TypeDef typeName constr
```

We have an invariant restricting the use of constructors. Let `ds : DataSystem typeName constr` be a data system, and `t : typeName`. The expression `ds t` returns a list of products. In every product `PROD c [t1, ..., tn]` in this list, the constructor c must have a type $c : t_1 \times, \ldots, \times t_n \to t$.

The functional representation is more general than data systems. This generality is the basis of extending the enumeration of ADTs to the enumeration of proofs, and we examine the relationship in Section 5.3.4.

In order to have a generic enumeration algorithm, we need a generic representation of terms. We can represent terms by a Rose tree:

$$\textbf{data}\quad \texttt{Tree}\ \textit{label}\ =\ \textsc{Node}\ \textit{label}\ [\texttt{Tree}\ \textit{label}]$$

instantiated with the type of data constructors: `Tree` *constr*.

The type of a data type generic enumeration is:

$$\mathsf{enumerateTerms}\ ::\ \texttt{DataSystem}\ \textit{typeName constr} \to \textit{typeName} \to \texttt{E}\ (\texttt{Tree}\ \textit{constr})$$

Given a finite set of type declarations (a data system), enumerateTerms returns an enumeration of terms (represented as Rose trees) for any type in the data system. The implementation follows the same structurally recursive pattern as previously described: we merely generalised the type of inputs.

5.3.2 Data systems and type systems

We want to enumerate proofs and type derivations, which, similarly to data systems, can also be represented as trees, where the labels are the judgements. We will illustrate this relationship between type systems and data systems with a very simple example.

Example 5.3.1 (Odd-Even type system)**.** *We define a type system that sorts the Peano numbers into two types. The terms are the Peano numbers, and the types are called* `Odd` *and* `Even` *(Figure 5.1).*

$$\begin{aligned} n \in \mathbb{N} \quad &::= \quad Z \mid S(n) \\ T \quad &::= \quad \texttt{Odd} \mid \texttt{Even} \end{aligned}$$

$$\frac{}{Z : \texttt{Even}}\ \textsc{Axiom} \qquad \frac{n : \texttt{Even}}{S(n) : \texttt{Odd}}\ \textsc{Odd intro} \qquad \frac{n : \texttt{Odd}}{S(n) : \texttt{Even}}\ \textsc{Even intro}$$

Figure 5.1: Odd-Even: Simple type system partitioning natural numbers

Note that this type system is a sort system. In fact, it can also be represented as a data system: we can write two mutually recursive data types representing the type derivations:

$$\begin{aligned} \textbf{\textit{data}}\ \texttt{Even} \quad &= \quad \mathrm{Z} \mid \mathrm{S}\ \texttt{Odd} \\ \textbf{\textit{data}}\ \texttt{Odd} \quad &= \quad \mathrm{S}\ \texttt{Even} \end{aligned}$$

The values of the data type `Even` *in the data system correspond to the type derivations for the type* `Even` *in the Odd_Even type system.*

Data systems can be considered special type systems in two regards: first, they only have a finite number of types, and second, they do not have context. We call type systems with a finite number of types (with or without context) *sort systems*, and call the types *sorts*, so data systems are context-free sort systems.

5.3.3 Abstract proof systems

When enumerating proofs, we use an abstract notion of proofs and proof systems, abstracting over the set of judgements. For our purpose, proofs are trees labelled with judgements. We are interested in sets of proofs where we can enumerate all proofs of a given judgement.

Definition 5.3.2 (Inductive proof set)**.** Let J be an arbitrary computably enumerable set: the set of judgements. We will use J^* for the set of finite sequences of judgements (which also contains the empty sequence).

Let $\mathcal{T}(J)$ be the set of trees with nodes labelled with elements of J. We call any subset $P \subset \mathcal{T}(J)$ a *proof set*. When P is unambiguous from the context, we call any $t \in P$ member a proof, the label of the root node the *conclusion* of the proof, a label of a child of the root node a *premise*, and the list of labels of the children of the root node the *premise list*. It is possible for the tree to only have only one node, when the premise list is empty. If $t \in P$ is a one-node tree with label j, we call j an *axiom*.

Let I be a relation between judgements and premise lists defined as follows:

$$\forall j \in J, \bar{p} \in J^*, I(j,\bar{p}) = \exists t \in P,\ j \text{ is the conclusion of } t,\ \bar{p} \text{ is the premise list of } t$$

We call I the *inference relation* of P.

Any proof set P is an *inductive proof set* if it satisfies the following four properties:

1. For every $t \in P$ all of its subtrees are also in P.
2. Let $t' \in P$ be any proof with the conclusion $j \in J$. Let $t \in P$ be any proof which has a node n labelled with j. If we replace the subtree having root n in t with t', the result will also be an element of P. (*proof irrelevance*)
3. The inference relation of P is computable (that is, given a premise list and a judgement, we can compute whether the judgement is a conclusion of the premise list).

Remark 12. Note that the inference relation uniquely determines the proof set. Let $t \in \mathcal{T}(J)$ be any tree, with conclusion j and premise list $\bar{p}$. We can check whether $t \in P$ by checking if $I(j, \bar{p})$ holds and recursively checking whether the subtrees are proofs. This also means that inductive proof sets are computable.

Definition 5.3.3 (Rule based proof system)**.** We call a computable partial function r from a premise list to a judgement a *rule*: $r : J^* \rightharpoonup J$. Let P be an inductive proof system with inference relation I. We say that P is a *rule based proof system* iff there is a countable set of rules $r_1, r_2, \ldots$, where

$$\forall j \in J, \forall \bar{p} \in J^*, \quad I(j, \bar{p}) \iff \exists n \in \mathbb{N}, \bar{p} \in dom(r_n) \text{ and } r_n(\bar{p}) = j$$

Remark 13. Note that what is usually called an inference rule in a proof system is something we would rather call a rule scheme. Given a list of premises, a rule scheme can yield multiple different consequences, as it frequently ranges over some parameters (like a formula) that are left open in the rule scheme, or it may need to select part of the judgement in a premise, which may happen multiple ways.

In the following, we give a functional representation of the inference relation for a rule based proof system:

Definition 5.3.4 (Abstract proof system)**.** Let `Judgement` be the type of judgements. Let the type `Rule` represent a rule:

$$\textbf{type}\ \texttt{Rule} = [\texttt{Judgement}] \to \texttt{Maybe Judgement}$$

By an *abstract proof system*, we mean a functional representation of the inference relation, with the following type:

$$\textbf{type}\ \texttt{InferenceRules} = \texttt{Judgement} \to \left[\ (\texttt{Rule}, [\texttt{Judgement}])\ \right]$$

For a given judgement j, the function call `InferenceRules`(j) returns a list where each element is a rule paired with a list of premises, with the condition that the rule can be applied to the premises, returning JUST j as the conclusion.

We can view the `InferenceRules` function as the interface for a backward chaining search. Backward chaining suits our needs, as our goal is to start from a judgement and enumerate all proofs of that judgement.

Remark 14. Note that an abstract proof system is *finitary*: there are only a finite number of premises in every inference, meaning that in any proof, the premise list is finite. This means that the inner lists returned by the `InferenceRules` function are finite.

One important property of abstract proof systems is that we can easily define their co-product in the following way:

$$(\oplus) : \texttt{InferenceRules} \to \texttt{InferenceRules} \to \texttt{InferenceRules}$$
$$(i_1 \oplus i_2)\ j = i_1\ j\ +\!+\ i_2\ j$$

where $+\!+$ is list concatenation. We can independently encode inference rules like modus ponens (which, as we noted, is a scheme for rules in our terminology) as abstract proof systems, and compositionally define the proof or type system as the co-product of a set of inference rules.

Note that we used an ordered representation of premises, that is, the premises are a list rather than a set, which corresponds to our proofs being represented as ordered trees. In addition, the set of premise lists is also represented as a list. The order in this second list is not unique for a given inductive proof set, but is instead the property of the enumeration. We generate an enumeration from the abstract proof system, which imposes an order on the proofs. The order of proofs in turn depends on the order of the premise lists. We include the order of the premise lists in our representation, giving the user control over the order of the enumeration. This means that the `InferenceRules` function contains more information than the inductive proof set generated by it.

Proofs can be represented by Rose trees, the judgements being the labels: `Tree Judgement`.

By a proof enumeration algorithm, we mean a computation that is polymorphic over the type of judgements, and, given an abstract proof system and a judgement, returns an enumeration of all proofs of that judgement. It has the following type:

$$\textsf{enumerateProofs} \;::\; \texttt{InferenceRules} \to \texttt{Judgement} \to \texttt{E}\,(\texttt{Tree}\ \texttt{Judgement})$$

It is clear from the Odd-Even example, as well as from their type, that data systems are special kinds of abstract proof systems, where the proofs are terms. We will elaborate on their relationship in the next section.

5.3.4 Enumerating finitely branching proof systems

We show that the algebraic enumeration algorithm presented in Section 5.3.1 for data systems can be applied to a restricted class of abstract proof systems without any changes, making it a proof enumeration algorithm (for this restricted class), provided that it is implemented lazily (with call-by-need semantics).

We start from the observation that the type `InferenceRules` is an instantiation of the type `DataSystem`, with `Judgement` as the *typeName* and `RuleName` as the type

representing data constructors *constr*. Therefore, the input of a proof enumeration algorithm has the same type as our data system enumeration algorithm.

The output is also an enumeration of Rose trees in both cases. Here, however, the correspondence is not perfect. On the one hand, the proofs are labelled by judgements, and the judgements correspond to type names in data systems. The terms of a data system, on the other hand, are labelled by data constructors, which instead correspond to rules in proof systems. This difference is not substantial, however, as given a tree labelled by rules, we can recursively produce a tree labelled by judgements: the rules are the very functions that do this. Therefore an enumeration of trees labelled by rules can directly be turned into an enumeration of proofs.

The remaining differences between the algebraic enumeration algorithm enumerating data systems and proof enumeration algorithms are as follows:

1. In data systems, the lists in the type `DataSystem` are assumed to be finite. While we are only interested in finite proofs (where the premise lists themselves are finite), the set of premise lists can still be infinite.

2. In data systems, the type *typeName* is assumed to have a finite number of values. In abstract proof systems, the set of judgements can be infinite.

3. In data systems, we assumed that the type names are atomic. In the case of proofs, we generally assume that judgements are themselves inductively defined. Similarly, in data systems, the function representing a data system is supposed to be given by a fixed finite map with a finite domain (the set of type names). An abstract proof system representing an inductive proof set can be an arbitrary computable function over an infinite domain (the set of judgements), where this domain has its own inductive structure.

Which of these differences is a prerequisite of the algebraic enumeration algorithm? Note that whether the type names are atomic or replaced by inductively defined judgements does not matter, since the algorithm is polymorphic over this type. Regarding the third condition (an abstract proof system being an arbitrary total computable function), note that the algebraic enumeration algorithm only uses it as an opaque function, so it does not matter whether it is a simple map or some more complex computation, as long as it terminates. But the first condition is necessary, as we iterate over these lists inside the algorithm, and these loops need to terminate.

Definition 5.3.5 (Finitely branching proof system)**.** We will call an abstract proof system a *finitely branching proof system* if the proof space is *finitely branching*, that is, for any given judgement j there are only finitely many premise lists $\bar{p}$ occuring in proofs of that judgement. This means that the outer lists returned by the `InferenceRules` function must be finite (similarly to the inner lists).

This condition considerably limits proof systems, as we will show in Section 5.4.1.

We will discuss what kind of proof systems are finitely branching in Section 5.4.1. For now, we will show that the algebraic enumeration algorithm can be applied to them.

The main insight behind applying the algebraic enumeration algorithm as a proof enumeration algorithm is that starting from a fixed judgement and bounding the size of proofs, a finitely branching proof system can only visit a finite set of other judgements. From this, it follows that if we implement the data system enumeration algorithm lazily (using call-by-need semantics), it will work for finitely branching proof systems as well. The reason lies in the implementation of the enumerations themselves in FEAT.

Every enumeration in FEAT is partitioned into sets by the size (number of constructors) of the terms, or in other words, the number of nodes of the enumerated trees. In the case of data systems, these partitions are all finite. As we have said, starting from a fixed judgement and bounding the size of proofs, a finitely branching proof system can only visit a finite set of other judgements. This means that the partitions—sets of proofs with a fixed size—are finite for finitely branching proof systems as well. When we calculate a partition of an enumeration, we can ignore the rest of the judgements: the part of the algorithm that computes the partition can be exactly the same as for data systems. Enumerating all proofs of a fixed size for a given judgement (a partition of the enumeration) is exactly the same as enumerating all of an ADT's terms of a fixed size. This means that for limited sizes, the finitely branching proof system can be represented as a data system. This is precisely what the enumeration achieves when using call-by-need evaluation, only adding new judgements (and new enumerations corresponding to those judgements) as necessary.

Having said that, some differences still remain. The biggest difference between data systems and proof systems is that for a proof system, it can be undecidable whether a given judgement has any proof at all, that is, whether an enumeration (specified by a proof system and a judgement) is empty. Therefore, indexing or iterating over the enumeration may not terminate when the enumeration is generated from a proof system and not from a data system. But for finitely branching proof systems, and for

a finite size, it is decidable whether there is a proof of that size or not. This means that for any partition, we can decide whether it is empty or not, and consequently, if a judgement does have a proof, the enumeration will produce it. The algorithm may of course diverge after reaching the last proof (or immediately, if there are no proofs at all).

Another difference is that encoding a data system in our functional representation is straightforward, while some programmer effort is needed to encode a proof system as a function generating the set of premise lists for a given judgement.

The algebraic enumeration algorithm enumerates proofs in the same order as a breadth-first search (BFS). The main difference compared to a naive BFS algorithm is that an algebraic enumeration algorithm is aware of the recursive nature of the search space, and will employ an efficient memoisation scheme to help revisit the same parts, i.e proofs proving the same judgement. This highlights that the efficiency gained by algebraic enumerations depends on the recursive references, and the algorithm performs better if we revisit the same judgement multiple times, rather than always reaching new judgements.

5.3.5 Finitely branching type systems

We want to enumerate typed programs by a correspondence between proofs (type derivations) and programs. This correspondence does not automatically mean that programs and proofs are the same, however. What we can expect is the ability to build up a program term following the structure of the proof. An *abstract type system* is a variant of an abstract proof system which, instead of rules, is labelled by program term constructors:

$$\begin{aligned}
&\textbf{type}\ \texttt{TypingRules} &&= \texttt{Judgement} \to \Big[\ \big(\texttt{TermConstructor}, [\texttt{Judgement}]\big)\ \Big] \\
&\textbf{type}\ \texttt{TermConstructor} &&= [\texttt{Program}] \to \texttt{Maybe Program}
\end{aligned}$$

Similarly to how we built proofs (trees labelled by judgements) from trees labelled by rules, we can build program terms from trees labelled by term constructors. In this case, we do not need to build an entire tree, just the program that would be the label of the root node.

In the case of a type system, the type `Judgement` should not contain the programs. For example, a typical typing judgement is written like this: $\Gamma \vdash p : \tau$, where Γ is a typing context, a function mapping a finite set of variables to types, p is the program and

τ is the type of the program under the typing context. In this case, the type `Judgement` is a pair of the form (Γ, τ).

Finitely branching type systems are abstract type systems where the proof space is finitely branching. Given a finitely branching type system, the algebraic enumeration algorithm can enumerate all programs with a given type.

5.4 Symmetries in proof and type systems

In the present section, we show what kind of proof and type systems are finitely branching, and discuss techniques for eliminating potential symmetries in these systems that make the enumeration inefficient. We will rely on these techniques in the following chapters. The material presented here is a standard part of structural proof theory [Negri et al., 2001], but we review these from the viewpoint of enumerating proofs.

5.4.1 Finitely branching proof systems

We could automatically turn an infinitely branching system into a finitely branching one, or handle infinitely many branches in a different way. However, even if we did extend the algorithm to handle an infinitely branching proof space, the resulting enumeration would not be very efficient.

Infinitely branching inference rules usually have some premises that can not be generated from the conclusion, but can be arbitrary in part, making proof search and proof enumeration difficult. The characteristic example is the MODUS PONENS rule in natural deduction, or the corresponding CUT rule in sequent calculus:

$$\text{MODUS PONENS}\ \frac{A \quad A \to B}{B} \qquad \text{Cut}\ \frac{\Gamma \vdash A, \Delta \quad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}$$

The A formula in the premise of the MODUS PONENS or the CUT rule cannot be guessed from the conclusion, and proof systems with these rules are not finitely branching. Most type systems have typing rules analogous to the MODUS PONENS rule: one such example is the APPLICATION rule in the simply typed lambda calculus (STLC):

$$\text{APPLICATION}\ \frac{\Gamma \vdash f : \tau_1 \to \tau_2 \quad \Gamma \vdash a : \tau_1}{\Gamma \vdash f\, a : \tau_2}$$

In the APPLICATION rule, we cannot guess the type τ_1 from the conclusion, which can be arbitrary.

Proof systems designed for proof search go to great lengths to avoid such rules. In proof search algorithms, it is standard to use proof systems without the cut rule, exactly for the reason we mentioned: certain parts of the premises could be anything, making them difficult to guess.

Remark 15 (Subformula property). In proof theory, it is common to aim for a proof system that has the *subformula property*, which means that in any inference rule, all formulas in the premises are subformulas of the formulas in the conclusion. The cut rule violates this condition, therefore a common technique is cut-elimination, to show that the cut rule is admissible (redundant) and the same judgements can be derived without it.

The subformula property and cut-elimination were introduced by Gentzen in his landmark paper [Gentzen, 1935, 1969], and its main usage is to establish meta-theoretic results, such as consistency, for proof systems. Indeed, the consistency of first-order logic immediately follows from Gentzen's *Hauptsatz*, the cut-elimination theorem for the first-order sequent calculus. Gentzen introduced two sequent calculi, the classical LK and the intuitionistic LJ, to show that the cut rule is admissible in these systems, and the cut-free fragments do have the subformula property.

The subformula property, the cut rule and cut-elimination all play a central role not just in the metatheory of proof systems, but in proof search as well.

Finitary proof systems with the subformula property are usually finitely branching. Note that in our abstract notion of proof systems, we only referred to judgements, not to formulas or propositions, therefore we cannot state the subformula property as a requirement without additional assumptions. Moreover, from the subformula property, it does not strictly follow that a proof system is finitely branching, and the finitely branching condition brings proof systems closer to (similarly finitely branching) data systems. In my own use case—setting up search spaces for the DEP—I use sort systems (type systems with a finite number of types, without type constructors), where the subformula property is not applicable anyway.

The STLC type system or the corresponding natural deduction proof system (the ones Howard [1980] originally noticed the equivalence of) are not finitely branching and do not have the subformula property, while the cut-free fragments of the sequent calculi LK and LJ are finitely branching, and do have the subformula property. A well-known finitely branching type system is based on the cut-free fragment of Gentzen's

LJ calculus.

Example 5.4.1 (Normal form lambda terms and the LJ calculus)**.** *The following example is standard, based on Gentzen's original LJ sequent calculus. I quote the variant from Herbelin [1994]. Compared to the original LJ calculus, he uses an unordered context* Γ*, in order to avoid the need for the* EXCHANGE *rule, and allows irrelevant formulas in the* AXIOM *rule to avoid the need for the* WEAKENING *rule. Also, as a fragment of LJ, it is limited to implication as the sole logical connective.*

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \textsc{Axiom} \qquad \frac{\Gamma, x : \tau_1, y : \tau_1 \vdash t : \tau_2}{\Gamma, x : \tau_1 \vdash t\{y := x\} : \tau_2}\ \textsc{Contraction}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2}\ \textsc{Abstraction}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma, x : \tau_2 \vdash t : \tau}{\Gamma, f : \tau_1 \to \tau_2 \vdash : t\{x := f\ t_1\} : \tau}\ \textsc{Application}$$

The identification of proofs in LJ and simply typed λ*-terms, explicit in our system, is also standard. Note that the context is unordered (as in every other system in this thesis), which means that contraction can be applied to any variable in the context (not just the last one). The rules as presented here omit* CUT*, making it a finitely branching system. Accordingly, this variant only generates* β*-normal form* λ*-terms.*

In practice, we should aim for a proof system with not only a finite but a low branching factor, because otherwise the search space grows too quickly. But we should highlight that there are proof systems with a high branching factor where most branches are empty because they lead to judgements with no proof. It is worth noting that the algebraic enumeration method will perform poorly on these proof systems. The reason is that we approximate the set of size-bounded proofs with a data system. If we can reach a large number of different judgements within a certain size, then the data system will be large even if most of these judgements have no proof.

5.4.2 Adequate representation of judgements

The algebraic enumeration method is (relatively) efficient if the memoisation mechanism can kick in, meaning that if we have already built an enumeration for a judgement we do not need to recalculate most of its structure. We should aim for a proof system which gives the algorithm the most opportunity for memoisation to help. If two

judgements are isomorphic in the sense that there is a simple structural one-to-one correspondence between their proofs, then we should only build an enumeration of their proofs once (and use it to derive the other enumeration).

Definition 5.4.1 (Adequate proof system)**.** We call a finitely branching proof system *adequate* if the isomorphic judgements have the same representation. The adequacy of a proof system depends on what judgements we consider isomorphic.

A common example is the elimination of the EXCHANGE rule, which normally allows permuting elements of a sequent or context, by using unordered contexts. A key requirement in our implementation is thus to define the widest equivalence class of judgements possible, and to use a canonical representation of each class, eliminating most symmetries in judgements. In the following chapters, all contexts used will be unordered.

5.4.3 Curry-Howard property

Type systems have a different kind of redundancy: the correspondence between proofs and programs. Even if enumerating type derivations is efficient, to ensure that our program enumeration is efficient as well, we need to require an additional property on the translation from type derivations to programs:

Definition 5.4.2 (Curry-Howard property)**.** A type system has the *Curry-Howard property* if there is a natural (syntax-based) 1-1 correspondence between proofs (type derivations) and the programs we wish to enumerate. This means that the function that converts proof trees to programs is a bijection: we get a different program for each proof tree in the enumeration, and every program we wish to enumerate can be generated from a proof in the enumeration.

This property follows the similarly named, well-known correspondence between proof and type systems. Howard's important observation was a syntax-based correspondence between these system, but in the case of turning an enumeration of type derivations into an enumeration of programs we merely need a 1-1 correspondence which is efficiently computable. But all easily computable bijections I know of are syntax-based, and it is natural to aim for a syntax-based correspondence in the design of type systems aimed for enumerating programs.

An efficiently computable bijection is important: a program with multiple type derivations is enumerated multiple times. A program that has no type derivations is

omitted from the enumeration.

Obviously, the Curry-Howard property depends not merely on the type system, but also on which class of programs we want to enumerate.

To highlight the importance of the Curry-Howard property, let us revisit Example 5.4.1, the type system based on the LJ calculus, which generates β-normal form terms of STLC.

The correspondence between natural deduction and STLC means that there is a one-to-one syntactic relationship between proofs of natural deduction and (α-equivalent classes of) λ-terms of STLC (essentially, we can consider them identical). These systems are not finitely branching, however, while the LJ system shown in Example 5.4.1 is finitely branching. The one-to-one relationship is lost with sequent calculus: the LJ system does not have the Curry-Howard property.

In the LJ system, multiple proofs correspond to the same λ-term, as proof rules can be permuted. The following example is taken once again from Herbelin [1994]:

Example 5.4.2 (Permutation of proof rules)**.** *In the (cut-free) LJ system, the following two derivations:*

$$\dfrac{\dfrac{\dfrac{}{y:\tau, z:\tau_1 \vdash y:\tau}\,\textsc{Axiom} \quad \dfrac{}{y:\tau, z:\tau_1, w:\tau_2 \vdash w:\tau_2}\,\textsc{Axiom}}{x:\tau\to\tau_2, y:\tau, z:\tau_1 \vdash (x\,y):\tau_2}\,\textsc{Application}}{x:\tau\to\tau_2, y:\tau \vdash \lambda z.(x\,y):\tau_1\to\tau_2}\,\textsc{Abstraction}$$

and

$$\dfrac{\dfrac{}{y:\tau \vdash y:\tau}\,\textsc{Axiom} \quad \dfrac{\dfrac{}{y:\tau, z:\tau_1, w:\tau_2 \vdash w:\tau_2}\,\textsc{Axiom}}{y:\tau, w:\tau_2 \vdash \lambda z.w:\tau_1\to\tau_2}\,\textsc{Abstraction}}{x:\tau\to\tau_2, y:\tau \vdash \lambda z.(x\,y):\tau_1\to\tau_2}\,\textsc{Application}$$

are both associated to the same typed λ-term $\lambda z.(x\,y):\tau_1\to\tau_2$ for a context in which $x:\tau\to\tau_2$ and $y:\tau$.

Therefore, the typing of β-normal form λ-terms with the cut-free LJ system does not have the Curry-Howard property.

The standard solution is *focusing*, the idea being to limit the availability of formulas/types in the context to a special element, called the *focus* or *stoup*.

Example 5.4.3 (LJT based type system for normal form STLC terms)**.** *The LJ system has a standard focused variant, called LJT. The following version is again taken from*

Herbelin [1994]. In this variant, the left side of the judgement has a special element, the focus, which can also be empty. In the notation, a semicolon separated the focus from the context. A dot means an empty focus. It can be proven that the two type systems are equivalent, in that they type the same set of λ-terms, but in LJT there is exactly one proof for each such term.

$$\frac{}{\Gamma; x : \tau \vdash x : \tau}\,\textsc{Axiom} \qquad \frac{\Gamma, x : \tau_1; x : \tau_1 \vdash t : \tau_2}{\Gamma, x : \tau_1; \cdot \vdash t : \tau_2}\,\textsc{Contraction}$$

$$\frac{\Gamma, x : \tau_1; \cdot \vdash t : \tau_2}{\Gamma; \cdot \vdash \lambda x.t : \tau_1 \to \tau_2}\,\textsc{Abstraction}$$

$$\frac{\Gamma; \cdot \vdash t_1 : \tau_1 \quad \Gamma; x : \tau_2 \vdash t : \tau}{\Gamma; f : \tau_1 \to \tau_2 \vdash : t\{x := f\,t_1\} : \tau}\,\textsc{Application}$$

The type system based on the LJ calculus only generated β normal form λ-terms, and this remains true for LJT. By changing the proof system to be finitely branching (removing the CUT rule), we also changed the set of enumerated objects from all λ-terms to β-normal form λ-terms. Unlike the finitary conditions, the Curry-Howard property is not a property of a proof system, as it depends on what objects we wish to enumerate – in our case, which programs we consider equivalent.

In program synthesis, it is common to consider λ-terms equivalent to their β normal forms, thus we may argue that limiting the system to generate only terms in β normal form is appropriate. Note that STLC is strongly normalising, meaning that every term has a unique β normal form. But LJT may still lack the Curry-Howard property if we consider η-reduction as well. The term $(\lambda f : \tau \to \tau.\ \lambda x : \tau.\ fx)\ :\ (\tau \to \tau) \to (\tau \to \tau)$ can be η-reduced to $(\lambda f : \tau \to \tau.\ f)\ :\ (\tau \to \tau) \to (\tau \to \tau)$, so we may consider them equivalent, yet they have two different type derivations in LJT. Such terms may or may not be equivalent in a particular language, and the exact set of programs that we wish to enumerate depends on the domain of application: our equivalence relation on programs.

5.5 Related work

Enumerating programs and program fragments for testing and synthesis purposes is a very general task, with a vast number of implementations. I did not compare how algebraic enumerations perform in this domain compared to other methods. Instead, I will only provide a glimpse into various research areas, in order to give a general overview.

Compiler testing Chen et al. [2020] provides a recent survey of test case generation for compilers. Many of the solutions they discuss can generate tests specified by a two-level grammar or an attribute grammar. The capabilities of these specifications surpass context-free grammars: they are known to be Turing-complete, and can express various type systems. Attribute grammars specifically are considered to be closely related to functional programming. But the focus of automatic test generation for compilers is test coverage (generating a smaller set of programs covering a specific set of features) rather than fast exhaustive enumeration. To the best of my knowledge, these techniques provide neither automatic caching, nor efficient indexing into the enumerations.

Program synthesis In the research literature, the most widely used specification format for program synthesis is syntax-guided synthesis (SyGuS) [Alur et al., 2013], which uses context-free grammars. SyGuS does not allow static semantics to be used in the specification. Consequently, many generic enumeration algorithms in program synthesis, like CVC4SY [Reynolds et al., 2019], are limited to context-free grammars as the input. Typing rules cannot be represented with context-free grammars, and no matter how fast they are, generate-and-test approaches cannot match the speed of enumerating programs directly corresponding to a particular type.

Proof search and proof enumeration The subformula property and the Curry-Howard property are standard requirements in proof search, see for example the survey by Dyckhoff and Pinto [1999]. Proof search and proof enumeration are related tasks, but with different emphases. In proof search, we assume a rich proof system, where finding even a single derivation of a judgement can be challenging. By contrast, proof enumeration assumes a simpler proof system, where the proofs of a judgement are abundant, but we want to enumerate all of them. We introduce a fast enumeration technique that can generate millions of programs very quickly. We will not be discussing higher-order type systems where the type limits the program so strictly that we would need search rather than enumeration. Rather, our targets are sort systems and first-order type systems, where the number of programs with a small size limit belonging to a type is still very large (although still a small fraction of all well-formed programs).

Some synthesis methods, like MYTH [Osera and Zdancewic, 2015], are based on proof search. Their solution is not enumerative: they use a rich type system as a logical specification, from which they derive the solution using input-output examples.

Wells and Yakobowski [2005] designed an enumeration algorithm specific to a concrete proof system with applications to program synthesis, a variation of the LJT system discussed earlier. Their solution is hand-crafted for this specific proof system, and thus cannot be adapted out-of-the-box to other proof systems. We showed a general programming technique that is relatively easy to adapt to various type and proof systems, requiring only moderate programmer effort.

Logic programming Inference rules can easily be encoded as logic programs. To ensure termination, we can use breadth-first search or iterative deepening, but without caching, we have found both methods to be far too slow for enumerating proofs. SLG resolution (tabled execution) [Chen and Warren, 1993] is an execution model for logic programs based on well-founded semantics that provides memoization (caching) and avoids looping by left recursion. However, it is designed to find all solutions of a problem, and we usually have infinitely many solutions. This means that we need to limit the search depth in advance. In my preliminary experiments, all attempts were orders of magnitude slower than the implementation presented here. SLG resolution does not provide efficient indexing into the enumeration. The benefit of logic programming is that encoding the inference rules as Prolog rules is more straightforward and requires less programmer effort than encoding them as functions, especially for type systems using unification, which is built into Prolog.

Algebraic enumerations We build directly on the FEAT library [Duregård et al., 2012]. The FEAT library has only been applied to algebraic data types, i.e. context-free structures. FEAT can automatically generate an enumeration for Haskell data types, a special case of datatype-generic programming [Gibbons, 2006]. The implementation relies on type classes and generalised deriving [Magalhães et al., 2010].

Recent work by Van Der Rest and Swierstra [2022] extended the datatype-generic generation of enumerations from ADTs to indexed type families, also called type-indexed data types [Hinze et al., 2004]. Type-indexed data types are more expressive than ADTs and allow for an infinite family of mutually recursive types, somewhat similarly to finitely branching proof systems, with the judgements playing the role of the indices. But indices in an indexed family are inductively defined, and the types of the family are generated through structural recursion. In abstract proof systems, we allowed an arbitrary computable function to generate the inference relation. The authors developed enumerations in a dependently typed setting, and they were able to

prove that their enumeration is exhaustive (contains all values) and unique (contains any value at most once). In the more general case of finitely branching proof systems, it is undecidable whether a judgement is provable (or whether a type is inhabited, in the case of type systems), and our algorithm to enumerate proofs may not terminate for particular proof systems and judgements.

The SCIFE Scala library [Kuraj et al., 2015] extended the algebra of enumerations to dependent enumerations, that is, enumerations parameterised by a value. This allows SCIFE to surpass the expressive power of ADTs and enumerate structures with complex invariants, such as search trees. Our enumeration of proofs is much like a dependent enumeration in SCIFE, in that it is parametrised by the judgement. However, SCIFE is only designed as a combinator library, allowing the user to build enumerations with algebraic combinators on dependent enumerations: unlike other works on algebraic enumerations, the focus of SCIFE is not datatype-generic programming. We found that turning inference rules of a proof system into dependent enumerations by combinators can be cumbersome. We built on the idea of dependent enumerations, but we showed that the basic datatype-generic algorithm of FEAT that generates enumerations from data systems can be extended to the class of finitely branching proof systems. It is worth noting that SCIFE was empirically compared to other enumeration methods, and found to perform very well.

Indexing function Algebraic enumerations are based on indexing, i.e. a bijective map from the natural numbers to the set of structures. The main building block of such functions are pairing functions, that is, bijections $\pi : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, such as the Cantor pairing function. More precisely, the enumerations rely on the inverse of the pairing functions. Pairing functions are known to have efficient implementations, and are used in many algorithms. Pairing functions and their inverses can be implemented in linear time and constant space. This limit is only reached for non-monotone pairing functions, as the best result for monotone pairing functions is linear time and log space [Regan, 1992]. Pairing functions are a building block of many algorithms and theoretical results [Rosenberg, 2002]. FEAT uses the most straightforward pairing function, the Cantor pairing function, and I did not experiment with other versions.

5.6 Summary

This chapter showed how a functional representation of type inference rules can be turned into an enumeration of type derivations and hence the enumeration of typed program fragments, given a type and a typing context. The implementation is based on two areas: algebraic enumerations and proof enumerations. We reviewed results showing that algebraic enumeration, a datatype-generic method, can turn the definition of a data system into an enumeration of each type. We highlighted that a finitely branching proof space ensures that proofs can be enumerated the same way as data systems. We also discussed techniques to eliminate symmetries and make the enumerations efficient.

I did not experimentally compare our enumeration algorithms to other alternatives, beyond making sure that the implementation is reasonable. The techniques presented here complement the description of our solutions of the desugaring extension problem and describe a straightforward generic technique for proof enumerations.

Chapter 6

Searching for desugarings

In Chapter 4, we defined the desugaring extension problem (DEP). Chapter 5 then described an implementation technique for enumerative synthesis, which we will now apply to the DEP, yielding a baseline solution for the benchmarks introduced in Chapter 3.

From the overall task of finding a desugaring translation from the source language to the core language, we have delineated the DEP by requiring three prerequisites:

1. A decomposition of the full desugaring task into a series of DEPs.
2. A test set for each desugaring extension problem.
3. A search space for each desugaring extension problem.

For our baseline solution, we assume that the user provides the first two—the decomposition and the test sets—while we focus on the third: defining the search space. The search space is as crucial as it is difficult for any program synthesis task, as it should "strike a good balance between expressiveness and efficiency" (Gulwani et al. [2017], section 1.3.2). Expressiveness and efficiency are contradictory: we want the search space to be large, so that it can express a wide variety of translation rules, while at the same time narrow, so that the search is still feasible. As this is at the heart of this chapter, we will be referring back to this challenge as the *balancing challenge*.

Defining the search space allows us to differentiate between the shallow translation rules (desugarings) targeted with automatic synthesis methods and the more complex translation rules, which we judge to be out of reach. This is a moving target: we expect research to progress along this dimension of the problem.

In the baseline solution, we do not construct a fixed search space to give a definitive answer to the balancing challenge. Rather, we construct an extensible search space,

assigning further responsibility to the user: domain knowledge can be used to fine-tune the search space for a particular DEP. This is a limitation of the current approach, admitting that the balancing challenge is hard. But it also gives us the opportunity to progress gradually, exploring the design space for a synthesis algorithm.

In Section 6.1 we follow KLE's work when defining search spaces. They proposed a basic model for compositional translations, and constructed the Pidgin benchmark to list the limitations of this model. The main contribution of the current chapter is a search space that is close to the model proposed by KLE, but is defined by typing rules. Defining the search space by typing rules fits our formal framework, makes the search space easily extensible by adding new typing rules for program templates, and allows us to apply our enumerative synthesis method, which expects the specification in the form of typing rules. We show that this search space can gradually be extended with templates to cover all problematic cases identified by KLE: one of our judgement criteria of search spaces is based on how many intended translations they contain from the Pidgin benchmark.

Since search space extensions are motivated by the Pidgin languages and the Pidgin languages also serve as our first benchmark, there is some inevitable overlap between motivation and evaluation, and a danger of overfitting the search space to the Pidgin benchmark. While acknowledging this point is necessary, we should still note that the main contribution of the present Chapter is the base search space, based on typing rules, that makes the extensions possible in the first place, rather than the individual extensions themselves, which only illustrate this possibility. We design an extensible search space and illustrate its flexibility with concrete extensions. The double role of the Pidgin languages in the design of the extensions can also be justified: they are specifically designed by renowned researchers to cover a range of problematic features. Accepting that the Pidgin languages have a double role, we illustrate search space extensions with the language constructs from the Pidgin benchmark but separate these illustrations from the main discussion of the design.

Defining a search space with typing rules is the final building block necessary for achieving a testable sub-task, allowing us to evaluate our ideas empirically for the first time. We evaluate our method empirically on the benchmarks in Section 6.2. We demonstrate that the benchmarks can be decomposed into a series of DEPs, that these desugaring expression problems are solvable by our baseline algorithm, and that the solution of their composition—the sequential learning problem—yields a full solution for the benchmarks. The search space is not fixed: it allows and indeed requires user

guidance. As the amount of user guidance needed is our main criterion to evaluate the solutions, we carefully list every type of user guidance needed in the baseline solution, so that future solutions can be compared along this aspect.

6.1 Search spaces for translation rules

In this section, we will examine search spaces of increasing expressiveness, the expressive power of each to be demonstrated on the Pidgin benchmark. After briefly examining a simple search space to justify our choice of leaving test generation to the user, we look at a search space that is close to tree transducers (the model proposed by KLE), but fits our framework. We show that it can be reformulated as a type system for a meta-language, and this type system can be extended by additional typing rules for program templates. We then proceed to show a series of extensions covering all cases identified as problematic by KLE.

We define search spaces for a single source constructor: following our formal definition, the search spaces are parametrised by a given signature $\sigma_1 \times \cdots \times \sigma_n \to \sigma$ derived from the signature of the source constructor. The full search space for a DEP, which may need to find translation rules for multiple source term constructors, is formed by a tuple of translation rules for each source term constructor in the extension, where each translation rule can be chosen independently. The full search space for the DEP is simply the Cartesian product of the search spaces for the individual source term constructors.

6.1.1 Relabelling

The simplest search space we consider, $\mathcal{H}_{\text{relabel}}$, is formed by the term constructors of the core language. With $\mathcal{H}_{\text{relabel}}$ we can express desugarings which map each constructor of the source language to a term constructor of the core language of the same signature: a relabelling. Note that any arguments of the source term constructor are translated recursively, and the order of arguments cannot be changed. We call the desugaring extension problem specialised to $\mathcal{H}_{\text{relabel}}$ the *relabelling problem*.

Example 6.1.1. *In the Pidgin benchmark, many intended translation rules can be expressed as relabellings. In fact, all of them can, except for* `STrue`, `SFalse`, `SPrim`,

SBetween and SFor:

$$
\begin{aligned}
[\![\mathtt{SNum}(n)]\!] &= \mathtt{CNum}(n)\\
[\![\mathtt{SVar}(i)]\!] &= \mathtt{CVar}(i)\\
[\![\mathtt{SStr}(s)]\!] &= \mathtt{CStr}(s)\\
[\![\mathtt{SIf}(t_1,t_2,t_3)]\!] &= \mathtt{CIf}([\![t_1]\!],[\![t_2]\!],[\![t_3]\!])\\
[\![\mathtt{SLam}(\bar{i},t)]\!] &= \mathtt{CLam}(\bar{i},[\![t]\!])\\
[\![\mathtt{SApp}(t_1,t_2)]\!] &= \mathtt{CApp}([\![t_1]\!],[\![t_2]\!])\\
[\![\mathtt{SLet}(i,t_1,t_2)]\!] &= \mathtt{CLet}(i,[\![t_1]\!],[\![t_2]\!])\\
[\![\mathtt{SLetRec}(i,t_1,t_2)]\!] &= \mathtt{CLetRec}(i,[\![t_1]\!],[\![t_2]\!])\\
[\![\mathtt{SAssign}(i,t)]\!] &= \mathtt{CAssign}(i,[\![t]\!])\\
[\![\mathtt{SList}(\bar{t})]\!] &= \mathtt{CList}([\![\bar{t}]\!])\\
[\![\mathtt{SListCase}(t_1,t_2,t_3)]\!] &= \mathtt{CListCase}([\![t_1]\!],[\![t_2]\!],[\![t_3]\!])
\end{aligned}
$$

This is not surprising, since the core language is supposed to be close to a subset of the source language.

Relabelling can be a natural search space for a DEP when the source language itself is an extension of the core language, and we want to find the rules that form this extension.

This search space is simplified to triviality: for many source signatures in the Pidgin benchmark (like `SNum`) it contains only one translation rule, and in some cases (like `SFor`) it is empty. But even this very limited search space may contain translation rules that are incorrect (and therefore not our intended translation):

Example 6.1.2. *In the Pidgin benchmark, the relabelling search space contains an incorrect translation rule:*

$$[\![\mathtt{SLetRec}(i,t_1,t_2)]\!] \stackrel{*}{=} \mathtt{CLet}(i,[\![t_1]\!],[\![t_2]\!])$$

The previous example shows that even if we assume that the intended translation is a relabelling, we still may need appropriate test cases to rule out incorrect translations, which are not always trivial to find. To distinguish between recursive and non-recursive **let** constructs, we need an example that actually behaves recursively when interpreted as **letrec**. The example of distinguishing **let** and **letrec** demonstrates that generating test cases can be a hard problem even for a severely restricted search space. Here, in our baseline solution, we assume an appropriate user-written test set, i.e., one that is part of the expected user guidance.

6.1.2 Substitutions

We now consider a search space that is close to the original tree transducer [Comon et al., 2007] model suggested by KLE. We define the search space $\mathcal{H}_{\text{subst}}$ using substitution on terms with variables.

Definition 6.1.1 (Terms with variables)**.** Let T_Ω be an abstract language over signature $\Omega(S_\Omega, \mathcal{F}_\Omega, sig_\Omega)$, where $S_\Omega = \{\sigma_1, \ldots, \sigma_n\}$. Let a function $\rho : S \to \mathbb{N}$ assign a natural number to every sort in the signature. Let X^ρ be a finite set of variables, containing $\rho(\sigma)$ variables for every sort σ:

$$X^\rho = \{x_1^{\sigma_1}, \ldots, x_{\rho(\sigma_1)}^{\sigma_1}, \cdots, x_1^{\sigma_n}, \ldots, x_{\rho(\sigma_n)}^{\sigma_n}\}$$

and disjoint from $\mathcal{F}_\Omega$. Extend the signature function to the new variables as

$$\forall \sigma \in S_\Omega, \forall i \in [1 \ldots \rho(\sigma)], sig(x_i^\sigma) = \sigma$$

.

Let Ω^ρ be the signature with the same set of sorts as Ω, the term constructors extended with the variables of X^ρ, and the signature function extended to the variables. We denote the set of terms with signature Ω^ρ as $T_\Omega(X^\rho)$, and we call these terms *terms with variables* over the language T_Ω.

The signature function *sig* is extended to terms with variables as well (similarly as it is extended to terms). Terms with variables with sort σ is denoted as $T_\Omega^\sigma(X^\rho)$.

Definition 6.1.2 (Substitutions)**.** Let Ω be the core signature and ρ a function determining the number of variables. For a sort $\sigma \in S_\Omega$ a term $t \in T_\Omega^\sigma(X^\rho)$ defines a function by substituting the arguments into the variables. We will refer to translations (and interpretations) that can be expressed by terms with variables as *substitutions*.

Let Σ be our source signature, and let $f \in \mathcal{F}_\Sigma, f : \sigma_1 \times \cdots \times \sigma_n \to \sigma$ be a source term constructor. Let $s : S_\Sigma \rightharpoonup S_\Omega$ be the sort mapping from source sorts to core sorts. For any sort $\sigma' \in S_\Omega$, let $\rho_f(\sigma')$ be the number of times $s^{-1}(\sigma')$ occurs in the domain of the signature of f (among the sorts $\sigma_1, \ldots, \sigma_n$). (As the mapping is not necessarily injective, $s^{-1}(\sigma')$ could be a set, and we sum the occurrences of each element.) The search space for f is defined as $\mathcal{H}_{\text{subst}}^f = T_\Omega^\sigma(X^{\rho_f})$, the set of substitutions for f.

Every relabelling is a substitution.

The translations defined by translation rules in $\mathcal{H}_{\text{subst}}$ are also definable by a deterministic top-down tree transducer (DTOP). But $\mathcal{H}_{\text{subst}}$ is more restricted:

- It guarantees that the output of a translation is well-typed according to the core language's signature (the output corresponds to the abstract syntax)
- It guarantees that the translation preserves the sort mapping from source sorts to core sorts.

We do not want translation rules that violate these restrictions, so substitutions are a better search space than DTOPs: more efficient, and similarly expressive in our domain. Unlike the finite set of states of a DTOP, the set of typing contexts in $\mathcal{H}_{\text{subst}}$ is infinite.

In $\mathcal{H}_{\text{subst}}$ and DTOPs a term constructor determines the translation rule.

Example 6.1.3. *To demonstrate the limitations of this determinism, let us look at the two translation rules for* `SPrim` *in the Pidgin benchmark. They cannot be expressed in either of these search spaces, as they require a different behaviour based on the number of arguments:*

$$\begin{aligned} [\![\texttt{SPrim}(o,[t_1])]\!] &= \texttt{CPrim1}(o,[\![t]\!]) \\ [\![\texttt{SPrim}(o,[t_1,t_2])]\!] &= \texttt{CPrim2}(o,[\![t_1]\!],[\![t_2]\!]) \end{aligned}$$

The translation rules covered by $\mathcal{H}_{\text{subst}}$ (along with DTOPs) is not only deterministic (requiring at most one translation rule for every source term constructor), but are also total.

Example 6.1.4. *The rules for* `SPrim`*, as shown in the previous example, also demonstrate non-total rules. They do not cover every case: they do not cover cases when the argument list is empty or contains more than two elements. Expressing the two rules for* `SPrim` *is possible with a bottom-up tree transducer or a non-deterministic top-down tree transducer (both of which are more expressive than DTOPs) but requires additional states that do not correspond to the sorts given by the abstract syntax.*

It is clear that $\mathcal{H}_{\text{subst}}$ is much more expressive than relabellings, but the Pidgin benchmark does not demonstrate this expressiveness.

Example 6.1.5. *In the Pidgin benchmark, only two additional rules can be covered by* $\mathcal{H}_{subst}$*:*

$$[\![\texttt{STrue}]\!] = \texttt{CBool}(\text{TRUE}) \qquad [\![\texttt{SFalse}]\!] = \texttt{CBool}(\text{FALSE})$$

The reason is that the Pidgin source and core languages were specifically chosen by KLE *to demonstrate potential problems with modelling the translation with a tree transducer; therefore, all of the more complex translation rules are designed to fall outside of this search space.*

6.1.3 Substitutions as a type system

So far we remained very close to KLE's original model, tree transducers. We now begin our endeavour to gradually extend our model, to cover the features that KLE found problematic: analysing arguments, re-arranging children, throwing errors, and generating fresh names. To achieve this, we reformulate $\mathcal{H}_{\text{subst}}$ to have an extensible search space.

We present $\mathcal{H}_{\text{subst}}$ as a subset of a general purpose meta-language—a simplified ML-like functional language with algebraic data types—where the subset is defined by typing rules. This subset can be extended with additional program templates written in the meta-language by providing their typing rules, lifting the main limitation of the tree transducer model. Moreover, presenting the meta-language through the typing rules allows us to directly apply the enumeration method introduced in Chapter 5.

The generic method to define the subsets via typing rules is the following. Let $\sigma_1, \ldots, \sigma_n$ and σ be core sorts. A translation rule is a computation with type $\sigma_1 \times \cdots \times \sigma_n \to \sigma$. We uniformly represent isomorphic types that differ in the order of the arguments by introducing an environment: $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$, where Γ is a function from a set of variables to core sorts, usually given as a set of pairs in the form of $x : \sigma$, where x is a *meta-variable* (not an identifier in the source or core languages), and σ is a core sort. We represent translation rules as judgements of the form

$$\Gamma \vdash T : \sigma$$

where T is a meta-program corresponding to a translation rule typed $\sigma_1 \times \ldots \times \sigma_n \to \sigma$. This uniform representation is necessary for an adequate representation of judgements, as defined in Section 5.1.

To avoid confusing the typing rules (defining various ways to build meta-programs, programs that define translation rules) and the translation rules (building blocks of compositional translations), we will refer to the former as *meta-rules*. A set of meta-rules define a meta-language for translation rules.

Let us now apply this to defining the base meta-language, equivalent to $\mathcal{H}_{\text{subst}}$. There are many ways to write the rules of a simple type system to achieve this goal. Our version was carefully designed such that it has the *Curry-Howard property* (see Section 5.1): we require every type derivation to uniquely correspond to a translation rule in $\mathcal{H}_{\text{subst}}$. The meta-language is shown in Figure 6.1.

$$\begin{array}{rcl} \sigma_1,\ldots,\sigma_n,\sigma & \in & S \\ f & \in & \mathcal{F} \\ t & ::= & x \mid c \mid f(t_1,\ldots,t_n) \end{array}$$

$$\frac{}{\Gamma; x:\sigma \vdash x:\sigma}\ \textsc{Axiom} \qquad \frac{c:\sigma}{\Gamma \vdash c:\sigma}\ \textsc{C-rule}$$

$$\frac{\Gamma \vdash t_1:\sigma_1 \quad \cdots \quad \Gamma \vdash t_n:\sigma_n \quad f:\sigma_1 \times \cdots \times \sigma_n \to \sigma}{\Gamma \vdash f(t_1,\ldots,t_n):\sigma}\ \textsc{F-rule}$$

Figure 6.1: Terms with variables as proofs

6.1.4 Analysing arguments

The first feature that cannot be expressed with $\mathcal{H}_{\text{subst}}$ is when we have multiple rules for the same source term constructor. In our first extension we allow differentiating between cases based on the structure of the arguments. We introduce a rule that conforms to the fold model: it cannot access the original source arguments, only their translation to the core language, and we branch based on the structure of the core arguments. We introduce case separation for core sorts, as follows:

Let $\sigma \in S_\Omega$ be a core sort. Let $\forall i \in [1\ldots n].f_i : \sigma_i^1 \times \cdots \times \sigma_i^{k_i} \to \sigma$ be the core term constructors with co-domain σ, where k_i is the number of arguments of f_i and $\sigma_i^1 \times \cdots \times \sigma_i^{k_i}$ is the domain of f_i. If a translation rule has an argument of sort σ, then that argument could be constructed in n ways (by one of the core term constructors with co-domain σ). We introduce a template for meta-rules that allows us to separate these n cases:

$$\frac{\Gamma; x_1^1:\sigma_1^1,\ldots,x_1^{k_1}:\sigma_1^{k_1} \vdash M_1:\sigma' \quad \ldots \quad \Gamma; x_n^1:\sigma_n^1,\ldots,x_n^{k_n}:\sigma_n^{k_n} \vdash M_n:\sigma'}{\Gamma; y:\sigma \vdash \mathbf{case}\, y\, \mathbf{of}\, f_1(x_1^1,\ldots,x_1^{k_1}) \to M_1 \ldots f_n(x_n^1,\ldots,x_n^{k_n}) \to M_n : \sigma'}\ \textsc{Case}\ \sigma$$

This template for meta-rules is parametrised by the core sort σ.

Example 6.1.6. *As a concrete example, let us use the* `List`$_{CTerm}$ *sort from the Pidgin core language as* σ*, as this is needed to express the two rules for* `SPrim`*:*

$$\frac{\Gamma \vdash M:\sigma' \quad \Gamma; x:\sigma, y:[\sigma] \vdash N:\sigma'}{\Gamma; y:[\sigma] \vdash \mathbf{case}\, y\, \mathbf{of}\, [] \to M; (x:y) \to N : \sigma'}\ \textsc{Case List}_{CTerm}$$

As we have already discussed in Section 4.4, compositional translations have a limited ability to analyse the structure of children, as any such analysis is only possible

after the recursive call. In the above example, this means that in order to faithfully represent the SPrim rules, we must rely on translating a list of source terms into a list of core terms. In general, in order for the case separation to be what we expect, the translation rules for the term constructors of the source sort must keep the structure of the arguments we branch on. This is an inherent limitation of the fold model: in this respect, this model is more limited in expressiveness than a non-deterministic tree transducer.

Note that the CASE rule is the first rule to change the environment. In $\mathcal{H}_{\text{subst}}$ the environment Γ is never changed by any meta-rule, therefore it remains fixed for the search space $\mathcal{H}^f_{\text{subst}}$ for a given source term constructor f. $\mathcal{H}^f_{\text{subst}}$ can be described with a context-free grammar (it can be represented by an ADT), while the extended meta-language that contains the CASE rule cannot. The CASE rule is the first rule relying on a more general implementation of enumerations, as introduced in Chapter 5.

6.1.5 Errors

While the CASE meta-rule allows multiple translation rules for the same term constructor, we are still limited to total translation rules in the resulting meta-language. To cover non-totality, we need a way to raise syntax errors for the cases not covered by the intended translation:

$$\frac{}{\Gamma \vdash \textbf{syntax\_error} : \sigma}\ \textsc{Throw}$$

Example 6.1.7. *With* CASE *and* THROW *the intended translation rule for* SPrim *can be expressed as a single rule:*

$$\begin{aligned}
[\![\mathit{SPrim}(o,ts)]\!] \quad = \quad & \textbf{case}\ [\![ts]\!]\ \textbf{of} \\
& [] \to \textbf{syntax_error} \\
& (t_1 : ts_1) \to \quad \textbf{case}\ ts_1\ \textbf{of} \\
& \qquad\qquad [] \to \mathit{CPrim1}(o,t_1); \\
& \qquad\qquad (t_2 : ts_2) \to \quad \textbf{case}\ ts_2\ \textbf{of} \\
& \qquad\qquad\qquad\qquad [] \to \mathit{CPrim2}(o,t_1,t_2); \\
& \qquad\qquad\qquad\qquad (_ : _) \to \textbf{syntax_error}
\end{aligned}$$

6.1.6 Transforming arguments

The next feature not covered by tree transducers is re-arranging the children. We do not define a search space for re-arrangements, but we allow extending the search space

with fixed, user defined re-arrangements.

Let $\sigma_1, \sigma_2 \in S_\Omega$ be two core sorts and $\mathsf{tr} : \sigma_1 \to \sigma_2$ a function in the meta-language that transforms between them, which re-arranges the structure contained in σ_1:

$$\frac{\Gamma; x : \sigma_2 \vdash M : \sigma}{\Gamma; y : \sigma_1 \vdash \mathbf{let}\, x = \mathsf{tr}(y)\, \mathbf{in}\, M : \sigma}\ \textsc{Transform}$$

Example 6.1.8. *In the Pidgin benchmark we have an intended translation rule that requires re-arranging the children: the rule for the* `SFor` *source constructor. However, simply adding a user-defined translation rule is not enough to express the intended translation rule. The argument we need to re-arrange belongs to the* $\texttt{List}_{\mathit{SForBind}}$ *source sort, which has no corresponding pair in the Pidgin core language; not even SForBind has a corresponding sort. In Section 4.4, we extended the notion of sort-preserving compositional translations to partial sort mappings by extending the core language with new sorts (data types) that are essentially copies of the source sorts.*

We may notice that the sort (data type) SForBind is isomorphic to pairs, and treat its copy as such in the meta-language: $\mathit{Id} \times \mathit{CTerm}$. *Furthermore, as* $\texttt{List}_{\mathit{SForBind}}$ *is isomorphic to a list of pairs, we can represent its copy in the meta-language as* $\texttt{List}\,(\mathit{Id} \times \mathit{CTerm})$. *The user-defined transformation needed for extending the meta-language is unzip: we want to re-arrange the list of pairs into two separate lists:*

$$\texttt{unzip} : \texttt{List}\,(\mathit{Id} \times \mathit{CTerm}) \to (\texttt{List}\,\mathit{Id}) \times (\texttt{List}\,\mathit{CTerm})$$

The result of unzipping this list is a pair of lists, which is yet another datatype (sort) that is not part of the core language. We need to once again extend the core language with a new sort, a list of identifiers and core terms, as well as its sole term constructor. With these extensions, we can then express the Unzip *rule as the specialised version of the* Transform *rule, where the* `tr` *transformation is specialised to the* `unzip` *meta-language function:*

$$\frac{\Gamma; x : (\texttt{List}\,\mathit{Id}) \times (\texttt{List}\,\mathit{CTerm}) \vdash M : \sigma}{\Gamma; y : \texttt{List}\,(\mathit{Id} \times \mathit{CTerm}) \vdash \mathbf{let}\, x = \texttt{unzip}(y)\, \mathbf{in}\, M : \sigma}\ \textsc{Unzip}$$

The new sort, $(\texttt{List}\,\mathit{Id}) \times (\texttt{List}\,\mathit{CTerm})$ *does not appear in any core term constructor, so adding the* Unzip *meta-rule alone leads us nowhere: we can not use the resulting value in any other meta-rules. We need to add the corresponding* Case *rule to the set of meta-rules, since otherwise this sort could not appear in any program terms.*

As demonstrated by the previous example, sometimes a transformation may result in a datatype that is not part of the core language (just the meta-language), so we know in advance that we need to analyse the result with a CASE rule. In these cases, we can combine the two meta-rules (the TRANSFORM rule and the CASE RULE) into one, significantly reducing the search space:

$$\frac{\Gamma; x_1^1 : \sigma_1^1, \ldots, x_1^{k_1} : \sigma_1^{k_1} \vdash M_1 : \sigma' \quad \ldots \quad \Gamma; x_n^1 : \sigma_n^1, \ldots, x_n^{k_n} : \sigma_n^{k_n} \vdash M_n : \sigma'}{\Gamma; y : \sigma_1 \vdash \mathbf{case}\, tr(y)\, \mathbf{of}\, f_1(x_1^1, \ldots, x_1^{k_1}) \to M_1 \ldots f_n(x_n^1, \ldots, x_n^{k_n}) \to M_n : \sigma'}\ \text{TRANSFORM-CASE}$$

Example 6.1.9. *The required meta-rule to express the translation rule of* `SFor` *can be obtained by specialising the* TRANSFORM-CASE *rule scheme to* unzip*. The* CASE *rule for the output of the* unzip *function has only one branch, as pairs have only one term constructor. In the meta-language we can replace general* **case** *pattern matching with* **let** *pattern matching:*

$$\frac{\Gamma; x_1 : \texttt{List}\,\sigma_1, x_2 : \texttt{List}\,\sigma_2 \vdash M : \sigma}{\Gamma; x : \texttt{List}\,(\sigma_1 \times \sigma_2) \vdash \mathbf{let}\,(x_1, x_2) = \mathit{unzip}(x)\,\mathbf{in}\, M : \sigma}\ \text{UNZIP'}$$

Just as in the case of translating `SPrim`*, faithfully translating* `SFor` *depends on the translation of its arguments. In this case the argument belongs to the SForBind sort: we rely on translating a list of pairs of identifiers and source terms to a corresponding list of pairs of identifiers and core terms.*

The TRANSFORM and CASE rules, as well as their combined TRANSFORM-CASE version, all rely on analysing the argument of the translation rule. This analysis is limited, as it happens after the recursive call, and they all share this limitation. Faitfhul representation of certain translation rules, as shown by the previous example, requires that the structure of the arguments is preserved. Note that this is exactly the condition that we set up in Section 4.4 when a source sort has no corresponding core sorts, and the sort-mapping is partial: we simply 'copy' these term constructors in the meta-language.

6.1.7 Layering

With the introduction of the CASE, TRANSFORM, TRANSFORM-CASE and ERROR rules, we have successfully extended the search space. However, we have also lost the Curry-Howard property of having exactly one type derivation for every translation

rule. There are two reasons for this: the new meta-rules can be permuted with the F-RULE, and the new meta-rules can be permuted with each other.

The former can be solved with layering. We introduce a new judgement type: $\vdash_E$, and change the new rules (CASE,TRANSFORM, TRANSFORM-CASE and ERROR) to use it. We connect the new rules with the basic rules through a one-way conversion from the basic judgement type to the extended judgement type $\vdash_E$:

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash_E M : \sigma}\ \textsc{Extend}$$

The latter can be solved with focusing (see Section 5.4.3): only allow application of the CASE and TRANSFORM rules in a controlled way (in a fixed order). We will ignore this complication for now. Transforming our type system to an equivalent one with the Curry-Howard property is challenging, and we do not have benchmarks where the CASE or TRANSFORM rules are applicable to more than one variable in the environment, thus we currently lack the method to evaluate any attempts to solve this problem. To a certain extent, the combined TRANSFORM-CASE rule reduces this problem, as it fixes the order of the TRANSFORM and CASE rules applied to the same variable.

6.1.8 Fresh name generation

The last feature noted by KLE as problematic is fresh name generation. Generating fresh names depends on the context of the translation rule (depending specifically on the other names occurring in the entire generated program), not merely on the arguments of the translation rule, hence it is not a mathematical function from terms to a term. Fresh name generation is internal to the translation itself, and needs to be handled by our meta-language.

Extending our meta-language with a name-generating effect is standard and straightforward.

$$\frac{\Gamma; y : \texttt{Id} \vdash M : \sigma}{\Gamma \vdash \textbf{let}\ y = \mathsf{gensym}()\ \textbf{in}\ M : \sigma}\ \textsc{Fresh'}$$

Unfortunately, adding the FRESH' rule blows up the search space, making synthesis unfeasible. We would like to take the binding structure of the core language into account, and use fresh name generation to introduce new identifiers only where the

core language allows it, limiting the scope of the introduced identifier. A straightforward solution is to add a separate meta-rule for every binding construct in the core language.

Example 6.1.10. *The following meta-rule is specific to the* `CLet` *construct in the Pidgin core language, taking its binding structure into account:*

$$\frac{\Gamma \vdash M_1 : \mathit{CTerm} \quad \Gamma; x : \mathit{CTerm} \vdash M_2 : \mathit{CTerm}}{\Gamma \vdash \textbf{let } i = \texttt{gensym}() \textbf{ in } \texttt{CLet}(i, M_1, M_2[\texttt{CVar}(i)/x]) : \mathit{CTerm}}\ \textsc{Fresh}$$

Similar rules can be added for `CLetRec` *or* `CLam` *as well.*

6.1.9 Summary: the complete meta-language for translation rules

We now collate all the extensions in a parametric search space $\mathcal{H}_{\text{meta}}$ that is based on all the meta-rules introduced so far. Our new feature is restriction: we control the exact set of meta-rules by parameters, allowing for user assistance.

We introduce four new parameters:

1. The set $\mathcal{B}$ of base cases: core term constructors that we allow to appear in the C-RULE or F-RULE.

2. The set $\mathcal{T}$ of transformations that we can apply to arguments.

3. The set $\mathcal{A}$ of case analyses, which contains the core sorts that can appear in the CASE rule, and the translation functions between core sorts that can appear in the TRANSFORM-CASE rule. Since using the identity function on sort σ as a translation in a TRANSFORM-CASE rule is equivalent to a CASE rule on σ, we handle them in a single parameter. (We can think of the TRANSFORM-CASE rule as a user-defined view of the data that we can analyse, and the CASE is a special case where this view is defined by the identity.)

4. The set $\mathcal{V}$ of binding constructs in the target language that we specify the FRESH rule for.

The search space $\mathcal{H}_{\text{meta}}(\mathcal{B}, \mathcal{T}, \mathcal{A}, \mathcal{V})$ is generated by the following set of meta-rules:

1. The AXIOM, EXTEND and ERROR rules are the fixed part of the search space.

2. The C-RULE and the F-RULE are restricted to term constructors in $\mathcal{B}$.

3. The TRANSFORM rule with the condition $\mathsf{tr} \in \mathcal{T}$, using the extended judgement $\vdash_E$.

4. The CASE and TRANSFORM-CASE rules with the condition $\mathsf{tr} \in \mathcal{A}$, using the extended judgement $\vdash_E$.

5. The FRESH rule, specified for binding constructs in $\mathcal{V}$.

6.2 Evaluation

6.2.1 Hypothesis and methodology

Our hypothesis is that for all of the benchmarks:

1. We can set up a sequential learning task, decomposing the full desugaring problem into a series of desugaring extension problems.

2. Each individual desugaring extension problem can be solved with our enumerative synthesis algorithm, using a search space based on $\mathcal{H}_{\text{meta}}$ as defined in Subsection 6.1.9.

3. The individual desugaring extension problems can be combined to solve the overall problem.

As noted earlier, our search space needs to meet several contradictory requirements. We do not expect to solve the balancing challenge, but this is one of the main dimensions along which we expect research to progress. This situation places us in a tight spot: it is hard to measure how well results generalise. As we set up a search space for our specific benchmarks, the balancing challenge manifests itself in the following way: a successful search in an overly strict search space would not carry any generalisable result. On the other hand, a search space that is too expansive would preclude any feasible enumerative search, making our endeavour completely hopeless.

We attempt to address this problem as follows. We rely on the parametric search space $\mathcal{H}_{\text{meta}}$, which allows user guidance in a controlled manner. We set up an *initial search space* for all benchmarks, based on assumptions we find intuitive. Next, we proceed in an incremental manner, repeating any failed tests with additional user

guidance: the parametric setup of $\mathcal{H}_{\text{meta}}$ allows us to construct new search spaces that are strictly contained in the previous ones. Although we report a number of different metrics from our tests, we believe that the amount of user assistance needed for each step to successfully complete is the most useful metric. We evaluate $\mathcal{H}_{\text{meta}}$ and the enumeration synthesis library on all benchmarks we introduced in Chapter 3 in this way.

The setup of each benchmark will consist of:

1. The syntax of the source and core languages.
2. The intended translation between them.
3. The sort mapping from source sorts to core sorts.
4. The initial sublanguage Σ_0 and its initial translation.
5. The initial search space.

When presenting the sort mapping (3), we also include the extra sorts we extended the original core language with, as they are represented as meta-language data types. Thus the mapping we show is total, but the codomain is an extension of the original core language. The partial sort mapping from source sorts to core sorts excludes sorts whose image is not a core sort; the extra part of the mapping is supposed to be generated automatically.

The test itself consist of a series of steps. We first attempt to decompose the rest of the source language into a series of increasing sublanguages, defining a sequence of individual desugaring extension problems. We write test sets for each desugaring extension problem, then check if they can be solved by the initial search space. If any step fails within our time and memory limits, we add additional user guidance, either by fixing some part of the translation, or by restricting the initial search space for the failed step.

The search times reported were obtained on an Intel E3-1245v5 @ 3.50GHz with 8 cores and 32 GB RAM, running Debian Linux 11.5 and GHC 9.0.2. We set up a one hour time limit and a 8GB memory limit for each individual desugaring extension problem.

6.2.2 Pidgin languages

The first test was conducted on the Pidgin benchmark, specifically designed by KLE to demonstrate the challenging aspects of defining a search space. While the Pidgin languages were not intended to serve as a benchmark for program synthesis, it is perfect for illustrating our notation, the steps required to set up a sequential learning problem, and the way we display and analyse the results.

Setup

Syntax and intended translation The syntax of the Pidgin source and core languages is shown in Figure 3.1a, and the intended translation from the Pidgin source to the Pidgin core language is shown in Figure 3.1.

Sort mapping As we do not have a core sort corresponding to the *SForBind* sort, we extended the core language with a new sort *(Id,CTerm)*, a pair of identifiers and core terms. Note that since we do not have any core term constructors that have parameters of this new sort, no terms with this sort can appear in core programs. This sort can only occur together with the CASE or TRANSFORM-CASE rules. The sort mapping was the following:

$Bool \to CTerm$	$Id \to Id$	$Number \to Number$
$String \to String$	$Op \to Op$	$STerm \to CTerm$
$\texttt{List}_{Id} \to \texttt{List}_{Id}$	$\texttt{List}_{STerm} \to \texttt{List}_{CTerm}$	$SForBind \to (Id,CTerm)$
$\texttt{List}_{SForBind} \to \texttt{List}_{(Id,CTerm)}$		

Σ_0 sublanguage and its translation The sublanguage Σ_0 contained numbers, strings, identifiers, the constructors for lists, and the `SFBind` term constructor. We defined the translation of Σ_0 as follows:

- We preserve numbers, strings, identifiers and lists of identifiers.
- Lists of source terms are translated to lists of core terms by individually translating the elements and preserving their order.
- A term with `SFBind` as the head is translated to a pair of an identifier (from the first argument) and a core term (by translating the second argument, a source term).

- Terms with sort $\texttt{List}_{\mathit{SForBind}}$ are translated into a list of pairs by translating the individual elements into pairs, preserving the order.

Initial search space For the initial search space $\mathcal{H}^1_{\text{Pidgin}} = \mathcal{H}_{\text{meta}}(\mathcal{B}, \mathcal{T}, \mathcal{A}, \mathcal{V})$ we set the four parameters as follows:

- The set of base rules $\mathcal{B}$ contained all core term constructors, including list constructors for term and identifier lists. It did not contain number, string or identifier constants. In general, we do not want to allow such constants in the search space: they rarely occur in desugaring rules, and they blow up the search space. However, we needed the Boolean constants TRUE and FALSE, since the Boolean sort is not part of the source language. We also included all operator symbols, which required some domain knowledge about their role: while operator symbols are constants, they correspond to various operations.

- The set of translations $\mathcal{T}$ is empty.

- The set of case analyses contains two functions: $\mathcal{A} = \left\{\mathsf{id}_{\texttt{List}\,\mathit{CTerm}}, \mathsf{unzip}_{\texttt{List}\,(\mathit{Id},\mathit{CTerm})}\right\}$. The two rules are $\textsc{Case}_{\textsc{List}\,\mathit{CTerm}}$ and $\textsc{Transform-Case}\ \mathsf{unzip}_{\texttt{List}\,(\mathit{Id},\mathit{CTerm})}$.

- The set of binding construct $\mathcal{V}$ contained only `CLet`. Our sole rule in this group is $\textsc{Fresh}_{\texttt{CLet}}$.

Results

We ran the first test with $\mathcal{H}^1_{\text{Pidgin}}$, and found all but one of the intended translation rules within the time limit: this test did not find the intended desugaring for `SBetween`. This is the largest translation rule, with an AST size of 14, and no subsequent steps depended on it, so this failure did not affect any other steps. We constructed a second, specialised search space $\mathcal{H}^2_{\text{Pidgin}}$ specifically for `SBetween`:

- The base rules retained operators from among the constants and the core term constructors `CVar` and `CPrim2`, but no other term constructors or constants.

- We did not include any transformations or case analysis rules.

- We kept the $\textsc{Fresh}_{\texttt{CLet}}$ binding rule.

Table 6.1: Benchmark – Pidgin languages

Task	New constructors	Hyp. space	AST size	Index	#test set	Time
Σ_1	`SNum`	$\mathcal{H}^1$	2	1	2	0s
Σ_2	`SStr`	$\mathcal{H}^1$	2	1	2	0s
Σ_3	`SPrim`	$\mathcal{H}^1$	12	16,567,096	5	2min 27s
Σ_4	`SVar`, `SLet`	$\mathcal{H}^1$	6	1,298	2	0s
Σ_5	`STrue`, `SFalse`	$\mathcal{H}^1$	4	10	4	0s
Σ_6	`SAssign`	$\mathcal{H}^1$	3	18	1	0s
Σ_7	`SBetween`	$\mathcal{H}^2$	14	157,160,392	9	28min 20s
Σ_8	`SIf`	$\mathcal{H}^1$	4	171	2	0s
Σ_9	`SLam`, `SApp`	$\mathcal{H}^1$	6	1,813	2	0s
Σ_{10}	`SLetRec`	$\mathcal{H}^1$	4	132	1	0s
Σ_{11}	`SList`	$\mathcal{H}^1$	2	3	2	0s
Σ_{12}	`SListCase`	$\mathcal{H}^1$	4	207	2	0s
Σ_{13}	`SFor`	$\mathcal{H}^1$	11	57,334,664	3	9min 44s
Σ	total				37	40min 32s

The modified sequential learning task yielded a full solution of the Pidgin benchmark. The results are summarised in Table 6.1.

The rows of the table correspond to desugaring extension problems. In each problem Σ_n the translations of the sublanguages $\Sigma_0, \ldots, \Sigma_{n-1}$ are assumed to be known. We work our way downwards: the test sets cannot use term constructors from below. To get some rough measurement of the complexity of each task, we noted the size of the intended desugaring (AST size), as well as the *index* of the translation found in the enumeration for each desugaring extension problem, that is, the number of attempts tried before a solution was found. We also show the number of handwritten tests we used in order to find the solution.

Translation rules found The solution found was identical to the intended translation in Figure 3.1 in all but three cases. In the case of `SPrim` and `SFor`, the difference was merely the syntax: the found translation rules were expressed with the syntax of our meta-language, as we have already shown when we introduced the CASE and TRANSFORM rules.

The SPrim rule:

$$\begin{aligned}
[\![\texttt{SPrim}(o,ts)]\!] \;=\; & \textbf{case } [\![ts]\!] \textbf{ of} \\
& [] \to \textbf{syntax_error} \\
& (t_1 : ts_1) \to \;\; \textbf{case } ts_1 \textbf{ of} \\
& \qquad [] \to \texttt{CPrim1}(o,t_1); \\
& \qquad (t_2 : ts_2) \to \;\; \textbf{case } ts_2 \textbf{ of} \\
& \qquad\qquad [] \to \texttt{CPrim2}(o,t_1,t_2); \\
& \qquad\qquad (_ : _) \to \textbf{syntax_error}
\end{aligned}$$

The SFor rule:

$$[\![\texttt{SFor}(t_1,t_2,t_3)]\!] \;=\; \textbf{let } (is,ts) = \texttt{unzip}([\![t_2]\!]) \textbf{ in } \texttt{CApp}([\![t_1]\!],[\texttt{CLam}(is,[\![t_3]\!]),\texttt{CList}(ts)])$$

In the case of SBetween, the algorithm found an expression semantically equivalent to our intended translation but smaller:

$$\begin{aligned}
[\![\texttt{SBetween}(t_1,t_2,t_3)]\!] \;=\; & \textbf{let } i_1 = \texttt{gensym}() \textbf{ in } \texttt{CLet}(i_1,[\![t_1]\!], \\
& \textbf{let } i_2 = \texttt{gensym}() \textbf{ in } \texttt{CLet}(i_2,[\![t_2]\!], \\
& \texttt{CPrim2}(\wedge,\texttt{CPrim2}(<,\texttt{CVar}(i_2),[\![t_3]\!]),\texttt{CPrim2}(<,\texttt{CVar}(i_1),\texttt{CVar}(i_2)))))
\end{aligned}$$

Decomposition The test sets of the desugaring extension problems cannot contain source term constructors from later extensions, but a test set still needs to exclude non-intended translations. Sometimes the translations of term constructors depend on each other and cannot be learnt separately. The Pidgin benchmark has three such cases:

1. We cannot learn SLam and SApp separately, as they only show their behaviour together.

2. KLE did not specify the exact semantics of the source and core languages — this gave us some freedom in implementing them. Our implementation does not allow for executing open programs containing non-declared variables; in other words, all variables must be declared in a SLam, SLet or SLetRec expression before use, otherwise an exception is raised. This means that we could not use SVar on its own without one of these term constructors: we grouped it together with SLet in task Σ_4.

3. The last case where we needed to group multiple term constructors together was the STrue, SFalse group. The symmetry of the Boolean constants means that we need to learn them together, relying on the already fixed translations of the $\wedge$ and $\vee$ operations to rule out translations that map to the opposite Boolean value in the core language.

Since all other groups only contain one term constructor, this decomposition is in line with our assumption that we often do not need to learn more than one rule at a time. This, in turn, supports our hypothesis that the language can be partitioned into small groups of term constructors.

Test programs The majority of test programs are very small and simple: in most cases, we do not need large test programs, and they could plausibly be generated by automatic testing. There are three exceptions:

1. To learn the full semantics of `SBetween`, we needed test programs where the evaluation order of arguments matters.
2. To distinguish `SLetRec` from `SLet`, we needed recursive examples.
3. The source term constructor `SFor` assumes a combinator function as its first argument, which leads to an example that dwarfs the rest. This last case is somewhat artificial, though.

In many cases, our algorithm found a desugaring equivalent to the intended semantics on pure terms, but changed the evaluation order of the arguments. The Pidgin source language has side effects, induced by the `SAssign` operator. We utilised it to add test cases that depend on the evaluation order.

Sometimes the order of the test programs in a test set is important. The majority of the time is spent evaluating various core programs that we get via the tested translations, especially when we need recursive test programs, such as in the case of `SFor`. Recursive test programs easily lead to non-terminating core programs, thus the time spent checking prospective translations on these test programs greatly depends on the resource limit. If the resource limit is too low, we may get an incorrect desugaring, if too high, the search may take a long time. It is possible to learn the intended semantics of `SFor` with only a single test program, but since it would need to be recursive and large, the search time would suffer and would depend on the resource limit, making the learning process impractical. As currently we can only tune the resource limit manually (automatic tuning is left to future research), it is important for performance to use simple, non-recursive source programs first in a test suite. We added such a test program to our test set, before the full example, and we found that in this case a reasonably high resource limit did not significantly impact the time of the search. Thus, no manual tuning was needed, while the search time was reduced by around 90% (compared to using only a single recursive test program).

$$
\begin{array}{rcl}
t \in \mathit{STerm} & ::= & \ldots \mid \texttt{SListComp}(t,q) \\
q \in \mathit{SQual} & ::= & \texttt{QEmpty} \mid \texttt{QBind}(i,t,q) \mid \texttt{QGuard}(t,q) \mid \texttt{QLet}(i,t,q) \\
e,e_1,e_2 \in \mathit{CTerm} & ::= & \ldots \mid \texttt{MVar}(i) \mid \texttt{MLam}(i,e) \mid \texttt{MApp}(e_1,e_2) \mid \texttt{Return}(e) \mid \texttt{Bind}(e_1,e_2)
\end{array}
$$

(a) Adjusted extension of Pidgin with basic list comprehensions – syntax

$$
\begin{array}{rcl}
[\![\texttt{SListComp}(t,q)]\!] & = & \texttt{MApp}([\![q]\!],[\![t]\!]) \\
[\![\texttt{QEmpty}]\!] & = & \texttt{MLam}(\%u,\texttt{Return}(\texttt{MVar}(\%u))) \\
[\![\texttt{QBind}(x,t,q)]\!] & = & \texttt{MLam}(\%u,\texttt{Bind}([\![t]\!],\texttt{CLam}([x],\texttt{MApp}([\![q]\!],\texttt{MVar}(\%u))))) \\
[\![\texttt{QGuard}(t,q)]\!] & = & \texttt{MLam}(\%u,\texttt{CIf}([\![t]\!],\texttt{MApp}([\![q]\!],\texttt{MVar}(\%u)),\texttt{CList}([]))) \\
[\![\texttt{QLet}(x,t,q)]\!] & = & \texttt{MLam}(\%u,\texttt{Let}(x,[\![t]\!],\texttt{MApp}([\![q]\!],\texttt{MVar}(\%u))))
\end{array}
$$

(b) Adjusted extension of Pidgin with basic list comprehensions – intended translation

Figure 6.2: Adjusted extension of Pidgin with basic list comprehensions

6.2.3 Basic list comprehensions

Our second benchmark is list comprehensions: we extended the Pidgin source and core languages with new constructs, mimicking the expected process of incremental learning.

This benchmark is special, as the intended translations express the desugarings of qualifiers with meta-level lambdas. This means the meta-language of the benchmark contains λ-abstraction and application, which our search space does not. Extending the search space with those is not completely trivial, as we have showed in Section 5.4.1: the application rule is not finitely branching. Certainly it would be possible to set up a search space based on the LJT calculus (Example 5.4.3), but we will leave this for future research.

Instead, we adjusted the benchmark. We extended the core language with macros: `MVar`, `MLam` and `MApp`, that provide textual substitution. Core language macros allow us to approximate a higher-order meta-language. The Figure 6.2a shows the new extended syntax of the source and core languages. The extension for the source language is not changed from the original formulation of the benchmark, but the extension of the core language is. Figure 6.2b contain the expected translation rules, in this case expressed with core language macros, where $\%u$ is a freshly chosen name that is guaranteed to be unique and not used elsewhere in the term.

Setup

Syntax and intended translation The syntax of the extended source and core languages is shown in Figure 3.2a, and the intended translation from the Pidgin source to the Pidgin core language is shown in Figure 3.2b.

Sort mapping The benchmark extended the source language with a new sort for qualifiers in the list comprehension expression. As we do not have function types, we map qualifiers to core terms, expecting a macro expression defined with `MLam`, representing a function on core terms implemented with substitution.

Σ_0 sublanguage and its translation As this benchmark is the continuation of the Pidgin benchmark, the initial language Σ_0 in this case is simply the whole Pidgin source language, and its translation is the intended translation shown in Figure 3.1.

Initial search space. Our search space $\mathcal{H}_{\text{lc}}$ is similar to $\mathcal{H}^1_{\text{Pidgin}}$.

- The set of base rules $\mathcal{B}$ contained all core term constructors from Pidgin, including list constructors for term and identifier lists, the new constructors `MLam`, `MVar`, `MApp` for macros, and the new `Return` and `Bind` operations based on the definition of monads. It did not contain number, string or identifier constants, but contained the Boolean constants TRUE and FALSE, and all operators as constants.
- The are no translations $\mathcal{T}$ or case analysis rules $\mathcal{A}$.
- In addition to $\text{FRESH}_{\mathit{CLet}}$ we added a new binding construct in $\mathcal{V}$ for macros, called the $\text{FRESH}_{\mathit{MLam}}$:

$$\frac{\Gamma; i : \texttt{Id} \vdash M : \mathit{CTerm}}{\Gamma \vdash \textbf{let}\ i = \texttt{gensym}()\ \textbf{in}\ \texttt{MLam}(i, M) : \mathit{CTerm}}\ \textsc{Meta-lambda}$$

Results

Decomposition This benchmark demonstrates that sometimes "large" (>2) constructor groups are necessary, and our enumerative method does not scale up to large groups.

The smallest constructor group required for any test program is `SListComp` and `QEmpty`, so they need to be included in the first step. But any list comprehension test program that does not use the other qualifiers simply reduces to `Return`, therefore we can not learn the intended semantics of this group without the others. Adding the

Table 6.2: Benchmark – List comprehensions

Constructor group	AST size	Index	#tests	time
QEmpty	3	6	1	0s
QBind	10	97,632,734	1	25min 27s
QLet	7	116,747	1	2s
QGuard	6	31,307	1	1s
total			4	25min 30s

simplest qualifier, QGuard, is still not enough: without bound variables, we cannot write test programs that exclude similarly erroneous desugarings. Setting up the first step of the sequential learning task with just SListComp and QEmpty or possibly even adding QGuard would commit to a wrong translation rule early, and would not be able to continue the learning process. This demonstrates the "stuck" state discussed in Section 4.9.

The first step needs to contain at least three source constructors, and one of them should be QBind or QLet. But this group is too large, with a combined AST size of 14, causing our search to time out.

As a second test, we used a hint from the user: we provided the semantics (desugaring rule) of SListComp. This allowed for learning the semantics of the qualifiers one-by-one.

Results The results we obtained after the user hint are presented in Table 6.2.

Translation rules found We found the intended translation rules in all but one case, QGuard, where the semantics found was equivalent to the intended one, but shorter:

$$[\![\texttt{QGuard}(t,q)]\!] \quad = \quad \texttt{CIf}([\![t]\!],[\![q]\!],\textbf{let}\ \_ = \mathsf{gensym}()\ \textbf{in}\ \texttt{MLam}(\_,\texttt{CList}([])))$$

Test programs We only needed a single test program for each individual learning task: one test program suffices per source constructor. These test programs are non-trivial: we needed to embed them, in order to demonstrate the non-trivial scoping behaviour of qualifiers. To hypothetically auto-generate such test programs, the binding and scoping rules of the source language must be taken into account.

6.2.4 Try/Catch/Finally

Setup

Initial search space To express the intended desugaring, we extended the set of binding constructs with `CTryCatch`. The new FRESH$_{CTry}$ rule is the following:

$$\frac{\Gamma \vdash M : \mathit{CTerm} \quad \Gamma; x : \mathit{CTerm} \vdash N : \mathit{CTerm}}{\Gamma \vdash \mathbf{let}\, i = \mathsf{gensym}()\, \mathbf{in}\, \texttt{CTryCatch}(M, i, N[\texttt{Var}(i)/x]) : \mathit{CTerm}}\ \textsc{Fresh-trycatch}$$

Our initial search space $\mathcal{H}^1_{\text{tcf}}$ was otherwise identical with $\mathcal{H}^1_{\text{Pidgin}}$.

Results

The Try/Catch/Finally benchmark marks the limits of our current, enumerative approach.

Additional user guidance We found that the size of the intended desugaring of `STryCatchFinally`, with an AST size of 13, is too large for the general search space. We therefore created a second, severely restricted search space $\mathcal{H}^2_{\text{tcf}}$.

- The set of base rules $\mathcal{B}$ are restricted to two core term constructors, the new `CThrow` and `CTryCatch`.
- The are no translations $\mathcal{T}$ or case analysis rules $\mathcal{A}$.
- We kept the set of $\mathcal{V}$ binding constructs: the FRESH$_{\texttt{CLET}}$ and new the FRESH$_{\texttt{CTRY}}$ binding rules.

Decomposition The relabelling of `SThrow` can easily be learned on its own, because it has a unique behaviour. This allows us to learn the two new term constructors in two steps.

Results

The desugaring rule for `STryCatchFinally` is very large (even without thunking), and we reached the timeout even with the very restricted search space $\mathcal{H}^{tcf}$. To see how much we missed the mark, we ran the search to completion. The results are shown in Table 6.3.

Translation rules found The translation rule synthesised for SThrow is identical with the intended one: this is a relabelling.

The synthesised translation rule for STryCatchFinally does not use thunking. One reason is that we did not even included the core constructs necessary for thunking (like CLam). But even if we were included it, the algorithm would not had found the thunked version. As we have already explained in the benchmark (Section 3.2.2), we have no effect in the core language that would differentiate between the two: they are observationally equivalent, but the non-thunked version is smaller.

The translation rule found is otherwise identical with the intended one:

$$
\begin{aligned}
&[\![\texttt{STryCatchFinally}(t_1, i, t_2, t_3)]\!] = \\
&\quad \textbf{let } v = \texttt{gensym}() \textbf{ in } \texttt{CLet}\Big(v, \\
&\qquad \textbf{let } e = \texttt{gensym}() \textbf{ in } \texttt{CTryCatch}\big(\texttt{CTryCatch}([\![t_1]\!], i, [\![t_2]\!]), e, \\
&\qquad\quad \textbf{let } \_ = \texttt{gensym}() \textbf{ in } \texttt{CLet}(\_, [\![t_3]\!], \texttt{CThrow}(\texttt{CVar}(e)))\big), \\
&\qquad \textbf{let } \_ = \texttt{gensym}() \textbf{ in } \texttt{CLet}(\_, [\![t_3]\!], \texttt{CVar}(v))\Big) \\
&[\![\texttt{CThrow}(t)]\!] = \texttt{CThrow}([\![t]\!])
\end{aligned}
$$

Test programs Learning the desugaring rule of STryCatchFinally needs many complex test programs. First, try-catch-finally can branch: the main block may or may not throw, and the catch block also may or may not throw. This means that we already need 3 test programs to cover all cases. We also need to ensure the evaluation order. In general, branching constructs needs at least as many test programs as potential paths of execution. To fix the evaluation order, we need examples that exclude all other permutations. The number of potential evaluation orders grows with the number of parameters: STryCatchFinally has 4 parameters, the most among all of our term constructors.

Table 6.3: Benchmark – Try-catch-finally

Constructor group	AST size	Index	#tests	time
SThrow	2	18	1	0s
STryCatchFinally	13	774,161,305	8	2h 5m 49s
total			9	2h 5m 49s

6.2.5 Human overhead

This Chapter has presented the first solution to KLE's challenge, which can hopefully serve as the baseline for future research. We must highlight that it is not ready for practical use due to the significant human overhead of the synthesis process.

This overhead can be divided into two categories. First, the user must set up the synthesis process, providing the decomposition of the language, the initial translation, the search space parameters, and the test set. Second, we may frequently need to deal with an unsuccessful synthesis, where the process times out or returns a rule that is not the intended one. In these cases more input is needed, either in the form of additional user guidance like new test cases or changed search space. In some cases there could be a need to backtrack to earlier synthesis steps and correct rules that turned out to be erroneous.

I do not view this as a fatal flaw of the approach. Rather, automating or semi-automating many of the current human overhead is the direction future research should aim in, therefore there is value in highlighting these tasks.

Chapter 7

A linear type system for the efficient enumeration of translation rules

In the desugaring extension problem, our task is to synthesise translation rules. A natural heuristic restriction on the search space for translation rules is to require that each meta-variable is used exactly once: all intended translation rules in Chapter 3 are such, and using a meta-variable multiple times would lead to code bloat. We will call variables that are enforced to be used exactly once *linear* (following the terminology of logic and type theory), and those that can be used an arbitrary number of times *unrestricted*. We will call a search space linear if all variables are linear, and unrestricted if all variables are unrestricted.

In this chapter we justify the usage of linear variables via a comparison of linear and unrestricted search spaces: we argue that for most target languages, the linear search space grows much slower than an unrestricted search space. We introduce a type system for efficiently enumerating terms with linear variables, to enforce the heuristics. We empirically evaluate the type system on the Pidgin benchmarks and the list comprehension benchmarks, demonstrating that using linear types to restrict the search space can lead to around 15x faster synthesis for our benchmarks.

7.1 Background

When Girard introduced linear logic as a refinement of classical and intuitionistic logic [Girard, 1987], he anticipated its widespread applications in computer science. Linear types are frequently used in many areas, from concurrency to artificial intelligence, and are most commonly associated with resource control (see for example

Pierce [2004]).

Linear logic has since been classified as a kind of substructural logic. In classical (and intuitionistic) logic, formulas can be freely used. This property is most pronounced in the sequent calculus presentation of classical (or intuitionistic) logic, as there are specific inference rules ensuring the free usage of formulas: the structural rules. If we have two copies of a formula in our condition, they do not accomplish anything more than one copy, so one of them can safely be omitted (CONTRACTION rule). Any formula added to the condition will not change the validity of an entailment (WEAKENING rule). There are other structural rules like the EXCHANGE rule, which allows us to freely permute the order of the conditions, but these are not changed in linear logic, so we will not be discussing them. The two rules associated with the free usage of formulas are the following:

$$\frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\ \textsc{Contraction} \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma, A, B \vdash \Delta}\ \textsc{Weakening}$$

Linear logic is dubbed as the logic of resources, because it omits these two rules. This means that every formula in the condition of the entailment needs to be used exactly once when deriving the conclusion. Similarly, in a linear type system, where type derivation rules are based on the inference rules of linear logic, variables must be used exactly once in every execution of the program. Note that by this we do not mean that every formula needs to appear exactly once syntactically in the proof, or that every variable needs to appear exactly once in the program! If case analysis is used in the proof or program, we need to use every formula or variable in every branch.

Linear logic changed a lot in comparison to classical logic, introducing new logical connectives and modalities. But we do not need its complex apparatus, as our type system is based on a multi-sorted system for terms with variables, without logical connectives, as we do not have product or sum types. (Note that we relied on product and sum types in the meta-language to represent "copies" of sorts of the source language that has no corresponding sort in the core language, see Section 4.4. But we do not allow products or sums of arbitrary types.) We change the basic (structural) rules, inspired by linear type systems, to keep track of the usage of meta-variables in the terms.

7.2 Motivation

In language translation rules we would naturally like to avoid any double usage of the meta-variables, as this could lead to code bloat. In our model of compositional translations, the meta-variables in the translation rules stand for the already translated sub-programs. These sub-programs themselves were translated by the structurally recursive translation, and may have used the same translation rule. If we use a meta-variable twice, it could lead to code bloat: the output (the translated code) can become exponentially larger than the input (the source code).

Example 7.2.1 (Code bloat)**.** *As a simple example let us consider a desugaring of a square function in arithmetic expressions. Let us have a source and core language of arithmetic, where the source language has some additional functions. Let x be a meta-variable ranging over source terms, and let $\mathtt{SQUARE}$ be a one-argument function of the source language that we wish to desugar. Consider the following desugaring rule:*

$$[\![\mathtt{SQUARE}(x)]\!] = [\![x]\!] * [\![x]\!]$$

While this simple translation seems natural, it has a serious drawback. The meta-variable x does not stand for a number: it is a source expression. If the term denoted by x contains an expression that uses the $\mathtt{SQUARE}$ construct itself, the translation could lead to an exponential growth of core expressions. For example, chaining the $\mathtt{SQUARE}$ function twice results in its argument inserted four times into the core expression:

$$[\![\mathtt{SQUARE}(\mathtt{SQUARE}(x))]\!] = ([\![x]\!] * [\![x]\!]) * ([\![x]\!] * [\![x]\!])$$

We also expect to use each meta-variable at least once: we expect comments and other source elements that the semantics do not depend on to be removed from the source abstract syntax tree (AST) at an earlier step.

Nevertheless, we cannot totally exclude cases where either duplication or omission is necessary. When a translation rule needs to use a meta-variable multiple times, a common technique to avoid code bloat is thunking. Thunking in general refers to explicitly delaying the evaluation of an expression by wrapping it into an anonymous function with zero arguments (or a single dummy argument), with an operation called "force" triggering the evaluation. In the context of translations, we use thunking to save the expression denoted by the meta-variable into a target expression without evaluating it, and bind this target expression to a target variable. The *force* operation evaluates this

variable later, where the original expression would appear, to avoid code bloat: even if it appears multiple times, we only refer to the target variable, not the whole expression. This technique was used in the intended translation of `STryCatchFinally`, as shown in Section 3.2.2, to avoid generating the finally block in two places.

Thunking is only available if the target language has facilities for the type of the meta-variable. For example, if the type of the meta-variable is a program expression, we may be able to wrap it into an anonymous function and evaluate it later, but if the type of the meta-variable is e.g. a list of parameter names or a variable declaration, we most likely cannot do this. Thunking is only available if the target language has appropriate thunk/force operations.

Another possibility to gain back expressiveness is to add structural rules for the duplication or omission of a (linear) meta-variable. While this allows for the unrestricted usage of linear variables, the resulting system still keeps track of meta-variable use.

Thunking only depends on the target language, so it does not directly affect the setup of a linear search space. Depending on the presence of structural rules, we can distinguish between two kinds of desugaring extension problems utilising linear typing rules. The first kind is a strict restriction, where we do not have structural rules to duplicate/omit linear variables, and so the search space may not contain useful intended translation rules. The second kind does have these structural rules, so it covers the same (semantic) set of translation rules as the unrestricted search space, but this set is ordered differently. Any terms violating linear usage are larger in this linear search space than in the unrestricted search space, as they require additional structural rules to derive. In the present chapter, we will focus solely on the first problem, and will not assume structural rules to ease the strict enforcing of linear usage of the meta-variables.

7.3 The effect of pruning non-linear terms

Let us assume that for a given problem, the intended rule is linear. In this case, the efficiency of the usage of linear meta-variables compared to unrestricted meta-variables relies on two properties. First, the asymptotic growth of the linear search space compared to the unrestricted one: how many terms we need to test for a given size. Second, how efficiently can we enumerate the linear rules, as compared to enumerating the unrestricted rules: in other words, the overhead of enforcing the heuristics.

In the present section, we obtain an estimate of the first property: the effect of linear usage on the asymptotic growth of the search space. Comparing the growth of

the search spaces in general is not easy, because the number of translation rules of a given size (for both linear and unrestricted terms) depends on the target language in a non-trivial way. We simplify the problem by focusing on the restricted search space $\mathcal{H}_{\text{subst}}$, substitutions, which contains terms with variables: we compare the asymptotic growth of terms with linear variables and terms with unrestricted variables. Moreover, we only calculate exactly the asymptotic growth for a concrete example, relying on informal arguments for generalising the result to other target languages. The benchmarks in Section 7.5 will give further empirical evidence for the efficiency of linear search spaces in some of our benchmarks.

Example 7.3.1. *Let the target language have only one sort σ, a binary term constructor $f : \sigma \times \sigma \to \sigma$, and a sole constant $c : \sigma$.*

In this simple case, the terms with variables are binary trees, with the leaves labelled with the variables and the constant. A binary tree with n leaves has $2n-1$ nodes, which is the size of our term. Thus, there is a simple linear relationship between the number of leaves and the size of the term, but it is easier to base our calculations on the number of leaves.

The number of differently shaped trees (with a given number of leaves) is given by the Catalan numbers: there are C_n different binary trees with $n+1$ leaves, where $C_n = \frac{1}{n+1}\binom{2n}{n}$ is the nth Catalan number.

If we have k linear variables, then for a given shape with n leaves we can have $\binom{n}{k}k! = \frac{n!}{(n-k)!} \approx n^k$ different terms. If the variables are unrestricted, we get $(k+1)^n$ terms. Thus, changing the variables from unrestricted to linear changes the asymptotic growth function from exponential (regarding the size of the terms) to polynomial. (Regarding the number of variables, the position is switched: the unrestricted terms grow polynomially, with the linear terms growing exponentially. But the number of meta-variables for a given problem is fixed and usually small, and we are interested in the dependence on the size of the term.) Note that the growth of the search space regarding the size of the terms is still roughly exponential in both cases, since Catalan numbers grow approximately exponentially.

As an example, the number of terms with a fixed shape for a context with 3 variables is shown in Table 7.1. In order to get the number of terms for the given size, these numbers should be multiplied by the number of shapes, i.e. C_{n-1}, but this multiplier is the same for both linear and unrestricted terms, and thus does not change the ratio.

While the actual numbers for a more complex target language could be quite dif-

size	**1**	**3**	**5**	**7**	**9**	**11**	**13**	**15**	**17**	**19**
#linear terms	0	0	6	24	60	120	210	336	504	720
#unrestricted terms	4	16	64	256	1,024	4,096	16,384	65,536	262,144	1,048,576

Table 7.1: The number of linear and unrestricted terms for one binary term constructor, one constant, and 3 variables in the context, for a fixed shape

ferent (especially if it has more constants), this simple example shows that the gain can be enormous.

The more exact calculations of the concrete example can be informally generalised in the following way. For a given target language, we can count the number of terms with (linear or unrestricted) variables by multiplying 1) the number of "shapes" b, terms (labelled trees) without leaves filled, and 2) the possible way we can assign variables and constants as leaves. For linear variables, we must choose one leaf for every variable in the context, which is a factor that grows polynomially with the number of leaves to choose from. Unrestricted variables act the same as constants, increasing the base of the exponential factor of the potential ways we can fill up the leaves with constants. While the asymptotic growth of both search spaces is generally exponential, changing a factor from exponential to polynomial still can result in a huge gain.

Linear variables may also have other effects on the search space. If the target language lacks constants for a sort σ, then only those shapes are allowed in a linear search space that have exactly the same number of leaves with sort σ as the number of linear variables with sort σ in the context. In other words, a linear search space may exclude certain shapes if there are no constants in the language to fill up the corresponding leaves, so for certain target languages and certain contexts of linear variables, the gain could be much larger.

So a linear search space can be very efficient. But there is a trade-off: efficiently enumerating only the linear terms is harder than enumerating the unrestricted terms. A generate-and-test approach is not feasible for such large scale pruning. In the following sections, we introduce a linear type system that enables us to efficiently enumerate the linear typed translation rules.

$$\frac{}{x:\sigma \vdash x:\sigma}\ \textsc{Axiom} \qquad \frac{c:\sigma}{\emptyset \vdash c:\sigma}\ \textsc{C-rule}$$

$$\frac{\Gamma_1 \vdash N_1:\sigma_1 \quad \ldots \quad \Gamma_n \vdash N_n:\sigma_n \quad f:\sigma_1 \times \cdots \times \sigma_n \to \sigma \quad \Gamma = \sqcup_{k=1}^{n} \Gamma_k}{\Gamma \vdash f(N_1,\ldots,N_n):\sigma}\ \textsc{F-rule}$$

Figure 7.1: Terms with linear variables as proofs – top-down version

7.4 A linear search space

We transform the search spaces defined in Chapter 6 into linear ones, following the incremental way search spaces were presented there.

7.4.1 Type system for terms with linear variables

We begin with the search space $\mathcal{H}_{\text{subst}}$, terms with (unrestricted) variables. The contribution of the present section is a type system that facilitates efficient search for its linear counterpart, $\mathcal{H}^{lin}_{\text{subst}}$, terms with linear variables.

Simply enforcing linear usage is quite standard, but we have another requirement for our typing rules: we want an efficient enumeration. To show why this is difficult, we introduce two standard versions of a linear type system for terms with variables, demonstrating that they are not efficient for enumerating linear terms.

We call the first version top-down, because the proof generates the term starting from the topmost term constructor. The top-down linear calculus rules are shown in Figure 7.1. They are very similar to the classic rules, but we need to split the context Γ in the F-RULE. While the new F-RULE is still finitely branching, the number of branches can be very high. This step is not easy to implement efficiently in a backward chaining search: we need to create a choice with every possible split, and the number of splits grows exponentially with the size of the context. This results in a highly branching search space for terms.

One standard idea to avoid this context splitting is to build the terms from the bottom, starting from the smaller terms. The bottom-up version of the rules for the linear terms are shown in Figure 7.2. These rules generate the same set of terms (with the help of a meta-level *let* construct), but all rules have at most one premise.

But these rules also have a severe problem with regard to searching: we lost the Curry-Howard property. Various applications of the F-rule can be permuted, resulting in a blow-up of the search space. To demonstrate the problem, let us see the following

$$\frac{}{x:\sigma \vdash x:\sigma}\ \textsc{Axiom} \qquad \frac{c:\sigma}{\emptyset \vdash c:\sigma}\ \textsc{C-rule}$$

$$\frac{\Gamma, x:\sigma \vdash N:\sigma' \quad f:\sigma_1 \times \cdots \times \sigma_n \to \sigma}{\Gamma, x_1:\sigma_1, \ldots, x_n:\sigma_n \vdash \mathbf{let}\, x = f(x_1,\ldots,x_n)\, \mathbf{in}\, N:\sigma'}\ \textsc{F-rule}$$

Figure 7.2: Terms with linear variables as proofs – bottom-up version

example.

Example 7.4.1. *Let us have three sorts $S = \{\sigma_1, \sigma_2, \sigma_3\}$. Let us have three term constructors: $F = \{f : \sigma_1 \times \sigma_2 \to \sigma_3, g : \sigma_1 \to \sigma_1, h : \sigma_2 \to \sigma_2\}$. And let us have two variables in the context: $x : \sigma_1$ and $y : \sigma_2$.*

With the bottom-up linear rules, the term $f(g(x), h(y))$ can be derived in two ways:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{z:\sigma_3 \vdash z:\sigma_3}\ \textsc{Axiom}}{x':\sigma_1, y':\sigma_2 \vdash \mathbf{let}\, z = f(x',y')\, \mathbf{in}\, z:\sigma_3}\ \textsc{F-rule (f)}}{x':\sigma_1, y:\sigma_2 \vdash \mathbf{let}\, y' = h(y)\, \mathbf{in}\, \mathbf{let}\, z = f(x',y')\, \mathbf{in}\, z:\sigma_3}\ \textsc{F-rule (h)}}{x:\sigma_1, y:\sigma_2 \vdash \mathbf{let}\, x' = g(x)\, \mathbf{in}\, \mathbf{let}\, y' = h(y)\, \mathbf{in}\, \mathbf{let}\, z = f(x',y')\, \mathbf{in}\, z:\sigma_3}\ \textsc{F-rule (g)}$$

and

$$\dfrac{\dfrac{\dfrac{\dfrac{}{z:\sigma_3 \vdash z:\sigma_3}\ \textsc{Axiom}}{x':\sigma_1, y':\sigma_2 \vdash \mathbf{let}\, z = f(x',y')\, \mathbf{in}\, z:\sigma_3}\ \textsc{F-rule (f)}}{x:\sigma_1, y':\sigma_2 \vdash \mathbf{let}\, x' = g(x)\, \mathbf{in}\, \mathbf{let}\, z = f(x',y')\, \mathbf{in}\, z:\sigma_3}\ \textsc{F-rule (g)}}{x:\sigma_1, y:\sigma_2 \vdash \mathbf{let}\, y' = h(y)\, \mathbf{in}\, \mathbf{let}\, x' = g(x)\, \mathbf{in}\, \mathbf{let}\, z = f(x',y')\, \mathbf{in}\, z:\sigma_3}\ \textsc{F-rule (h)}$$

This example is very similar to Example 5.4.2.

In other words, we can permute applications of the F-RULE, with many different type derivations resulting in the same term. This is a serious inefficiency, which prevents us from enumerating linear terms effectively with this type system.

To make the search feasible, we use a third style of linear rules, combining the benefits of the two styles. We will not split the linear context, using only one premise for the F-RULE. But we will still build the term top-down. To achieve this, we share the linear context amongst multiple terms: instead of multiple premises in the proof rule, we will use multiple outputs (term-type pairs on the right side of the turnstile) in

$$\frac{}{\emptyset \vdash \emptyset}\ \textsc{Axiom}$$

$$\frac{\Gamma \vdash \Delta \quad \Delta = \emptyset \Rightarrow \Gamma = \emptyset}{\Gamma, x : \sigma \vdash x : \sigma, \Delta}\ \textsc{Var-lin}$$

$$\frac{\Gamma \vdash \Delta \quad \Delta = \emptyset \Rightarrow \Gamma = \emptyset \quad c : \sigma}{\Gamma \vdash c : \sigma, \Delta}\ \textsc{C-rule}$$

$$\frac{\Gamma \vdash \mathbf{sort}\big(N_1 : \sigma_1, \ldots, N_n : \sigma_n, \Delta\big) \quad f : \sigma_1 \times \cdots \times \sigma_n \to \sigma}{\Gamma \vdash f(N_1, \ldots, N_n) : \sigma, \Delta}\ \textsc{F-rule}$$

Figure 7.3: Terms with linear variables as proofs – multiple outputs with focusing

our judgements. To avoid permutations, however, we only work on one of the outputs at a time. To achieve this, we use ordered outputs. Note that this makes the two sides of the turnstile different: the linear context on the left side is unordered, while the output terms in the right side is ordered.

We need to be careful with introducing order: the order may distinguish otherwise equivalent judgements, which, as we have highlighted in Section 5.4.2, violates the adequacy requirement. We need to fix the order, based on the types of the outputs. The outputs on the right side of the turnstile are always ordered according to a total ordering on their types. We achieve this by the **sort** function, which sorts the outputs based on the types. We only need to call **sort** in the F-RULE, since that's the only rule which introduces new outputs.

The new rules are shown in Figure 7.3.

With our last version, terms are easily enumerable: we have exactly one derivation for every linear term, and we can directly calculate the context of the premise. This is the version that serves as the basis for extended linear search spaces, we will call it $\mathcal{H}^{\text{lin}}_{\text{subst}}$, or *linear substitutions*.

7.4.2 The CASE, ERROR and TRANSFORM rules

Extending $\mathcal{H}^{\text{lin}}_{\text{subst}}$ with the CASE, ERROR and TRANSFORM rules is fairly straightforward. These rules remain similar to their unrestricted versions. Due to our layering, they belong to a different kind of judgement, and this layer contains no F-RULE. Therefore, for this layer (the extension judgement with the notation $\vdash_E$), we do not need multiple outputs. This means that the rules can stay exactly the same.

We will now give a brief overview of these rules in a linear context, and we will be

evaluating the extended search spaces empirically on some selected benchmarks later.

The CASE rule is the main reason why keeping track of linear variables is more complicated than simply counting their number of occurrences: we want to use linear variables exactly once for any input, and in the CASE rule, only one branch is executed for a given input. Therefore, we want to use every linear variable exactly once in every branch. Fortunately, this requirement is very easy to add, as it means that the linear and the unrestricted versions of the CASE rule are essentially identical:

$$\frac{\Gamma, x_1^1 : \sigma_1^1, \ldots, x_1^{k_1} : \sigma_1^{k_1} \vdash_E M_1 : \sigma' \quad \ldots \quad \Gamma, x_n^1 : \sigma_n^1, \ldots, x_n^{k_n} : \sigma_n^{k_n} \vdash_E M_n : \sigma'}{\Gamma, y : \sigma \vdash_E \textbf{case}\, y\, \textbf{of}\, f_1(x_1^1, \ldots, x_1^{k_1}) \to M_1 \ldots f_n(x_n^1, \ldots, x_n^{k_n}) \to M_n : \sigma'}\ \textsc{Case}$$

The TRANSFORM rule relies on user-defined transformations. We have already used these transformations in a linear manner: the transformed argument was removed from the context. This can be seen as a heuristic in the unrestricted search space, as defined: if we re-arrange the arguments, we do not need the original arrangement anymore. We also expect the user-defined transformations to not duplicate (or omit) parts of the arguments, but this is not required or enforced: whether the user-defined transformation function is linear is up to the user. We also derived a combined version of the TRANSFORM and CASE rules, called TRANSFORM-CASE; this, too, can remain unchanged.

$$\frac{\Gamma, x : \sigma_2 \vdash M : \sigma}{\Gamma, y : \sigma_1 \vdash \textbf{let}\, x = \mathtt{tr}(y)\, \textbf{in}\, M : \sigma}\ \textsc{Transform}$$

$$\frac{\Gamma, x_1^1 : \sigma_1^1, \ldots, x_1^{k_1} : \sigma_1^{k_1} \vdash M_1 : \sigma' \quad \ldots \quad \Gamma, x_n^1 : \sigma_n^1, \ldots, x_n^{k_n} : \sigma_n^{k_n} \vdash M_n : \sigma'}{\Gamma, y : \sigma_1 \vdash \textbf{case}\, tr(y)\, \textbf{of}\, f_1(x_1^1, \ldots, x_1^{k_1}) \to M_1 \ldots f_n(x_n^1, \ldots, x_n^{k_n}) \to M_n : \sigma'}\ \textsc{Transform-Case}$$

The ERROR rule is also straightforward. In the case of syntax error we do not want the (linear) variables to be used at all, which is the same behaviour as in the unrestricted version.

$$\frac{}{\Gamma \vdash \textbf{syntax\_error} : \sigma}\ \textsc{Throw}$$

7.4.3 Bound linear variables

The FRESH meta-rule and its variants introduce new variables into the context for modelling binding constructs in the target language. These variables do not represent

an argument of the translation rule: they do not stand for already translated parts of the whole program, but instead are bound target language variables with a restricted scope. We used meta-variables to represent local target language variables.

In general, we do not assume target language variables to be linear. To model bound target language variables as unrestricted, we need two separate environments: one for linear and one for unrestricted variables.

The problem with the FRESH meta-rule is that the unrestricted context behaves very differently from the linear context. While the linear context is shared between all outputs, the unrestricted context is not shared, since some variables are only available in certain output terms (they have a scope).

We change our judgements once again. They will be of the form:

$$\Gamma \vdash (\Theta_1 \vdash_u M_1 : \sigma_1); \ldots; (\Theta_n \vdash_u M_n : \sigma_n)$$

Here Γ is a context of linear variables, shared between all output terms $M_1, \ldots, M_n$. Every output term has a separate context of unrestricted variables: Θ_i, which is written in front of the term separated by $\vdash_u$. The additional unrestricted contexts do not significantly change the already introduced rules: the unrestricted context is simply copied to all introduced outputs in the premise of the F-RULE. The arguments of a term constructor have the same unrestricted context.

With a new context, we need a new rule to introduce variables. The rule of unrestricted variables is analogous to using constants, just the variable needs to be present in the unrestricted context:

$$\frac{\Gamma \vdash \Delta \quad \Delta = \emptyset \Rightarrow \Gamma = \emptyset}{\Gamma \vdash (\Theta, x : \sigma \vdash_u x : \sigma), \Delta} \textsc{Var-unr}$$

With the double contexts, we can introduce the variants of the FRESH rule. We show an example for the `CLet` target binding construct in the Core Pidgin language:

$$\frac{\Gamma \vdash \mathbf{sort}\big((\Theta \vdash_{int} t_1 : \mathit{CTerm}), (\Theta, x : \mathit{CTerm} \vdash_{int} t_2 : \mathit{CTerm}), \Delta\big)}{\Gamma \vdash (\Theta \vdash_{int} \mathbf{let}\, i = \texttt{gensym}()\, \mathbf{in}\, \texttt{CLet}(i, t_1, t_2[\texttt{CVar}(i)/x]) : \mathit{CTerm}), \Delta} \textsc{Fresh}$$

The rule is similar to its unrestricted counterpart, as the introduced variables are unrestricted: instead of multiple premises, we introduce new outputs, similarly to the F-RULE. However, the unrestricted contexts are not the same for the FRESH rule case.

Note that the FRESH rule is in the basic layer, meaning that it has multiple outputs, and it does introduce new outputs. Therefore, we need to call the **sort** function as well to ensure adequacy of the representation. When we extended the type of the outputs (including unrestricted context for all output), we also changed the order: the sort function takes into account the unrestricted contexts as well as the types (but not the terms) in the output.

7.4.4 Summary - a linear search space for desugarings

Table 7.4 gives a summary of all introduced rules forming the linear search space $\mathcal{H}^{\text{lin}}_{\text{meta}}$, the linear counterpart of $\mathcal{H}_{\text{meta}}$.

The final search space has some unique characteristic dictated by our task at hand. It is common to have two contexts, a linear and an unrestricted one. But we treat them completely differently. The linear context is shared between multiple terms, as the condition that every variable is used exactly once applies to all of them. The unrestricted context, however, allows introducing new variables with a scope. This means that we need to split them to different contexts when a term is being split into subterms.

It is an interesting research question whether it is possible to combine the two: that is, whether there is a system that:

1. Allows constraints on variable usage, like allowing linear variables that can only be used once.

2. Allows introducing variables with a scope (e.g. linear λ terms).

3. Efficiently enumerable: do not use context splitting and has the Curry-Howard property.

I do not know the answer for this, and leave it to future research. Fortunately, we do not need to introduce linear variables with a scope, as we did not assumed the core language to be linear, merely the meta-language, where we do not introduce new variables. The system presented here suffices for our use case.

$$\frac{}{\emptyset \nmid\!\!\vdash \emptyset}\ \textsc{Axiom} \qquad \frac{\Gamma \nmid\!\!\vdash \Delta \quad \Delta = \emptyset \Rightarrow \Gamma = \emptyset}{\Gamma, x:\sigma \nmid\!\!\vdash (\Theta \vdash_u x:\sigma), \Delta}\ \textsc{Var-Lin}$$

$$\frac{\Gamma \nmid\!\!\vdash \Delta \quad \Delta = \emptyset \Rightarrow \Gamma = \emptyset}{\Gamma \nmid\!\!\vdash (\Theta, x:\sigma \vdash_u x:\sigma), \Delta}\ \textsc{Var-Unr} \qquad \frac{\Gamma \nmid\!\!\vdash \Delta \quad \Delta = \emptyset \Rightarrow \Gamma = \emptyset \quad c:\sigma}{\Gamma \nmid\!\!\vdash (\Theta \vdash_u c:\sigma), \Delta}\ \textsc{C-Rule}$$

$$\frac{\Gamma \nmid\!\!\vdash \mathbf{sort}\big((\Theta \vdash_u N_1:\sigma_1), \ldots, (\Theta \vdash_u N_n:\sigma_n), \Delta\big) \quad f:\sigma_1 \times \cdots \times \sigma_n \to \sigma}{\Gamma \nmid\!\!\vdash (\Theta \vdash_u f(N_1, \ldots, N_n):\sigma), \Delta}\ \textsc{F-Rule}$$

(a) Linear substitutions

~

$$\frac{\Gamma, x_1^1:\sigma_1^1, \ldots, x_1^{k_1}:\sigma_1^{k_1} \vdash_E M_1:\sigma' \quad \ldots \quad \Gamma, x_n^1:\sigma_n^1, \ldots, x_n^{k_n}:\sigma_n^{k_n} \vdash_E M_n:\sigma'}{\Gamma, y:\sigma \vdash_E \mathbf{case}\, y\, \mathbf{of}\, f_1(x_1^1, \ldots, x_1^{k_1}) \to M_1 \ldots f_n(x_n^1, \ldots, x_n^{k_n}) \to M_n : \sigma'}\ \textsc{Case}$$

$$\frac{\Gamma, x:\sigma_2 \vdash_E M:\sigma}{\Gamma, y:\sigma_1 \vdash_E \mathbf{let}\, x = \mathtt{tr}(y)\, \mathbf{in}\, M:\sigma}\ \textsc{Transform} \qquad \frac{\Gamma \nmid\!\!\vdash (\emptyset \vdash_u M:\sigma)}{\Gamma \vdash_E M:\sigma}\ \textsc{Extend}$$

$$\frac{\Gamma, x_1^1:\sigma_1^1, \ldots, x_1^{k_1}:\sigma_1^{k_1} \vdash_E M_1:\sigma' \quad \ldots \quad \Gamma, x_n^1:\sigma_n^1, \ldots, x_n^{k_n}:\sigma_n^{k_n} \vdash_E M_n:\sigma'}{\Gamma, y:\sigma_1 \vdash_E \mathbf{case}\, tr(y)\, \mathbf{of}\, f_1(x_1^1, \ldots, x_1^{k_1}) \to M_1 \ldots f_n(x_n^1, \ldots, x_n^{k_n}) \to M_n : \sigma'}\ \textsc{Transform-Case}$$

(b) Extended rules

~

$$\frac{\Gamma \nmid\!\!\vdash \mathbf{sort}\big((\Theta \vdash_u t_1 : \mathit{CTerm}), (\Theta, x:\mathit{CTerm} \vdash_u t_2 : \mathit{CTerm}), \Delta\big)}{\Gamma \nmid\!\!\vdash (\Theta \vdash_u \mathbf{let}\, i = \mathtt{gensym}()\, \mathbf{in}\, \mathtt{CLet}(i, t_1, t_2[\mathtt{CVar}(i)/x]) : \mathit{CTerm}), \Delta}\ \textsc{Fresh}$$

(c) An example of introducing fresh variables

Figure 7.4: A linear search space for desugarings

7.5 Empirical evaluation

7.5.1 Pidgin benchmark and list comprehensions benchmark

Our linear search space is parallel to the unrestricted one, with the same parameters, so we can directly compare the linear heuristic with the baseline (unrestricted) solution. I repeated all benchmarks of the Pidgin language and the list comprehension extension, but this time with a linear search space. All intended translation rules in these benchmarks are linear.

We display the result of the comparison for the Pidgin benchmark (Table 7.2) and

Table 7.2: Comparing linear and unrestricted search spaces – Pidgin languages

Constructor group	AST size	Index (unrestricted)	Index (linear)	Time (unrestricted)	Time (linear)
SNum	2	1	1	0s	0s
SStr	2	1	1	0s	0s
SPrim	12	16,567,096	1,073,578	2min 27s	33s
SVar, *SLet*	6	1,298	38	0s	0s
STrue, *SFalse*	4	10	10	0s	0s
SAssign	3	18	18	0s	0s
SBetween	14	157,160,392	3,618,042	28min 20s	1min 45s
SIf	4	171	1	0s	0s
SLam, *SApp*	6	1,813	60	0s	0s
SLetRec	4	132	3	0s	0s
SList	2	3	1	0s	0s
SListCase	4	207	7	0s	0s
SFor	11	57,334,664	24,428	9min 44s	4s
Full				40min 32s	2min 25s

for the list comprehension benchmark (Table 7.3). The tables show the index of the translation(s) found in the respective search spaces and the time of the search for both the unrestricted (original) and linear search spaces; the original results are displayed in columns with grey backgrounds. The corresponding benchmarks are identical in all other respects: the user-defined restrictions on the individual search spaces, the user provided examples, etc.. We omit some columns that do not differ between the unrestricted and linear search spaces, such as the size of the test set (see Table 6.1 for the original benchmarks for these). In all cases, the respective benchmarks always find the same semantic translation rules.

As we expected, using the linear search space greatly sped up the synthesis. The empirical evaluation shows $15-17$ times faster search times for the linear search space, and this multiplier can potentially grow even larger for larger translation rules.

7.5.2 Try/catch/finally benchmark

In the Try/Catch/Finally benchmark, the intended translation rule for *STryCatchFinally* is also linear: it uses thunking to evaluate the **finally** block twice without inserting it twice in the translation. We wrap the twice-used meta-variable in an anonymous func-

Table 7.3: Comparing linear and unrestricted search spaces – List comprehensions

Constructor group	AST size	Index (unrestricted)	Index (linear)	Time (unrestricted)	Time (linear)
QEmpty	3	6	6	0s	0s
QBind	10	97,632,734	3,853,408	25min 27s	1min 47s
QLet	7	116,747	2,553	2s	1s
QGuard	6	31,307	5,819	1s	0s
Full				25min 30s	1m 49s

tion to delay evaluation, saving it in a target variable. But this version has a very large AST size (20), which is outside of the scope of our search method even when using a linear search space; note that the size is increased not only through creating the thunk, but also by using it, as it requires the application of a function.

We can also set up different search spaces, in order to provide additional user guidance. We run two preliminary tests on the Try/Catch/Finally benchmark as well, to assess directions in which the linear search space can be extended. In the first, we used a fixed, user-provided high-level template. The template wrapped the meta-variable standing for the **finally** block in a thunk, thus effectively changing the last argument of the translation rule from linear to unrestricted. In the second test, we instead duplicated this meta-variable, enforcing that it be used exactly twice. For this second test, we needed to change the implementation slightly, changing the context of linear variables from sets to multisets, keeping track of the number of times each variable can be used.

Table 7.4 contains the comparison of the three search spaces:

1. the unrestricted search space, already presented in Chapter 6,
2. the search space which wraps the finally block in a thunk, turning this variable from linear to unrestricted, and
3. the search space which duplicated the variable for the finally block, forcing it to be used exactly twice.

We can see that both heuristics result in an impressive speed-up. The second test, relying on the number of times each variable is used, ran approximately 47 times faster. This result is encouraging, suggesting that the linear search space could be useful outside of a strictly linear context, where the user is allowed to specify some variables as

Table 7.4: Comparing search spaces for the Try/catch/finally benchmark

Search space	Index	Time
Unrestricted	774,161,305	2h 5m 49s
Linear with unrestricted finally block (thunked)	23,727,106	12m 31s
Linear with duplicated finally block	6,483,105	2m 40s

unrestricted, or specify their number of usage.

Chapter 8

Conclusion and future work

8.1 Summary

Developing correct semantics rules for real-world languages is a necessary but arduous prerequisite to formal analysis. Semi-automatically inferring the semantic rules of a programming language based on observing an implementation is a new research area. Krishnamurthi et al. [2019] posed the research challenge, and apart from my own Master's thesis [Bartha and Cheney, 2020]—which was based on an early case study—the present work is the first to address this challenge, to the best of my knowledge.

Having ascertained the high degree of difficulty of the problem, I anticipated gradual, piecemeal successes, and introduced a series of benchmarks to measure progress in Chapter 3. I have carefully analysed the problem in Chapter 2 and highlighted two key aspects: the decomposability of language feature learning into a sequence of easier desugaring extension problems, and the need to carefully select the search space and the meta-language in which the desugaring rules are expressed. Based on these ideas, I gave a formal definition of the problem in Chapter 4.

The design of a program synthesis algorithm can be divided into three aspects: the search space, the search technique and the user intent [Gulwani et al., 2017]. I deliberately focused on the first, and employed an enumerative synthesis algorithm, as the success of enumerative synthesis directly depends on the search space. While conceptually simple, the implementation of an enumeration algorithm with the purpose of comparing hypothesis spaces based on complex invariants is surprisingly challenging. I describe the technical background of an enumeration algorithm for typed program fragments based on proof enumeration in Chapter 5. I applied well-known algebraic techniques of defining enumerations to a new task: proof enumeration.

I developed and evaluated a baseline search space for the desugaring extension problem in Chapter 6.

A natural heuristic for translation rules is enforcing that every variable be used exactly once: i.e. using a meta-language with linear variables. In Chapter 7 I carefully designed a linear type system aimed at fast term enumeration, and demonstrated that a search space based on linear variables can increase the speed of synthesis by over an order of magnitude.

8.2 Weak points

As the research area is quite new, we cannot compare our results to previous works. We cannot yet evaluate our methods on real-world programming languages, because real desugarings are too large. Therefore, our evaluation relied on hand-crafted benchmarks, where we set up the goals ourselves. Thus, there is a danger of overestimating the results. Possible objections and potential weak points are the following:

- The benchmarks were fairly small, and enumerative synthesis does not scale well to larger examples.
- There were a limited number of benchmarks (since each required the implementation of two languages). The method may not generalise well.
- The benchmarks were hand-written. Pre-defined examples always carry the danger of inadvertently cherry-picking the results.
- Since I knew the intended desugarings in advance, extensive experimentation with user guidance, such as restrictions on the hypothesis space, was not required. Thus, there is a risk of inadvertently underestimating the amount of user guidance needed.

I do not regard these limitations as fatal flaws. Instead, they may illustrate that my approach aims to produce a “desugaring synthesis assistant”, rather than a fully automatic synthesis tool. I view the present research as a necessary step towards future algorithms that scale and generalise better.

8.3 Future work

The premise of the present work was an open-ended problem, and one of my contributions was to delineate a sub-task that could be solved. In order to achieve this, we deliberately ignored many aspects of the problem domain. These simplifications led us to new avenues of research.

Search technique We only explored enumerative synthesis. In other words, we focused on the definition of the search space, not the search technique. An obvious task would be to explore state-of-art search techniques. As program synthesis is a rapidly expanding area with new results appearing every year, summarising potential algorithms is beyond the scope of this work. Two examples of recent synthesis algorithms are Duet [Lee, 2021] which combines an enumerative algorithm with top-down propagation for inductive program synthesis, and Prose [Barke et al., 2020], which augments enumerative synthesis with a probabilistic model learnt during the synthesis from partial successes. Both promise much faster synthesis over diverse domains, and while they are not directly applicable to our unusual testing framework, the general ideas could be useful for the desugaring extension problem as well.

User intent The current benchmarks used hand-written Haskell code. One important future task would be to integrate the search methods to existing frameworks for semantic engineering. For this, a prerequisite is finding a good user interface: a language in which the user can easily express the user guidance needed for the search.

Automatic decomposition of the language An interesting task is whether the assumed decomposition of the language, which is a burden we placed on the user, could be automated or semi-automated.

In any new area of research, answering a question opens up a thousand new ones. While this thesis is but a single step toward the distant goal of learning the next 700 language semantics, our results provide grounds for optimism, and hopefully can be built upon.

Appendix A

Reference implementation of Pidgin

For concreteness, we describe our reference implementation of the Pidgin source language, core language, the core language interpreter, and the intended desugaring translation in a Haskell-like pseudocode. The source language behaviour is defined as the composition of the desugaring and core language interpreter (though in our implementation we give a direct definition). Our implementation shows the full details as runnable Haskell code.

```
newtype Id = Id String

data Op = UMin | Not | Plus | Minus | And | Or | LT | GT

data STerm = SVar String
           | SPrim Op [STerm]
           | SBetween STerm STerm STerm
           | SNum Int
           | SStr String
           | STrue
           | SFalse
           | SIf STerm STerm STerm
           | SLet Id STerm STerm
           | SLetRec Id STerm STerm
           | SLam [Id] STerm
           | SApp STerm [STerm]
           | SAssign [Id] STerm
           | SList [STerm]
           | SListCase STerm STerm STerm
           | SFor STerm [(Id,STerm)] STerm

data CTerm = CVar Id
           | CPrim1 Op CExp
           | CPrim2 Op CExp
           | CNum Int
           | CStr String
           | CBool Bool
           | CIf CTerm CTerm CTerm
           | CLet Id CTerm CTerm
           | CLetRec Id CTerm CTerm
           | CLam [Id] CTerm
           | CApp CTerm [CTerm]
           | CAssign Id CTerm
           | CList [CTerm]
           | CListCase CTerm CTerm CTerm

rename :: Id -> Id -> CTerm -> CTerm
```

Figure A.1: Source and target term languages

```
type Env = Map Id CTerm
type Result a = ... -- error and state monad
raise :: Error -> Result a
fresh :: Result Id
read :: Id -> Result CTerm
write :: Id -> CTerm -> Result ()
doUnop :: Op -> CTerm -> Result CTerm
doBinop :: Op -> CTerm -> CTerm -> Result CTerm
run : Result a -> Env -> Either a Error
```

Figure A.2: Helper functions for evaluation and desugaring

```
eval :: CTerm -> Either CTerm Error
eval t = run (eval' t) emptyEnv

eval' :: Map Id CTerm -> CTerm -> Result CTerm
eval' (CVar x)   = do
  e <- read x  -- might be an expression bound by letrec
  eval' e
eval' (CPrim1 op e) = do
  v <- eval' e
  case op of
    UMin -> doUnOp (negate @Int) v
    Not  -> doUnOp not v
    _    -> raise TypeError
eval' (CPrim2 op e1 e2) = do
  v1 <- eval' e1
  v2 <- eval' e2
  case op of
    Plus  ->
      case v1 of
        (CNum _) -> doBinOp ((+) @Int) v1 v2
        (CList l1) -> case v2 of
                        CList l2 -> return (CList (l1 ++ l2))
                        _ -> raise TypeError
        _ -> raise TypeError
    Minus -> doBinOp ((-) @Int) v1 v2
    And   -> doBinOp (&&) v1 v2
    Or    -> doBinOp (||) v1 v2
    LT    -> doBinOp ((<) @Int) v1 v2
    GT    -> doBinOp ((>) @Int) v1 v2
    -> raise TypeError
```

```
eval' (CIf ex1 ex2 ex3) = do
  c <- eval' ex1
  case c of
    CBool b  -> if b then eval' ex2 else eval' ex3
    _ -> raise TypeError
eval' (CLet i ex1 ex2) = do
  v <- eval' ex1  -- let is eager
  i2 <- fresh
  write i2 v
  eval' (rename i id ex2)
eval' (CLetRec i ex1 ex2) = do
  i2 <- fresh
  write i2 (rename i id ex1)  -- any recursive references will be expanded later
  eval' (rename i id ex2)
eval' ex@(CLam _ _) = return ex
eval' (CApp ex es) = do
  f <- eval' ex
  case f of
    CLam is body -> do
        vs <- mapM eval' es
        case length is == length vs of
          True -> do iterM write (zip is vs)
                     eval' body
          False -> case vs of
                     [ CLang' (CList vs')] ->
                        case length is == length vs' of
                          True -> do iterM write (zip is vs')
                                     eval' body
                          False -> raise ArgumentNumberMismatchError
                     _ -> raise ArgumentNumberMismatchError
    _ -> raise TypeError
eval' (CAssign x ex) = do
  v <- eval' ex
  write x v
  return v
eval' (CList es) = do
  vs <- mapM eval' es
  return (CList vs)
eval' (CListCase e1 e2 e3) = do
  l <- eval' e1
  case l of
```

```
ds :: STerm -> Either CTerm Error
ds t = run (ds' t) emptyEnv

ds' :: STerm -> Result STerm
ds' (SVar i) = return (CVar i)
ds' (SNum n) = return (CNum n)
ds' (SStr s) = return (CStr s)
ds' (SIf x y z) = do
     x' <- ds' x
     y' <- ds' y
     z' <- ds' z
     return (CIf x' y' z')
ds' (SLet i x y ) = do
     x' <- ds' x
     y' <- ds' y
     return (CLet i x' y')
ds' (SLetRec i x y ) = do
     x' <- ds' x
     y' <- ds' y
     return (CLetRec i x' y')
ds' (SLam ids x) = do
     x' <- ds' x
     return (CLam ids x')
ds' (SApp x xs)) = do
     x'  <- ds' x
     xs' <- mapM ds' xs
     return (CApp x' xs')
ds' (SAssign i x)) = do
     x' <- ds' x
     return (CAssign i x')
ds' (SList xs) = do
     xs' <- mapM ds' xs
     return (CList xs')
ds' (SListCase x y z) = do
     x' <- ds' x
     y' <- ds' y
     z' <- ds' z
     return (CListCase x' y' z')
```

Figure A.5: Desugaring reference implementation, part 1: straightforward cases.

```
ds' STrue = return (CBool True)
ds' SFalse = return (CBool False)
ds' (SPrim o [s]) = do
     s'  <- ds' s
     CPrim1(o,s')
ds' (SPrim o [s1,s2]) = do
     s1'  <- ds' s1
     s2' <- ds' s2
     CPrim2(o,s1',s2')
ds' (SPrim (_,_)) = errorResult (syntaxError)
ds' (SBetween x y z) = do
     x' <- ds' x
     y' <- ds' y
     z' <- ds' z
     v1 <- freshVar
     v2 <- freshVar
     v2 <- freshVar
     return (CLet(v1,x',CLet(v2,y',CLet(v3,z',
                     CAnd (CPrim2(LT, CVar v1, CVar v2),
                                  CPrim2(LT, CVar v2, CVar v3))))))
ds' (SFor x bs y) = do
     x'  <- ds' x
     bs' <- mapM dsBind bs
     y'  <- ds' y
     (ids, zs) <- unzip bs'
     return (CApp (x', [CLam(ids, y'), CList(zs)]))

dsBind :: (Id,STerm) -> Result (Id,CTerm)
dsBind (i,x) = do
        x' <- ds' x
        return (i,x')
```

Figure A.6: Desugaring reference implementation, part 2: nontrivial cases.

Bibliography

Abelson, H., Dybvig, R. K., Haynes, C. T., Rozas, G. J., Adams, N. I., Friedman, D. P., Kohlbecker, E., Steele, G. L., Bartley, D. H., Halstead, R., Oxley, D., Sussman, G. J., Brooks, G., Hanson, C., Pitman, K. M., Wand, M., Clinger, W., and Rees, J. (1991). Revised4 report on the algorithmic language Scheme. *SIGPLAN Lisp Pointers*, IV(3):1–55.

Abramsky, S., Gabbay, D. M., and Maibaum, T. S. E., editors (1995). *Handbook of Logic in Computer Science (Vol. 3): Semantic Structures*. Oxford University Press, Inc., USA.

Akiba, T., Imajo, K., Iwami, H., Iwata, Y., Kataoka, T., Takahashi, N., Moskal, M., and Swamy, N. (2013). Calibrating research in program synthesis using 72,000 hours of programmer time. Technical report, Microsoft Research.

Alur, R., Bodik, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. (2013). Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design*, FMCAD, pages 1–8. IEEE.

Alur, R., Radhakrishna, A., and Udupa, A. (2017). Scaling enumerative program synthesis via divide and conquer. In Legay, A. and Margaria, T., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 319–336, Berlin, Heidelberg. Springer.

Amin, N. and Tate, R. (2016). Java and Scala's type systems are unsound: The existential crisis of null pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 838–848, New York, NY, USA. Association for Computing Machinery.

Awodey, S. (2006). *Category Theory*. Oxford University Press.

Barke, S., Peleg, H., and Polikarpova, N. (2020). Just-in-time learning for bottom-up enumerative synthesis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA).

Bartha, S. and Cheney, J. (2020). Towards meta-interpretive learning of programming language semantics. In *Proceedings of the 29th International Conference on Inductive Logic Programming (ILP 2019)*, number 11770 in LNCS, pages 16–25.

Bartha, S., Cheney, J., and Belle, V. (2021). One down, 699 to go: Or, synthesising compositional desugarings. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA).

Bodin, M., Charguéraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A., and Smith, G. (2014). A trusted mechanised JavaScript specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 87–100, New York, NY, USA. Association for Computing Machinery.

Bodin, M., Diaz, T., and Tanter, E. (2018). A trustworthy mechanized formalization of r. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2018, pages 13–24, New York, NY, USA. Association for Computing Machinery.

Bogdanas, D. and Roşu, G. (2015). K-java: A complete semantics of java. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 445–456, New York, NY, USA. Association for Computing Machinery.

Burstall, R. M. (1969). Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48.

Charguéraud, A., Schmitt, A., and Wood, T. (2018). JSExplain: A double debugger for JavaScript. In *Companion Proceedings of the The Web Conference 2018*, WWW '18, pages 691–699, Republic and Canton of Geneva, CHE. International World Wide Web Conferences Steering Committee.

Chen, J., Patra, J., Pradel, M., Xiong, Y., Zhang, H., Hao, D., and Zhang, L. (2020). A survey of compiler testing. *ACM Computing Surveys*, 53(1).

Chen, W. and Warren, D. S. (1993). Query evaluation under the well-founded semantics. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '93, pages 168–179, New York, NY, USA. Association for Computing Machinery.

Clark, A. and Lappin, S. (2010). Unsupervised learning and grammar induction. In *The Handbook of Computational Linguistics and Natural Language Processing*, pages 197–220.

Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. (2007). Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`. release October, 12th 2007.

Cropper, A. and Muggleton, S. H. (2016). Metagol system. `https://github.com/metagol/metagol`.

Dasgupta, S., Park, D., Kasampalis, T., Adve, V. S., and Roşu, G. (2019). A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 1133–1148, New York, NY, USA. Association for Computing Machinery.

David, R., Raffalli, C., Theyssier, G., Grygiel, K., Kozik, J., and Zaionc, M. (2009). Some properties of random lambda terms. *Logical Methods in Computer Science*, 9(1).

Duregård, J., Jansson, P., and Wang, M. (2012). Feat: Functional enumeration of algebraic types. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, page 61–72, New York, NY, USA. Association for Computing Machinery.

Dyckhoff, R. and Pinto, L. (1999). Proof search in constructive logics. In *Sets and Proofs: invited papers from Logic Colloquium'97*, pages 53–65.

Ellison, C. and Roşu, G. (2012). An executable formal semantics of C with applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 533–544, New York, NY, USA. Association for Computing Machinery.

Felleisen, M. (1991). On the expressive power of programming languages. *Science of Computer Programming*, 17(1):35–75.

Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. (2001). *How to Design Programs*. MIT Press.

Filaretti, D. and Maffeis, S. (2014). An executable formal semantics of PHP. In Jones, R., editor, *Proceedings of the 28th European Conference on Object Oriented Programming (ECOOP 2014)*, pages 567–592, Berlin, Heidelberg. Springer-Verlag.

Frankle, J., Osera, P.-M., Walker, D., and Zdancewic, S. (2016). Example-directed synthesis: A type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 802–815, New York, NY, USA. Association for Computing Machinery.

Gentzen, G. (1935). Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39(1):176–210.

Gentzen, G. (1969). Investigations into logical deduction. In Szabo, M., editor, *The Collected Papers of Gerhard Gentzen*, chapter 3, pages 68–131. North-Holland Publishing Company, Amsterdam.

Gibbons, J. (2006). Datatype-generic programming. In *Proceedings of the 2006 International Conference on Datatype-Generic Programming*, SSDGP'06, pages 1–71, Berlin, Heidelberg. Springer-Verlag.

Girard, J.-Y. (1987). Linear logic. *Theoretical Computer Science*, 50(1):1–102.

Guha, A., Saftoiu, C., and Krishnamurthi, S. (2010). The essence of JavaScript. In *Proceedings of the 24th European Conference on Object Oriented Programming (ECOOP 2010)*, pages 126–150.

Gulwani, S., Polozov, O., and Singh, R. (2017). Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119.

Hathhorn, C., Ellison, C., and Roşu, G. (2015). Defining the undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 336–345, New York, NY, USA. Association for Computing Machinery.

Herbelin, H. (1994). A lambda-calculus structure isomorphic to Gentzen-style sequent calculus structure. In *Selected Papers from the 8th International Workshop on Computer Science Logic*, CSL '94, pages 61–75, Berlin, Heidelberg. Springer-Verlag.

Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A., and Roşu, G. (2018). KEVM: A complete formal semantics of the Ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217.

Hinze, R., Jeuring, J., and Löh, A. (2004). Type-indexed data types. *Science of Computer Programming*, 51(1):117–151. Mathematics of Program Construction (MPC 2002).

Howard, W. A. (1980). The formulae-as-types notion of construction. In Curry, H., B., H., Roger, S. J., and Jonathan, P., editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press.

Inala, J. P., Polikarpova, N., Qiu, X., Lerner, B. S., and Solar-Lezama, A. (2017). Synthesis of recursive ADT transformations from reusable templates. In Legay, A. and Margaria, T., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 247–263, Berlin, Heidelberg. Springer.

Jha, S., Gulwani, S., Seshia, S. A., and Tiwari, A. (2010). Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 215–224.

Jung, R., Jourdan, J.-H., Krebbers, R., and Dreyer, D. (2017). RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL).

Katayama, S. (2013). MagicHaskeller on the web: Automated programming as a service. In *Haskell Symposium*.

Kern, C. and Greenstreet, M. R. (1999). Formal verification in hardware design: A survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193.

Kheradmand, A. and Roşu, G. (2018). P4K: A formal semantics of p4 and applications. Technical report, University of Illinois at Urbana-Champaign. `https://arxiv.org/abs/1804.01468`.

Kohlbecker, E. (1986). *Syntactic Extensions in the Programming Language LISP*. PhD thesis, USA. UMI Order No. GAX86-27998.

Krishnamurthi, S., Lerner, B. S., and Elberty, L. (2019). The next 700 semantics: A research challenge. In Lerner, B. S., Bodík, R., and Krishnamurthi, S., editors, *3rd Summit on Advances in Programming Languages (SNAPL 2019)*, volume 136 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:14, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Kuraj, I., Kuncak, V., and Jackson, D. (2015). Programming with enumerable sets of structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 37–56, New York, NY, USA. Association for Computing Machinery.

Landin, P. J. (1964). The mechanical evaluation of expressions. *Computer Journal*, 6:308–320.

Landin, P. J. (1966). The next 700 programming languages. *Communications of the ACM*, 9(3):157–166.

Lee, W. (2021). Combining the top-down propagation and bottom-up enumeration for inductive program synthesis. *Proceedings of the ACM on Programming Languages*, 5.

MacLane, S. (1971). *Categories for the Working Mathematician*. Springer-Verlag, New York. Graduate Texts in Mathematics, Vol. 5.

Maffeis, S., Mitchell, J. C., and Taly, A. (2008). An operational semantics for JavaScript. In Ramalingam, G., editor, *Programming Languages and Systems, APLAS 2008*, volume 5356 of *Lecture Notes in Computer Science*, pages 307–325, Berlin, Heidelberg. Springer.

Magalhães, J. P., Dijkstra, A., Jeuring, J., and Löh, A. (2010). A generic deriving mechanism for Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 37–48, New York, NY, USA. Association for Computing Machinery.

McCarthy, J. (1966). A formal definition of a subset of Algol. *Formal Language Description Languages for Computer Programming*, pages 1–12.

Meinke, K. and Tucker, J. V. (1993). Universal algebra. In *Handbook of Logic in Computer Science (Vol. 1): Background: Mathematical Structures*, pages 189–368. Oxford University Press, Inc., USA.

Menon, A., Tamuz, O., Gulwani, S., Lampson, B., and Kalai, A. (2013). A machine learning framework for programming by example. In Dasgupta, S. and McAllester, D., editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 187–195, Atlanta, Georgia, USA. PMLR.

Milner, R., Tofte, M., and Macqueen, D. (1997). *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA.

Moggi, E. (1991). Notions of computation and monads. *Inf. Comput.*, 93(1):55–92.

Morandat, F., Hill, B., Osvald, L., and Vitek, J. (2012). Evaluating the design of the R language: Objects and functions for data analysis. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, pages 104–131, Berlin, Heidelberg. Springer-Verlag.

Muggleton, S. and de Raedt, L. (1994). Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19-20:629–679. Special Issue: Ten Years of Logic Programming.

Muggleton, S. H., Lin, D., Pahlavi, N., and Tamaddoni-Nezhad, A. (2014). Meta-interpretive learning: Application to grammatical inference. *Mach. Learn.*, 94(1):25–49.

Muralidaran, V., Spasić, I., and Knight, D. (2021). A systematic review of unsupervised approaches to grammar induction. *Natural Language Engineering*, 27(6):647–689.

Negri, S., von Plato, J., and Ranta, A. (2001). *Structural Proof Theory*. Cambridge University Press.

New, M. S., Fetscher, B., Findler, R. B., and McCarthy, J. (2017). Fair enumeration combinators. *Journal of Functional Programming*, 27:e19.

Nienhuis, K., Memarian, K., and Sewell, P. (2016). An operational semantics for C/C++11 concurrency. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 111–128, New York, NY, USA. Association for Computing Machinery.

Osera, P. and Zdancewic, S. (2015). Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 619–630.

Park, D., Stefănescu, A., and Roşu, G. (2015). KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 346–356, New York, NY, USA. Association for Computing Machinery.

Patsantzis, S. and Muggleton, S. H. (2022). Meta-interpretive learning as metarule specialisation. *Machine Learning*, 111(10):3703–3731.

Pierce, B. C. (2004). *Advanced Topics in Types and Programming Languages*. The MIT Press.

Plotkin, G. D. (1981). A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus university, Aarhus, Denmark.
Also published in: *Journal of Logic and Algebraic Programming* 60–61:17–140, 2004.

Polikarpova, N., Kuraj, I., and Solar-Lezama, A. (2016). Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 522–538, New York, NY, USA. Association for Computing Machinery.

Politz, J. G., Carroll, M. J., Lerner, B. S., Pombrio, J., and Krishnamurthi, S. (2012). A tested semantics for getters, setters, and eval in javascript. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 1–16, New York, NY, USA. Association for Computing Machinery.

Politz, J. G., Martinez, A., Milano, M., Warren, S., Patterson, D., Li, J., Chitipothu, A., and Krishnamurthi, S. (2013). Python: The full monty. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '13, pages 217–232, New York, NY, USA. Association for Computing Machinery.

Polozov, O. and Gulwani, S. (2015). FlashMeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on*

Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, pages 107–126, New York, NY, USA. Association for Computing Machinery.

Regan, K. W. (1992). Minimum-complexity pairing functions. *Journal of Computer and System Sciences*, 45(3):285–295.

Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C., and Tinelli, C. (2019). cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In Dillig, I. and Tasiran, S., editors, *Computer Aided Verification*, pages 74–83, Cham. Springer International Publishing.

Reynolds, J. C. (1997). The essence of Algol. In O'Hearn, P. W. and Tennent, R. D., editors, *Algol-like Languages*, pages 67–88. Birkhäuser Boston, Boston, MA.

Rosenberg, A. L. (2002). Efficient pairing functions - and why you should care. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, page 134, USA. IEEE Computer Society.

Solar-Lezama, A., Jones, C. G., and Bodík, R. (2008). Sketching concurrent data structures. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 136–148.

Solar-Lezama, A., Rabbah, R. M., Bodík, R., and Ebcioglu, K. (2005). Programming by sketching for bit-streaming programs. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 281–294.

Sundholm, G. (1983). *Proof Theory: a survey of the omega-rule*. PhD thesis, University of Oxford.

Van Der Rest, C. and Swierstra, W. (2022). A completely unique account of enumeration. *Proceedings of the ACM on Programming Languages*, 6(ICFP).

Wadler, P. (1992). Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493.

Wells, J. B. and Yakobowski, B. (2005). Graph-based proof counting and enumeration with applications for program fragment synthesis. In Etalle, S., editor, *Logic Based Program Synthesis and Transformation*, pages 262–277, Berlin, Heidelberg. Springer.