Cumulative Learning in the Lambda Calculus

Robert John Henderson

Thesis submitted for the degree of
Doctor of Philosophy
Department of Computing
Imperial College London

May 2014
Copyright Declaration

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.
Abstract

I design a machine learning system capable of ‘cumulative learning’, which means that it automatically acquires the knowledge necessary for solving harder problems through experience of solving easier ones. Working within the learning framework of inductive programming, I propose that the technique of abstraction, familiar from software engineering, is a suitable mechanism for accumulating knowledge. In abstraction, syntactic patterns in solutions to past problems are isolated as re-usable units and added to background knowledge.

For the system’s knowledge representation language, I argue that lambda calculus is a more suitable choice than first-order logic because lambda calculus supports abstraction readily. However, more mature and theoretically well-founded base inference techniques are available within first-order Inductive Logic Programming (ILP). Therefore, my approach is to adapt ILP inference techniques to lambda calculus.

Central to ILP is the idea of ‘generality’, and I show that a suitable concept of generality in lambda calculus arises from its standard denotational semantics. Consequently, notions of entailment, subsumption, refinement, and inverse deduction have direct analogues in the lambda calculus setting. I argue that the conventional ‘compression’ measure used in ILP is inflexible in capturing prior assumptions, particularly in the context of an expanding background knowledge. Instead I introduce a non-parametric Bayesian prior over hypotheses and background knowledge. I then design an inductive inference algorithm for the lambda calculus setting based on refinement and proof-directed search. I give a formal proof of correctness of this algorithm.

To enable automatic invention of abstractions, I design two algorithms. The first is a heuristic search that uses anti-unification to discovering opportunities for abstraction within a corpus of knowledge. I give a formal characterisation of its search space. The second algorithm performs inverse deduction in order to refactor knowledge in terms of an abstraction. I prove that this refactoring process is semantics-preserving.
Acknowledgements

I would like to thank my supervisor, Stephen Muggleton, without whose guidance and advice this project would not have been possible. I would also like to thank Dianhuan Lin, José Santos, Tristan Allwood, and Antonia Mayer for many helpful discussions. This project was supported by an EPSRC studentship.
Declaration of Originality

I declare that the work presented in this thesis is my own, except where appropriately stated and referenced.
# Contents

1 Introduction 4  
1.1 The RUFINSKY System ........................................ 6  
1.2 First-Order Logic vs. Lambda Calculus .................... 8  
1.3 Contributions of this Project .............................. 12

2 Background 13  
2.1 Cumulative Learning ........................................... 13  
2.2 Inductive Logic Programming ............................... 18  
2.2.1 Main Principles ........................................ 19  
2.2.2 Inverse Deduction .................................... 21  
2.2.3 Refinement ............................................... 25  
2.2.4 Multiple Refinement Points and Proof-Directed Search ............................................. 27  
2.3 Other Inductive Programming Techniques .................. 31  
2.3.1 Higher-order Background Knowledge as Declarative Bias ........................................ 31  
2.3.2 Non-termination and Levin Search .................... 33  
2.4 Abstraction Invention ....................................... 35  
2.4.1 Anti-unification as a Means of Abstraction ........... 35  
2.4.2 Predicate Invention in Inductive Logic Programming ........................................ 37  
2.4.3 Limitations of Predicate Invention due to First-Order Logic ........................................ 40

3 The Lambda Calculus Setting for Learning 43  
3.1 Syntax ......................................................... 43  
3.1.1 Types .................................................. 43  
3.1.2 Terms .................................................. 44  
3.1.3 Declarations ......................................... 46  
3.2 Denotational Semantics ...................................... 47  
3.2.1 Types .................................................. 47  
3.2.2 Terms .................................................. 48  
3.3 Generality Orders ........................................... 50  
3.4 Setting for Learning ....................................... 52
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.1</td>
<td>Partial terms represent sets of hypotheses</td>
<td>53</td>
</tr>
<tr>
<td>4</td>
<td>Prior Probability Distributions</td>
<td>54</td>
</tr>
<tr>
<td>4.1</td>
<td>The Base Prior</td>
<td>56</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Some examples</td>
<td>63</td>
</tr>
<tr>
<td>4.2</td>
<td>The Nonparametric Prior</td>
<td>65</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Worked example</td>
<td>75</td>
</tr>
<tr>
<td>4.3</td>
<td>The Cumulative Learning Prior</td>
<td>77</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Worked example</td>
<td>80</td>
</tr>
<tr>
<td>5</td>
<td>Refinement and RUFUS</td>
<td>84</td>
</tr>
<tr>
<td>5.1</td>
<td>Refinement</td>
<td>84</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Refinement operators and their properties</td>
<td>85</td>
</tr>
<tr>
<td>5.1.2</td>
<td>The RUFUS refinement operator</td>
<td>86</td>
</tr>
<tr>
<td>5.2</td>
<td>Proof-Directed Search</td>
<td>88</td>
</tr>
<tr>
<td>5.2.1</td>
<td>The evaluator</td>
<td>90</td>
</tr>
<tr>
<td>5.2.2</td>
<td>The runThreads algorithm</td>
<td>91</td>
</tr>
<tr>
<td>5.2.3</td>
<td>The RUFUS search tree</td>
<td>96</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Guiding the search</td>
<td>100</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Worked example</td>
<td>103</td>
</tr>
<tr>
<td>6</td>
<td>Anti-unification Search</td>
<td>105</td>
</tr>
<tr>
<td>6.1</td>
<td>Tree-Structured Terms</td>
<td>106</td>
</tr>
<tr>
<td>6.2</td>
<td>The auSearch Specification</td>
<td>108</td>
</tr>
<tr>
<td>6.3</td>
<td>The auSearch Algorithm</td>
<td>111</td>
</tr>
<tr>
<td>6.4</td>
<td>The mnos Algorithm</td>
<td>121</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Gavril’s maximum independent set algorithm</td>
<td>121</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Using Gavril’s algorithm to find maximum non-overlapping subsets</td>
<td>122</td>
</tr>
<tr>
<td>7</td>
<td>Abstraction Invention and KANDINSKY</td>
<td>125</td>
</tr>
<tr>
<td>7.1</td>
<td>The absInv Algorithm</td>
<td>125</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Worked example</td>
<td>135</td>
</tr>
<tr>
<td>7.2</td>
<td>Proof of Correctness of absInv</td>
<td>136</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Preliminaries</td>
<td>136</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Eta-expansion</td>
<td>139</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Refactoring the master term</td>
<td>139</td>
</tr>
<tr>
<td>8</td>
<td>Discussion</td>
<td>145</td>
</tr>
<tr>
<td>8.1</td>
<td>A ‘Proof-of-Concept’ Demonstration of RUFINSKY</td>
<td>145</td>
</tr>
<tr>
<td>8.2</td>
<td>How RUFINSKY could be Experimentally Evaluated</td>
<td>151</td>
</tr>
</tbody>
</table>
8.3 Limitations of RUFINSKY’s Cumulative Learning Mechanism ........................................ 154

8.4 Issues Raised about the Design of Inductive Programming Systems 155
  8.4.1 First order logic and lambda calculus ............................................................ 155
  8.4.2 Bayesian inference ......................................................................................... 157

8.5 Related Work ...................................................................................................... 158
  8.5.1 Search strategies ............................................................................................ 158
  8.5.2 Hybrid functional logic languages ................................................................. 159
  8.5.3 Program transformation ................................................................................. 159
  8.5.4 Analogical reasoning ..................................................................................... 160

9 Conclusions and Future Work .............................................................................. 161
  9.1 Summary of Thesis and Main Results ................................................................. 161
  9.2 Directions for Future Work .............................................................................. 162
  9.3 Final Thoughts .................................................................................................. 164

A Mathematical Conventions ................................................................................. 165
  A.1 Set Comprehension Notation ........................................................................... 165

B Proof of Lemma 6.53 ............................................................................................. 167

Bibliography ............................................................................................................. 170
Chapter 1

Introduction

The aim of this PhD project is to explore one approach to the problem of cumulative learning [Michelucci and Oblinger, 2010] in artificial intelligence. In other words: how might a computer system accumulate knowledge or expertise over time, and thus incrementally improve its own ability to learn and solve problems?

To illustrate what is meant by the term ‘cumulative learning’, let us consider two examples. The first is that of a human learning a skill, for example, a child learning to speak and to understand their native language. The second is that of the scientific method, the process by which the human race as a whole increases its understanding of the nature of the universe and the laws governing it, by performing experiments so as to guide the development of increasingly accurate theories. Both of these processes, though they occur at very different scales, have two features in common:

1. *Inductive inference* is used to solve problems.

2. Progression to more difficult problems is made possible via the *incremental accumulation of knowledge*.

For the purpose of this thesis, I define ‘cumulative learning’ as any process that possesses these two features.

In this PhD, the goal is to design and implement a machine learning algorithm capable of cumulative learning as defined by the two features above. Why pursue such a goal? There are two main reasons, one philosophical and the other practical. The philosophical motivation is that by implementing an algorithm that performs a process, we can improve our understanding of the nature of that process itself. In other words, if human learning and the scientific method are both instances of cumulative learning, then if one can construct a demonstrably effective algorithm that is also an instance of cumulative learning, it should provide some insight into how and why processes such as human and
A more practical motivation for studying cumulative learning relates to the field of machine learning and data mining [Bishop, 2006; Kononenko and Kukar, 2007], the sub-field of artificial intelligence which may be defined as ‘the use of computer algorithms to discover patterns in data’. Machine learning currently has many industrial applications, to name a few:

- analysis of scientific data (particularly in biology)
- analysis of financial or stock market data
- medical diagnosis
- robot control (e.g. robots to help in disaster situations; autonomous vehicles; robots for exploring other planets)
- information retrieval (e.g. internet search engines)
- automated translation
- handwriting and speech recognition
- artificial intelligence in computer games

Despite these many applications, modern machine learning algorithms (for example: feedforward neural networks, Bayesian networks, or support vector machines) are typically not capable of cumulative learning as defined by the two criteria given above. Most would only satisfy the first criterion, i.e. they use inductive inference to solve problems, often within a Bayesian framework. On the other hand, almost all machine learning algorithms currently in practical use lack any significant ability to ‘progress to more difficult problems via the incremental accumulation of knowledge’.

This lack of an ability for the machine to learn cumulatively is a significant drawback, and it means that applying machine learning systems to real-world problems is currently a relatively labour-intensive and expensive activity. Custom pattern-recognition algorithms are often implemented on a per-application basis, or otherwise data has to be heavily pre-processed in order to be suitable for input into stock algorithms. The standard mechanism for ‘progressing to more difficult problems’ is the input of human expertise. On the other hand, if we can develop machine learning systems that accumulate their own expertise and domain-specific knowledge through problem-solving experience, then such systems will be more flexible, and they will be applicable in a much wider variety of situations without the need for reprogramming or tuning by a human operator.
Lastly, it would be disingenuous not to say that a significant source of motivation for this PhD project has been to contribute towards the dream of ‘strong AI’ or ‘artificial general intelligence’. Strong AI refers to machines that possess intelligence at or above the level of humans, where intelligence is defined as ‘an agent’s ability to achieve goals in a wide range of environments’ [Legg and Hutter, 2006]. Since the 1970s, strong AI has been an emotive, almost taboo topic among artificial intelligence researchers. This is largely due to over-optimistic claims made by the early AI pioneers about the speed with which AI research would progress, and a subsequent backlash against strong AI research in general [Russell and Norvig, 2010].

However, in recent years some prominent AI scientists are starting to talk about strong AI as a serious research topic in public again [Bengio et al., 2013; Hutter, 2012; Schmidhuber et al., 2011; Maei and Sutton, 2010]. At the same time, others within and outside of the AI research community are expressing concern about the potential dangers of developing strong AI [Yudkowsky, 2008; Bostrom, 2012; Yampolskiy, 2013; Muehlhauser and Salamon, 2012; Fox and Shulman, 2010]. It is my opinion that a scientist who is consciously playing a part, even if small, in a research effort whose ultimate goal is strong AI has a responsibility to say so openly. We must take any long-term risks seriously as the field advances and balance them against potential benefits.

1.1 The RUFINSKY System

A design for a cumulative learning system must feature two things: a mechanism for performing inductive inference, and a mechanism for progressing to more difficult problems by incrementally accumulating knowledge. For the inductive inference mechanism, a natural place to start is the framework of Solomonoff [1964]. Solomonoff showed that if one uses a Turing-complete representation for inductive hypotheses, then it is possible (and indeed relatively straightforward) to express all inductive inference problems in a single, unified form, and hence conceive of a completely general mechanism for inductive inference.

The closest thing to a concrete realisation of Solomonoff’s framework is the modern field of inductive programming [Muggleton and De Raedt, 1994; Kitzelmann, 2009a], which studies algorithms for doing inductive inference using a Turing-complete programming language (often Prolog or Haskell) as a unified representation for knowledge. The field of inductive programming has produced some powerful theory and practical inference techniques. Examples of techniques include refinement, the use of a partial order over a Turing complete space of inductive hypotheses to make it amenable to guided search, and inverse deduction, a technique of using deduction procedures run in reverse as
a principled means of performing induction. Modern inductive programming has a solid grounding in other areas of computer science and mathematics such as logic [Nienhuys-Cheng and de Wolf, 1997], programming language theory [Pierce, 2002], and Bayesian machine learning [Bishop, 2006]. In this project, I have chosen to use techniques from inductive programming in order to realise a suitable generic inductive inference mechanism.

The second necessary feature of a cumulative learning system is a method for automatically accumulating knowledge, and for making use of that knowledge so as to enable progression to successively harder problems. For this, let us turn for inspiration to what may seem initially like an unexpected source: software engineering. Software engineering would not (usually) be regarded as an inductive inference process, so it does not fit my earlier definition of cumulative learning. However, there is much in the practice of software engineering that fits the second criterion of ‘progression to more difficult problems via the incremental accumulation of knowledge’. Software engineering is the discipline or craft of computer programming on the medium-to-large scale. Through this process, humans are able to collaborate on the construction of complex computer programs, despite the fact that the design of such a program (a modern operating system such as GNU/Linux, for example) can sometimes be so large and so rich in detail that no individual human could ever hope to understand all of it even in a lifetime. Thus, at the heart of software engineering is the idea of managing complexity, and the primary means for doing this is enshrined in something called the principle of abstraction. This may be stated as follows: ideas, designs, or techniques that need to be used more than once should be separated out from the specific contexts in which they appear, and encapsulated as re-usable units.

As an example of the principle of abstraction, consider the technique of sorting a list. It is likely that in a large computer program, there may be many distinct contexts in which it is necessary to sort lists of numbers or other items. As a result, it makes sense to encapsulate the process of sorting as a re-usable function in a library, so that the effort of implementing a sorting algorithm need only be performed once and by one person only. Other programmers can then use the sorting algorithm many times, in different situations, without having to re-write it or even understand how it is implemented.

In fact, any kind of repetitive structure or pattern in a computer program is usually a sign that there is a single idea there that is being used more than once. A good software engineer will always be on the lookout for such patterns, because each one provides an opportunity for abstracting out a potentially time-saving re-usable program unit. Indeed, putting the principle of abstraction into
practice has at least three benefits:

- It makes the program *smaller* (because repetitive code is eliminated), and hence easier to maintain and debug.
- It makes the program more modular and hence much easier to comprehend, because distinct ideas are separated out from one another rather than being intertwined as ‘spaghetti code’.
- A library of re-usable components is accumulated which guides the construction of new programs.

As a result of this discipline of abstraction, software engineers are able to ‘stand on the shoulders of giants’ when constructing large systems. As more and more re-usable program units are created, it allows for more advanced programs to be constructed with less effort, and hence for new problems to be solved that would have been much more difficult, or even impossible, otherwise. Indeed, this is a clear example of ‘progression to more difficult problems via the incremental accumulation of knowledge’.

There is a crucial difference between the knowledge accumulation mechanism in software engineering, as opposed to that in a process such as human language learning or the scientific method. In the latter processes, the representation of knowledge, and the algorithms for creating new knowledge, are only vaguely understood and to a large extent hidden away in the workings of the human brain. Hence it is very difficult to directly formalise or emulate these processes.

By contrast, in the case of software engineering the ‘knowledge’ takes the form of an explicit computer program, and mechanisms used by software engineers to abstract out re-usable programs units are amenable to formalisation. In this project, I take this idea of abstraction from software engineering and apply it to inductive programming. The product of this shall be an *abstraction invention* system: an algorithm that emulates the process of abstraction as carried out by human software engineers.

To summarise this section, the main contribution of this PhD project is the design and implementation of an inductive programming system that uses the mechanism of abstraction, borrowed from software engineering, in order to accumulate useful knowledge and hence improve its own problem-solving ability with experience. The system’s name is *RUFINSKY*; for a brief overview of its structure see Fig. 1.1.

1.2 First-Order Logic vs. Lambda Calculus

One of the most important issues in the design of RUFINSKY is the choice of which language to use to represent knowledge. As explained in the previ-
Figure 1.1: Overview of RUFINSKY: an inductive programming system capable of cumulative learning. It consists of two modules: RUFUS and KANDINSKY. RUFUS performs inductive inference over input data, subject to Background Knowledge (BK). The BK is a library of re-usable program units which RUFUS uses to construct the output: hypotheses consistent with the data. KANDINSKY performs abstraction invention over hypotheses and BK, potentially generating new BK which can be passed back to RUFUS.
ous section, RUFINsky’s inference mechanism shall be based on techniques from inductive programming. However, the field of inductive programming is somewhat split into two schools of thought. On the one hand, there is Inductive Logic Programming (ILP) [Muggleton and De Raedt, 1994], in which the standard practice is to use a Prolog-like language, i.e. first-order logic, to represent knowledge. On the other hand, there is Inductive Functional Programming (IFP) [Kittelmann, 2009a], in which the standard practice is to use a Haskell-like language, i.e. typed lambda calculus, to represent knowledge.

Since the most significant part of this project is the design of an abstraction invention procedure, the most important consideration when choosing the knowledge representation language had to be this: ‘how well does the language support the software engineering principle of abstraction?’. Now, Prolog¹ and Haskell are both expressive, declarative languages (both have a denotational semantics), which is what makes both suitable for generic knowledge representation in inductive programming. Prolog’s particular strengths lie in its support for powerful methods of deductive reasoning, in particular unification of terms, and multimodal predicates. Prolog and Haskell both support non-determinism and representation of grammars [Matsushita, 1998, Chap. 3]. However, when it comes to support for abstraction, Haskell wins hands down. This is because functions have first-class status in Haskell (and in other languages based on lambda calculus such as ML or Lisp). This means that, in Haskell, functions can be passed as arguments to other functions and returned as values. First-class functions enable abstraction of patterns in programs in ways that are impossible, or at best extremely convoluted to express, in a language such as Prolog in which the equivalent of functions (i.e. predicates) are not first-class.

The following quote from Abelson and Sussman [1996, Sect. 1.3] illustrates this point (read ‘procedure’ as a synonym for ‘function’):

\begin{quote}
Even in numerical processing we will be severely limited in our ability to create abstractions if we are restricted to procedures whose parameters must be numbers. Often the same programming pattern will be used with a number of different procedures. To express such patterns as concepts, we will need to construct procedures that can accept procedures as arguments or return procedures as values.
\end{quote}

A well-known example of an abstraction that can only expressed with first-class functions is the map operation, which has the following type signature in Haskell:

\footnote{By Prolog in this context I mean the pure subset of Prolog typically used to represent knowledge in ILP, not the larger, impure language used for actual software engineering with its metalogical predicates, etc.}

10
\texttt{map :: (a -> b) -> [a] -> [b]}

*Map* represents the concept of ‘transforming each element of a list with some unary operation’, for example \texttt{map (+ 3)} is the function which adds three to every element of a list.

A lambda calculus based language may be more suitable for making abstractions, but we must also consider what inductive inference techniques are available in the two sub-fields of inductive programming, ILP and IFP. In IFP, the focus of research has tended to have been from a perspective of recursive program synthesis, or ‘automatic programming’, rather than general inductive inference [Kitzelmann, 2009a]. The state-of-the-art techniques in IFP reflect this, for example the ‘analytical functional approach’ [Kitzelmann, 2009b]. On the other hand, state-of-the-art techniques in ILP are oriented much more towards generic inductive inference of the kind described by Solomonoff [1964]. For example, the technique of \textit{refinement} and the theory of \textit{refinement operators} [Nienhuys-Cheng and de Wolf, 1997] provide a principled and effective means for searching through a space of arbitrary inductive hypotheses for one that is consistent with given data. In refinement, the use of a \textit{generality ordering} over hypotheses enables pruning of large tracts of the hypothesis space in a principled manner, and the use of a compression-based \textit{coverage measure} (equivalent to a Bayesian posterior) enables positive guidance of the search towards more promising regions of the space.

It is clear that the existing techniques from Prolog-based ILP are more suitable as generic inductive inference mechanisms than those from IFP, and this has tended to produce a belief among the research community that first-order logic is the only knowledge representation language worth taking seriously for generic inductive inference. However, in this thesis I shall argue that this is in fact not the case, and that the most powerful techniques from ILP can be adapted quite readily to work with lambda calculus as the knowledge representation language. Hence, in this project I adapt the ILP technique of refinement to work with lambda calculus, the product being the base inductive inference system RUFUS (Fig. 1.1), which works by means of a refinement-based guided search through a space of lambda calculus programs. Thus, RUFINSKY benefits both from an effective generic inductive inference technique borrowed from Prolog-based ILP, as well as having all the facility for abstraction provided by first-class functions in lambda calculus.
1.3 Contributions of this Project

This thesis consists of a detailed design for the RUFINSKY cumulative learning system, justified by formal proofs of the properties of its various parts. The main contributions of the thesis are as follows:

- A review of the literature related to cumulative learning, inductive programming, and abstraction invention (Chap. 2).
- A formal specification of a Bayesian setting for inductive inference in which simply typed lambda calculus is the hypothesis language (Chap. 3).
- The design of a family of suitable prior probability distributions over lambda calculus terms, with formal proofs of their properties (Chap. 4).
- The design of a base inductive inference system, RUFUS, for which I adapted the ILP techniques of refinement and proof-directed search to work in the lambda calculus setting (Chap. 5). I give a formal proof that RUFUS’ search space is sound and complete.
- The design of an efficient search algorithm, ‘anti-unification search’, which can discover opportunities for abstraction within a corpus of lambda calculus terms (Chap. 6). I give a formal characterisation of the search space of this algorithm.
- The design of an abstraction invention system, KANDINSKY (Chap. 7), which incorporates ‘anti-unification search’ as well as an algorithm for refactoring a corpus of lambda calculus terms in order to construct an abstraction. I prove that this refactoring process is semantics-preserving.
- A critical discussion of the RUFINSKY design and of the issues encountered while exploring this design space. I also give a ‘proof of concept’ demonstration of the working RUFINSKY system, and discuss how RUFINSKY could be evaluated experimentally (Chap. 8).

Note that a description of an early incarnation of KANDINSKY was published mid-way through this project as a short paper ([Henderson and Muggleton, 2012]).
Chapter 2

Background

2.1 Cumulative Learning

At the beginning of Chap. 1 I defined *cumulative learning*, for the purpose of this thesis, to mean any process in which inductive inference is used to solve problems and in which progression to more difficult problems is made possible via the incremental accumulation of knowledge. In this section I review what work has been done on cumulative learning in artificial intelligence. Now, automated inductive inference is the domain of *machine learning*, therefore we shall be looking here at what attempts have been made to design machine learning systems that are able to improve their performance over the course of multiple learning problems by acquiring knowledge.

What do we mean precisely by knowledge? Broadly, we can take this to mean *inductive bias*, i.e. any information or set of assumptions possessed by a learning system that causes it to prefer one hypothesis over another when both are consistent with observed data [Mitchell, 1980].

In conventional statistical machine learning (neural networks, Bayesian networks, etc.) the field of *transfer learning* has strong connections with the idea of cumulative learning. It studies how inductive bias can be shifted as a result of experience of one set of learning tasks, in order to improve performance on another, related set of learning tasks.

For example, Raina et al. [2006] looked at transfer learning using a logistic regression model in a text classification domain. In their study, the distribution of successful hypotheses learned on one set of classification problems was used to make informed adjustments to the parameters of a prior probability distribution over all hypotheses, so as to bias it in favour of finding similar hypotheses in future. The adjusted prior was shown to produce an improvement in predictive accuracy when applied to new problems from the same text classification domain.
As another example, Niculescu-Mizil and Caruana [2007] developed a method of transfer learning for Bayesian network structure learning. Here, a prior probability distribution was devised that took into account the relatedness of a set of problems, assigning higher probability to a hypothesised network for one problem if it was similar to networks learned for other problems in the set. Their method required that each problem have the same domain of random variables, and similarity between networks was measured by counting the occurrences of edges that were present / not present in both.

Transfer learning techniques like those of Raina et al. and Niculescu-Mizil and Caruana can yield performance improvements, however the forms of ‘knowledge’ that these systems can accumulate through experience are limited. The problem is that the statistical learning techniques used here and in other similar studies are restricted, by design, to learning hypotheses from particular narrow classes (be it logistic regression models, Bayesian networks, neural networks, decision trees, etc.). Thus, while these transfer learning methods are able to adjust the bias towards some promising region of parameter space within one of these model classes, the underlying model class itself is unchangeable and this arguably has a much more significant effect on what the system is usefully able to learn.

As a result, in these kinds of transfer learning studies the shifts in bias tend to be enough to produce improvements in predictive accuracy in the presence of sparse data, but not enough to allow systems to progress to fundamentally more difficult problems, i.e. problems that would be computationally intractable, even with large amounts of data available, before the shift in bias. As an analogy, consider how a human learns to do mathematics. Learning arithmetic gives you useful knowledge that enables you to learn simple algebra. Then, with a knowledge of algebra you are in a position to learn to understand calculus. However, if one were to attempt to learn advanced calculus straight off without any prior knowledge of arithmetic or algebra, you would likely find it intractably difficult no matter how much ‘data’ you were given, i.e. training examples of calculus problems and their correct answers. Thus, the kind of transfer learning that we see demonstrated with these statistical learning techniques cannot be called true cumulative learning, because it enables no progression to fundamentally new or more difficult problems that would have been intractable for the system to solve before its shift in bias.

There is one area of machine learning, *inductive programming*, that differs from the more conventional techniques just described in that, rather than a learner being restricted to hypotheses of some particular narrow class, a generic *language* of hypotheses is used. This provides a far greater degree of flexibility
both in what hypotheses can potentially be learned by the system, and also in what kinds of inductive bias may be specified in order to cause the system to favour certain kinds of hypotheses rather than others. Indeed, using such a language one can express inductive bias explicitly as background knowledge: a set of definitions of concepts that the learner is allowed to compose together to construct hypotheses. By varying the background knowledge, one can radically change the system’s inductive bias, as well has control whether the bias is strong, favouring one chosen class of hypotheses, or weak, evenly favouring hypotheses with a variety of structural forms.

The flexibility of background knowledge has proved very useful for applying inductive programming to various applications (see Sect. 2.2), because it allows a detailed domain-specific inductive bias to be specified by a human expert without any need to customise the learning algorithm used to perform inductive inference. However, from a point of view of cumulative learning, background knowledge can also potentially be learned automatically by the system. In particular, because it is expressed in the same language as hypotheses, one can potentially use the same or similar learning techniques for learning background knowledge as one uses for hypotheses.

Perhaps surprisingly, despite the promise of background knowledge as a means for shifting bias, the amount of work that has been done on cumulative learning in inductive programming is still relatively small. I discuss some reasons for this in Sect. 2.4.2. For now, let us review what work has been done in this area.

Khan et al. [1998] investigated a form of transfer learning in ILP under the name ‘repeat learning’. They used a problem domain of learning the definitions of legal moves in chess. Using the ILP system Progol, they showed that a new piece of background knowledge invented while learning the definition of a move for one chess piece (e.g. the knight), could improve the predictive accuracy of the definition learned for a second chess piece (e.g. the king). However, the method they used to invent new background knowledge required, for efficiency reasons, that significant information about the form of the invented piece of knowledge be specified by a human in advance (in the form of a type declaration – see Sect. 2.4.2). Furthermore, they did not demonstrate progression to more difficult problems, instead merely demonstrating an inductive transfer of knowledge between two problems of similar difficulty. As in studies of transfer learning in logistic regression and Bayesian networks discussed earlier, the only benefit from transfer learning that Khan et al. showed was an improvement in predictive accuracy when the amount of training data was sparse: once the amount of training data was increased significantly the shift in bias due to
inductive transfer no longer provided any advantage.

Davis and Domingos [2009] studied a technique for transfer learning in a probabilistic variant of ILP (known as Markov logic – see [Domingos et al., 2006] for an overview). Their means for shifting inductive bias consisted not of learning new background knowledge directly, but of learning ‘templates’ describing commonly occurring structural motifs in knowledge. These templates were expressed in an ad-hoc form of second-order logic (Markov logic itself is a form of first-order logic). The templates were learned in some source problem domain, and then used as ‘seeds’ for instantiating candidate hypotheses in a target problem domain. However, much like Khan et al. and others, no progression to more difficult problems was demonstrated, only inductive transfer between problems of similar difficulty. Furthermore, Moore and Danyluk [2010] later showed that much of the apparent benefit to predictive accuracy found by Davis and Domingos was not due to a shift in inductive bias from the source domain at all, and that this interpretation had been to some extent an artifact of their experimental method.

Schmidhuber et al. [1997] studied a technique for cumulative learning in inductive programming which they termed ‘adaptive Levin search’. In this approach, inductive bias is controlled by assigning a numerical weight to each primitive in the background knowledge; a high weight indicates that the primitive should be used more frequently when constructing hypothesis programs. As the system solves a succession of problems, these weights are incrementally updated according to how frequently each primitive occurs in successful hypotheses. In this way, the system becomes biased towards re-using elements of background knowledge that occurred in successful hypotheses in the past. Adaptive Levin search was shown to produce a performance improvement on a selection of problem sequences, one involving guiding an agent through a maze, and another involving synthesis of simple mathematical functions. Schmidhuber later followed up this work with a more complicated cumulative learning system called OOPS [Schmidhuber, 2004]. OOPS supported a weight-modification mechanism similar to that of adaptive Levin search, as well as an ability to invoke arbitrary chunks of program code from past successful hypotheses in solutions to new problems. However, in the problem sequence that Schmidhuber tested, which involved solving the ‘towers of Hanoi’ problem, only the weight modification mechanism was shown to provide a performance benefit.

In my MSc project [Henderson, 2010], I studied a simple approach to cumulative learning in IFP, using a brute-force search based inference system modelled on MagicHaskeller [Katayama, 2007]. The system was presented with sequences of related, but successively more difficult problems. To modify its inductive bias,
it incorporated each solution program into its background knowledge as it progressed through a sequence. In this way, solutions to earlier problems were available to be invoked as library procedures by solutions to later ones. I evaluated the system on four sequences of list processing problems; for example, in one of the sequences the system was tasked with learning a sorting algorithm via four intermediate problems including removing a given element from a list, and finding the smallest element in a list. On each problem sequence tested, the system solved the entire sequence with the help of these bias shifts at least thirty times faster than it took to solve just the final problem in the sequence with no bias shifts.

Schmidhuber’s adaptive Levin search and my approach of adding solution programs to the background knowledge can both produce an improvement in learning ability over a sequence of successively more difficult problems. In both cases, as discussed above, a system was shown to solve sequences of problems some orders of magnitude faster with bias-shifting enabled than without, albeit under controlled conditions. However, both techniques still have some quite severe limitations. Adaptive Levin search, though a useful approach, is on its own fundamentally limited because it shifts inductive bias only by modifying how a system chooses to use existing background knowledge; it provides no means for acquiring new knowledge. I believe that a technique like adaptive Levin search would therefore work best in combination with some other bias-shifting technique that does produce new knowledge.

As for the method of adding past solution programs to a system’s background knowledge, the problem here is that the success or failure of bias-shifting is very sensitive to the exact choice and order of problems within a sequence. Indeed, under this approach bias-shifting will only have a useful effect if solutions to later problems can be expressed directly in terms of earlier solutions; it is not enough for the problems simply to be related in some way, for example if their solutions would share some common structure. In this PhD project, the technique of abstraction invention is designed to overcome this limitation: instead of using solution programs themselves as background knowledge, it derives abstractions from common structure in groups of solution programs, and then use these abstractions as the new knowledge. In this way it bears some resemblance to the technique of Davis and Domingos discussed above, but with the advantage that the re-usable abstractions are represented as regular background knowledge rather than as ad-hoc ‘templates’. In this project I will also use a weight modification mechanism similar to adaptive Levin search in order to complement the abstraction invention technique.

Overall, what is missing from previous work in cumulative learning is a really
convincing demonstration of how knowledge acquired on one set of problems can give a system the ability to solve a whole new class of harder problems that it found intractable before. Most of the work in transfer learning, as discussed, is concerned with transferring knowledge between problems of similar difficulty in order to gain some improvements in predictive accuracy in the presence of sparse training data. Only the work of Schmidhuber and my MSc project involve progression through a sequence of problems of increasing difficulty, however both of these demonstrations were on a very small scale and demonstrate only basic proof of concept. In both cases the problem sequences were very short (four or five different problems at most), and furthermore the sequences were rather contrived: the choice and ordering of problems was designed by hand in order to be amenable to the particular bias-shift mechanisms being demonstrated.

2.2 Inductive Logic Programming

In the last section I argued that inductive programming is the most promising branch of machine learning within which to investigate cumulative learning, particularly due to its support for flexible specification of inductive bias in the form of background knowledge. In this and in the following section I review the field of inductive programming.

Inductive Logic Programming (ILP) is by far the largest and most mature branch of inductive programming. It can be characterised as the study of algorithms for automating inductive inference using first-order logic as a unified knowledge representation language. The roots of ILP lie in early work by Plotkin [1969, 1971] and Shapiro [1983], however the field became firmly established in the early 1990s [Muggleton, 1991]. ILP is unique in that it is the only branch of machine learning that has yet made a substantial practical attempt at automating truly generic inductive inference. In other areas of machine learning, different classes of inductive hypotheses require different learning algorithms (decision trees, feedforward neural networks, or Bayesian networks, for example). On the other hand, the focus in ILP is on developing algorithms that can deal with the general class of all effectively computable hypotheses. This is achieved by using a Turing-complete language, for which first-order logic has been the conventional choice, as a universal medium for representing all hypotheses, data, and other forms of knowledge. Furthermore, ILP systems possess a unique ability to accept detailed domain-specific inductive bias as input in the form of background knowledge, which is also expressed in the language of first-order logic. By providing different background knowledge, it is thus feasible to adapt a single generic ILP inference algorithm to a wide variety of application domains without ever having to modify the internal workings of the learning
algorithm itself. ILP has found significant practical application, particularly as a tool aiding scientific discovery in areas such as biochemistry [King et al., 1996; Turcotte et al., 2001], and also in engineering [Dolsak and Muggleton, 1992; Feng, 1992].

Perhaps the most valuable contribution of the field of ILP, above the design of individual inductive inference systems, has been the formulation of an overarching theoretical framework within which such designs can be understood and compared, and which, in partnership with empirical evaluation, guides the design of new, more effective inference systems. In the next few subsections I review the main ideas of this framework that are relevant to this thesis: inverse deduction, refinement, and proof-directed search. One of the main points of this thesis is that these techniques, though they were developed within first-order logic based ILP, can in fact be seen to transcend first-order logic, and are readily transferable intact to other languages such as lambda calculus. In this PhD project I adapt the two techniques of refinement and proof-directed search to lambda calculus in RUFUS, and I adapt the technique of inverse deduction to lambda calculus in KANDINSKY.

One area of ILP that has particular relevance to this project is predicate invention, which concerns the automatic invention of novel background knowledge concepts. I defer a discussion of predicate invention until Sect. 2.4.2. In the next four subsections I start with an overview of the basic principles of ILP, followed by a review of some main techniques for inferring hypotheses from data.

Note that due to the large size of the field of ILP, what follows is not an exhaustive account of the state of the art; rather, I focus in detail on techniques that are relevant to this thesis. In particular I do not cover non-monotonic learning or probabilistic logic representations. For a comprehensive review of the field of ILP including these topics see [Muggleton et al., 2012a].

2.2.1 Main Principles

In the standard setting for learning in ILP [Muggleton and De Raedt, 1994], the goal is to infer a hypothesis that, in combination with background knowledge, correctly predicts some observed data. See Fig. 2.1 for a worked example. Hypotheses, background knowledge, and observed data are all expressed in first-order logic, typically in the form of Horn clauses. The observed data consists of positive and negative examples for a target predicate, specifying instances where the target predicate is true or false respectively. A hypothesis takes the form of a definition for the target predicate, and the background knowledge takes the form of a set of definitions for supplementary predicates.

We say that a hypothesis covers an example if it and the background knowl-
Positive and negative examples:
reverse([], []).
reverse([1, 2, 3], [3, 2, 1]).
:- reverse([], [5]).
:- reverse([1, 2, 3], [1, 2, 3]).

Background knowledge:
nil([]).
cons(H, T, [H|T]).
append_elem([], X, [X]).
append_elem([H|T], X, [H|Y]) :- append_elem(T, X, Y).

Mode declarations:
modeh(reverse(+list, -list)).
modeb(reverse(+list, -list)).
modeb(nil(+list)).
modeb(cons(-int, -list, +list)).
modeb(append_elem(+list, +int, -list)).

A correct hypothesis:
reverse(X, X) :- nil(X).
reverse(X, Y) :- cons(H, T, X), reverse(T, T2), append_elem(T2, H, Y).

Figure 2.1: An inductive inference problem framed in the ILP setting for learning, in which the aim is to synthesise a logic program that reverses a list of numbers. The examples, background knowledge, and mode declarations constitute the problem specification. The hypothesis shown is one possible solution to the problem. Following ILP notational convention, negative examples are written as headless clauses. In mode declarations: \texttt{modeh} means that the predicate is allowed to appear in a head of a hypothesised clause while \texttt{modeb} means that it may appear in a body; +/- symbols indicate input/output modes respectively; \texttt{list/int} are argument types.
edge together logically entail that example. We may speak of the coverage of a hypothesis to refer to the numbers of positive/negative examples that it covers. The aim in the ILP setting is as follows: given some background knowledge and examples, find a hypothesis that covers all of the positive examples and none of the negatives.

It is standard in ILP to constrain the hypothesis language using static type declarations called mode declarations. These assign types to the arguments of predicates, as well as input/output modes, which give predicates an operational interpretation as functions from inputs to outputs.

To choose between multiple hypotheses that have the same coverage with respect to the examples, it is typical to prefer the simplest hypothesis, i.e. whichever one has the shortest syntactic description. This is a form of the Occam’s razor principle from the natural sciences. It be understood as equivalent to using a Bayesian prior that assigns higher probability to shorter hypotheses [Muggleton and De Raedt, 1994].

When one wants to take into account both the coverage and size of a hypothesis simultaneously, it is common in ILP to use a compression measure as follows:

\[
\text{compression} = \text{no. positive examples covered} - \text{no. negative examples covered} - \text{size of hypothesis}
\] (2.1)

The size of a hypothesis is usually taken to be the number of literals it contains. Compression can be understood as a Bayesian log-posterior, with the coverage of examples corresponding to log-likelihood and the size of the hypothesis corresponding to log-prior [Muggleton and De Raedt, 1994; Muggleton, 1995]. This compression measure is very important in ILP because it is the standard ‘goodness of fit’ measure for hypotheses, and can be used as an objective function to guide a heuristic search.

### 2.2.2 Inverse Deduction

A fundamental idea to have come out of ILP is the principle of inverse deduction, which may be stated as follows: an inductive inference process can be created from a deductive inference process by running it in reverse. To see why this should be the case, recall in the setting for learning for ILP that the goal is to find a hypothesis that predicts observed data. Now, this ‘prediction’ occurs by some process of logical deduction, i.e. we deduce the predicted observation from the hypothesis. Therefore, it seems reasonable that if we could run deduction backwards, we could ‘anti-deduce’ a suitable hypothesis from some observations. Indeed, it is quite possible to do just that.
Deductive inference in first-order logic uses the \textit{resolution} deduction rule [Nienhuys-Cheng and de Wolf, 1997]. Muggleton and Buntine [1988] proposed the idea of \textit{inverse resolution}. They introduced two pairs of primitive operators for inverting resolution steps, the so-called “V” and “W” operators. A “V” operator is an inversion of a single binary resolution step. Now, a single deductive resolution step takes two clauses $A, B$ as input, resolves on a positive literal in $A$ and a corresponding negative literal in $B$, and yields an output clause $C$ that is a logical consequence of $A$ and $B$:

$$A, B \rightarrow C$$

Each of the “V” operators inverts this transformation with respect to one of the inputs. Thus, for first “V” operator, known as \textit{absorption}, clause $B$ becomes the output:

$$A, C \rightarrow B$$

For the second “V” operator, known as \textit{identification}, clause $A$ becomes the output:

$$B, C \rightarrow A$$

Both of these operators perform generalisation, and the absorption operator in particular is capable of generalising a recursive clause from initially non-recursive rules. See Fig. 2.2 for a concrete example of these operators.

The idea behind the “W” operators is that, to ‘complete the set’ of inverse resolution operators, one would like to invert a resolution step with respect to both inputs:

$$C \rightarrow A, B$$

However, on its own, such a transformation is highly non-deterministic, and what’s more it is usually never compressive. To achieve a more constrained, compressive inversion with respect to both arguments, Muggleton’s solution
was to perform two or more such transformations simultaneously, and to have one of the outputs shared by both steps. When \( A \) (in which resolution occurs on a positive literal) is shared, the operator is called \textit{inter-construction}:

\[
C_1 \rightarrow A, B_1 \\
C_2 \rightarrow A, B_2
\]

When \( B \) (in which resolution occurs on a negative literal) is shared, the operator is called \textit{intra-construction}:

\[
C_1 \rightarrow A_1, B \\
C_2 \rightarrow A_2, B
\]

Due to the sharing, it is now possible for these transformations to result in compression. See Fig. 2.3 for a concrete example. Unlike the “\( V \)” operators, the “\( W \)” operators do not perform generalisation of existing predicates, but rather they re-express them without generalisation in terms of new auxiliary predicates, a process known as \textit{predicate invention}.

Muggleton and Buntine implemented the absorption and intra-construction operators in a system called CIGOL. CIGOL was capable of learning recursive logic programs from examples by means of successively applying these operators. It did this by means of a greedy search, in which the most compressive transformation that was consistent with user-supplied positive and negative examples was chosen at each step. CIGOL was capable of inventing auxiliary predicates during the learning process. For example, when learning a predicate to reverse a list, it invented the ‘append’ predicate using intra-construction, as an auxiliary step.

However, CIGOL’s implementation of absorption and intra-construction was somewhat constrained. In particular, in order to apply the absorption operator, the input clause \( A \) (which contains the positive resolved literal) had to consist of only a single literal. Furthermore, for intra-construction the output clauses \( A_i \) (again containing the positive resolved literals) also had to consist of a single literal each. The motivation for this constraint was to cut down the high amount of non-determinism in these operators, and hence reduce the size of CIGOL’s search space. Rouveirol [1992] later lifted this constraint while still keeping the degree of non-determinism manageable by using a ‘flattened’ representation (clauses are initially pre-processed such that they do not contain any function symbols, as in Figs. 2.2 and 2.3). Neither Muggleton and Buntine nor Rouveirol implemented the inter-construction operator, though it was implemented by Wogulis and Langley [1989] in the RINCON system. RINCON was not an inductive learning system, however; it’s purpose was to compress a body of
a).

\[ \text{sparrow}(X) := \text{bird}(X), \text{small}(X), \text{brown}(X). \]
\[ \text{crow}(X) := \text{bird}(X), \text{large}(X), \text{black}(X). \]
\[ \text{sparrow}(X) := \text{wings}(X), \text{lays_eggs}(X), \text{small}(X), \text{brown}(X). \]
\[ \text{crow}(X) := \text{wings}(X), \text{lays_eggs}(X), \text{large}(X), \text{black}(X). \]

b).

\[ \text{uncle}(X, Y) := \text{brother}(X, Z), \text{mother}(Z, Y). \]
\[ \text{uncle}(X, Y) := \text{brother}(X, Z), \text{parent}(Z, Y). \]
\[ \text{uncle}(X, Y) := \text{brother}(X, Z), \text{father}(Z, Y). \]

Figure 2.3: Examples of the inverse resolution “W” operators a) inter-construction and b) intra-construction. Inter-construction takes \( C_1, C_2 \) as input and outputs \( A, B_1, B_2 \). Intra-construction takes \( C_1, C_2 \) as input and outputs \( A_1, A_2, B \). Both operators result in the invention of a new predicate (the literals of that predicate are indicated with boxes). Notice that in the case of inter-construction, the clause body of the invented predicate consists of literals that are common to both \( C_1 \) and \( C_2 \), whereas in intra-construction the clause bodies of the invented predicate consist of literals that represent the differences between \( C_1 \) and \( C_2 \). Notice also that in the case of intra-construction \( C_1 \) and \( C_2 \) must be clauses of the same predicate, whereas for inter-construction this is not necessary. Finally, it is worth pointing out that the English names ‘bird’ and ‘parent’ in this example would of course not be inferred as part of the inverse resolution process; the system would instead either create uninformative predicate names such as ‘concept1’, ‘concept2’, or alternatively it could request that a human user inspect the invented predicates and assign appropriate names to them.
knowledge, via inventing auxiliary concepts, into a form that allowed deductive inference to be performed from it more efficiently.

In inverse resolution, there tends to be a large search space of possibilities even for a single application of a “V” or “W” operator: there are many combinations of clauses, many literals to anti-resolve on, and many anti-substitutions of terms or variables to make. As a result, inverse resolution is usually only computationally feasible when performed greedily, what Muggleton [1995] called local generalisation. In other words, you always pick whichever inverse resolution step has the greatest immediate compression, and you never look ahead more than one step in the tree of all possible sequences of “V”/“W” operator applications. However, as Muggleton pointed out, there are many situations in which this greedy policy will fail. For example, it might be the case that the first three steps in a sequence of operator applications produces no compression, but then a high degree of compression occurs on the fourth step. Greedy search will not find such sequences, yet exhaustive search of all sequences of steps has an intractably large search space, particularly in the presence of a significant amount of background knowledge. Due to these problems, inverse resolution has lost favour since the 1990s as a primary means for inductive generalisation in ILP. Techniques based on refinement, which I discuss in the next section, have become much more popular.

Despite inverse resolution’s problems of tractability outside of a greedy search, it is certainly not a technique to be forgotten about or ignored. The reason for this is that unlike most other ILP techniques developed since, inverse resolution is capable of predicate invention, i.e. its inter- and intra-construction operators provide a means for a learning system to invent entirely novel auxiliary concepts of its own accord, as part of the inference process. Predicate invention is strongly related to abstraction invention, and is a potential way to learn new background knowledge through experience. In Sect. 2.4 I discuss predicate invention in detail, so I shall come back to inverse resolution in that context then.

2.2.3 Refinement

In this subsection I describe a popular approach to ILP that is rather different from the inverse deduction technique outlined in the last subsection. This different approach is of a ‘generate-and-test’ nature. Recall that in inverse deduction, hypotheses are derived directly from the examples by means of a reverse proof. Hence, such hypotheses are guaranteed to logically entail the examples by construction. On the other hand, in the ‘generate-and-test’ paradigm the idea is to generate hypotheses in some more arbitrary way, without any ‘by construction’
guarantee that they cover the examples, and then to filter these hypotheses for coverage of the examples by testing the predictions of each one using forward deduction. Due to its support for more direct methods of hypothesis construction, the ‘generate-and-test’ approach often has a much more manageable search complexity than inverse deduction.

The theory of refinement and refinement operators [Nienhuys-Cheng and de Wolf, 1997, Chap. 17], originally due to Shapiro [1983], provides a principled framework within which to design search algorithms for ‘generate-and-test’ style inductive programming that have a number of desirable features, in particular:

- **Guidance:** rather than simply enumerating hypotheses in an order determined a-priori, the search algorithm makes use of information about coverage of the examples of intermediate hypotheses in order to guide the search incrementally towards more promising regions of the hypothesis space.

- **Pruning:** large regions of the hypothesis space can be proven in advance to contain no plausible hypotheses that are consistent with the observed data, hence those regions can be pruned from the search.

The technique of refinement requires two structures to be defined with respect to the hypothesis space of interest: a refinement operator and a generality ordering. A refinement operator is a function mapping each hypothesis to a set of successor hypotheses. Equivalently, one can think of this function as defining a (possibly infinite) directed graph in which the hypotheses are the nodes; hence one often refers to the structure induced by a refinement operator as a refinement graph. A generality ordering is a quasi-order\(^1\) over the set of hypotheses, defined such that hypotheses that are ‘more general’ according to the ordering make stronger predictions about what data one will observe.\(^2\)

Usually in refinement, the generality ordering comes from established theory and therefore imposes a natural and informative structure over the space of hypotheses. In the case of first-order logic the generality ordering of theta-subsumption [Nienhuys-Cheng and de Wolf, 1997] is frequently used in the context of refinement of clauses. The generality of a hypothesis is closely related to its coverage of examples, and hence to its compression. In this way, choosing a refinement graph that closely follows the structure of a generality ordering such as theta-subsumption is often a good design strategy for making the search space amenable to heuristic guidance by compression.

\(^1\)A reflexive, transitive binary relation.

\(^2\)In essence, this is the meaning of generality: a more general hypothesis makes stronger predictions about what data one will observe. Inductive inference is seen as a process of generalisation from observed data to a hypothesis, because a useful hypothesis not only correctly predicts the data observed so far but it also predicts the outcome of future observations.
It is important for a refinement operator to exhibit a property called *soundness* with respect to the generality ordering. In the case of a so-called *upward* refinement operator, soundness means that refining a hypothesis always yields successors that are more or equally general. For a *downward* refinement operator, successors are always less or equally general. Soundness enables pruning, because it allows reasoning like the following: suppose that a hypothesis is reached during a search through the refinement graph that is found to make incorrect predictions about observed data. If we have, say, an upward refinement operator that we know is sound, then by the definition of generality we also know that all the descendants of this hypothesis (those than can be reached from it by further applications of the refinement operator) must also make the same incorrect predictions. Therefore, all the descendants can immediately be pruned from the search space, avoiding a potentially large tract of fruitless search. In the case of a downward refinement operator arguments along similar lines can also be made.

Two of the most well-known and successful ILP implementations, FOIL [Quinlan, 1990] and Progol [Muggleton, 1995], use the technique of refinement to search a space of first-order clauses. Both use downward refinement operators under the generality ordering of theta-subsumption, and both take advantage of coverage-directed guidance as well as pruning.

2.2.4 Multiple Refinement Points and Proof-Directed Search

In the context of refinement, it may often be the case that hypotheses consist of multiple components which can in effect be refined independently. In ILP hypotheses typically consist of multiple *clauses*, each clause being an individual logical fact or rule. However, this poses a significant problem to the tractability of refinement-based search: how does one choose which subset of the clauses in a hypothesis to refine, and in what order? To try all possibilities will usually involve a combinatorially large number of choices.

The tractability of refining multiple clauses simultaneously has proved such a significant problem that until recently, most refinement-based ILP systems were so-called *single-clause learners*, of which prominent examples are FOIL [Quinlan, 1990] and Progol [Muggleton, 1995]. What characterises a single-clause learner is that it will always refine one clause to completion before moving on to the next. Furthermore, each successive clause must of itself make new predictions about observed data in order to be accepted. Though this is an effective technique for keeping the search space down to a manageable complexity, it comes at a price: single-clause learners are *incomplete*, in that there are
many useful hypotheses that they simply cannot find. Specifically, single clause learners cannot discover hypotheses that contain dependencies between clauses such that asserting any one of the clauses alone does not make new predictions about observed data, whereas asserting two or more clauses simultaneously does make new predictions. Muggleton et al. [2012b] give a good example of such a hypothesis; a simplified version of their example is given here in Fig. 2.4.

Since the late 1990s there has been an effort in ILP to develop multi-clause learners [Bratko, 1999; Inoue, 2004; Corapi et al., 2010]. These systems generate whole groups of clauses at once in order to explain observed data, and hence are not subject to the incompleteness suffered by single-clause learners. However, such systems must employ clever search techniques if they are to remain efficient in the face of the inevitable larger complexity of their search spaces. One recent technique, developed in the TopLog family of ILP systems [Muggleton et al., 2008, 2012b]3, we can characterise under the name proof-directed search.

The idea behind proof-directed search is as follows. Starting with an under-specified hypothesis in need of refinement, one attempts to prove, using some mechanical deduction procedure, that this hypothesis correctly predicts observed data. Now, because the hypothesis is under-specified it will be necessary to refine parts of it into order to make progress with the proof. The trick is that the proof procedure itself selects which parts of the hypothesis need to be refined, and in what order. Thus, the choice of which part of the hypothesis to refine at any given moment does not have to be made arbitrarily or non-deterministically, avoiding a good deal of the extra combinatorial search usually associated with multiple refinement points and, crucially, without sacrificing completeness.

Let us now look in a bit more detail at the mechanism of the TopLog family of systems (this shall be relevant to the design of the RUFUS system in Chap. 5, which also uses a proof-directed search). In these systems the hypothesis space is defined by means of a logic program called a top theory, which consists of two sets of clauses $\top_{\text{non-terminal}}$ and $\top_{\text{terminal}}$ (see Fig. 2.5 for an example). $\top_{\text{non-terminal}}$ can be viewed as a declarative specification for a refinement operator, whereas $\top_{\text{terminal}}$ is an initial underspecified hypothesis, ready to be refined. Each non-terminal literal (prefixed by a ‘$’ in Fig. 2.5) in the body of a clause in $\top_{\text{terminal}}$ represents a point at which refinement can occur. The refinement graph is defined using first-order logic’s resolution deduction rule as follows. To refine a hypothesis (initially $\top_{\text{terminal}}$), transform one of its clauses by resolving a non-terminal literal in the body of that clause with the head of a clause in $\top_{\text{non-terminal}}$. It is a direct consequence of the soundness of resolution [Nienhuys-Cheng and de Wolf, 1997] that this refinement operator is sound with respect

---

3The two systems cited here are respectively TopLog and MC-TopLog. TopLog was the seminal work, though not actually a multi-clause learner. MC-TopLog is a multi-clause learner.
Positive and negative examples:

sentence([a, small, dog, eats, the, tasty, food], []).  
:- sentence([small, eats, dog, a, food, tasty], []).  
etc...

Background knowledge:

article([a|X], X).  
article([the|X], X).  
adjective([small|X], X).  
adjective([tasty|X], X).  
noun([dog|X], X).  
noun([food|X], X).  
verb([eats|X], X).  
etc...

Hypothesis:

sentence(A, D) :- noun_phrase(A, B), verb(B, C),  
noun_phrase(C, D).  
noun_phrase(A, D) :- article(A, B), adjective(B, C), noun(C, D).

Figure 2.4: In this ILP grammar learning task similar to one described by Muggleton et al. [2012b], the hypothesis that is shown cannot be found by single-clause learners such as FOIL or Progol. The target predicate sentence is a parser for sentences, of which positive and negative examples are given along with background knowledge. The hypothesis consists of two clauses, one for the target predicate and the other for a helper predicate noun_phrase. It is clear from the relationship between these two clauses that neither clause alone is enough to cover any examples, yet both clauses together will cover the given positive example.
to the generality ordering of logical entailment relative to $\top_{\text{non-terminal}}$.

TopLog’s proof-directed search procedure itself works like this: an attempt is made to prove one or more examples from the top theory and background knowledge using the mechanical deduction procedure $SLD$-resolution, the constrained form of resolution commonly used by implementations of the Prolog programming language. A hypothesis is not actually refined explicitly step-by-step, but rather an entire proof of the examples is constructed, then a final hypothesis is extracted from it at the end by identifying which steps in the proof correspond to refinements (i.e. resolutions of non-terminal literals). The SLD-resolution procedure will backtrack to generate multiple proofs and hence multiple alternative hypotheses. Also, pruning of regions of the search space that are inconsistent with the examples occurs as a natural consequence of the proof process.

Note that outside of inductive programming, a technique very similar to proof-directed search has been used in the context of the Haskell programming language to improve the efficiency of automated property testing of programs [Runciman et al., 2008; Allwood et al., 2011]. Here, the object being refined is not itself a program, but rather a data structure serving as a test case for a program.
2.3 Other Inductive Programming Techniques

ILP is quite a unified field in the sense that there is a strong consensus on its main guiding principles. However, in other areas of inductive programming outside of ILP such as Inductive Functional Programming (IFP), the range of techniques available is somewhat more of a mish-mash. Rather than give an overview of everything available, in this section I describe certain inductive programming techniques from outside of ILP that are relevant to this thesis. For a discussion of alternative techniques that I do not build on directly in this project see Sect. 8.5.

2.3.1 Higher-order Background Knowledge as Declarative Bias

It is often useful in inductive programming to be able to specify a strong inductive bias in declarative form, in order to customise a system to some problem domain. Such a ‘declarative bias’ provides a specification of a hypothesis space, and indeed background knowledge and associated type declarations are one form of declarative bias. However, in first-order ILP, background knowledge and type declarations alone are often not flexible enough to provide the kind of strong bias required for some applications. Thus, many ILP systems provide special mechanisms, separate from background knowledge, for specifying declarative bias.

For example, in the TopLog family of ILP systems which was discussed in Sect. 2.2.4, the top theory provides such a means: one can use it to specify a custom refinement operator, and hence achieve much more flexibility in controlling the structure of the hypothesis space than is possible with background knowledge and mode declarations alone. In another ILP technique known as the schema-directed approach [Flener, 1995, 1997], program templates called ‘schemas’ are used to create a strong bias towards hypotheses that conform to particular structural or recursive patterns. Flener gives an example of a ‘divide-and-conquer schema’, which provides a template from which a number of recursive programs including sorting algorithms may be instantiated.

Though specifying a strong bias in such ways can be useful for applying inductive programming to specific applications, these domain-specific biases typically must be provided by human experts. On the other hand, for cumulative learning, one would wish an inductive programming system to be able to automatically learn its own strong bias towards a problem domain, starting from only a very weak initial bias. Additional bias-specification mechanisms such as top theories or program schemas are therefore not ideal, because in order to obtain the full benefits of bias shifting in learning from experience, a system
would need a mechanism for learning its own top theory or learning its own program schemas, in addition to whatever mechanism it has for learning new background knowledge. Overall it is rather awkward from a cumulative learning point of view to be representing declarative bias in such a multitude of forms.

As an alternative approach, let us ask the following question: is it possible to obtain greater flexibility in the way bias can be specified purely through background knowledge and type declarations? As we shall see, if one is able to switch to a knowledge representation language that supports first-class functions, then the answer to this question is ‘yes’.

Katayama [2007] demonstrated the power of higher-order functions in background knowledge as a means for specifying a strong declarative bias. A higher-order function is a function that takes one or more other functions as arguments. Though higher-order functions cannot be represented in first-order logic, they can be represented in any knowledge representation language that supports first-class functions, such as lambda calculus.

One can use higher-order functions to encapsulate high-level structural patterns in programs. For example, the well-known operations ‘map’ and ‘fold’ from functional programming are higher-order functions, each representing a particular high-level pattern of recursion. Indeed, higher-order functions can often be viewed as program templates much like in the ‘schema-directed approach’ to ILP; however, now the templates are represented as regular background knowledge rather than requiring an additional ad-hoc formalism.

Katayama’s inductive programming system MagicHaskeller uses lambda calculus as its knowledge representation language, and hence supports the use of higher-order background knowledge. MagicHaskeller searches for functions to fit input-output examples using a brute-force generate-and-test search through a space of strongly-typed lambda expressions. MagicHaskeller was demonstrated in a ‘program synthesis’ domain, synthesising list-processing programs such as list reversal, nth element of a list, etc. It made use of higher-order functions known as morphisms [Augusteijn, 1998] in the background knowledge in order to provide an appropriate domain-specific bias. Katayama showed that, by making use of strong declarative bias in the form of this higher-order background knowledge, a system such as MagicHaskeller that generates hypotheses using a simple brute-force search can synthesise a variety of useful programs at a speed competitive with other inductive programming systems that use much more advanced, data-guided search mechanisms.

In this thesis, I take Katayama’s work further in two ways. Firstly, I adapt ILP’s technique of refinement to work with lambda calculus (Chap. 5), in order get the benefits of both an efficient guided search and the flexibility of higher-
order background knowledge. Secondly, I develop an abstraction invention procedure (Chap. 7) capable of learning new background knowledge, higher-order or otherwise, from experience. The aim is thus to enable an inductive programming system to adapt to a problem domain by automatically acquiring an appropriate strong bias, rather than this bias needing to be provided a priori by a human user.

2.3.2 Non-termination and Levin Search

Any inductive programming system that supports a Turing-complete hypothesis language and deductively computes the predictions of candidate hypotheses as part of its inference algorithm must have some policy for dealing with potentially non-terminating hypothesis programs. It is impossible to prove in general whether an arbitrary program will terminate for given input, due to the halting problem of Turing. Therefore, in order to avoid incompleteness, i.e. a situation where there exist some computable functions that a system cannot hypothesise, a system must be prepared to test candidate hypotheses for arbitrarily long runtimes.

The majority of work in ILP and other areas of inductive programming has tended to side-step this issue of potentially non-terminating programs simply by living with incompleteness. This is usually done in one of two ways. Either, the hypothesis space is restricted to one containing only provably terminating programs (for example, MagicHaskeller [Katayama, 2007]). Alternatively, some pre-set limit on the number of computation steps for which any hypothesis may be tested is used, and any hypothesis that exceeds this limit is simply discarded (for example, Progol [Muggleton, 1995] and HYPER [Bratko, 1999]).

However, there are big drawbacks to both of these options. Limiting the hypothesis space to some particular class of provably terminating programs inevitably involves making rather strong a priori assumptions about what class of hypotheses are suitable for solving the problem at hand. Such strong assumptions are at odds with the goals of cumulative learning, in which the aim is to start off with only weak initial assumptions, and to have the system itself learn about the nature of the problem domain from its own exploration.

On the other hand, using a pre-set step limit is also an inadequate solution when one knows little about the problem domain a priori. This is because it is impossible to choose a suitable value for the step limit without making strong assumptions about the number of computation steps required to test the target hypothesis. If one gets it wrong and sets the step limit too low then the system will fail to find the target hypothesis at all. On the other hand, if one sets the step limit too high then much time will be wasted unnecessarily testing
non-terminating hypotheses. In the cases of Progol and HYPER, this issue is
dealt with by relying on a human user knowing something about the nature of
the target hypothesis and choosing a step limit on a problem-by-problem basis.
However, if one is to make a convincing demonstration of cumulative learning,
then such a priori assumptions cannot be made.

Fortunately, there does exist a practical technique for overcoming the non-
termination problem that neither sacrifices completeness nor requires making
strong assumptions about the nature of the target hypothesis. This technique,
which appears to be somewhat little-known in the ILP community, is known as
*Levin search* or *universal search*. It was originally postulated by Levin [1973] as a
theoretical device for solving a problem in algorithmic complexity theory, how-
ever it was later recognised [Schmidhuber et al., 1997; Schmidhuber, 2004] for its
potential as a practical hypothesis search algorithm in inductive programming.

Schmidhuber [2004] gives an example of a Levin search procedure in the
form of a brute-force generate-and-test algorithm, which I shall reproduce here
(modified slightly for the sake of clarity). The algorithm requires that some
prior probability distribution \( p(x) \) is defined over one’s hypothesis space in such
a way that one can easily enumerate all hypotheses \( x \) in descending order of
probability \( p(x) \).\(^4\) Its description is as follows:

For \( N = 1, 2, 4, 8, 16 \ldots \): generate each hypothesis \( x \) for which
\[ Np(x) \geq 1 \] and test it for at most \( Np(x) \) steps.

Notice that the Levin search procedure consists of an iterative deepening search.
We can make the following remarks about it:

- The value \( N \), which doubles at each iteration, is an upper bound on the
total number of computation steps spent testing hypotheses on that iter-
ation.

- At each iteration, the number of hypotheses generated is finite; indeed it
is at most \( N \).

- At each iteration, every possible hypothesis is tested either to termination
or for a number of steps proportional to its prior probability, within one
step.

Schmidhuber calls this last property *bias-optimality*, i.e. the algorithm allocates
computation time to each hypothesis in proportion to its prior probability. It is
also relatively straightforward to see that Levin search is *complete* in the follow-
ing sense: given any terminating hypothesis with non-zero prior probability, the

\(^4\)Note that this is not a difficult requirement to satisfy, particularly if one considers that in
inductive programming one typically uses a prior on hypotheses that decreases monotonically
with their syntactic size.
Levin search algorithm will have tested it to termination (and hence discovered whether or not it is a solution to the inductive inference problem) within a finite amount of time.

Schmidhuber demonstrated Levin search in a brute-force form as given above. Of course, any brute-force search algorithm is going to be strongly limited due to the combinatorial nature of hypothesis spaces typical to inductive programming. Therefore, in this thesis (Chap. 5) I shall combine Levin search with the proof-directed refinement technique of ILP discussed earlier, in order to obtain an efficient, informed search without compromising on Turing-completeness, in spite of potentially non-terminating hypotheses.

2.4 Abstraction Invention

In Chap. 1 I introduced abstraction invention as the process of automatically deriving new, re-usable concepts from repetitive patterns in existing knowledge. I develop a form of abstraction invention in this PhD project as a mechanism for learning background knowledge from experience. Abstraction invention mimics the process of abstraction used by software engineers in order to build up a library of re-usable procedures and data structures. In this section I review work that is related to abstraction invention, in particular anti-unification, a well-known mechanical means for abstracting a common template from a group of syntactically similar programs, and predicate invention, an area of ILP that intersects with abstraction invention and concerns the automatic construction of new predicates.

2.4.1 Anti-unification as a Means of Abstraction

Abelson and Sussman [1996, Sect. 1.3.1], and also Felleisen et al. [2001, Sect. 21.1], identify a standard ‘recipe’ which human programmers often use to derive abstractions from syntactic patterns in functional programs. Felleisen et al. describe this recipe explicitly:

“[Given two similar function definitions], we compare them and mark the differences with boxes. . . . Next we replace the contents of corresponding pairs of boxes with new names and add these names to the parameter list.”

As an example, consider the following pair of functional programs, the first of which increments all the elements in a list, and the second of which decrements all the elements in a list:
incElems \ lst = \begin{cases} \text{nil} & \text{if (null \ lst)} \\ \text{(cons \ \boxed{\text{inc}} (\text{head \ lst}))} & \text{incElems \ (\text{tail \ lst}))} \end{cases}

decElems \ lst = \begin{cases} \text{nil} & \text{if (null \ lst)} \\ \text{(cons \ \boxed{\text{dec}} (\text{head \ lst}))} & \text{decElems \ (\text{tail \ lst}))} \end{cases}

These programs only differ at a single location, which we have marked with a box. Following the recipe, if we replace the contents of the box with a new variable f, and then make f a new function parameter, we obtain the following abstraction:

\text{map} \ f \ \text{lst} = \begin{cases} \text{nil} & \text{if (null \ lst)} \\ \text{(cons \ f (\text{head \ lst}))} & \text{(map \ f \ (\text{tail \ lst}))} \end{cases}

\text{incElems} = \text{map} \ \text{inc} \\
\text{decElems} = \text{map} \ \text{dec}

\text{map} \text{ encapsulates the pattern of applying some function f to each element of a list. Also shown above are the two programs incElems and decElems now re-expressed concisely in terms of the map abstraction.}

The above informal abstraction recipe is recognisable as a form of anti-unification. In anti-unification, one obtains a generalisation or schema from a set of tree-structured terms by pattern matching from the root down, replacing any mismatch points with new variables. Automatic anti-unification of terms in first-order logic is generally regarded as a solved problem: a widely-used algorithm was first discovered by Reynolds [1969] and Plotkin [1969]. On the other hand, anti-unification of lambda calculus terms, as would be needed to automate the above recipe, is not quite so straightforward. The main issue lies in deciding how to correctly handle lambda parameters and bound variables.

Some algorithms for anti-unification of lambda terms have been proposed, though with various different design goals in mind. For example, Feng and Muggleton [1992] gave one that applies only to a restricted subset of lambda calculus; they made this restriction in order to guarantee that their algorithm will produce unique ‘least general generalisations’ when terms are interpreted as higher-order logic formulae. A similar approach was also taken by Schmid
et al. [2001]. On the other hand, Bakewell and Runciman [1999] formulated an anti-unification algorithm that derives procedural abstractions from function definitions expressed in a Haskell-like language with a significantly more complicated syntax than pure lambda calculus. Their algorithm is essentially a straightforward automation of the abstraction recipe of Abelson and Sussman/Felleisen et al. given above.

From an abstraction invention point of view, the main limitation of anti-unification algorithms such as that of Bakewell and Runciman and others is that they are designed only to anti-unify whole terms. On the other hand, a truly flexible abstraction invention algorithm should be able to make abstractions from commonality between arbitrary subterms of a larger program. To this end, in Chap. 6 of this thesis I design an anti-unification search algorithm, which performs a heuristic search for a common pattern over all possible combinations of subterms of a lambda calculus term.

### 2.4.2 Predicate Invention in Inductive Logic Programming

*Predicate invention* [Stahl, 1996] in ILP refers to any process by which novel background knowledge predicates are introduced automatically during the learning process. These are predicates for which no partial specification or set of examples has been given in advance; the learning system simply decides of its own accord that a new concept is required to better express its knowledge. The process is analogous to ‘coining a new word’ in human language.

Particularly in the early 1990s, there was significant effort within the ILP community to develop practical techniques for predicate invention. However, different studies of predicate invention had rather different goals in mind. Some approaches focused on restructuring a body of knowledge without generalisation, either for the purpose of making it more ‘meaningful’, i.e. more understandable to humans [Flach, 1993], or for improving the computational efficiency with which deductions could be made from it [Wogulis and Langley, 1989]. In contrast, other studies focused on inventing auxiliary predicates as an intermediate step during the generalisation of an inductive hypothesis from examples. Permitting such auxiliary predicates changes the language bias, which can allow some target hypotheses to be expressed more compactly and hence enable more efficient learning and improve predictive accuracy [Muggleton and Buntine, 1988; Bain and Muggleton, 1991; Wrobel, 1994; Leban et al., 2008; Muggleton et al., 2012c]. Furthermore, in the case of some recursive target concepts, expression of the concepts is not possible at all in first-order logic without the introduction of recursive auxiliary predicates [Flener, 1995].

Perhaps surprisingly, there have been only a small number of studies of
predicate invention as an explicit mechanism for cumulative or transfer learning (i.e. using predicates invented while solving one set of problems to help solve another); one of the few notable investigations along these lines is by Khan et al. [1998]. To try to see why this has been the case, in the rest of this subsection I shall compare the predicate invention techniques mentioned above from the point of view their suitability for cumulative learning. Later I shall argue that predicate invention as a cumulative learning mechanism in ILP is limited by certain constraints of the language of first-order logic, which may explain why studies of cumulative learning in ILP have been somewhat few and far between.

The predicate invention techniques most relevant to cumulative learning are those that are likely to produce re-usable predicates. In such techniques, the decision procedure for choosing which predicates to invent should favour those for which there is some evidence that they will be re-usable. It might seem reasonable that compression can provide such evidence, because compression often occurs as a result of re-use (the re-used predicate then constitutes an abstraction). However, compression per se is not always a strong indicator of re-use or re-usability. To see why, consider the inverse resolution “W” operators described in Sect. 2.2.2: inter-construction and intra-construction. Both of these operators perform compression-guided predicate invention. In the case of inter-construction, the invented predicate is derived from commonality in two or more clauses of a knowledge base, and applying the inter-construction transformation results in the new predicate being used in at least two places. Thus, the compression produced by inter-construction is a direct result of there being multiple uses of the invented predicate, which given reasonable prior assumptions is a good indicator of further re-usability in the future. On the other hand, in the case of intra-construction, the invented predicate is derived from differences in two clauses of the same concept, and applying the intra-construction transformation only results in a single use of the invented predicate. Indeed, it appears that intra-construction is essentially a way of getting round a constraint used in ILP that knowledge must be expressed in conjunctive normal form (i.e. as a conjunction of clauses, where each clause is a disjunction of literals). If it were not for this constraint, then any intra-construction transformation could be expressed with at least as much compression without predicate invention simply by introducing a disjunction within a clause body. In the ‘uncle’ example of Fig. 2.3b, this would yield the following (a semicolon represents disjunction, as in standard Prolog syntax):

\[
\text{uncle}(X, Y) :- \text{brother}(X, Z), (\text{father}(Z, Y); \text{mother}(Z, Y)).
\]

Thus, compression seems to be a good indicator of the re-usability of an invented predicate when it occurs due to the abstraction of a repeated pattern in a pro-
gram, as in inter-construction, whereas in other cases such as intra-construction, compression is not a strong indicator of re-usability.

Of the numerous predicate invention techniques mentioned earlier, the majority in fact do not choose which predicates to invent based on potential for re-usability. In the technique of ‘closed world specialisation’ [Bain and Muggleton, 1991], as well as the ‘concept formation’ technique of the MOBAL system [Wrobel, 1994], the motivation for inventing a predicate is much the same as for intra-construction: to mimic the effect of a disjunctive sub-expression within the body of a clause while still conforming to conjunctive normal form. Also, in the ‘schema-guided approach’ advocated by Flener [1995], recursive auxiliary predicates are introduced because they are necessary in order to express certain hypotheses within the constraints of first-order logic, but again without any particular intention of re-usability in mind.

A number of studies have used techniques that we may characterise as ‘brute-force’ predication invention. Unlike other mechanisms such as the inverse resolution “W” operators, these ‘brute-force’ techniques do not make use of syntactic patterns in existing knowledge (such as commonality or differences between clauses) in order to make an informed choice of which predicate to invent. Instead, they systematically generate candidate auxiliary predicates in a somewhat uninformed, ‘brute-force’ manner, as part of the process of searching for a hypothesis using some standard search method such as refinement. For example, Leban et al. [2008] allowed an arbitrary single-clause auxiliary predicate to be refined as part of a multi-clause target hypothesis during a refinement-based search with the multi-clause ILP system HYPER. Khan et al. [1998] augmented the Progol system with an ability to systematically generate auxiliary predicates. Now, although this brute-force approach is simple and flexible, it can understandably suffer from severe problems of computational tractability due to the unrestricted combinatorial search space of potential new predicates that is being considered. To mitigate this, both Leban et al. and Khan et al. had to specify the arity and argument types of their invented predicates a priori, rather than deal with the larger search space of predicates of arbitrary arity and type. This had the unfortunate consequence of making their demonstrations of predicate invention seem rather contrived, since in order to specify the correct type signature for an invented predicate you really have to know what predicate you want the system to invent before you have started. In contrast, in the case of the inverse resolution “W” operators the decision of what arity

Note that in a language that supports first-class functions such as lambda calculus, there is no requirement for recursive auxiliary concepts is be given new, named definitions. Such concepts can be represented simply as subexpressions of a larger concept by invoking a higher-order function called the fixed-point operator.
and argument types to give an invented predicate is made automatically in a natural way by analysing the structure of the clauses of the knowledge base that are being transformed by the operator.

An interesting variation on the ‘brute-force’ approach to predicate invention is known as ‘meta-interpretive learning’ [Muggleton et al., 2012c], which gets round the problem of the large combinatorial search space of potential new predicates by using a very strong declarative bias. Candidate hypotheses are constrained to being members of some particular domain-specific language, and an interpreter for this language (the ‘meta-interpreter’) is provided a priori as background knowledge. Muggleton et al. demonstrated their technique in a grammar-learning domain, achieving efficient inference of multi-predicate hypotheses that were constrained to be regular or context-free grammars. However, meta-interpretive learning does not seem suitable as a predicate invention mechanism for cumulative learning for two reasons. Firstly, the invented predicates do not come with any particularly evidence for re-usability. Secondly, meta-interpretive learning requires that one already has a strong inductive bias provided a priori, whereas in cumulative learning one assumes that only a weak initial bias is available, and that a strong bias should be learned automatically by means of the cumulative learning process itself.

We have seen that of the variety of predicate invention techniques that have been developed in ILP, only the inverse resolution operator inter-construction seems highly suited as a means for inventing re-usable background knowledge in a cumulative learning scenario. The other techniques either do not invent predicates for which there is any particular evidence of future re-usability, or in the case of the ‘brute-force’ approach they suffer from severe combinatorial search problems. However, given inter-construction’s apparent promise, why has no one apparently made a serious attempt to actually use it to implement cumulative learning? In the next subsection I argue that first-order logic’s lack of support for first-class functions limits the power of predicate invention mechanisms like inter-construction. However, as we shall see, all is by no means lost because it is possible to adapt inter-construction to other languages such as lambda calculus and in doing so shed this limitation.

2.4.3 Limitations of Predicate Invention due to First-Order Logic

Because predicates are not first-class in first-order logic, it means that you cannot create any kind of abstraction that would require parametrising over a predicate symbol. In particular:

- You cannot create multi-clause abstractions, i.e. there is no straightforward
<table>
<thead>
<tr>
<th>First-order logic</th>
<th>Lambda calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>p₁(X) :- a(X).</td>
<td>p₁ = \ x -&gt; or (a x)</td>
</tr>
<tr>
<td>p₁(X) :- b(X), c(X).</td>
<td>(and (b x) (c x))</td>
</tr>
<tr>
<td>p₂(X) :- a(X).</td>
<td>p₂ = \ x -&gt; or (a x)</td>
</tr>
<tr>
<td>p₂(X) :- b(X), d(X).</td>
<td>(and (b x) (d x))</td>
</tr>
<tr>
<td>↓</td>
<td>↓</td>
</tr>
<tr>
<td>?</td>
<td>p₁ = g c</td>
</tr>
<tr>
<td></td>
<td>p₂ = g d</td>
</tr>
<tr>
<td></td>
<td>g = \ y x -&gt; or (a x)</td>
</tr>
<tr>
<td></td>
<td>(and (b x) (y x))</td>
</tr>
</tbody>
</table>

Figure 2.6: On the left is a simple example of a logic program containing a syntactic pattern that spans multiple clauses: notice that the definitions of p₁ and p₂ are nearly identical apart from a difference of one literal in the second clause. Within first-order logic, there is no straightforward way to construct a predicate that abstracts this pattern. On the right is a translation of the logic program into lambda calculus, along with a re-expressed version where the pattern has been abstracted into a function g. Notice how the predicates c and d are passed as arguments to g.

way to abstract over commonality that spans more than one clause, as in Fig. 2.6.

- You cannot abstract over similar patterns of arguments where predicate symbols differ, as in Fig. 2.7.

These limitations are quite severe, particularly the inability to make multi-clause abstractions. Multi-clause concepts are ubiquitous in most logic programs, and will frequently contain commonality spanning multiple clauses.

Unfortunately, one can only conclude that (pure) first-order logic is simply not a good choice of language if one wishes to make abstractions. It is no wonder that Prolog programs of any significant size written by humans tend to rely heavily on extensions of the language beyond first-order logic: meta-logical, extra-logical, and higher-order features [Sterling and Shapiro, 1994].

The abstractions shown in Figures 2.6 and 2.7 can be made in lambda calculus very straightforwardly using the anti-unification ‘recipe’ described earlier in Sect. 2.4.1. In Chap. 7, I formalise this recipe as an inverse deduction process for lambda calculus, and in fact we shall see that it is very much a lambda-calculus analogue of inter-construction.
<table>
<thead>
<tr>
<th>First-order logic</th>
<th>Lambda calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1(X, Y) :- a(X, Y), a(Y, X).</td>
<td>p1 = \ x y -&gt;</td>
</tr>
<tr>
<td></td>
<td>and (a x y) (a y x)</td>
</tr>
<tr>
<td>p2(X, Y) :- b(X, Y), b(Y, X).</td>
<td>p2 = \ x y -&gt;</td>
</tr>
<tr>
<td></td>
<td>and (b x y) (b y x)</td>
</tr>
<tr>
<td></td>
<td>↓</td>
</tr>
<tr>
<td></td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>p1 = g a</td>
</tr>
<tr>
<td></td>
<td>p2 = g b</td>
</tr>
<tr>
<td></td>
<td>g = \ z x y -&gt;</td>
</tr>
<tr>
<td></td>
<td>and (z x y) (z y x)</td>
</tr>
</tbody>
</table>

Figure 2.7: The logic program on the left exhibits a syntactic pattern in which different predicate is invoked with an identical pattern of arguments. Again, with first-order logic there is no straightforward way to construct a predicate that abstracts this pattern. However, on the right we see that an abstraction can be made if we translate into lambda calculus.
Chapter 3

The Lambda Calculus

Setting for Learning

In this chapter I first I describe the syntax and semantics of a formulation of simply typed lambda calculus, originally due to Church [1940]. Secondly I define a notion of generality in lambda calculus, and introduce entailment and subsumption orders over lambda terms. Thirdly I introduce a setting for machine learning in which lambda calculus is the knowledge representation language.

The language described in this chapter may be be regarded as a minimalist, simply typed version of the Haskell programming language [Peyton Jones, 2003; Hudak, 2000]. I have used Haskell as a model both for its syntactic style and for its non-strict semantics. My overview of the denotational semantics of lambda calculus follows the style of presentation in [Peyton Jones, 1987]. For an account covering the more subtle technical details, see [Stoy, 1981] or [Streicher, 2006]. The lambda calculus of this chapter is explicitly typed as in [Huet, 1975].

Before proceeding, the reader may wish to refer to Appendix A for a description of some notational conventions used in this and the following chapters.

3.1 Syntax

I shall use the convention of writing object-level syntax in a typewriter font, and meta-level syntax in an italic roman font.

3.1.1 Types

Definition 3.1 (data/function type). A type is either:

- a data type consisting of an alpha-numeric character string;
- or a function type of the form \((\alpha \rightarrow \beta)\), where \(\alpha\) and \(\beta\) are types.
The operator \( \to \) is taken to be right associative, and I shall usually omit parentheses where possible. So, for example, \( ((X \to X) \to (X \to X)) \) may be written \( (X \to X) \to X \to X \).

**Definition 3.2** (arity, return type). Observe that any type can be written uniquely in the following form, where \( n \geq 0 \) and \( R \) is a data type:

\[
\alpha_1 \to \ldots \to \alpha_n \to R
\]

Let us call \( n \) the *arity* and \( R \) the *return type* of the type expression.

### 3.1.2 Terms

**Definition 3.3** (symbol, name, hole, type annotation). A *symbol* takes the form \( s:\alpha \) where \( s \) is an alpha-numeric character string and \( \alpha \) is a type. \( s \) is known as the *name* of the symbol. A *hole* takes the form \( \#:\alpha \) where \( \alpha \) is a type. In both cases the \( :\alpha \) part is known as the *type annotation*.\(^1\)

To avoid clutter, I shall usually omit writing the type annotations on symbols and holes, both when a type is clear from context, and when knowing the type of a symbol or hole is not important for the discussion. For example, symbols such as \( a1:\text{Bool} \) and \( \text{cons}: (\text{Int} \to \text{List} \to \text{List}) \) may be written as \( a1 \) and \( \text{cons} \), and a hole may be written as \( \# \).

**Definition 3.4** (term, variable, application, lambda abstraction, parameter, body). *Terms* are defined as follows:

- if \( x \) is a symbol or hole whose type is \( \alpha \), then the *variable* \( x \) is a term of type \( \alpha \);
- if \( M \) is a term of type \( \alpha \to \beta \) and \( N \) is term of type \( \alpha \), then the *application* \( (M \ N) \) is a term of type \( \beta \);
- if \( x \) is a symbol whose type is \( \alpha \) and \( M \) is a term of type \( \beta \), then the *lambda abstraction* \( (\ x \to \ M) \) is a term of type \( \alpha \to \beta \); \( x \) is known as the *parameter* of the lambda abstraction (or *lambda parameter*), and \( M \) the *body*.

I shall use the following conventions for making terms easier to read:

- The application operator is taken to be left associative, and we will usually omit parentheses where possible. So for example, \( ((a \ b) \ (c \ d)) \) may be written \( a \ b \ (c \ d) \).

\(^1\)Note that the use of the colon symbol here to indicate a type annotation is distinct from my use of it elsewhere to mean the list ‘cons’ operator.
We may write \((\ x_1 \mapsto (\ x_2 \mapsto \ldots (\ x_n \mapsto M) \ldots ))\), where the \(x_i\) are symbols and \(M\) is a term, as \((\ x_1 \ x_2 \ldots \ x_n \mapsto M)\). So for example, 
\((\ x \mapsto (\ y \mapsto x))\) becomes \((\ x \ y \mapsto x)\).

**Definition 3.5** (bound, free). An occurrence of a variable within a term is known as *bound* if it is inside the body of some lambda abstraction whose parameter is the same as that variable. If an occurrence of a variable is not bound, we say it is *free*.

For example, in \(x \ (\ x \mapsto x \ (\ y \mapsto x \ z))\), the first occurrence of the variable \(x\) is free, whilst the other two are are bound. The occurrence of \(z\) is free. Note that occurrences of holes in terms are always free because a hole cannot be a parameter of lambda abstraction.

**Definition 3.6** (trivial, total, partial). A *trivial* term is a hole. A *total term* is a term that contains no holes. A *partial term* is a term that may or may not contain holes (in other words, ‘partial term’ is just a synonym for ‘term’).

**Definition 3.7** (size). The *size* of a term is equal to the total number of applications, lambda abstractions, and non-hole variables that it contains:

\[
\text{size}(x) = \begin{cases} 
1 & \text{if } x \text{ is a symbol} \\
0 & \text{if } x \text{ is a hole}
\end{cases}
\]

\[
\text{size}(MN) = 1 + \text{size}(M) + \text{size}(N)
\]

\[
\text{size}(\ x \mapsto M) = 1 + \text{size}(M)
\]

**Definition 3.8** (place, subterm). A *place* is a list of natural numbers marking an occurrence of a *subterm* within a term. The set of places in a term and their corresponding subterms are defined as follows:

- The subterm at place \([\ ]\) in a term \(T\) is equal to \(T\).
- The subterm at place \(1:p\) in a term \((MN)\) is equal to the subterm at place \(p\) in \(M\).
- The subterm at place \(2:p\) in a term \((MN)\) is equal to the subterm at place \(p\) in \(N\).
- The subterm at place \(1:p\) in a term \(( \ x \mapsto M)\) is equal to the subterm at place \(p\) in \(M\).

**Definition 3.9** (in scope). A lambda parameter \(x\) is said to be *in scope* at a place \(p\) in a term \(T\) if the subterm at that place \(p\) in \(T\) is inside the body of some lambda abstraction whose parameter is \(x\).
Definition 3.10 (ancestor, descendant). Let \( p, q \) be places. \( p \) is an improper ancestor of \( q \) (\( q \) is an improper descendant of \( p \)) if \( p \) is a prefix of \( q \). \( p \) is an ancestor of \( q \) (\( q \) is a descendant of \( p \)) if \( p \) is an improper ancestor of \( q \) and \( p \neq q \).

Definition 3.11 (place-substitution). A place-substitution on terms, written \( \{p_1/T_1, \ldots, p_n/T_n\} \), represents a mapping from distinct places \( p_i \) to terms \( T_i \). If \( A \) is a term and \( \sigma \) is place-substitution, then \( A\sigma \) is the term obtained by replacing the subterm at place \( p_i \) in \( A \) with \( T_i \), for all \( p_i/T_i \in \sigma \). Note that \( A\sigma \) is defined only if the following conditions hold:

- for all \( i \): \( p_i \) is a valid place in \( A \), and \( T_i \) has the same type as the subterm at place \( p_i \) in \( A \);
- for all \( i, j \): \( p_i \) is not an ancestor of \( p_j \).

Application of a place-substitution to a term is taken to be left-associative. So for example, \( (A\sigma)\tau \) may be written \( A\sigma\tau \), where \( A \) is a term and \( \sigma, \tau \) are place-substitutions.

Definition 3.12 (size). The size of a place-substitution \( \{p_1/T_1, \ldots, p_n/T_n\} \) is equal to the sum of the sizes of the terms \( T_i \).

Proposition 3.13. For terms \( A, B, C_1 \ldots C_n \) and places \( p, q_1 \ldots q_n \) it holds that:

\[
A\{p/B\{q_1/C_1 \ldots q_n/C_n\}\} = A\{p/B\{(p ++ q_1)/C_1 \ldots (p ++ q_n)/C_n\}\}
\]

Proof. The proof follows by structural induction on \( p \). We omit the details. \( \square \)

Definition 3.14 (hole-substitution). Let \( \sigma = \{p_1/T_1, \ldots, p_n/T_n\} \) be a place-substitution and let \( A \) be a term. We say that \( \sigma \) is a hole-substitution on \( A \) if for each \( i \in \{1 \ldots n\} \) it holds that the subterm at place \( p_i \) in \( A \) is a hole.

3.1.3 Declarations

Definition 3.15 (data type declaration, constructor). A data type declaration takes the form:

```
data X = C_1 \alpha_{11} \ldots \alpha_{1k_1}
| \ldots
| C_n \alpha_{n1} \ldots \alpha_{nk_n}
```

where \( X \) is a data type, \( C_i \) are alpha-numeric character strings called constructors, and \( \alpha_{ij} \) are types.

Definition 3.16 (value declaration). A value declaration takes the form:

\[
x = M
\]

where \( x \) is a symbol and \( M \) is a term of the same type as \( x \).
3.2 Denotational Semantics

3.2.1 Types

A type denotes a set known as the domain of the type. Elements of the domain of a type are known as values of that type. A partial order \( \sqsubseteq \) called the semantic approximation ordering exists over every domain. One element in each domain, written \( \bot \), represents an ‘undefined’ value.

**Definition 3.17** (domains). The domain of a type is defined as follows:

- Given a data type declaration for a data type \( X \), as in Defn. 3.15, then the domain of \( X \) contains a value \( \bot \), as well as all possible constructor application values of the form \((C_i \ a_{i1} \ldots \ a_{ik_i})\) where \( C_i \) is a constructor of \( X \) with argument types \( \alpha_{i1} \ldots \alpha_{ik_i} \) and each \( a_{ij} \) is a value in the domain of the corresponding \( \alpha_{ij} \). For \( x, y \) in the domain of \( X \):
  \[ x \sqsubseteq y \quad \text{if} \quad x = \bot \]
  or if \( x \) is of the form \((C \ a_{1} \ldots \ a_{n})\) and \( y \) is of the form \((C \ b_{1} \ldots \ b_{n})\) for the same constructor \( C \), such that \( a_{j} \sqsubseteq b_{j} \) for each \( j \).

- The domain of a function type \( \alpha \rightarrow \beta \) contains monotonic (with respect to \( \sqsubseteq \) ) functions from the domain of \( \alpha \) to the domain of \( \beta \). The \( \bot \) element of the domain of a function type is the function that maps all its arguments to \( \bot \). For \( f, g \) in the domain of a function type:
  \[ f \sqsubseteq g \quad \text{if} \quad \forall x. f(x) \sqsubseteq g(x) \]

**Definition 3.18** (non-trivial entries). The set of non-trivial entries of a value \( f \) of function type \( \alpha \rightarrow \beta \) is equal to:

\[ \{ a \triangleleft \text{values of type } \alpha \mid f(a) \neq \bot \} \]

**Definition 3.19** (size). The size of a value of a data type is given by:

\[ \text{size}(\bot) = 1 \]
\[ \text{size}(C \ a_{1} \ldots \ a_{n}) = 1 + \sum_{i=1}^{n} \text{size}(a_{i}) \]

The size of a value of a function type is given by:

\[ \text{size}(f) = 1 + \sum_{i=1}^{n} \text{size}(b_{i}) \]

where \( \langle a_{1}, b_{1} \rangle \ldots \langle a_{n}, b_{n} \rangle \) are the non-trivial entries of \( f \).

**Definition 3.20** (value-place). A value-place is a list containing natural numbers and values. It marks an occurrence of a subvalue within a value. The set of value-places in a value and their corresponding subvalues are defined as follows:
• The subvalue at place \( [] \) in a value \( x \) is equal to \( x \).

• The subvalue at place \( i : p \) in a value \((C \ a_1 \ldots \ a_n)\) is equal to the subvalue at place \( p \) in \( a_i \).

• The subvalue at place \( a : p \) in a value \( f \) of function type is equal to the subvalue at place \( p \) in \( f(a) \).

3.2.2 Terms

We define a structure called an environment, whose purpose is to give meaning to the free variables in terms. An environment is analogous to an interpretation in first-order logic.

Definition 3.21 (environment). An environment is a function that takes a symbol and returns a value of the same type as that symbol. Though environments themselves are not values in the domain of any type, it is convenient to have the set of all environments subject to the semantic approximation ordering.

For environments \( E_1, E_2 \):

\[
E_1 \sqsubseteq E_2 \text{ if } \forall x. E_1(x) \sqsubseteq E_2(x)
\]

Definition 3.22 (substitution). A substitution on environments, written \( \{x_1/a_1, \ldots, x_n/a_n\} \), represents a mapping from distinct symbols \( x_i \) to values \( a_i \), where each \( a_i \) has the same type as \( x_i \). If \( E \) is an environment and \( \theta \) is a substitution, then \( E\theta \) is a new environment whose application to a symbol \( x \) is defined as follows:

• if \( \theta \) contains an element \( x_i/a_i \) such that \( x = x_i \), then \( E\theta \cdot x = a_i \);

• otherwise, \( E\theta \cdot x = E \cdot x \).

A term denotes, with respect to a given environment, a value of the same type as that term. We shall express this denotation by means of a semantic function called eval (short for ‘evaluate’).

Definition 3.23 (eval). The function eval takes a term and an environment,
and returns a value of the same type as the term:

\[ eval(x, E) \]

where \( x \) is a symbol

\[ = E(x) \] (3.24)

\[ eval(x, E) \]

where \( x \) is a hole

\[ = \bot \] (3.25)

\[ eval((M \ N), E) \]

\[ = eval(M, E) \cdot eval(N, E) \] (3.26)

\[ eval(\lambda \ x \to M), E) \cdot a \]

where \( a \) is a value of the same type as \( x \)

\[ = eval(M, E[x/a]) \] (3.27)

To complete the definition of our lambda calculus language, we define a toplevel environment with respect to a given set of data type and value declarations.

**Definition 3.28** (fundamental environment). The fundamental environment \( E_0 \) is defined as follows.

- For all types \( \alpha \):

  \[ E_0(\text{void}: \alpha) = \bot \]

  \[ E_0(\text{error}: \alpha) = \bot \]

- For all types \( \alpha \):

  \[ E_0(\text{fix}:(((\alpha \to \alpha) \to \alpha) \to \alpha)) \cdot f = \text{the least (under} \ \subseteq \text{) fixed point of} \ f \]

For each declaration of a data type \( X \) with constructors \( C_i \) and constructor argument types \( \alpha_{ij} \), as in Defn. 3.15, it holds that:

- For \( i \in \{1 \ldots n\} \):

  \[ E_0(C_i:(\alpha_{i1} \to \ldots \to \alpha_{ik_i} \to X)) \cdot a_1 \cdot \ldots \cdot a_{k_i} \]

  where each \( a_j \) is an arbitrary value of appropriate type

  \[ = (C_i \ a_1 \ldots \ a_{k_i}) \]
• For all types $\beta$ and for $i \in \{1 \ldots n\}$:

$$E_0((\text{case } \cdot X):\tau_{\text{case}}) \cdot \bot = \bot$$
$$E_0((\text{case } \cdot X):\tau_{\text{case}}) \cdot (C_i a_1 \ldots a_{k_i}) \cdot f_1 \cdot \ldots \cdot f_n$$

where each $a_j$ and $f_j$ is an arbitrary value of appropriate type

$$= f_i \cdot a_1 \cdot \ldots \cdot a_{k_i}$$

In the above, $\cdot$ is a string concatenation operator and $\tau_{\text{case}}$ is defined as follows:

$$\tau_{\text{case}} = X \rightarrow (\alpha_{11} \rightarrow \ldots \rightarrow \alpha_{1k_1} \rightarrow \beta)$$
$$\rightarrow \ldots$$
$$\rightarrow (\alpha_{n1} \rightarrow \ldots \rightarrow \alpha_{nk_n} \rightarrow \beta)$$
$$\rightarrow \beta$$

Finally, for any symbol $x$ not accounted for above, $E_0(x) = \bot$.

**Definition 3.29** (toplevel environment). Given value declarations $x_i = M_i$ for $i \in \{1 \ldots n\}$, the toplevel environment $E$ is defined as the least (under $\sqsubseteq$) fixed point of the following function $f$:

$$f(E) = E_0\{x_1/\text{eval}(M_1, E), \ldots, x_n/\text{eval}(M_n, E)\}$$

where $E_0$ is the fundamental environment.

### 3.3 Generality Orders

In order to specify a setting for learning, we need to identify a suitable formal notion of *generality* in lambda calculus. What do we mean by ‘generality’? In inductive inference, one expects that if a hypothesis is *more general* than some training data, then it will:

1. not contradict the training data;
2. make predictions about unseen test data.

In first-order logic, the idea of generality is closely related to the idea of the *truth* of a formula: $X$ is more general than $Y$ if $X$ is true in a subset of the interpretations (‘possible worlds’) in which $Y$ is true. It is easy to see how this ‘subset of possible worlds’ definition fits with the two desiderata mentioned above, i.e. $X$ will not contradict $Y$ and $X$ will tend to have more predictive power than $Y$ because it narrows down our knowledge about which ‘possible world’ we are in.
On the other hand, in our lambda calculus language there is no concept of the ‘truth’ of a term (except in the special case of a term of boolean type). Instead we have the concept of ‘definedness’ provided by the semantic approximation ordering. However, it just so happens that definedness meets our desiderata for a good definition of generality, i.e. if a value \( X \) is more defined than a value \( Y \), then \( X \) does not contradict \( Y \), and furthermore \( X \) will tend to contain extra information not contained in \( Y \). Therefore, in the context of lambda calculus, it seems to make good sense to identify ‘more general’ with ‘more defined’. Throughout much of this thesis I shall be exploring the consequences of this choice.

I shall now define generality orders over lambda calculus terms that are somewhat analogous to the entailment and subsumption orders over clauses commonly used in first-order ILP. Entailment gives us a fundamental definition of what it means for one lambda term to be more general than another. Subsumption is a logically stronger, more computationally tractable approximation to entailment.

**Definition 3.30** (\( \lambda \)-entailment). Let \( A, B \) be terms. We say that \( B \) \( \lambda \)-entails \( A \) if for all environments \( E \) it holds that:

\[
\text{eval}(B, E) \sqsubseteq \text{eval}(A, E)
\]

**Definition 3.31** (\( \lambda \)-subsumption). Let \( A, B \) be terms. We say that \( B \) \( \lambda \)-subsumes \( A \) if there exists a hole-substitution \( \sigma \) on \( A \) such that \( A\sigma = B \).

For the rest of this thesis, ‘entails’ shall be taken to mean ‘\( \lambda \)-entails’ and ‘subsumes’ shall be taken to mean ‘\( \lambda \)-subsumes’, unless otherwise specified.

**Definition 3.32** (properly subsumes, subsume-equivalent). Let \( A, B \) be terms. \( B \) properly subsumes \( A \) if \( B \) subsumes \( A \) and \( A \) does not subsume \( B \). \( A \) and \( B \) are subsume-equivalent if \( B \) subsumes \( A \) and \( A \) subsumes \( B \).

It is straightforward to see that entailment and subsumption are both quasi-orders. The next theorem shows us that subsumption is in fact a partial order (modulo alpha-conversion).

**Lemma 3.33.** Let \( A, B \) be terms such that \( B \) subsumes \( A \). It holds that either \( B = A \) or size(\( B \)) > size(\( A \)).

**Proof.** The proof follows by Defn. 3.31, Defn. 3.14, and Defn. 3.7.

**Theorem 3.34.** Terms \( A, B \) are subsume-equivalent i.f.f. they are equal.

**Proof.** The forward implication follows by Lemma 3.33. Its converse follows by Defn. 3.31.
The fact that subsumption is logically stronger than entailment is shown in the following theorem:

**Theorem 3.35.** For terms $A, B$, if $B$ subsumes $A$ then $B$ entails $A$.

**Proof.** Assume $B$ subsumes $A$. By Defn. 3.31, there exists a hole-substitution $\sigma$ on $A$ such that $\sigma A = B$. Proceed by structural induction on $A$:

1. $A$ is a symbol. Since $A$ contains no holes, $\sigma$ is the empty substitution. Hence $B = A$, from which the proof follows.

2. $A$ is a hole. The proof follows by Eqn. 3.25 and Defn. 3.17.

3. $A = (M N)$. Then $\sigma$ can be partitioned as follows where $\sigma_M, \sigma_N$ are hole-substitutions on $M, N$ respectively:

$$\sigma = \{p/T \sigma_M \cdot (1 : p)/T\} \cup \{p/T \sigma_N \cdot (2 : p)/T\}$$

Observe that $B = (M \sigma_M N \sigma_N)$. The proof then proceeds as follows, for all $E$:

$$eval(B, E) \supseteq eval(M \sigma_M, E) \cdot eval(N \sigma_N, E)$$  
(by 3.26)

$$\supseteq eval(M, E) \cdot eval(N \sigma_N, E)$$  
(by the inductive hypothesis and 3.17)

$$\supseteq eval(M, E) \cdot eval(N, E)$$  
(by the inductive hypothesis and 3.17)

$$\supseteq eval(A, E)$$  
(by 3.26)

4. $A = (\lambda x \rightarrow M)$. Then $\sigma$ can be written as follows where $\sigma_M$ is a hole-substitution on $M$:

$$\sigma = \{p/T \sigma_M \cdot (1 : p)/T\}$$

Observe that $B = (\lambda x \rightarrow M \sigma_M)$. It then holds for all $E, a$ that:

$$eval(B, E) \cdot a \supseteq eval(M \sigma_M, E\{x/a\})$$  
(by 3.27)

$$\supseteq eval(M, E\{x/a\})$$  
(by the inductive hypothesis)

$$\supseteq eval(A, E) \cdot a$$  
(by 3.27)

Thus, the proof follows by Defn. 3.17.

$\square$

### 3.4 Setting for Learning

I now define a setting for learning which will allow us to perform inductive inference using lambda calculus as the knowledge representation language. In this setting, the learner takes the following information as input:
1. A set of data type and value declarations. Together these define a toplevel environment $B$ which we call the background knowledge.

2. A value $D$ called the target datum (analogous to the ‘examples’ in ILP).
   
   The type of $D$ is called the target type.

The hypothesis space considered by the learner is the set of total terms whose type is equal to the target type.

Learning proceeds by Bayesian inference, with a prior and a likelihood function defined over the hypothesis space. The likelihood of a total term $H$ is defined as:

\[
P(D|H, B) = \begin{cases} 
1 & \text{if } \text{eval}(H, B) \sqsubseteq D \\
0 & \text{otherwise} 
\end{cases}
\]  

The posterior probability of $H$ is given by Bayes’ rule:

\[
P(H|D, B) \propto P(D|H, B)P(H)
\]

where $P(H)$ is the prior probability of $H$. Note that $P(H)$ is independent of $B$. I describe a suitable choice of prior in the next chapter.

Given the form of the likelihood function above, note that finding $H$ that maximises the posterior probability is equivalent to finding $H$ that maximises the prior probability subject to the constraint that $\text{eval}(H, B) \sqsubseteq D$.

### 3.4.1 Partial terms represent sets of hypotheses

Since a partial term may contain holes, it can be thought of as a ‘partially defined’ term. One can imagine converting a partial term to a total term by specifying what subterms should go in the holes. Since there may be many possible ways to fill in the holes, we can regard a partial term as representing a set of hypotheses, i.e. the set of total terms that can be obtained from it by filling in the holes. We can express this idea using the $\lambda$-subsumption relation defined in the previous section:

**Definition 3.37** (interpretation of a partial term as a set of total terms). Given a total term $T$ and a partial term $H$, we say that $T \in H$ if $T$ subsumes $H$.

Since a partial term represents a set of hypotheses, it can be thought of as an event in the sense of probability theory, where each hypothesis is a possible outcome. Thus, the prior probability of a partial term $H$ is equal to:

\[
P(H) = \sum_{T \in H} P(T)
\]

Furthermore, it follows by elementary probability theory that the likelihood of a partial term $H$ is equal to:

\[
P(D|H, B) = \sum_{T \in H} \frac{P(D|T, B)P(T)}{P(H)}
\]  

53
Chapter 4

Prior Probability Distributions

It is standard in ILP to use a compression measure (Eqn. 2.1) to choose between competing hypotheses. This compression measure is understood to represent a Bayesian posterior in the following sense: \textit{if hypothesis }X\textit{ has greater compression than hypothesis }Y, then hypothesis }X\textit{ has greater posterior probability than hypothesis }Y. In other words, posterior is some monotonically increasing function of compression. However, usually a precise \textit{quantitative} relationship between compression and posterior probability is not made explicit.

Relying on the compression measure is attractive because compression is simple to define and easy to compute. However, not being able to calculate explicit probabilities of hypotheses has significant drawbacks. It limits ones mechanism of inference to \textit{Bayes MAP} (Maximium A Posteriori), in which one chooses a single hypothesis that maximises posterior probability, and then makes predictions based on that one hypothesis. On the other hand, one is unable to do \textit{Bayes prediction}, in which one makes predictions by taking a weighted average by posterior probability, over all hypotheses. This is a serious deficiency because there exist situations in which Bayes prediction can be expected to significantly outperform Bayes MAP in terms of predictive accuracy (see Fig. 4.1).

I propose that in inductive programming we should use explicit prior and posterior probability distributions rather than relying on a compression measure. The benefits of doing this are twofold. Firstly, users of ILP systems will then have a choice between Bayes MAP and Bayes prediction as an inference mechanism, rather being restricted to Bayes MAP. Secondly, by making our prior probability distributions explicit we make them easier to customise, therefore we can specify priors that are more flexible and more appropriate to particular learning scenarios.
Figure 4.1: An example of a scenario where Bayes MAP (Maximum A Posteriori) makes poor predictions compared with Bayes prediction. In this example, the hypothesis space contains two clusters of hypotheses. The Bayes MAP hypothesis is contained in the left-hand cluster (highest point of the curve). However, the right-hand cluster has much larger total probability mass (area under curve) compared with the left-hand cluster. Therefore hypotheses from the right-hand cluster will make predictions that are more representative of the distribution as a whole. The problem with Bayes MAP is that it is unable to take into account situations like this where a large number of lower-probability hypotheses outweigh one higher-probability hypothesis.
With this in mind, in this chapter I define three successively more flexible prior probability distributions over lambda terms. These priors are designed for use within the setting for learning introduced in Chap. 3. Each of the three priors builds on the last, and as we shall see, the second and third add features specifically designed to allow cumulative learning.

4.1 The Base Prior

In this section I specify a prior suitable for base inductive inference, i.e. without cumulative learning. I designed this prior in order to meet the following five criteria:

1. **Occam’s Razor.** The probability of a hypothesis should tend to decrease exponentially with its syntactic size. This is the same design principle that underlies the standard ILP compression measure. There are information theoretic reasons why this is a good design principle [Muggleton and De Raedt, 1994].

2. **Minimisation of redundancy.** By restricting the terms with non-zero probability under the prior to a form known as long normal form, a large class of redundant hypotheses are eliminated. We can think of this as a lambda calculus equivalent of the restriction to definite clause form typically used in first-order ILP.

3. **Static typing taken into account.** The structure of the prior fits closely with the structure of the hypothesis space as imposed by static typing. This is a significant advantage because it gives one the ability to independently control the form of the prior probability distribution in different subspaces of the hypothesis space merely by introducing appropriate type constraints. This level of control is not possible with the standard ILP compression measure, which ignores type constraints.

4. **Customisable weights on background knowledge.** Each element of background knowledge is assigned a customisable primitive probability value a priori. This can be used to reflect the fact the certain elements of background knowledge are more useful than others in a given learning scenario. It is an important source of flexibility which helps us to ensure that our prior accurately reflects our assumptions. This is in contrast with the standard ILP compression measure, which enforces a prior assumption that all elements of background knowledge are equally likely.

5. **Extensibility to non-parametric form.** In the next section we extend this prior to a non-parametric form, in which the weights on elements of
background knowledge are inferred automatically as part of the learning process, rather than provided in advance. A number of aspects of this base prior are designed so as to easily facilitate this extension.

Throughout this chapter I shall use a standard convention of Barendregt [Barendregt, 1984] that lambda terms are taken to be syntactically equal if they are identical up to the renaming of bound variables. So, for example, the terms \( (\lambda x \to x) \) and \( (\lambda y \to y) \) are taken to be equal provided that \( x \) and \( y \) have the same type.

The first step in defining this base prior is to introduce the following constraint on lambda terms known as long normal form [Dowek, 2001]:

**Definition 4.1.** A term, of type \( \alpha_1 \to \ldots \to \alpha_m \to R \) where \( R \) is a data type, is in long normal form if it is of the form:

\[
(\lambda x_1 \ldots x_m \to f T_1 \ldots T_n)
\]

where:

- \( f \) is a variable whose type has arity \( n \);
- each \( T_j \) is a term in long normal form;
- if \( f \) is a hole then \( n = 0 \).

It is known [Dowek, 2001, Pg. 1022–1023] that for any lambda term \( T \), there always exists some lambda term \( T' \) in long normal form such that \( T \) and \( T' \) are semantically equivalent, i.e. that \( \text{eval}(T, E) = \text{eval}(T', E') \) for all environments \( E \). Therefore, by excluding all terms from our hypothesis space except those in long normal form, we eliminate redundancy without constraining the set of possible values that can be represented by hypotheses. Katayama [2007] was the first to make use of long normal form in this way to eliminate redundancy in the context of inductive programming. I shall make use of the same technique here.

Now I shall give a collection of definitions leading up to that of the prior itself. In order to calculate the prior of a term, we determine probability values called *weights* for all of the variables it contains (Defn. 4.18). The final prior probability is then simply the product of these weights (Defn. 4.23). These weight values come from an object called a *pool* (Defn. 4.9), which stores the types and weights of all variables in scope at some place within a term. A pool consists of multiple *frames* (Defn. 4.7), reflecting the multiple levels of nesting at which variables may be bound. Finally, *even and odd weighted types* (Defn. 4.3 and Defn. 4.4) reflect the fact that weights must be associated not just with variables that are in scope, but also with any higher-order elements of the types.
of those variables: this is necessary because when any variable gets invoked as a higher-order function it introduces new variables, and those new variables need weight values too.

**Definition 4.2** (weight). A weight is a probability value.

The following two definitions are mutually recursive:

**Definition 4.3** (even weighted type, even weight annotation, primary weight). An even weighted type takes the form:

$$\alpha_1 \rightarrow \cdots \rightarrow \alpha_m \rightarrow R^w$$

where $m \geq 0$, each $\alpha_i$ is an odd weighted type, $R$ is a data type, and $w$ is a weight. We call $w$ an even weight annotation. We also say that $w$ is the primary weight of this even weighted type.

**Definition 4.4** (odd weighted type, odd weight annotation). An odd weighted type takes the form:

$$\alpha_1 \rightarrow \cdots \rightarrow \alpha_m \rightarrow R^X$$

where $m \geq 0$, each $\alpha_i$ is an even weighted type, $R$ is a data type, and $X$ is a set of pairs of the form $\{\langle S_1, w_1 \rangle, \ldots, \langle S_n, w_n \rangle\}$ where the $S_j$ are distinct data types and the $w_j$ are weights. We call $X$ an odd weight annotation.

**Definition 4.5** (stripWeights). For an even or odd weighted type $\tau$, we define $\text{stripWeights}(\tau)$ to be the type obtained by stripping $\tau$ of all weight annotations.

**Definition 4.6.** The function $\text{zeroWeightsOdd}$ takes a type $\tau$ and returns the odd weighted type that is uniquely defined by the following constraints:

- $\text{stripWeights}(\text{zeroWeightsOdd}(\tau)) = \tau$;
- each even weight annotation within $\text{zeroWeightsOdd}(\tau)$ is equal to zero, and each odd weight annotation is equal to the empty set.

**Definition 4.7** (frame). A frame takes the form $\{\langle x_1, \alpha_1 \rangle, \ldots, \langle x_m, \alpha_m \rangle\}^X$ where the $x_i$ are distinct symbols, each $\alpha_i$ is an even weighted type such that $\text{stripWeights}(\alpha_i)$ is equal to the type of $x_i$, and $X$ is an odd weight annotation as in Defn. 4.4.

**Definition 4.8** (frameSymbols). The function $\text{frameSymbols}$ takes a frame and returns the set of symbols in that frame:

$$\text{frameSymbols}(F^X) = \{\langle x, \alpha \rangle \in F \cdot x\}$$

**Definition 4.9** (pool). A pool is a list of frames $[F_1 \ldots F_n]$ such that the following two conditions hold:
the sets $\text{frameSymbols}(F_i)$ are all disjoint from one another;

none of the sets $\text{frameSymbols}(F_i)$ contain the symbol $\text{void}:\alpha$ for any type $\alpha$.

**Definition 4.10 (dropThruWeights).** The function $\text{dropThruWeights}$ takes a data type and an odd weight annotation, and returns a list of weights:

$$\text{dropThruWeights}(R, X) = \begin{cases} \{w\} & \text{if } \langle R, w \rangle \in X \text{ for some weight } w \\ [] & \text{otherwise} \end{cases}$$

**Definition 4.11 (lookupArgs, lookupWeights).** The function $\text{lookupArgs}$ takes a symbol $x$ and a pool $\Gamma$, and returns a list of odd weighted types. The function $\text{lookupWeights}$ again takes a symbol $x$ and a pool $\Gamma$, and returns a list of weights. Let $\alpha_1 \rightarrow \ldots \rightarrow \alpha_m \rightarrow R$ be the type of $x$ where $R$ is a data type. The definitions of $\text{lookupArgs}$ and $\text{lookupWeights}$ consist of three of cases:

1. $\Gamma = []$. For each $i \in \{1 \ldots m\}$ let $\alpha'_i = \text{zeroWeightsOdd}(\alpha_i)$. It holds that:

$$\text{lookupArgs}(x, \Gamma) = [\alpha'_1, \ldots, \alpha'_m] \quad (4.12)$$

$$\text{lookupWeights}(x, \Gamma) = \begin{cases} \[] & \text{if } m = 0 \text{ and } x = \text{void} \\ [0] & \text{otherwise} \end{cases} \quad (4.13)$$

2. $\Gamma$ is of the form $F^X : \Gamma'$ such that $\langle x, \tau' \rangle \in F$ for some even weighted type $\tau'$. We can write $\tau'$ in the following form:

$$\alpha'_1 \rightarrow \ldots \rightarrow \alpha'_m \rightarrow R^w$$

It then holds that:

$$\text{lookupArgs}(x, \Gamma) = [\alpha'_1, \ldots, \alpha'_m] \quad (4.14)$$

$$\text{lookupWeights}(x, \Gamma) = [w] \quad (4.15)$$

3. $\Gamma$ is of the form $F^X : \Gamma'$ and $\langle x, \tau' \rangle \notin F$ for any even weighted type $\tau'$:

$$\text{lookupArgs}(x, \Gamma) = \text{lookupArgs}(x, \Gamma') \quad (4.16)$$

$$\text{lookupWeights}(x, \Gamma) = \text{dropThruWeights}(R, X) + \text{lookupWeights}(x, \Gamma') \quad (4.17)$$

**Definition 4.18 (priorWeights).** The function $\text{priorWeights}$ takes a term $T$ in long normal form, an odd weighted type $\tau$ such that $\text{stripWeights}(\tau)$ is equal to the type of $T$, and a pool $\Gamma$. $\text{priorWeights}$ returns a list of weights. Since $T$ is in long normal form it can be written uniquely as follows, where $f$ is a variable:

$$T = (\\ x_1 \ldots x_m \rightarrow f \ Y_1 \ldots Y_n)$$
We can also write \( \tau \) as follows:

\[
\tau = \alpha_1 \rightarrow \ldots \rightarrow \alpha_m \rightarrow R^X
\]

The definition of \( \text{priorWeights} \) then consists of two cases:

1. \( f \) is a hole. Then, \( \text{priorWeights}(T, \tau, \Gamma) = [] \).
2. \( f \) is a symbol.
   
   Let \( \Gamma' = \{ (x_1, \alpha_1) \ldots (x_m, \alpha_m) \}^X : \Gamma \).
   
   Let \( [\beta_1 \ldots \beta_n] = \text{lookupArgs}(f, \Gamma') \).
   
   Then, \( \text{priorWeights} \) is given by:

\[
\text{priorWeights}(T, \tau, \Gamma) = \text{lookupWeights}(f, \Gamma')
+ \text{priorWeights}(Y_1, \beta_1, \Gamma')
+ \ldots
+ \text{priorWeights}(Y_n, \beta_n, \Gamma')
\]

\textbf{Definition 4.19 (weightSum).} The function \( \text{weightSum} \) takes a data type and a list of even weighted types, and returns a probability value:

\[
\text{weightSum}(R, [\alpha_1 \ldots \alpha_m]) = \\
\sum_{i=1}^{m} \begin{cases} 
\text{primary weight of } \alpha_i & \text{if } \alpha_i \text{ has return type } R \\
0 & \text{otherwise}
\end{cases}
\]

\textbf{Definition 4.20 (normalised).} We say that an even weighted type \( \alpha_1 \rightarrow \ldots \rightarrow \alpha_m \rightarrow R^w \) is \textit{normalised} if all the \( \alpha_i \) are normalised. We say that an odd weighted type \( \alpha_1 \rightarrow \ldots \rightarrow \alpha_m \rightarrow R^X \) is normalised if all the \( \alpha_i \) are normalised and it holds for all data types \( S \) that:

\[
\text{weightSum}(S, [\alpha_1 \ldots \alpha_m]) + \text{product}(\text{dropThruWeights}(S, X)) = 1
\]

Likewise, we say that a frame \( \{ (x_1, \alpha_1) \ldots (x_m, \alpha_m) \}^X \) is normalised if all the \( \alpha_i \) are normalised and it holds for all data types \( S \) that:

\[
\text{weightSum}(S, [\alpha_1 \ldots \alpha_m]) + \text{product}(\text{dropThruWeights}(S, X)) = 1
\]

We say that a pool \([F_1 \ldots F_n]\) is normalised if all the \( F_i \) are normalised.

\textbf{Lemma 4.21.} Let \( \Gamma = [F_1 \ldots F_n] \) be a pool, and let \( x \) be a symbol such that \( x \notin \text{frameSymbols}(F_i) \) for all \( i \in \{1 \ldots n\} \) and \( x \neq \text{void} \). It follows that:

\[
\text{product}(\text{lookupWeights}(x, \Gamma)) = 0
\]

\textbf{Proof.} By induction on \( n \):

1. \( n = 0 \). The proof follows by Eqn. 4.13.
2. $n \geq 1$. The proof follows by Eqn. 4.17 and the inductive hypothesis.

Lemma 4.22. Let $R$ be a data type, let $X$ be the set of all symbols with return type $R$, and let $\Gamma$ be a normalised pool. It follows that:

$$\sum_{x \in X} \text{product}(\text{lookupWeights}(x, \Gamma)) = 1$$

Proof. By structural induction on $\Gamma$:

1. $\Gamma = \emptyset$. The proof follows by Eqn. 4.13.

2. $\Gamma = \{(y_1, \beta_1) \ldots (y_n, \beta_n) : \Gamma'\}$. Let $Y = \{y_1 \ldots y_n\}$, then proceed:

$$LHS = \sum_{x \in X \cap Y} \text{product}(\text{lookupWeights}(x, \Gamma)) + \sum_{x \in X \setminus Y} \text{product}(\text{lookupWeights}(x, \Gamma))$$

$$= \text{weightsum}(R, [\beta_1 \ldots \beta_n]) + \sum_{x \in X \setminus Y} \text{product}(\text{lookupWeights}(x, \Gamma)) \quad \text{(by 4.15 and 4.19)}$$

$$= \text{weightsum}(R, [\beta_1 \ldots \beta_n]) + (1 - \text{weightsum}(R, [\beta_1 \ldots \beta_n])) \times \sum_{x \in X \setminus Y} \text{product}(\text{lookupWeights}(x, \Gamma')) \quad \text{(by 4.17 and 4.20)}$$

$$= RHS \quad \text{(by 4.9, 4.21, and the inductive hypothesis)}$$

Definition 4.23 (prior). The function prior takes a term $T$ in long normal form, a normalised odd weighted type $\tau$ such that $\text{stripWeights}(\tau)$ is equal to the type of $T$, and a normalised pool $\Gamma$. It returns a probability value:

$$\text{prior}(T, \tau, \Gamma) = \text{product}(\text{priorWeights}(T, \tau, \Gamma))$$

Working within the setting for learning of Sect. 3.4, the base prior is a probability distribution over the set of all long normal terms of some target type. It is parametrised by a normalised pool $\Gamma$ and a normalised odd weighted type $\tau$ such that $\text{stripWeights}(\tau)$ is equal to the target type. Note that the pool $\Gamma$ is analogous to ‘modeb’ declarations in ILP. Assuming some fixed values for $\Gamma$ and $\tau$, the prior probability of a long normal term $H$ is given by:

$$P(H) = \text{prior}(H, \tau, \Gamma)$$

Having defined the form of our prior, we must check that it represents a valid probability distribution. There are two questions to ask. Firstly, is the
Stochastic function $\text{priorProc}(\tau, \Gamma)$:

**Input:** $\tau$, a normalised odd weighted type;
$\Gamma$, a normalised pool.

**Output:** a long normal, total term of type $\text{stripWeights}(\tau)$.

1. Let $x_1 \ldots x_m$ be fresh symbols where $m$ is the arity of $\tau$.
2. Define $R$ and $\Gamma'$ as in Defn. 4.18.
3. Pick any symbol $f$ of return type $R$, with probability $\text{product}(\text{lookupWeights}(f, \Gamma'))$.
4. Define $\beta_1 \ldots \beta_n$ as in Defn. 4.18.
5. For $i \in \{1 \ldots n\}$, pick $Y_i$ by calling $\text{priorProc}(\beta_i, \Gamma')$.
6. Return $(\\lambda \ x_1 \ldots x_m \rightarrow f \ Y_1 \ldots Y_n)$.

Figure 4.2: Algorithm for sampling from the base prior. Note that the validity of line 3 is guaranteed by Lemma 4.22. Furthermore, at line 5, the $\beta_i$ are guaranteed to be normalised by Defn. 4.11.

prior normalised, i.e. do probabilities sum to one over all long normal, total terms of the target type? Secondly, are partial terms treated appropriately, i.e. does our prior obey Eqn. 3.38?

To help us answer these two questions, we need to take an alternative viewpoint of the prior over total terms as a stochastic process. This is given in Fig. 4.2. The following theorem shows that these ‘distribution’ and ‘process’ views of the prior are equivalent:

**Theorem 4.24** (equivalence of distribution and process views for prior). Let $T$ be a total term in long normal form, let $\tau$ be a normalised odd weighted type such that $\text{stripWeights}(\tau)$ is equal to the type of $T$, and let $\Gamma$ be a normalised pool. It holds that:

$$P(\text{priorProc}(\tau, \Gamma) \text{ returns } T) = \text{prior}(T; \tau, \Gamma)$$

**Proof.** The proof follows straightforwardly by structural induction on $T$, using the definition of $\text{priorProc}$ (Fig. 4.2) and Defn. 4.18. □

The next pair of theorems confirm that our prior is indeed normalised, and that it respects our identification of partial terms with sets of total terms (Eqn. 3.38). Note the condition that $\text{priorProc}$ must terminate with probability one, which is not guaranteed for arbitrary $\Gamma$; this is a significant caveat which I shall discuss shortly.

**Theorem 4.25** (prior is normalised). Let $\tau$ be a normalised odd weighted type, let $\Gamma$ be a normalised pool, and let $S$ be the set of all long normal, total terms
of type \text{stripWeights}(\tau)$. Given the precondition that that \text{priorProc}(\tau, \Gamma) terminates with probability one, it holds that:

$$\sum_{T \in S} \text{prior}(T, \tau, \Gamma) = 1$$

Proof.

$$LHS = \sum_{T \in S} P(\text{priorProc}(\tau, \Gamma) \text{ returns } T) \quad \text{(by 4.24)}$$

$$= RHS \quad \text{(by the precondition)}$$

\[ \square \]

**Theorem 4.26** (prior respects partial terms). Let $T$ be a partial term in long normal form, let $\tau$ be a normalised odd weighted type such that \text{stripWeights}(\tau) is equal to the type of $T$, and let $\Gamma$ be a normalised pool. Given the precondition that \text{priorProc}(\tau, \Gamma) terminates with probability one, it holds that:

$$\text{prior}(T, \tau, \Gamma) = \sum_{X \in T} \text{prior}(X, \tau, \Gamma)$$

Proof. Define $x_1 \ldots x_m$, $f$, and $Y_1 \ldots Y_n$ as in Defn. 4.18. Proceed by structural induction on $T$:

1. $f$ is a hole:

$$LHS = 1 \quad \text{(by 4.23 and 4.18)}$$

$$= RHS \quad \text{(by 4.25 and the precondition)}$$

2. $f$ is a symbol. Define $\Gamma'$ and $\beta_1 \ldots \beta_n$ as in Defn. 4.18:

$$LHS = \text{product}(\text{lookupWeights}(f, \Gamma')) \times \prod_{i=1}^{n} \text{prior}(Y_i, \beta_i, \Gamma') \quad \text{(by 4.23 and 4.18)}$$

$$= \text{product}(\text{lookupWeights}(f, \Gamma')) \times \prod_{i=1}^{n} \sum_{Z_i \in Y_i} \text{prior}(Z_i, \beta_i, \Gamma') \quad \text{(by the inductive hypothesis and the precondition)}$$

$$= \sum_{Z_1 \in Y_1} \ldots \sum_{Z_n \in Y_n} \text{prior}((\wedge x_1 \ldots x_m \rightarrow f \; Z_1 \ldots Z_n), \tau, \Gamma) \quad \text{(by 4.23 and 4.18)}$$

$$= RHS \quad \text{(by 3.31)}$$

\[ \square \]

### 4.1.1 Some examples

Here are some simple examples to illustrate the form of the base prior probability distribution. We shall need the following data type declaration:
data Nat = Zero | Succ Nat

In our first example, consider the following normalised odd weighted type \( \tau \) and normalised pool \( \Gamma \):

\[
\tau = \text{Nat}^1
\]
\[
\Gamma = \left\{ \langle \text{zero}, \text{Nat}^0 \rangle, \langle \text{succ}, \text{Nat}^1 \rightarrow \text{Nat}^{1-\alpha} \rangle \right\}^{(\text{Nat}^0)}
\]

The following table shows terms \( T \) in descending order of probability under \( \text{prior}(T, \tau, \Gamma) \):

<table>
<thead>
<tr>
<th>Term</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>( \alpha )</td>
</tr>
<tr>
<td>Succ Zero</td>
<td>( \alpha(1-\alpha) )</td>
</tr>
<tr>
<td>Succ (Succ Zero)</td>
<td>( \alpha(1-\alpha)^2 )</td>
</tr>
<tr>
<td>Succ (Succ (Succ Zero))</td>
<td>( \alpha(1-\alpha)^3 )</td>
</tr>
<tr>
<td>etc...</td>
<td></td>
</tr>
</tbody>
</table>

Here is another example, this time with the higher order function \( \text{fix} \) in the pool:

\[
\tau = \text{Nat}^1
\]
\[
\Gamma = \left\{ \langle \text{fix}, (\text{Nat}^0 \rightarrow \text{Nat}^{(\text{Nat}^1-\alpha)}) \rightarrow \text{Nat}^1 \rangle \right\}^{(\text{Nat}^0)}
\]

Observe how when a higher order function is invoked, it introduces new local variables which affect the form of the probability distribution:

<table>
<thead>
<tr>
<th>Term</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>fix (\ a -&gt; a)</td>
<td>( \alpha )</td>
</tr>
<tr>
<td>fix (\ a -&gt; fix (\ b -&gt; b))</td>
<td>( \alpha(1-\alpha) )</td>
</tr>
<tr>
<td>fix (\ a -&gt; fix (\ b -&gt; a))</td>
<td>( \alpha(1-\alpha)^2 )</td>
</tr>
<tr>
<td>fix (\ a -&gt; fix (\ b -&gt; fix (\ c -&gt; c)))</td>
<td>( \alpha(1-\alpha)^3 )</td>
</tr>
<tr>
<td>fix (\ a -&gt; fix (\ b -&gt; fix (\ c -&gt; b)))</td>
<td>( \alpha(1-\alpha)^4 )</td>
</tr>
<tr>
<td>fix (\ a -&gt; fix (\ b -&gt; fix (\ c -&gt; a)))</td>
<td>( \alpha(1-\alpha)^5 )</td>
</tr>
<tr>
<td>etc...</td>
<td></td>
</tr>
</tbody>
</table>

Finally, let us discuss the caveat mentioned earlier. Under what sort of circumstances might \( \text{priorProc} \) fail to terminate? Here is such an example:

\[
\tau = \text{Nat}^1
\]
\[
\Gamma = \left\{ \langle \text{succ}, \text{Nat}^1 \rightarrow \text{Nat}^1 \rangle \right\}^{(\text{Nat}^0)}
\]

In this example, there is no variable of arity zero that may terminate the growth of the term, so \( \text{priorProc} \) terminates with probability zero. It appears to be easy enough to avoid this kind of pathological situation simply by adding appropriate arity zero elements to the pool. However, it is clear that non-termination is
an issue to watch out for when using this prior. A theoretical question for
future work might be: what conditions can be placed on \( \tau \) and \( \Gamma \) to guarantee
termination of \( \text{priorProc} \)?

### 4.2 The Nonparametric Prior

The base prior of the previous section features weights attached to elements
of background knowledge that allow a degree of customisation. By modifying
the weights we can change the inductive bias, expressing an assumption that
particular elements of background knowledge are more likely to be useful than
others. However, it is not ideal to have to rely on a human to choose these
weight values. It would be preferable if the machine learning system were able
to infer the best weight values for a particular problem domain automatically.
This amounts to a form of cumulative learning similar to the ‘adaptive Levin
search’ of Schmidhuber et al. [1997].

In this section we shall extend the base prior so as to enable automatic learn-
ing of weights. To do this, we follow the example of \textit{nonparametric Bayesian
methods} [Gershman and Blei, 2012]. The basic idea behind nonparametric
Bayesian methods is as follows. Consider a Bayesian model such as a Gauss-
ian mixture model. The mixture model is parametrised by an integer number
of clusters \( k \). How does one go about choosing the most suitable value of \( k \)
for a particular inference problem? One solution might be to repeatedly fit the
mixture model to our data, each time using a different value of \( k \). Then at the
end, we could choose whichever value of \( k \) produced the best fit to the data
according to some goodness of fit metric (which might be predictive accuracy
on a validation dataset). On the other hand, the nonparametric Bayesian ap-
proach is much more elegant. It tells us to design our model so that uncertainty
in the value of \( k \) is built in, i.e. the model is flexible enough to consider all
possibilities for \( k \) simultaneously. The nonparametric version of a mixture model
is called a \textit{Chinese restaurant process model}. By analogy, I shall construct a
nonparametric version of last section’s base prior in which uncertainty in the
weight values is built in.

First, some preliminaries. I shall be making use of a standard probability
distribution called the \textit{Dirichlet distribution} [Frigyik et al., 2010]. The Dirichlet
distribution is a probability distribution over categorical probability distribu-
tions:

**Definition 4.27** (Dirichlet distribution). An \( n \)-component \textit{Dirichlet distri-
bution} is a probability distribution whose domain is equal to the following:

\[
\{ [w_1 \ldots w_n] \in \text{lists of } n \text{ weights } \bar{\sum_{i=1}^{n} w_i = 1} \}
\]
where \( n \geq 1 \). The distribution is parametrised by \( n \) positive real numbers denoted below as \( a_1 \ldots a_n \). Its probability density function \( \text{dirichletDensity} \) is as follows:

\[
\text{dirichletDensity}([w_1 \ldots w_n], [a_1 \ldots a_n]) = G(A) \prod_{i=1}^{n} \frac{w_i^{a_i-1}}{G(a_i)}
\]

where \( A = \sum_{i=1}^{n} a_i \)

\( G \) is the gamma function

Note that \( \text{dirichletDensity} \) normalises to 1 when integrated over its \((n - 1)\)-dimensional domain.

We see that when one samples from a Dirichlet distribution, the result is an \( n \)-component categorical distribution. Suppose that, in turn, one were to sample repeatedly from this categorical distribution and observe a sequence of outcomes. Let \( x_i \) be the number of times that an outcome of the \( i \)th category occurs in the sequence. The following identity then gives us the probability of observing such a sequence, marginalised over the (un-observed) weight values of the categorical distribution:

**Proposition 4.28.** Let \( a_1 \ldots a_n \) be positive real numbers and let \( x_1 \ldots x_n \) be non-negative integers, with \( n \geq 1 \). The following then holds, where the integral is over the domain of the \( n \)-component Dirichlet distribution:

\[
\int \left( \prod_{i=1}^{n} w_i^{x_i} \right) \text{dirichletDensity}([w_1 \ldots w_n], [a_1 \ldots a_n]) dw_1 \ldots dw_{n-1} = \frac{G(A)}{G(A + X)} \prod_{i=1}^{n} G(a_i + x_i)
\]

where \( A = \sum_{i=1}^{n} a_i \)

\( X = \sum_{i=1}^{n} x_i \)

\( G \) is the gamma function

**Proof.** See [Frigyik et al., 2010, Sect. 1.4].

I shall now describe the nonparametric extension of last section’s base prior. Just as for the base prior, there is a ‘process’ view and a ‘distribution’ view of the nonparametric prior. The most intuitive way to understand this prior is to start with the process view, to let us discuss that first.

Figure 4.5 gives the definition of the non-parametric prior in the form of a stochastic process \( npPriorProc \). Observe that the first action of \( npPriorProc \) is to generate some weight values using stochastic functions \( \text{pickWeightsOdd} \) and \( \text{pickPoolWeights} \). Having chosen these weights, \( npPriorProc \) then delegates to the base prior process in order to generate a term. \( \text{pickWeightsOdd} \) (Fig. 4.3)
and \textit{pickPoolWeights} (Fig. 4.4) both work by randomly sampling weight values from Dirichlet distributions. The Dirichlet distributions depend on two hyper-parameters (Defn. 4.29).

\textbf{Definition 4.29} \((npPriorHyperParam_1, npPriorHyperParam_2)\). The non-parametric prior has two positive, real-valued hyperparameters: \(npPriorHyperParam_1\) and \(npPriorHyperParam_2\). Both are concentration parameters for Dirichlet distributions (see Fig. 4.3 and Fig. 4.4).

We have seen that the process view gives us a simple way to understand what the nonparametric prior means: it is essentially the same as the base prior, except that weight values are generated by randomly sampling from Dirichlet distributions rather than being pre-specified. On the other hand, the process view alone does not give us an effective procedure for computing the prior probability of a given term. For this we need to convert to a \textit{distribution} view, i.e. we need a probability mass function for our prior.

How can we perform this conversion from process to distribution view? Recall that \textit{npPriorProc} works by first sampling the weight values from Dirichlet distributions, then it delegates to the base prior to sample the lambda term itself. Since we know the probability density function for a Dirichlet distribution (Defn. 4.27), and we know the probability mass function for the base prior (Defn. 4.23), we can write down the joint probability density of a given configuration of weights \textit{and} a given lambda term. In order to recover the probability mass function for a lambda term we simply need to marginalise this joint distribution over all possible weight values. This yields our distribution view of the nonparametric prior, \textit{npPrior}, given in Fig. 4.7.

\textit{npPrior} uses the auxiliary functions \textit{pickWeightIndicesOdd} and \textit{pickPoolWeightIndices} (Fig. 4.6) to generate an \textit{indirected} odd weighted type and an \textit{indirected} pool (Defn. 4.31). These indirected structures are a way of keeping track of where weights should go without actually generating the weight values themselves. At line 4 of Fig. 4.7, \textit{npPrior} delegates to \textit{priorWeights} in order to determine the number of times each weight gets used in the calculating the probability of the term. The final expression for the probability of the term at line 10 of Fig. 4.7 comes from marginalisation over the weights by means of Proposition 4.28. For the proof of equivalence between the process and distributional views of the nonparametric prior see Theorem 4.34 below.

\textbf{Definition 4.30} (weight index). A \textit{weight index} is either 0, or it is a pair of the form \(\langle u, c \rangle\) where \(u\) and \(c\) are positive integers. We call \(u\) the \textit{urn index} and \(c\) the \textit{colour}.

\textbf{Definition 4.31} (indirected weighted type, indirected frame, indirected pool). An \textit{indirected even/odd weighted type} is the same as an even/odd weighted type
**stochastic function** \( \text{pickWeightsOdd}(\tau) \):

**input**: \( \tau \), a type.

**output**: a normalised odd weighted type.

1. Let \( \alpha_1 \rightarrow \ldots \rightarrow \alpha_m \rightarrow R = \tau \), where \( R \) is a data type.
2. Partition \( \{1 \ldots m\} \) into non-empty disjoint subsets, each of the form \( \{i \in \{1 \ldots m\} \mid \text{the return type of } \alpha_i \text{ is equal to } S \} \) for some data type \( S \).
   
   Let \( M \) be the number of such subsets, and for each \( k \)th subset \( \{i_{k1} \ldots i_{k_n_k}\} \) corresponding to data type \( S_k \):
3. Let \( \gamma_1 \ldots \gamma_{n_k} \) all be equal to \( npPriorHyperParam_1 \). Let \( \gamma_{n_k+1} \) be equal to \( npPriorHyperParam_2 \).
4. Pick weights \( [w_{k1} \ldots w_{k(n_k+1)}] \) by sampling from an \((n_k+1)\)-component Dirichlet distribution with parameter vector \( [\gamma_1 \ldots \gamma_{n_k+1}] \).
5. For \( j \in \{1 \ldots n_k\} \), pick \( \alpha_i'_{kj} \) by calling \( \text{pickWeightsEven}(\alpha_{kj}, w_{kj}) \).
6. Let \( X \), an odd weight annotation, equal \( \{k \in \{1 \ldots M\} \mid \langle S_k, w_{k(n_k+1)} \rangle \} \).
7. Return \( \alpha_1' \rightarrow \ldots \rightarrow \alpha_m' \rightarrow R^X \).

**stochastic function** \( \text{pickWeightsEven}(\tau, w) \):

**input**: \( \tau \), a type;

\( w \), a weight.

**output**: a normalised even weighted type.

1. Let \( \alpha_1 \rightarrow \ldots \rightarrow \alpha_m \rightarrow R = \tau \), where \( R \) is a data type.
2. For \( i \in \{1 \ldots m\} \), pick \( \alpha_i' \) by calling \( \text{pickWeightsOdd}(\alpha_i) \).
3. Return \( \alpha_1' \rightarrow \ldots \rightarrow \alpha_m' \rightarrow R^w \).

Figure 4.3: \( \text{pickWeightsOdd} \) is a stochastic process for assigning weight annotations to a type. Weights are obtained by sampling from Dirichlet distributions. \( \text{pickWeightsEven} \) is an auxiliary function. The weighted types output by these functions are guaranteed to be normalised, a fact which follows straightforwardly by structural induction on \( \tau \) using Defn. 4.20.
stochastic function \textit{pickPoolWeights}(\Gamma):
\begin{enumerate}
\item Let \{x_1 \ldots x_m\} = \Gamma, and let \(\alpha_i\) be the type of \(x_i\) for each \(i \in \{1 \ldots m\}\).
\item Partition \{1 \ldots m\} into non-empty disjoint subsets, each of the form \(\{i \in \{1 \ldots m\} \mid\text{the return type of } \alpha_i\text{ is equal to } S\}\) for some data type \(S\). Let \(M\) be the number of such subsets, and for each \(k\)th subset \(\{i_k1 \ldots i_kn_k\}\) corresponding to data type \(S_k\):
\item Let \(\gamma_1 \ldots \gamma_{n_k}\) all be equal to \(npPriorHyperParam_1\).
\item Pick weights \([w_{k1} \ldots w_{kn_k}]\) by sampling from an \(n\)-component Dirichlet distribution with parameter vector \([\gamma_1 \ldots \gamma_{n_k}]\).
\item For \(j \in \{1 \ldots n_k\}\), pick \(\alpha'_{ikj}\) by calling \(pickWeightsEven(\alpha_{ikj}, w_{kj})\).
\item Let \(X\), an odd weight annotation, equal \(\{k \in \{1 \ldots M\} \bullet (S_k, 0)\}\).
\item Return \([\langle x_1, \alpha'_{11} \rangle \ldots \langle x_m, \alpha'_{m_{n_k}}\rangle]^X\].
\end{enumerate}

Figure 4.4: Stochastic process for assigning weight annotations to a set of symbols so as to create a pool consisting of a single frame. By Defn. 4.20, this pool is guaranteed to be normalised.

stochastic function \textit{npPriorProc}(\tau, \Gamma):
\begin{enumerate}
\item Pick \(\tau'\) by calling \(pickWeightsOdd(\tau)\).
\item Pick \(\Gamma'\) by calling \(pickPoolWeights(\Gamma)\).
\item Return \(priorProc(\tau', \Gamma')\).
\end{enumerate}

Figure 4.5: Algorithm for sampling from the non-parametric prior.
except that weight indices are used in all contexts instead of weights, i.e. as even weight annotations and within odd weight annotations. Indirected frame and indirected pool are defined in the same manner. The functions stripWeights, zeroWeightsOdd, frameSymbols, dropThruWeights, lookupArgs, lookupWeights, and priorWeights are extended to apply in the context of indirected weighted types, indirected frames, and indirected pools. Again, this is achieved by dealing with weight indices wherever weights were used before.

**Definition 4.32** (urn). An urn is a list of real numbers.

**Lemma 4.33.** Let $T$ be a partial term in long normal form, let $\tau$ be the type of $T$, and let $\Gamma$ be a set of symbols not containing void. Given the precondition that $npPriorProc(\tau, \Gamma)$ terminates with probability one, it holds that:

$$\sum_{Y \in T} P(npPriorProc(\tau, \Gamma) \text{ returns } Y) = npPrior(T, \Gamma)$$

**Proof.** Define $m$ and $W$ as in Fig. 4.7, and for each $i \in \{1 \ldots m\}$ define $a_{i1} \ldots a_{in_i}$, $A_i$, $x_{i1} \ldots x_{in_i}$, and $X_i$ as in Fig. 4.7. If no element of $W$ is equal to 0, then proceed as follows. Note that we use the somewhat informal notation $\int d\tau'd\Gamma'$ to express a multi-dimensional integral over the weights within a weighted type.
stateful function \textit{pickWeightIndicesOdd}(\tau):

\textbf{input:} \ \tau, a type.

\textbf{output:} an indirected odd weighted type.

\textbf{state:} \ U, a list of urns.

The implementation of this function is identical to that of \textit{pickWeightsOdd}, except that lines 4 and 5 are different as follows:

4. \ \ U \leftarrow U + + [\gamma_1 \ldots \gamma_{n_{k+1}}]. \quad \text{For } j \in \{1 \ldots n_{k+1}\}, \text{ let } w_{kj} = \langle u,j \rangle \text{ where } u \text{ is the new length of } U.

5. \quad \text{For } j \in \{1 \ldots n_k\}, \text{ call } \textit{pickWeightIndicesEven}(\alpha_{i_k,j}, w_{kj}) \text{ and let } \alpha'_{i_k,j} \text{ be the result.}

stateful function \textit{pickWeightIndicesEven}(\tau,w):

\textbf{input:} \ \tau, a type;

\textit{w}, a weight index.

\textbf{output:} an indirected even weighted type.

\textbf{state:} \ U, a list of urns.

The implementation of this function is identical to that of \textit{pickWeightsEven}, except that lines 2 is different as follows:

2. \quad \text{For } i \in \{1 \ldots m\}, \text{ call } \textit{pickWeightIndicesOdd}(\alpha_i) \text{ and let } \alpha'_i \text{ be the result.}

stateful function \textit{pickPoolWeightIndices}(\Gamma):

\textbf{input:} \ \Gamma, a set of symbols not containing \texttt{void}.

\textbf{output:} an indirected pool.

\textbf{state:} \ U, a list of urns.

The implementation of this function is identical to that of \textit{pickPoolWeights}, except that lines 4 and 5 are different as follows:

4. \quad \ U \leftarrow U + + [\gamma_1 \ldots \gamma_{n_k}]. \quad \text{For } j \in \{1 \ldots n_k\}, \text{ let } w_{kj} = \langle u,j \rangle \text{ where } u \text{ is the new length of } U.

5. \quad \text{For } j \in \{1 \ldots n_k\}, \text{ call } \textit{pickWeightIndicesEven}(\alpha_{i_k,j}, w_{kj}) \text{ and let } \alpha'_{i_k,j} \text{ be the result.}

Figure 4.6: Procedures for assigning weight indices to types and pools. These weight indices refer to urns and colours within an accumulated list of urns \( U \).
function npPrior(T, Γ):

    input: T, a term in long normal form;
          Γ, a set of symbols not containing void.

    output: a probability value.

1. Let τ be the type of T.

2. Execute pickWeightIndicesOdd(τ) with initial state []. Let τ’ be the output and let S be the final state.

3. Execute pickPoolWeightIndices(Γ) with initial state S. Let Γ’ be the output and let [u₁…uₘ] be the final state.

4. Let W, a list of weight indices, equal priorWeights(T, τ’, Γ’).

5. For each i ∈ {1…m}:


7. Let Aᵢ = ∑_{j=1}^{nᵢ} aᵢⱼ.

8. For each j ∈ {1…nᵢ}, let xᵢⱼ equal the number of occurrences of the weight index (i,j) in W.

9. Let Xᵢ = ∑_{j=1}^{nᵢ} xᵢⱼ.

10. If any element of W is equal to 0 then return 0, else return:

    \[
    \prod_{i=1}^{m} \left[ \frac{G(Aᵢ)}{G(Aᵢ + Xᵢ)} \prod_{j=1}^{nᵢ} \frac{G(aᵢⱼ + xᵢⱼ)}{G(aᵢⱼ)} \right]
    \]

    where G is the gamma function.

Figure 4.7: Procedure for calculating the probability of a term under the non-parametric prior.
and pool:

\[ LHS = \sum_{Y \in T} \int d\tau' d\Gamma' \times \]

(probability density of \(pickWeightsOdd(\tau)\) at \(\tau')\) \times

(probability density of \(pickPoolWeights(\Gamma)\) at \(\Gamma')\) \times

\(prior(Y, \tau', \Gamma')\) (by the definition of \(npPriorProc\) and 4.24)

\[ = \int d\tau' d\Gamma' \times \]

(probability density of \(pickWeightsOdd(\tau)\) at \(\tau')\) \times

(probability density of \(pickPoolWeights(\Gamma)\) at \(\Gamma')\) \times

\(prior(T, \tau', \Gamma')\) (by 4.26 and the precondition)

\[ = \prod_{i=1}^{m} \int dw_{i1} \ldots dw_{i(n_{i}-1)} \times \]

\[ \prod_{i=1}^{m} \prod_{j=1}^{n_{i}} \text{dirichletDensity}([w_{i1}, \ldots, w_{in_{i}}], [a_{i1}, \ldots, a_{in_{i}}]) \times \]

\[ \prod_{i=1}^{m} n_{i} \prod_{j=1}^{w_{ij}} \] (by 4.23 and the definitions of \(pickWeightsOdd\), \(npPrior\), etc.)

\[ = \prod_{i=1}^{m} \int dw_{i1} \ldots dw_{i(n_{i}-1)} \times \]

\[ \text{dirichletDensity}([w_{i1}, \ldots, w_{in_{i}}], [a_{i1}, \ldots, a_{in_{i}}]) \prod_{j=1}^{w_{ij}} \] (by 4.28 and the definition of \(npPrior\))

On the other hand, if some element of \(W\) is equal to 0, then the first two steps in the proof are the same as above. The remainder of the proof then follows using the fact that when \(W\) contains 0 it holds that \(prior(T, \tau', \Gamma') = 0\) for all \(\tau', \Gamma'\) in the domain of the integral.

\[ \square \]

**Theorem 4.34** (equivalence of distribution and process views for \(npPrior\)).

Let \(T\) be a total term in long normal form, let \(\tau\) be the type of \(T\), and let \(\Gamma\) be a set of symbols not containing void. It holds that:

\[ P(npPriorProc(\tau, \Gamma) \text{ returns } T) = npPrior(T, \Gamma) \]

**Proof.** The proof follows in identical fashion to that of Lemma 4.33, except that there is no need for the precondition at the second line of working. \[ \square \]

The following two theorems confirm, as was earlier shown for the base prior, that the nonparametric prior is normalised and that it respects partial terms:

**Theorem 4.35** (\(npPrior\) is normalised). Let \(\tau\) be a type, let \(\Gamma\) be a set of symbols not containing void, and let \(S\) be the set of all long normal, total terms of type
τ. Given the precondition that npPriorProc(τ, Γ) terminates with probability one, it holds that:

\[
\sum_{T \in S} npPrior(T, Γ) = 1
\]

Proof.

\[
LHS = \sum_{T \in S} P(npPriorProc(τ, Γ) \text{ returns } T) \quad (\text{by 4.34})
\]
\[
= RHS \quad (\text{by the precondition})
\]

\[\square\]

Theorem 4.36 (npPrior respects partial terms). Let T be a partial term in long normal form, let τ be the type of T, and let Γ be a set of symbols not containing void. Given the precondition that npPriorProc(τ, Γ) terminates with probability one, it holds that:

\[
npPrior(T, Γ) = \sum_{X \in T} npPrior(X, Γ)
\]

Proof. The proof follows by Theorem 4.34 and Lemma 4.33. \[\square\]

Lastly, let us discuss how the nonparametric prior may be used to do a form of cumulative learning, i.e. to automatically optimise weight values over the course of a sequence of inductive inference problems. Suppose we have a list of target data \(D_1 \ldots D_n\) with corresponding target types \(α_1 \ldots α_n\), for \(n\) related inductive inference problems. If we introduce the following data type declaration:

```
data Tuple = Tuple α₁ ... αₙ
```

then we can express all of these problems together as a single, monolithic inductive inference problem with the following target datum:

\[D = Tuple D_1 \ldots D_n\]

Hypotheses for the monolithic problem take the form:

\[H = Tuple H_1 \ldots H_n\]

Suppose we use the nonparametric prior as the prior for this monolithic problem, i.e. \(P(H) = npPrior(H, Γ)\) for some appropriate choice of Γ. Also, we ensure that Γ contains precisely one symbol of return type Tuple, namely the symbol Tuple. Under these conditions, we can decompose the prior probability of a hypothesis for the monolithic problem as follows:

\[
P(H) = \prod_{i=1}^{n} P(H_i|H_1 \ldots H_{i-1})
\]
Note that these conditional probabilities are straightforward to calculate using the definition of \( npPrior \) and the fact that \( npPrior \) respects partial terms.

We can attempt to solve the monolithic problem by sequentially solving each of the sub-problems, in each case using \( P(H_i|H_1 \ldots H_{i-1}) \) as the prior for that sub-problem. Notice that due to the way we have used the nonparametric prior, the hypotheses \( H_1 \ldots H_n \) are not independent random variables. This is due to the sharing of the same weight values among all of the sub-problems. Therefore, conditioning on the hypotheses for earlier problems in this way amounts to a shift in inductive bias as one progresses through the sequence.

### 4.2.1 Worked example

Let us compare the forms of the non-parametric prior and the base prior on a simple hypothesis space. We shall also see in more detail how the non-parametric prior may be used to automatically adapt weight values over the course of multiple problems. In this example we make use of the following data type declaration, which is equivalent to a grammar for words consisting of any number of 'A's and 'B's followed by a 'C':

```plaintext
data Word = A Word |
        B Word |
        C
```

Consider the base prior over long normal terms \( T \) of type \( Word \), given by \( prior(T, \tau, \Gamma) \) where:

\[
\tau = Word^{(1)}
\]

\[
\Gamma = \{ [\{ A, Word^{(1)} \rightarrow Word^{\alpha} \}, \{ B, Word^{(1)} \rightarrow Word^{\beta} \}, \{ C, Word^{\gamma} \}] \}^{\{Word,0\}^1}
\]

Here \( \alpha, \beta, \gamma \) are weights such that \( \alpha + \beta + \gamma = 1 \). The following table shows the probability values for the first few terms under this distribution:

<table>
<thead>
<tr>
<th>Term</th>
<th>Probability</th>
<th>Probability (( \alpha = \beta = \gamma ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>( \gamma )</td>
<td>( \frac{1}{3} )</td>
</tr>
<tr>
<td>A C</td>
<td>( \alpha\gamma )</td>
<td>( \frac{1}{9} )</td>
</tr>
<tr>
<td>B C</td>
<td>( \beta\gamma )</td>
<td>( \frac{1}{9} )</td>
</tr>
<tr>
<td>A (A C)</td>
<td>( \alpha^2\gamma )</td>
<td>( \frac{1}{27} )</td>
</tr>
<tr>
<td>A (B C)</td>
<td>( \alpha\beta\gamma )</td>
<td>( \frac{1}{27} )</td>
</tr>
<tr>
<td>B (A C)</td>
<td>( \beta\alpha\gamma )</td>
<td>( \frac{1}{27} )</td>
</tr>
<tr>
<td>B (B C)</td>
<td>( \beta^2\gamma )</td>
<td>( \frac{1}{27} )</td>
</tr>
<tr>
<td>etc...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For the sake of concreteness, the right-hand column shows the probability values in the special case of uniform weights.
Now consider the non-parametric prior over the same hypothesis space, given by \( npPrior(T, \Gamma) \) where:

\[
\Gamma = \{ A : \text{Word} \rightarrow \text{Word}, \\
B : \text{Word} \rightarrow \text{Word}, \\
C : \text{Word} \}
\]

The probability values for the first few terms under this distribution are as follows, where \( h = npPriorHyperParam_1 \) (Defn. 4.29). The special case of \( h = 1 \) is also shown for concreteness:

<table>
<thead>
<tr>
<th>Term</th>
<th>Probability</th>
<th>Probability ((h = 1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>( \frac{1}{3} \times \frac{h}{3h+1} )</td>
<td>( \frac{1}{3} )</td>
</tr>
<tr>
<td>A C</td>
<td>( \frac{1}{3} \times \frac{h}{3h+1} )</td>
<td>( \frac{1}{12} )</td>
</tr>
<tr>
<td>B C</td>
<td>( \frac{1}{3} \times \frac{h}{3h+1} )</td>
<td>( \frac{1}{12} )</td>
</tr>
<tr>
<td>A (A C)</td>
<td>( \frac{1}{3} \times \frac{h+1}{3h+1} \times \frac{h}{3h+2} )</td>
<td>( \frac{1}{30} )</td>
</tr>
<tr>
<td>A (B C)</td>
<td>( \frac{1}{3} \times \frac{h}{3h+1} \times \frac{h}{3h+2} )</td>
<td>( \frac{1}{50} )</td>
</tr>
<tr>
<td>B (A C)</td>
<td>( \frac{1}{3} \times \frac{h}{3h+1} \times \frac{h}{3h+2} )</td>
<td>( \frac{1}{50} )</td>
</tr>
<tr>
<td>B (B C)</td>
<td>( \frac{1}{3} \times \frac{h+1}{3h+1} \times \frac{h}{3h+2} )</td>
<td>( \frac{1}{50} )</td>
</tr>
</tbody>
</table>

With the base prior, each symbol in a term receives a fixed weight, and the total probability is obtained by multiplying all of these weights together. On the other hand, for the non-parametric prior, we see that the weight of a symbol increases the more frequently that symbol occurs. For example, \( A \ (A \ C) \) receives a higher probability than \( A \ (B \ C) \) in the table above, because the first occurrence of \( A \) effectively boosts the probability of the second occurrence. We may say that the weights ‘adapt’ according to how frequently the corresponding symbols actually occur.

The hyperparameter \( h \) determines how quickly the weights adapt. When \( h \) is small the weights will adapt rapidly. When \( h \rightarrow \infty \), the weights do not change at all, and the non-parametric prior becomes identical to the base prior with uniform weights.

Let us now look at how weight adaptation can shift inductive bias over the course of multiple problems. Following the scheme outlined at the end of the last subsection, consider a tuple data type for a monolithic problem consisting of two sub-problems:

```haskell
data Tuple = Tuple Word Word
```
Let us use a non-parametric prior over monolithic hypotheses $T$ of type \texttt{Tuple}, given by $\texttt{npPrior}(T, \Gamma)$ where:

$$
\Gamma = \{ \text{Tuple}: (\text{Word} \to \text{Word} \to \text{Tuple}), \\
A: (\text{Word} \to \text{Word}), \\
B: (\text{Word} \to \text{Word}), \\
C: \text{Word} \}
$$

Note that any non-trivial $T$ must be of the form $\text{Tuple} \ H_1 \ H_2$ where $H_1$ and $H_2$ are each of type \texttt{Word}. $H_1$ and $H_2$ are hypotheses for the first and second sub-problems respectively. Now, suppose that the first of these two sub-problems has just been solved (with respect to some target datum) and that the following concrete value has been chosen for $H_1$:

$$
H_1 = A (A (A C))
$$

We may write down the conditional probability of $H_2$ given $H_1$, as follows:

$$
P(H_2|H_1) = \frac{P(H_1, H_2)}{P(H_1)}
= \frac{\texttt{npPrior}(\text{Tuple} \ H_1 \ H_2, \Gamma)}{\texttt{npPrior}(\text{Tuple} \ H_1 \ #, \Gamma)}
$$

The following table shows the conditional probability values for the first few terms $H_2$, given that $H_1 = A (A (A C))$:

| $H_2$ | $P(H_2|H_1)$ | $P(H_2|H_1)$ when $h = 1$ |
|-------|-------------|-----------------------------|
| C     | $\frac{h+1}{3h+4}$ | $\frac{2}{7}$ |
| A C   | $\frac{h+3}{3h+4}$ $\times$ $\frac{h+1}{3h+5}$ | $\frac{1}{7}$ |
| B C   | $\frac{h}{3h+4}$ $\times$ $\frac{h+1}{3h+5}$ | $\frac{1}{28}$ |
| A (A C) | $\frac{h+3}{3h+4}$ $\times$ $\frac{h+1}{3h+5}$ $\times$ $\frac{h+1}{3h+6}$ | $\frac{5}{63}$ |
| A (B C) | $\frac{h+3}{3h+4}$ $\times$ $\frac{h}{3h+5}$ $\times$ $\frac{h+1}{3h+6}$ | $\frac{1}{63}$ |
| B (A C) | $\frac{h}{3h+4}$ $\times$ $\frac{h+3}{3h+5}$ $\times$ $\frac{h+1}{3h+6}$ | $\frac{1}{63}$ |
| B (B C) | $\frac{h}{3h+4}$ $\times$ $\frac{h+1}{3h+5}$ $\times$ $\frac{h+1}{3h+6}$ | $\frac{1}{126}$ |

We can see from the above table that conditioning on the known value of $H_1$ has shifted the bias for the second problem, favouring hypotheses $H_2$ that contain more $A$s than $B$s. This has occurred because $H_1$ contains mostly $A$s and no $B$s.

### 4.3 The Cumulative Learning Prior

We have seen that the nonparametric prior can be used for cumulative learning of the probability weights attached to elements of background knowledge. In
this section I introduce an extension of this prior to allow cumulative learning of the elements of background knowledge themselves, as well as their weights. I call this the cumulative learning prior.

The cumulative learning prior shall require the following infinite family of data types and value declarations to be included in the definition of the toplevel environment. For all types \( \alpha_1 \ldots \alpha_m \) where \( m \geq 0 \), we introduce the following data type declaration:

\[
\text{data } \text{Tuple}_{\alpha_1 \ldots \alpha_m} = \text{Tuple } \alpha_1 \ldots \alpha_m
\]

where each \( \text{Tuple}_{\alpha_1 \ldots \alpha_m} \) is a unique reserved data type. Then, for all \( m, n \geq 0 \) and for all appropriate type annotations of a symbol \text{fixtuple} and a symbol \text{master}, we introduce the following value declarations:

\[
\text{fixTuple} = \text{fix } (\lambda \text{r f} \rightarrow f (\text{caseTuple } (\text{r f}) (\lambda a_1 \ldots a_m \rightarrow a_1)) \ldots (\text{caseTuple } (\text{r f}) (\lambda a_1 \ldots a_m \rightarrow a_m)))
\]

\[
\text{master} = (\lambda f \rightarrow \text{caseTuple} \ (\text{fixTuple } (\lambda a_1 \ldots a_m b_1 \ldots b_n \rightarrow f a_1 \ldots a_m)) \ (\lambda a_1 \ldots a_m b_1 \ldots b_n \rightarrow \text{Tuple } b_1 \ldots b_n))
\]

\text{fixtuple} is a variation of \text{fix}. It constructs a tuple whose elements are defined mutually recursively in terms of one another. Each allowed type annotation for \text{fixtuple} has the form:

\[
(\alpha_1 \rightarrow \ldots \rightarrow \alpha_m \rightarrow \text{Tuple}_{\alpha_1 \ldots \alpha_m}) \rightarrow \text{Tuple}_{\alpha_1 \ldots \alpha_m}
\]

Each allowed type annotation for \text{master} has the form:

\[
(\alpha_1 \rightarrow \ldots \rightarrow \alpha_m \rightarrow \text{Tuple}_{\alpha_1 \ldots \alpha_m \beta_1 \ldots \beta_n}) \rightarrow \text{Tuple}_{\beta_1 \ldots \beta_n}
\]

Here, the \( \alpha_i \) are types of \( m \) invented elements of background knowledge and the \( \beta_j \) are target types for \( n \) inductive inference problems. The value declaration for \text{master} uses \text{fixTuple} to construct a recursive definition of the \( m \) invented elements of background knowledge in terms of one another, as well as definitions of hypotheses for the \( n \) inference problems in terms of the invented background knowledge. It is this sharing of the same invented background knowledge among all of the hypotheses that allows cumulative learning.

The cumulative learning prior is parametrised by two probability distributions, \text{numAbstractionsDist} and \text{abstractionTypeDist} (Defn. 4.37). \text{numAbstractionsDist} is used to choose the number of invented elements of background knowledge; a suitable choice of distribution might be a geometric or a zeta
stochastic function \( clPriorProc(tys, \Gamma) \):

- **input**: \( tys \), a list of types;
  \( \Gamma \), a set of symbols not containing `void`.

- **output**: a long normal, total term.

1. Pick \( m \) by sampling from \( numAbstractionsDist \).
2. For each \( i \in \{1 \ldots m\} \):
   3. Pick \( \tau_i \) by sampling from \( abstractionTypeDist \).
   4. Let \( g_i \) be a fresh symbol of type \( \tau_i \).
   5. Pick \( \Gamma' \) by calling \( pickPoolWeights(\Gamma \cup \{g_1 \ldots g_m\}) \).
6. Let \( [\tau_{m+1} \ldots \tau_{m+n}] = tys \).
7. For each \( i \in \{1 \ldots m + n\} \):
   8. Pick \( \tau'_i \) by calling \( pickWeightsOdd(\tau_i) \).
   9. Pick \( T_i \) by calling \( priorProc(\tau'_i, \Gamma') \).
10. Return:

\[
\text{master } (\setminus g_1 \ldots g_m \rightarrow \text{tuple } T_1 \ldots T_{m+n})
\]

Figure 4.8: Algorithm for sampling from the cumulative learning prior. In lines 1–4, the number and types of abstractions (invented elements of background knowledge) are chosen. In lines 5–9, weights and terms are chosen as in \( npPriorProc \) (Fig. 4.5). Finally in line 10, a monolithic hypothesis is constructed and returned.

distribution. \( abstractionTypeDist \) is used to choose the type of each invented element of background knowledge. Here the simplest choice of distribution would be a uniform distribution over some finite set of types. On the other hand one might define a distribution on types along the same lines as the base prior or nonparametric prior on terms, in order to build in the principle of Occam’s Razor into the choice of type.

**Definition 4.37** (\( numAbstractionsDist, abstractionTypeDist \)).
\( numAbstractionDist \) is some probability distribution over the non-negative integers. \( abstractionTypeDist \) is some probability distribution over types.

The definition of the cumulative learning prior is given in Fig. 4.8 in the form of a stochastic process. It is straightforward to convert this from process form to distribution form using the same technique as for the nonparametric prior of the previous section, so I shall not give the details here.

Let us now look at how to actually perform cumulative learning with the cumulative learning prior. As in the weights adaptation scheme of the previous section, we work with a monolithic inductive inference problem whose target
datum has the following form:

\[ D = \text{Tuple} \ D_1 \ldots \ D_n \]

However, this time a hypothesis for the monolithic problem has this form:

\[ H = \text{master} \ (\ \backslash \ a_1 \ldots a_m \ \rightarrow \ \text{Tuple} \ A_1 \ldots A_m \ H_1 \ldots H_n) \]

where \( m \) may be any non-negative integer. Here, the \( A_i \) are abstractions, i.e. invented elements of background knowledge, and the \( H_i \) are hypotheses for the individual sub-problems.

In order to cumulatively learn the hypotheses \( H_i \), one would solve the sub-problems sequentially, in each case using \( P(H_i|A_1 \ldots A_m, H_1 \ldots H_{i-1}) \) as the prior for the sub-problem. In order to learn the abstractions \( A_i \), one needs abstraction invention, a mechanism for which I shall describe in Chap. 7.

### 4.3.1 Worked example

Let us now look at a concrete example of how to use the cumulative learning prior. We shall again make use of the \texttt{Word} data type that was introduced in the example of Sect. 4.2.1:

\begin{verbatim}
data Word = A Word
    | B Word
    | C
\end{verbatim}

Consider the hypothesis space of master terms \( T \) of the following form:

\[ T = \text{master} \ (\ \backslash \ a_1 \ldots a_m \ \rightarrow \ \text{Tuple} \ A_1 \ldots A_m \ H_1 \ H_2) \]

where \( m \geq 0 \) and \( H_1, H_2 \) both have type \texttt{Word}. The type of \( T \) is \texttt{Tuple\textunderscore Word\textunderscore Word}. In this master term, hypotheses for two sub-problems \( H_1 \) and \( H_2 \) are being defined with respect to a number of abstractions. Let the prior over this hypothesis space be given by the stochastic process \( clPriorProc([\texttt{Word}, \texttt{Word}], \Gamma) \) where:

\[ \Gamma = \{ A: (\texttt{Word} \rightarrow \texttt{Word}), \quad B: (\texttt{Word} \rightarrow \texttt{Word}), \quad C: \texttt{Word} \} \]

Let \( clPrior \) be the distribution form of \( clPriorProc \), satisfying the following identity:

\[ clPrior(T, \Gamma) = P(clPriorProc([\texttt{Word}, \texttt{Word}], \Gamma) \text{ returns } T) \]
If we initially assume that the number of abstractions $m$ is zero, then the prior probability of a given hypothesis $H_1$ for the first sub-problem is equal to:

$$
P(H_1|m = 0) = \frac{\text{clPrior}(\text{master } (\text{Tuple } H_1 \#), \Gamma)}{\text{clPrior}(\text{master } (\text{Tuple } \# \#), \Gamma)} = \text{npPrior}(H_1, \Gamma)
$$

(by the definitions of $\text{clPrior}$ and $\text{npPrior}$)

Thus we see that when there are no abstractions, the cumulative learning prior effectively reduces to the non-parametric prior of the previous section.

Now, suppose that the following value of $H_1$ happens to be our chosen solution to the first sub-problem:

$$
H_1 = A (B (A (B (B (B (A (B (B (B (A (B (B (A (B (B C))))))))))))))
$$

The prior probability of this solution is as follows where $h = \text{npPriorHyperParam}_1$:

$$
P(H_1|m = 0) = \frac{h^3(h + 1)^2(h + 2)^2(h + 5)^2(h + 6)(h + 7)\ldots(h + 11)}{(3h)(3h + 1)(3h + 2)\ldots(3h + 18)} = 1.35 \times 10^{-8} \text{ when } h = 1
$$

Given the obvious repeating pattern ‘ABB’ in $H_1$, we might consider introducing the following abstraction so as to express $H_1$ more concisely:

$$
A_1 = (\lambda x \rightarrow A (B (B x)))
$$

Binding the abstraction $A_1$ to an arbitrary symbol $f$, we can re-express $H_1$ as $H_1'$:

$$
H_1' = f (f (f (f (f (f C))))))
$$

and we rewrite the master term as:

$$
\text{master } (\lambda f \rightarrow \text{Tuple } A_1 H_1' \#)
$$

We can calculate the joint prior probability of this new abstraction $A_1$ and the re-expressed hypothesis $H_1'$, conditioning on the value of $m$ and the type of $A_1$. Defining $d = \text{npPriorHyperParam}_2$ we obtain:

$$
P(A_1, H_1' | m = 1 \text{ and } A_1 \text{ has type } \text{Word} \rightarrow \text{Word}) = \frac{\text{clPrior}(\text{master } (\lambda f \rightarrow \text{Tuple } A_1 H_1' \#), \Gamma)}{\text{clPrior}(\text{master } (\lambda f \rightarrow \text{Tuple } \# \#), \Gamma)} = \frac{h^4(h + 1)^2(h + 2)(h + 3)(h + 4)(h + 5)}{(4h)(4h + 1)(4h + 2)\ldots(4h + 9)} \times \frac{hd(d + 1)(d + 2)}{(h + d)(h + d + 1)(h + d + 2)(h + d + 3)} = 6.94 \times 10^{-8} \text{ when } h = d = 1
$$
Let us assume for convenience that \( \text{numAbstractionsDist} \) and \( \text{abstractionTypeDist} \) (Defn. 4.37) have been chosen such that:

\[
P(m = 0) = P(m = 1) \text{ and } A_1 \text{ has type } \text{Word} \to \text{Word}
\]

and furthermore that \( h = d = 1 \). Under these conditions we see that introducing the abstraction \( A_1 \) increases the prior probability of the master term as a whole by a factor of:

\[
\frac{\text{clPrior}(\text{master } \langle f \to \text{Tuple } A_1 H'_1 H_2 \rangle, \Gamma)}{\text{clPrior}(\text{master } \langle \text{Tuple } H'_1 \# \rangle, \Gamma)} = 6.94 \times 10^{-8} = 5.10
\]

This increase in probability indicates that the abstraction \( A_1 \) is likely to be beneficial for future learning and is therefore worth keeping.

Let us now see how the introduction of the abstraction \( A_1 \) affects the conditional prior probability distribution over hypotheses \( H_2 \) for the second sub-problem:

\[
P(H_2 | A_1, H'_1) = \frac{\text{clPrior}(\text{master } \langle f \to \text{Tuple } A_1 H'_1 H_2 \rangle, \Gamma)}{\text{clPrior}(\text{master } \langle f \to \text{Tuple } A_1 H'_1 \# \rangle, \Gamma)}
\]

The following table shows these conditional probability values for the first few terms \( H_2 \):

| \( H_2 \) | value(\( H_2 \)) | \( P(H_2 | A_1, H'_1) \) when \( h = 1 \) |
|---|---|---|
| C | C | \( \frac{h+1}{4h+10} \) |
| A C | A C | \( \frac{h+2}{4h+10} \times \frac{h+1}{4h+11} \) |
| B C | B C | \( \frac{h+1}{4h+10} \times \frac{h+1}{4h+11} \) |
| f C | A (B (B C)) | \( \frac{h+6}{4h+10} \times \frac{h+1}{4h+11} \) |
| A (A C) | A (A C) | \( \frac{h+1}{4h+10} \times \frac{h+2}{4h+11} \times \frac{h+1}{4h+12} \) |
| A (B C) | A (B C) | \( \frac{h+1}{4h+10} \times \frac{h+2}{4h+11} \times \frac{h+1}{4h+12} \) |
| A (f C) | A (A (B (B C))) | \( \frac{h+1}{4h+10} \times \frac{h+6}{4h+11} \times \frac{h+1}{4h+12} \) |

etc...

Note that \( \text{value}(H_2) \) is the value of \( H_2 \) after expanding out applications of \( f \).

Let \( B \) be the toplevel environment (Defn. 3.29) defined with respect to data type declarations for \text{Word} and \text{Tuple} and value declarations for \text{fixTuple} and \text{master}. Then, \( \text{value}(H_2) \) may be precisely specified as:

\[
\text{value}(H_2) = X_2
\]

where \( \text{Tuple } X_1 X_2 = \text{eval(master } \langle f \to \text{Tuple } A_1 H'_1 H_2 \rangle, B) \)

We see from the above table that introducing the abstraction \( A_1 \) produces a significant bias towards hypotheses for the second sub-problem that contain instances of the pattern ‘ABB’. Recall that this abstraction was originally derived from a repeating pattern ‘ABB’ in the hypothesis \( H_1 \) found for the first sub-problem. Hence we see how this cumulative learning mechanism biases the
learner to favour hypotheses for new problems that have elements in common with hypotheses already learned on previous problems.

Finally, it is worth discussing to what extent abstractions may be regarded as new background knowledge under this cumulative learning scheme. On the one hand, from the point of view of a monolithic learning problem it is clear than abstractions form part of a hypothesis; they do not affect the background knowledge $B$ as defined in the setting for learning, which remains constant throughout the cumulative learning process. On the other hand, from the point of view of an individual sub-problem such as the second sub-problem in the example above, we see that abstractions behave indistinguishably from elements of background knowledge both in how they determine the meaning of a hypothesis and in how they affect the form of the (conditional) prior. In this sense, i.e. with respect to a sub-problem, it is legitimate to regard abstractions as ‘new background knowledge’. 
Chapter 5

Refinement and RUFUS

In the last two chapters I introduced a setting for learning and suitable prior probability distributions to go with it. In this chapter I describe the design of a practical learning algorithm for use within this setting, which I have implemented as the inductive programming system RUFUS.

The design of RUFUS incorporates the techniques of refinement (Sect. 2.2.3) and proof-directed search (Sect. 2.2.4) adapted from first-order ILP. Therefore, RUFUS has aspects in common with ILP systems like Progol [Muggleton, 1995] and MC-TopLog [Muggleton et al., 2012b]. In particular, RUFUS’ heuristic search through a refinement graph is much like that of Progol’s, though RUFUS’ search space is not bounded by anything akin to a ‘most-specific clause’. Also, RUFUS uses a coverage-testing proof procedure to determine which parts of a hypothesis to refine, much as MC-TopLog uses its coverage-testing proof procedure to determine which clauses to refine.

RUFUS has some significant novel features not usually seen in first-order ILP. In particular, its lambda calculus representation language allows RUFUS to handle higher-order background knowledge (discussed in Sect. 2.3.1). RUFUS also incorporates a form of Levin search for dealing with potentially non-terminating hypothesis programs (discussed in Sect. 2.3.2).

5.1 Refinement

Having defined generality orders over lambda calculus terms back in Sect. 3.3, we shall see that it is relatively straightforward to adapt the technique of refinement to the lambda calculus setting. The canonical theory of refinement operators is not specific to first-order logic – it applies to arbitrary quasi-ordered sets [Nienhuys-Cheng and de Wolf, 1997, Chap. 17]. Therefore, we can immediately use much of this theory without modification.

On the other hand, the generality orderings of \(\lambda\)-entailment and \(\lambda\)-subsumption
(Sect. 3.3) do have somewhat different properties from their counterparts in first-order logic. One consequence of this is that so-called ideal refinement operators exist in the lambda calculus setting. In this section I shall construct an ideal refinement operator that is of practical use.

5.1.1 Refinement operators and their properties

The material in this subsection follows Nienhuys-Cheng and de Wolf [1997, Chap. 17].

Given some space of hypotheses, a refinement operator is a function that maps each hypothesis to a set of successor hypotheses. Associated with a refinement operator is a generality ordering over the same space of hypotheses.

Definition 5.1 (refinement operator, i-step refinements, refinements). A refinement operator $\rho$ over a quasi-ordered set $\langle L, \leq \rangle$ is a function that takes an element of $L$ and returns a subset of $L$. For $A \in L$ and non-negative integers $i$, we define the set $\rho^i(A)$ of i-step refinements as follows:

$$\rho^0(A) = \{A\}$$

$$\rho^i(A) = \{B \prec \rho^{i-1}(A); C \prec \rho(B) \bullet C\} \text{ where } i \geq 1$$

We also define the set $\rho^*(A)$ of refinements:

$$\rho^*(A) = \rho^0(A) \cup \rho^1(A) \cup \rho^2(A) \ldots$$

Definition 5.2 (upward/downward, sound, complete, locally finite, proper). Let $\rho$ be a refinement operator over a quasi-ordered set $\langle L, \leq \rangle$. When $\rho$ is an upward refinement operator, the following holds:

- $\rho$ is sound if for all $A \in L$ and $B \in \rho(A)$ it holds that $B \geq A$.
- $\rho$ is complete if for all $A, B \in L$ such that $B \geq A$, it holds that there is some $B' \in \rho^*(A)$ such that $B$ and $B'$ are equivalent under $\leq$.
- $\rho$ is locally finite if for all $A \in L$ it holds that the cardinality of $\rho(A)$ is finite.
- $\rho$ is proper is for all $A \in L$ and $B \in \rho(A)$ it holds that $B > A$. Note that properness implies soundness.
- $\rho$ is ideal if it is complete, locally finite, and proper.

We may also define the dual concept of a downward refinement operator by reversing the direction of the inequalities in the above properties.

\footnote{Note that my definition of $\rho^*(A)$ differs slightly from that of Nienhuys-Cheng and de Wolf in that I include $\rho^0(A)$ in $\rho^*(A)$; I find this to be somewhat more convenient.}
Soundness of a refinement operator would usually be regarded as necessary for the refinement operator to be of any practical use; soundness makes pruning possible in a heuristic search. On the other hand, the three properties of completeness, locally finiteness, and properness might be regarded as desirable but optional. Completeness is desirable because otherwise there will always be some hypotheses that the system is incapable of learning. Locally finiteness simplifies the design of search algorithms because one can rely on being able to exhaustively generate the set of one-step refinements of any given hypothesis. Finally, properness implies that the refinement graph is acyclic, which means that no search path will get stuck in a loop visiting the same hypothesis repeatedly.

5.1.2 The RUFUS refinement operator

I shall now specify a hypothesis space and associated refinement operator for the inductive inference system RUFUS.

As in the previous chapter, throughout this chapter I shall use the convention of Barendregt that lambda terms are taken to be syntactically equal if they are identical up to the renaming of bound variables.

RUFUS works within the setting for learning described in Sect. 3.4, and uses the nonparametric prior of Sect. 4.2. Its hypothesis space \( \mathcal{L} \) consists of terms in long normal form of some target type \( \tau_{\text{target}} \), and is parametrised by a set of symbols \( \Gamma \). \( \tau_{\text{target}} \) and \( \Gamma \) correspond to the arguments \( \tau \) and \( \Gamma \) of \( \text{npPriorProc} \) (Fig. 4.5).

**Definition 5.3** (\( \tau_{\text{target}}, \Gamma, \mathcal{L} \)). \( \tau_{\text{target}} \) is some type. \( \Gamma \) is some finite set of symbols not containing \text{void}. \( \mathcal{L} \) is the set of all long normal terms of type \( \tau \) whose free variables are either holes or elements of \( \Gamma \).

The RUFUS refinement operator \( \rho \) is an upward refinement operator defined over the set of terms \( \mathcal{L} \) ordered by \( \lambda \)-subsumption:

**Definition 5.4** (RUFUS refinement operator). \( \rho \) is a function that takes an element of \( \mathcal{L} \) and returns a subset of \( \mathcal{L} \):

\[
\rho(T) = \text{the set of all possible values of } \text{refine}(T)
\]

where the nondeterministic function \( \text{refine} \) is given in Fig. 5.1.

**Lemma 5.5.** \( \rho \) is a complete upward refinement operator over \( \mathcal{L} \) with respect to the \( \lambda \)-subsumption order.

**Proof.** Consider \( A, B \in \mathcal{L} \) where \( B \) subsumes \( A \). By Defn. 3.31, there exists a hole-substitution \( \sigma \) on \( A \) such that \( A\sigma = B \). By induction on the size of \( \sigma \):

1. \( \sigma \) has size 0. Then, \( B = A \) and hence \( B \in \rho^*(A) \).
nondeterministic function \textit{refine}(T):

\begin{itemize}
\item \textbf{input:} \(T\), a term in long normal form.
\item \textbf{output:} a term in long normal form.
\end{itemize}

1. Let \(p\) be any place in \(T\) at which a hole occurs.
2. Return \textit{refineAt}(\(p, T\)).

nondeterministic function \textit{refineAt}(p, T):

\begin{itemize}
\item \textbf{input:} \(p\), a place in \(T\) at which a hole occurs; \(T\), a term in long normal form.
\item \textbf{output:} a term in long normal form.
\end{itemize}

1. Let \(R\), a data type, be the the type of the hole at place \(p\) in \(T\), and let \(\Lambda\) be the set of lambda parameters in scope at place \(p\) in \(T\).
2. Let \(f\), a symbol, be any element of \(\Lambda \cup \Gamma\) whose return type is equal to \(R\).
   Let \(\alpha_1 \rightarrow \cdots \rightarrow \alpha_n \rightarrow R\) be the type of \(f\).
3. Let \(X\) be the term in long normal form written
   \[
   (f \ (\lambda x_{11} \cdots x_{1i_1} \rightarrow \#) \cdots (\lambda x_{n1} \cdots x_{nk_n} \rightarrow \#))
   \]
   where the \(x_{ij}\) are fresh symbols.
4. Return the term obtained from \(T\) by replacing the hole at \(p\) with \(X\).

Figure 5.1: The \textit{refine} and \textit{refineAt} algorithms.
2. $\sigma$ has size 1 or more. Then, $\sigma$ be written in the form $\{p/T\} \cup \tau$ where $T$ is non-trivial and $p/T \notin \tau$. $T$ is in long normal form and the type of $T$ has arity zero, therefore we can write $T$ as:

$$T = (f \langle x_{11} \ldots x_{1k_1} \rightarrow A_1 \rangle \ldots \langle x_{n1} \ldots x_{nk_n} \rightarrow A_n \rangle)$$

where $f$ is a symbol, the $x_{ij}$ are symbols and the $A_i$ are terms in long normal form. For $i \in \{1 \ldots n\}$, let $q_i$ be the place in $T$ at which the corresponding $A_i$ occurs. Let $T' = T[q_1/# \ldots q_n/#]$, from which it follows that $T = T'[q_1/A_1 \ldots q_n/A_n]$. We can now rewrite $B$ as follows:

$$B = A(p/T)\tau$$

$$= A(p/T')\{q_1/A_1 \ldots q_n/A_n\}\tau$$

$$= A(p/T)'\{(p ++ q_1)/A_1 \ldots (p ++ q_n)/A_n\}\tau$$  \hspace{1cm} \text{(by} \hspace{1cm} \text{3.13})

$$= A(p/T)'\{(p ++ q_1)/A_1 \ldots (p ++ q_n)/A_n\} \cup \tau$$

Observe that the last line above has the form $B = A'\sigma'$, where $A' = A(p/T')$. Noting that the size of $\sigma'$ is less than size of $\sigma$, it follows by the inductive hypothesis that $B \in \rho^*(A')$. Furthermore, by Defn. 5.4 it holds that $A' \in \rho(A)$, and thus by Defn. 5.1 that $B \in \rho^*(A)$.

\[ \square \]

**Theorem 5.6.** \( \rho \) is an ideal upward refinement operator over \( L \) with respect to the subsumption order.

**Proof.** \( \rho \) is complete by Lemma 5.5. \( \rho \) is locally finite by Defn. 5.4 and the fact that that \( \Gamma' \) is finite. \( \rho \) is proper by Defn. 5.4. \[ \square \]

Thus, RUFUS’ refinement operator is an ideal refinement operator. This may seem surprising, particularly as it is known that ideal refinement operators for first-order clausal hypothesis spaces do not exist ([Nienhuys-Cheng and de Wolf, 1997, Sect. 17.3]). On the other hand, lambda terms are more similar in structure to first-order atoms than to first-order clauses, and indeed, ideal refinement operators for sets of first order atoms do exist ([Nienhuys-Cheng and de Wolf, 1997, Sect. 17.2]). In fact, the ideal refinement operator for first-order atoms that Nienhuys-Cheng and de Wolf describe in that section has a lot in common with the RUFUS refinement operator for lambda terms.

The existence of ideal refinement operators can be seen as one advantage of a term-based language, such as lambda calculus, over a clausal one.

### 5.2 Proof-Directed Search

In the specification for RUFUS’ refinement operator in Fig. 5.1, there are two sources of non-determinism: firstly at line 1 of refine where a hole is chosen, and
secondly at line 2 of refineAt where a symbol f is chosen. It is clear that this refinement operator exhibits a significant amount of redundancy: when building up a total term via a sequence of refinements, it does not matter in what order the holes are chosen. As long as the same symbols are chosen at the same places, then the same total term will always be constructed (Fig. 5.2).

Since in the setting for learning one is interested in total terms as final hypotheses, it would make sense to eliminate this redundancy by constraining the order in which holes are chosen for refinement. For example, one could use a policy of always picking the left-most hole. Though this would not affect the completeness of the refinement operator with respect to total terms, such a policy ignores an important fact: some paths through the refinement graph offer more informative guidance to heuristic search than others.

For example, in Fig. 5.2 we see two paths through the refinement graph which both end up at the same total term. Imagine testing each partial hypothesis against some target datum during a search through the refinement graph. It is clear in this particular example that refining the left-hand hole first would offer more informative guidance than refining the right-hand hole first. Why is this? The reason is that, here, the left-hand choice allows the program that is being refined to produce useful output for some input: it will output 0 when the input is odd. On the other hand, if one takes the right-hand choice first, the resulting program produces no useful output regardless of input.

Though in the above example the left-most hole happens to give better guidance, in many cases it could be the right-most hole or some hole in a completely arbitrary position. As a result, we need a more informed approach to choosing which hole to refine, rather than a naive policy like ‘always pick the left-most hole’. The same issue exists in first-order ILP with multi-clause hypotheses: one needs an informed approach to choosing which clause to refine. As discussed in Sect. 2.2.4, the ILP system MC-TopLog solved this problem by using proof-directed search, i.e. by attempting a proof that some partial hypothesis covers
examples, and refining a clause whenever it becomes the limiting factor in the progress of the proof. This is a good strategy for achieving informative guidance, because information about coverage is returned by the proof procedure, and this is precisely the information that is necessary to guide and prune the search. In RUFUS, we shall use this technique to choose which hole to refine.

5.2.1 The evaluator

The proof procedure that RUFUS uses to test a hypothesis’ coverage is known as an evaluator. It is an implementation of the \textit{eval} function of Defn.3.23. RUFUS’ evaluator uses a strategy for performing these proofs known as \textit{call-by-need evaluation}, as described in [Abelson and Sussman, 1996, Sect. 4.2]. Call-by-need evaluation is a standard technique used in interpreters for functional programming languages, so I shall not discuss its details here.

I shall now give a high-level specification for the behaviour of RUFUS’ evaluator, in the form of the function \textit{evalProc} (Defn.5.9). \textit{evalProc} returns a kind of computational process called a \textit{reader process} (Defn.5.7). The concept of an \textit{evaluation process} (Defn.5.8) forms the heart of the specification, linking the behaviour of \textit{evalProc} to the definition of \textit{eval} (Defn.3.23). We can think of \textit{evalProc} as a computable approximation to the uncomputable function \textit{eval}.

The specification for \textit{evalProc} has a number of features that support proof-directed search. In the first condition of Defn.5.8, we see that when the output of \textit{eval} is equal to \( \perp \) due to the presence of a hole in the term being evaluated, then the evaluation process will return the place at which this hole occurs. Also, the fourth and fifth conditions of Defn.5.8 ensure that it safe to refine a term mid-way through evaluation.

\textbf{Definition 5.7 (reader process).} A \textit{reader process} is a computational process that proceeds in steps. At each step it reads in some input, then either a) continues on to the next step or b) halts and returns some final output. We define the following operations on reader processes:

- \textit{run}(R, n, x) represents the action of running reader process \( R \) for \( n \) steps or until it halts, repeatedly passing in the value \( x \) as input on each step.

- \textit{run}(R, \infty, x) represents the action of running reader process \( R \) forever or until it halts, repeatedly passing in the value \( x \) as input on each step.

\textbf{Definition 5.8 (evaluation process).} An \textit{evaluation process} is a four-valued tuple \((R, p, T, E)\) where \( R \) is a reader process, \( T \) is a term, \( E \) is a toplevel environment, and \( p \) is a value-place in \textit{eval}(\( T, E \)), such that the following conditions hold. Let \( x \) be the subvalue at place \( p \) in \textit{eval}(\( T, E \)), and let \( \tau \) be the type of \( x \). If \( \tau \) is a data type then:
1. If \( x = \bot \) then one of the following two conditions holds:
   - either \( \text{run}(R, \infty, T) \) returns \( \langle \text{"hole"}, h, R' \rangle \) where \( h \) is a place in \( T \) at which a hole occurs and \( R' \) is a reader process such that \( \langle R', p, T, E \rangle \) is an evaluation process;
   - or \( \text{run}(R, \infty, T) \) either returns “error” or does not halt, and for all \( X \in T \) it holds that the subvalue at place \( p \) in \( \text{eval}(X, E) \) is equal to \( \bot \).

2. If \( x \) is of the form \( (C \ a_1 \ldots a_k) \) then \( \text{run}(R, \infty, T) \) returns
   \[ \langle \text{"data"}, C, [R_1 \ldots R_k]\rangle \]
   where each \( R_i \) is a reader process such that \( \langle R_i, p + + [i], T, E \rangle \) is an evaluation process.

If \( \tau \) is a function type \( \alpha \rightarrow \beta \) then:

3. \( \text{run}(R, \infty, T) \) returns \( \langle \text{"function"}, R' \rangle \) where \( R' \) is a function that takes a value of type \( \alpha \) and returns a reader process, such that for all values \( a \) of type \( \alpha \) it holds that \( \langle R'(a), p + + [a], T, E \rangle \) is an evaluation process.

Finally, regardless of \( \tau \):

4. For any positive integer \( n \) such that \( \text{run}(R, n, T) \) does not halt, it holds that \( \langle R', p, T, E \rangle \) is an evaluation process where \( R' \) is the state of reader process \( R \) after executing \( \text{run}(R, n, T) \).

5. For any term \( T' \) that properly subsumes \( T \) it holds that \( \langle R, p, T', E \rangle \) is an evaluation process. Furthermore, if \( \text{run}(R, \infty, T') \) halts in a finite number of steps then so does \( \text{run}(R, \infty, T) \).

**Definition 5.9 (RUFUS evaluator).** The function \( \text{evalProc} \) takes a term \( T \) and a toplevel environment \( E \), and returns a reader process \( R \) such that \( \langle R, [\], T, E \rangle \) is an evaluation process.

In the RUFUS system, I have implemented an evaluator that meets the specification of Defn. 5.9. Internally, it uses the call-by-need evaluation strategy mentioned above.

### 5.2.2 The \textit{runThreads} algorithm

In the second and third conditions of Defn. 5.8 in the last subsection, we saw that an evaluation process may halt and return multiple new evaluation processes. We can think of these processes as ‘threads of evaluation’, each corresponding to different arguments of a constructor application or to different input-output entries of a function. In order to manage these multiple threads of evaluation during coverage testing, RUFUS shall make use of the following object:
Definition 5.10 (evaluation thread structure). An evaluation thread structure is a five-valued tuple:

\[ [(R_1 \ldots R_n), [p_1 \ldots p_n], T, E, [x_1 \ldots x_n]] \]

where:

- each \( (R_i, p_i, T, E) \) is an evaluation process;
- each \( x_i \) is a value of finite size of the same type as the subvalue at \( p_i \) in \( eval(T, E) \).

In the above definition, each \( x_i \) is intended to be a subvalue of the target datum, against which a corresponding subvalue of \( eval(T, E) \) is being compared for coverage. For each \( i \in \{1 \ldots n\} \) the \( R_i, p_i, \) and \( x_i \) together constitute the \( i \)th thread:

Definition 5.11 (thread). A thread is a three-element tuple \( \langle R, p, x \rangle \) where \( R \) is a reader process, \( p \) is a value-place and \( x \) is a value of finite size.

RUFUS’ coverage testing algorithm, \( runThreads \), is given in Fig. 5.3. It takes a list of threads along with the term being evaluated, and its action is to run each of the threads in turn, performing evaluation and comparing each result against the corresponding subvalue of the target datum. Each time any thread halts and spawns child threads, these child threads are then run too.

In order to handle potential non-termination of an evaluation process, \( runThreads \) ‘times out’ after performing evaluation for \( evalLimit \) steps (Defn. 5.12). However, when such a time-out occurs, the return value of \( runThreads \) includes the state of all currently running threads, allowing evaluation to be resumed later. Indeed, as we shall see, RUFUS never gives up on any hypothesis no matter how many steps are required in order to evaluate it. This enables RUFUS to achieve completeness despite non-termination, in the manner of Levin search.

Definition 5.12 (evalLimit). The \( runThreads \) algorithm is parametrised by a positive integer \( evalLimit \). This sets a limit on the number of steps for which the evaluator may run at a single invocation.

\( runThreads \) may return one of four possible results. “covers” indicates that all threads ran to completion and that each result of evaluation covered the corresponding subvalue of the target datum. “contradicts” indicates that at least one result of evaluation contradicted its corresponding subvalue of the target datum. “timeout” indicates that a thread ran for \( evalLimit \) steps without halting. “refine” indicates that a thread is currently unable to progress due to the presence of a hole in the term being evaluated, i.e. it constitutes a request
for refinement. A formal specification for all of this behaviour is given as Lemma 5.16 below.

The state variables of runThreads have the following meanings. $y$ is the result of evaluation of the term $T$; it is gradually built up as evaluation progresses. $stepCount$ is the number of steps for which the currently running thread has proceeded so far; each time runThreads finishes executing a thread and moves onto a new one, $stepCount$ is reset to zero.

**Definition 5.13 (covers, contradicts).** Let $T$ be a term, let $E$ be an environment, let $p$ be a value-place in $eval(T,E)$, and let $x$ be a value of the same type as the subvalue at $p$ in $eval(T,E)$. The predicates covers and contradicts are defined as follows:

\[
\text{covers}(p,T,E,x) \text{ if } (\text{the subvalue at } p \text{ in } eval(T,E)) \sqsubseteq x \tag{5.14}
\]

\[
\text{contradicts}(p,T,E,x) \text{ if } \nexists H \in T \text{ such that covers}(p,H,E,x) \tag{5.15}
\]

**Lemma 5.16.** Given an evaluation thread structure:

\[
\langle \llbracket R_1 \ldots R_n \rrbracket, \llbracket p_1 \ldots p_n \rrbracket, T, E, [x_1 \ldots x_n] \rrbracket
\]

let Result be the output of:

\[
\text{runThreads}((\llbracket \langle R_1, p_1, x_1 \rangle \ldots \langle R_n, p_n, x_n \rangle \rrbracket, T)\]

executed from some arbitrary initial state. The following facts then hold:

1. If Result = “covers” then covers($p_i, T, E, x_i$) for all $i \in \{1 \ldots n\}$.

2. If Result = “contradicts” then contradicts($p_i, T, E, x_i$) for some $i \in \{1 \ldots n\}$.

3. If Result is of either of following two forms:

\[
\langle \text{“timeout”}, \langle \llbracket R'_1, p'_1, x'_1 \rrbracket \ldots \langle R'_m, p'_m, x'_m \rrbracket \rangle \rangle
\]

\[
\langle \text{“refine”}, \langle \llbracket R'_1, p'_1, x'_1 \rrbracket \ldots \langle R'_m, p'_m, x'_m \rrbracket, h \rangle \rangle
\]

then:

\[
\langle \llbracket R'_1 \ldots R'_m \rrbracket, \llbracket p'_1 \ldots p'_m \rrbracket, T, E, [x'_1 \ldots x'_m] \rrbracket
\]

is an evaluation thread structure, and the following holds for all $H \in T$:

If covers($p'_i, H, E, x'_i$) for all $i \in \{1 \ldots m\}$ then

covers($p_i, H, E, x_i$) for all $i \in \{1 \ldots n\}$.

Furthermore, in the case of the “refine” form of Result it holds that $h$ is a place in $T$ at which a hole occurs.

**Proof.** By induction on $N = \sum_{i=1}^{n} \text{size}(x_i)$:
stateful function \textit{runThreads}(ts, T):

\begin{itemize}
  \item[1.] If \( ts = [] \) then return “covers”.
  \item[2.] Else if \( ts \) is of the form \( \langle R, p, x \rangle : ts' \) then:
    \begin{itemize}
      \item[3.] If \( x = \bot \) then return \textit{runThreads}(ts', T).
      \item[4.] Else if \textit{run}(R, evalLimit, T) has not halted then:
        \begin{itemize}
          \item[5.] \( \text{stepCount} \leftarrow \text{stepCount} + \text{evalLimit} \)
          \item[6.] Return \( \langle \text{“timeout”}, \langle R', p, x \rangle : ts' \rangle \) where \( R' \) is the remainder of the reader process \( R \) after running \textit{run}(R, evalLimit, T).
        \end{itemize}
    \end{itemize}
  \item[7.] Else if \textit{run}(R, evalLimit, T) returns “error” then return “contradicts”.
  \item[8.] Else if \textit{run}(R, evalLimit, T) returns \( \langle \text{“hole”}, h, R' \rangle \) then:
    \begin{itemize}
      \item[9.] Let \( n \) be the number of steps taken for \textit{run}(R, evalLimit, T) to halt.
      \item[10.] \( \text{stepCount} \leftarrow \text{stepCount} + n. \)
      \item[11.] Return \( \langle \text{“refine”}, \langle R', p, x \rangle : ts', h \rangle \).
    \end{itemize}
\end{itemize}
continued on next page...
...continued from previous page.

12. Else if run(R, evalLimit, T) returns \(\langle \text{"data"}, C, [R_1 \cdots R_k]\rangle\) and \(x\) is of the form \(\langle C' a_1 \cdots a_k \rangle\) then:

13. \(y \leftarrow \text{the value obtained by replacing the subvalue at } p \text{ in } y \text{ with } \langle C \perp \cdots \perp \rangle.\)

14. If \(C = C'\) then:

\[
\text{stepCount} \leftarrow 0
\]

15. For \(i \in \{1 \cdots k\}\), let \(u_i\), a thread, equal \(\langle R_i, p + + [i], a_i \rangle.\)

16. Return \(\text{runThreads}(\langle u_1 \cdots u_k \rangle : ts', T).\)

17. Else return “contradicts”.

18. Else if run(R, evalLimit, T) returns \(\langle \text{"function"}, R' \rangle\) and \(x\) is a function then:

19. \(\text{stepCount} \leftarrow 0\)

20. Let \(\langle a_1, b_1 \rangle \cdots \langle a_k, b_k \rangle\) be the non-trivial entries of \(x.\)

21. For \(i \in \{1 \cdots k\}\), let \(u_i\), a thread, equal \(\langle R'(a_i), p + + [a_i], b_i \rangle.\)

22. Return \(\text{runThreads}(\langle u_1 \cdots u_k \rangle : ts', T).\)

Figure 5.3: The \textit{runThreads} algorithm.
1. \( N = 0 \), and hence \( n = 0 \). \textit{runThreads} returns “covers” at line 1, from which the proof follows.

2. \( N \geq 1 \), and hence \( n \geq 1 \). \textit{runThreads} proceeds to line 3. There are seven cases to consider:

(a) \textit{runThreads} returns at line 3. By Defn. 3.17 it holds that \textit{covers}(p_1, T, E, x_1). The proof then follows by the inductive hypothesis.

(b) \textit{runThreads} returns at line 6. The proof follows by the fourth condition of Defn. 5.8.

(c) \textit{runThreads} returns at line 7. By Defn. 5.8 it holds for all \( H \in T \) that the subvalue at \( p_1 \) in \( \text{eval}(H, E) \) is equal to \( \bot \). Since \textit{runThreads} did not return at line 3 we also know that \( x_1 \neq \bot \). Hence it holds that \textit{contradicts}(p_1, T, E, x_1) from which the proof follows.

(d) \textit{runThreads} returns at line 11. The proof follows by Defn. 5.8.

(e) \textit{runThreads} returns at line 17. The proof follows by Defn. 5.8 and the inductive hypothesis.

(f) \textit{runThreads} returns at line 18. By Defn. 5.8 and Defn. 3.17 it holds that \textit{contradicts}(p_1, T, E, x_1), from which the proof follows.

(g) \textit{runThreads} returns at line 23. The proof follows by Defn. 5.8 and the inductive hypothesis.

\[ \square \]

5.2.3 The RUFUS search tree

Combining coverage testing together with refinement, I shall now specify the proof-directed search space of RUFUS. This search space takes the form of a tree, whose internal nodes are as follows:

\textbf{Definition 5.17} (node). A \textit{node} is a tuple of the form \((ts, T, y, stepCount)\) where \( ts \) is a list of threads, \( T \) is a term, \( y \) is a value of finite size of the same type as \( T \), and \( stepCount \) is a non-negative integer.

The RUFUS search tree is defined in Fig. 5.4. The root node of the tree, given by \textit{startNode}, consists of a thread that evaluates an initial ‘empty’ hypothesis and tests it for coverage against the target datum. The action of \textit{expandNode} is to run the threads contained in a node. If a timeout occurs, then the currently running threads are saved and returned as a child node. If evaluation stops due to the presence of a hole in the hypothesis, then all possible one-step refinements
are made at that hole, with one child node being returned for each different possible refinement.

Each leaf of the RUFUS search tree is known as a ‘goal’. Below I prove that the RUFUS search tree is sound, in that every goal in the tree consists of a hypothesis that covers the target datum (Theorem 5.20). I also prove that the RUFUS search tree is complete, in that every possible hypothesis that covers the target datum occurs as a goal in the tree (Theorem 5.22).

Note that pruning is already incorporated into the definition of the RUFUS search tree. Pruning occurs whenever runThreads returns “contradicts” (lines 7 and 18 of Fig. 5.3). In such cases, as soon as a contradiction is found between the current partial hypothesis and the target datum, then the search path is terminated and expandNode returns an empty set of child nodes. We see that this occurs without the need for all threads to run to completion on that search path.

**Definition 5.18 (goal path).** A goal path is a pair \( \langle [N_1 \ldots N_d], H \rangle \) where \([N_1 \ldots N_d]\) is a list of nodes with \(d \geq 1\), \(H\) is a term, and the following conditions hold:

- for each \(i \in \{1 \ldots d - 1\}\) it holds that expandNode\((N_i)\) is of the form \("nodes", Ns\) and \(N_{i+1} \in Ns\);
- \(\) expandNode\((N_d)\) = \("goal", H\).

We say that \(\langle [N_1 \ldots N_d], H \rangle\) is a goal path from \(N_1\) to \(H\).

**Lemma 5.19.** Let the following be an evaluation thread structure:

\(\langle [R_1 \ldots R_n], [p_1 \ldots p_n], T, E, [x_1 \ldots x_n] \rangle\)

such that \(T \in L\). Let \(\langle [N_1 \ldots N_d], H \rangle\) be a goal path where:

\(N_1 = \langle ([R_1, p_1, x_1] \ldots [R_n, p_n, x_n]), T, y, stepCount \rangle\)

for some \(y\) and \(stepCount\). It holds that \(H \in L\) and covers\((p_i, H, E, x_i)\) for all \(i \in \{1 \ldots n\}\).

**Proof.** Let \(ts = ([R_1, p_1, x_1] \ldots [R_n, p_n, x_n])\). By induction on \(d\):

1. \(d = 1\). By Defn. 5.18, expandNode\((N_1)\) = \("goal", H\). It follows by the definition of expandNode (Fig. 5.4) that \(H = T\) and that the output of runThreads\((ts, T)\) is equal to “covers”. The proof follows by Lemma 5.16.

2. \(d > 1\). By Defn. 5.18, expandNode\((N_1)\) is of the form \("nodes", Ns\). Let \(R\) be the output of runThreads\((ts, T)\). By the definition of expandNode, there are then two cases to consider:
function *startNode*(*B*, *D*):

**input:** *B*, a toplevel environment;

D, a value of finite size and of type τtarget.

**output:** a node.

1. Let *T*, a long normal term of type τtarget, be equal to (∧ *x*1 ... *x*n → #)
   where the *x*i are fresh symbols and *n* is the arity of τtarget.
2. Return ⟨[[evalProc(*T*, *B*), []], *D*], *T*, ⊥, 0⟩.

function *expandNode*(*N*):

**input:** *N*, a node.

**output:** either ⟨“goal”, *H*⟩ where *H* is a term

or ⟨“nodes”, *Ns*⟩ where *Ns* is a set of nodes.

1. Let ⟨*ts*, *T*, *y*, stepCount⟩ = *N*.
2. Execute runThreads(*ts*, *T*) with initial state *y*, stepCount. Let *R* be the
   output and let *y’*, stepCount’ be the final state.
3. If *R* = “covers” then return ⟨“goal”, *T*⟩.
4. Else if *R* = “contradicts” then return ⟨“nodes”, {}⟩.
5. Else if *R* = ⟨“timeout”, *ts’⟩ then return
   ⟨“nodes”, ⟨*ts’*, *T*, *y’*, stepCount’⟩⟩.
6. Else if *R* = ⟨“refine”, *ts’*, *h*⟩ then return
   ⟨“nodes”, {*T’ <$\bowtie$> possible values of refineAt(*h*, *T*)
   • ⟨*ts’*, *T’*, *y’*, stepCount’⟩}⟩.

Figure 5.4: Specification for RUFUS’ search tree. *startNode* constructs the root node.

*expandNode* constructs the children of a given node, returning either a ‘goal’ or a new
set of nodes.
(a) $R$ is of the form $\langle \text{“timeout”}, ts' \rangle$ and $N_2 = \langle ts', T, y', \text{stepCount}' \rangle$ for some $y'$ and $\text{stepCount}'$. The proof follows by Lemma 5.16 and the inductive hypothesis.

(b) $R$ is of the form $\langle \text{“refine”}, ts', h \rangle$ and $N_2 = \langle ts', T', y', \text{stepCount}' \rangle$ for some possible value $T'$ of $\text{refineAt}(h, T)$, some $y'$, and some $\text{stepCount}'$. The proof follows by Lemma 5.16 and the inductive hypothesis.

\begin{theorem}[soundness of RUFUS] Let $B$ be a toplevel environment, let $D$ be a value of finite size and of type $\tau_{\text{target}}$, and let $H$ be a term. If there exists a goal path from $\text{startNode}(B, D)$ to $H$, then it follows that $H \in \mathcal{L}$ and $\text{eval}(H, B) \sqsupseteq D$.

\end{theorem}

Proof. Let $\langle \langle R, [], x \rangle, T, y, \text{stepCount} \rangle = \text{startNode}(B, D)$. By the definition of $\text{startNode}$ (Fig. 5.4) it holds that $R = \text{evalProc}(T, B)$ and $x = D$ and $T \in \mathcal{L}$. By Defn. 5.9 and Defn. 5.10 it holds that $\langle \langle R, [], T, B, [x] \rangle \rangle$ is an evaluation thread structure. The proof then follows by Lemma 5.19.

\begin{lemma} \label{lem:evaluation-thread} Let the following be an evaluation thread structure:

$$\langle \langle R_1 \ldots R_n, [p_1 \ldots p_n], T, E, [x_1 \ldots x_n] \rangle \rangle$$

where $T \in \mathcal{L}$. Let $H$ be an element of $\mathcal{L}$ that subsumes $T$ and such that covers$(p_i, H, E, x_i)$ for all $i \in \{1 \ldots n\}$. Let $N_1$ be the following node:

$$N_1 = \langle \langle R_1, p_1, x_1 \rangle \ldots \langle R_n, p_n, x_n \rangle \rangle, T, y, \text{stepCount} \rangle$$

for some $y$ and $\text{stepCount}$. It follows that there exist nodes $N_2 \ldots N_d$ with $d \geq 1$ and there exists a term $H'$, such that $\langle \langle N_1 \ldots N_d, H' \rangle \rangle$ is a goal path and $H$ subsumes $H'$.

\end{lemma}

Proof. Let $ts = \langle \langle R_1, p_1, x_1 \rangle \ldots \langle R_n, p_n, x_n \rangle \rangle$. Let $R$ be the output of $\text{runThreads}(ts, T)$. The proof proceeds by a triple mathematical induction on three variables: the number of holes in $T$, the set of places $\{p_1 \ldots p_n\}$, and the number of steps required for $\text{run}(R_1, \infty, T)$ to halt. Note that by Defn. 5.8, $\text{run}(R_1, \infty, T)$ is guaranteed to halt in a finite number of steps provided that $x_1 \neq \bot$. There are three cases to consider:

1. $R = \text{“covers”}$. By the definition of $\text{expandNode}$ (Fig. 5.4) it follows that $\langle \langle N_1, T \rangle \rangle$ is a goal path. We are also given that $H$ subsumes $T$, from which the proof follows.

2. $R = \text{“contradicts”}$. By Lemma 5.16 and Eqn. 5.15 it holds for some $i \in \{1 \ldots n\}$ that $\exists X \in T$ such that $\text{covers}(p_i, X, E, x_i)$. However, this is contradicted by the existence of $H$, therefore this case can never occur.
3. $R$ is of the form \(\langle \text{“timeout”}, ts' \rangle\). The proof follows by Lemma 5.16 and the inductive hypothesis.

4. $R$ is of the form \(\langle \text{“refine”}, ts', h \rangle\). By the definition of refineAt (Fig. 5.1), there exists a possible value $T'$ of refineAt$(h, T)$ such that $T' \in \mathcal{L}$ and $H$ subsumes $T'$. The proof follows by Lemma 5.16 and the inductive hypothesis.

\[\square\]

**Theorem 5.22** (completeness of RUFUS). Let $B$ be a toplevel environment, let $D$ be a value of finite size and of type $\tau_{\text{target}}$, and let $H$ be an element of $\mathcal{L}$ such that eval$(H, B) \supseteq D$. It follows that there exists a goal path from startNode$(B, D)$ to some term $H'$ such that $H$ subsumes $H'$.

*Proof.* The proof follows in identical manner to that of Theorem 5.20 except that the last step follows by Lemma 5.21. \[\square\]

### 5.2.4 Guiding the search

So far I have defined RUFUS’ search tree and proved that it is sound and complete. All that remains now is the question of how to traverse the search tree appropriately in order to perform inference using either Bayes MAP or Bayes prediction. For Bayes MAP, one wishes to perform optimisation, i.e. to search for the goal node in the tree that has the highest prior probability (and hence the highest posterior, since all the goals have the same likelihood of 1). On the other hand, for Bayes prediction one would likely wish to perform Monte Carlo sampling from the set of goal nodes according to their prior probabilities. It is an intended feature of the design of RUFUS that both inference strategies are feasible using the same search tree.

In this rest of this chapter I shall focus on one strategy for Bayes MAP inference. The chosen approach shall be a *best-first search*, using a heuristic estimate of the posterior probability at each intermediate node in the tree in order to guide the search. This heuristic estimate takes into account any coverage of the target datum by the partial hypothesis at an intermediate node. The resulting search strategy is not unlike that found in FOIL or Progol (see Sect. 2.2.3).

My heuristic estimate of the posterior probability, or *score*, of a node is given in Defn. 5.28. This score consists of three components. First, the prior probability of the partial hypothesis $T$ is calculated. This is then multiplied by an estimate $\text{likelihoodEst}$ of the (unnormalised) likelihood of $T$. Note that $\text{likelihoodEst}$ is a function of $y$, the part of the target datum so far correctly predicted by $T$. Finally, this is divided by $\text{stepCount}$, the number of steps for which the node’s currently active thread has been running so far without halting.
Note that a small constant offset value \textit{stepCountOffset} (Defn. 5.27) is added to the step-count. The purpose of \textit{stepCountOffset} is mainly to avoid a division by zero when \textit{stepCount} is zero; overall it serves to dampen the influence of very small values of \textit{stepCount} on the score.

By dividing the posterior estimate by the step-count, we approximate the \textit{bias-optimality} property of Levin search that was discussed in Sect. 2.3.2. Recall that to achieve bias-optimality, the search process must allocate computation time to each hypothesis in proportion to its probability. Consider two hypotheses \(T_1,T_2\) with posterior probability estimates \(p_1,p_2\) respectively, and which have so far been tested for step-counts of \(n_1,n_2\) respectively. The two hypotheses will have equal scores when the following condition holds (ignoring the effect of \textit{stepCountOffset}):

\[
\frac{p_1}{n_1} = \frac{p_2}{n_2} \implies \frac{n_1}{n_2} = \frac{p_1}{p_2}
\]

Hence, we see that the ratio of the step-counts is equal to the ratio of the probabilities, precisely the condition for bias-optimality.

The likelihood estimate that I shall use is defined in terms of a probability distribution \textit{valProb} over values of the target type (Defn. 5.25). \textit{valProb} is in turn defined in terms of an arbitrary assignment \textit{conWeight} of probabilities to data type constructors (Defn. 5.24). Suppose that \(D\) is the target datum, and that \(y\) is the portion of \(D\) that is covered by a partial hypothesis \(T\). I shall then use the following as an estimate of the likelihood of \(D\) given \(T\):

\[
P(D|T) \approx \frac{\text{valProb}(D)}{\text{valProb}(y)}
\]

which is equal to the probability under \textit{valProb} of the remaining portion of the target datum that is not yet covered by \(T\). Notice that the numerator in Eqn. 5.23 is a constant; we may therefore omit it for convenience, since multiplying all scores by a constant scaling factor will have no effect on the outcome of RUFUS’ search. This yields the \textit{likelihoodEst} function given in Defn. 5.26. Though \textit{likelihoodEst} is a crude estimate of likelihood, it can still provide useful guidance to RUFUS’ search. In fact, this kind of likelihood estimate is very similar to the ‘number of examples covered’ heuristic typically used in first-order ILP (Eqn. 2.1). Finally, note that one could avoid the need for the arbitrary weights of Defn. 5.24 by extending \textit{valProb} to a non-parametric form. This could be done in a similar way to how the base prior distribution was extended to non-parametric form back in Chap. 4.

\textbf{Definition 5.24 (\textit{conWeight})}. The function \textit{conWeight} takes a constructor and returns a probability value. It is subject to the following constraint for each data
type declaration with constructors $C_1 \ldots C_n$:

$$\sum_{i=1}^{n} \text{conWeight}(C_i) = 1$$

**Definition 5.25** (*valProb*). The function *valProb* takes a value and returns a probability. For a value of a data type:

$$\text{valProb}(\bot) = 1$$

$$\text{valProb}((C \ a_1 \ldots a_n)) = \text{conWeight}(C) \times \prod_{i=1}^{n} \text{valProb}(a_i)$$

For a value of a function type:

$$\text{valProb}(f) = \prod_{i=1}^{n} \text{valProb}(b_i)$$

where $\langle a_1, b_1 \rangle \ldots \langle a_n, b_n \rangle$ are the non-trivial entries of $f$.

**Definition 5.26** (*likelihoodEst*). The function *likelihoodEst* takes a value and returns a real number:

$$\text{likelihoodEst}(y) = \frac{1}{\text{valProb}(y)}$$

**Definition 5.27** (*stepCountOffset*). *stepCountOffset* is some positive real number.

**Definition 5.28** (*score*). The *score* of a node is equal to:

$$\text{score}((ts,T,y,\text{stepCount})) = \frac{\text{npPrior}(T,\Gamma) \times \text{likelihoodEst}(y)}{\text{stepCount} + \text{stepCountOffset}}$$

Note that an alternative prior function may be substituted for $\text{npPrior}(T,\Gamma)$ in the above expression if appropriate.

In Fig. 5.5 I give a best-first search algorithm for traversing the RUFUS search tree. Using this search strategy, we now have a self-contained inference algorithm *runRufus* (Defn. 5.29). Note that due to the heuristic nature of the search, the goal returned by *runRufus* is not guaranteed to be Bayes MAP optimal. It may therefore be desirable to extend the search so as to obtain multiple goals, and to test them against validation data in order to avoid overfitting.

**Definition 5.29** (*runRufus*). The function *runRufus* takes a toplevel environment (background knowledge) and a value of finite size (target datum). It returns either “nogoal” or $\langle$“goal”, $H$ $\rangle$ where $H$ is a term:

$$\text{runRufus}(B,D) = \text{bestFirstSearch}([\text{startNode}(B,D)])$$
function bestFirstSearch(fringe):
    input: fringe, a set of nodes.
    output: either “nogoal”
            or ⟨“goal”, T⟩ where T is a term.
1. If fringe = {} then return “nogoal”
2. Else:
3. Let N be the node in fringe with the highest score, and let fringe’ be equal to fringe with the element N removed.
4. If expandNode(N) is of the form (“goal”, T) then return (“goal”, T).
5. Else if expandNode(N) is of the form (“nodes”, Ns) then return bestFirstSearch(fringe’ ∪ Ns).

Figure 5.5: The bestFirstSearch algorithm. “nogoal” is returned if the entire search tree has been traversed and no goal has been found. Otherwise, the first goal to be encountered is returned. Note that if two or more nodes are tied for the highest score at line 3, then one may be chosen arbitrarily.

5.2.5 Worked example

Let us look at a concrete example of RUFUS in action on a simple inductive inference problem. The aim shall be to learn a predicate to test if a natural number is even, from input-output examples.

Following the setting for learning (Sect.3.4), we start by specifying some background knowledge and a target datum. Let the background knowledge B be the toplevel environment (Defn.3.29) defined with respect to the empty set of value declarations and the following set of data type declarations:

\[ \text{data Nat} = \text{Zero} | \text{Succ Nat} \]
\[ \text{data Bool} = \text{True} | \text{False} \]

Let the target type be \( \text{Nat} \rightarrow \text{Bool} \). Let the target datum \( D \) be the least value under \( \sqsubseteq \) in the domain of the target type such that the following constraints hold:

\[ D(\text{Zero}) = \text{True} \]
\[ D(\text{Succ Zero}) = \text{False} \]
\[ D(\text{Succ (Succ Zero)}) = \text{True} \]
\[ D(\text{Succ (Succ (Succ Zero)))} = \text{False} \]
\[ D(\text{Succ (Succ (Succ (Succ Zero)))}) = \text{True} \]
For the prior over hypothesis terms, we shall use the following:

\[ P(H) = npPrior(H, \Gamma) \]

where \( \Gamma = \{ \text{True}:\text{Bool}, \text{False}:\text{Bool}, \text{caseNat}:(\text{Nat} \to \text{Bool} \to (\text{Nat} \to \text{Bool}) \to \text{Bool}), \text{fix}(((\text{Nat} \to \text{Bool}) \to \text{Nat} \to \text{Bool}) \to \text{Nat} \to \text{Bool}) \} \)

We also specify some reasonable values for the following parameters of RUFUS that have so far been left unspecified:

\[ npPriorHyperParam_1 = 8 \]
\[ npPriorHyperParam_2 = 16 \]
\[ evalLimit = 50 \]
\[ conWeight(x) = \frac{1}{2} \text{ for } x \in \{ \text{Zero}, \text{Succ}, \text{True}, \text{False} \} \]
\[ stepCountOffset = 1 \]

Under these conditions, \( \text{runRufus}(B, D) \) returns \( \langle \text{"goal"}, H \rangle \) where \( H \) is equal to the following term:

\[
(\lambda a_1 \rightarrow \text{fix}(\lambda a_2 a_3 \rightarrow \text{caseNat} a_3 \\
\text{True} \\
(\lambda a_4 \rightarrow \text{caseNat} a_4 \\
\text{False} \\
(\lambda a_5 \rightarrow a_2 a_5))) a_1)
\]

Note that this hypothesis is a correct general implementation of the ‘even’ predicate. This was tested with a prototype implementation of RUFUS.
Chapter 6

Anti-unification Search

In this chapter and the next I describe the design of a practical abstraction invention method for use within the lambda calculus setting for learning, which I have implemented as the KANDINSKY system. The method consists of two stages. In the first stage, a search for syntactic commonality is performed over a given set of terms. In the second stage, common syntactic structure is abstracted out to form a new piece of background knowledge, by means of inverse deduction. In this chapter I shall describe the first stage, i.e. the search for commonality, and in Chap. 7 I shall describe the second stage.

In order to search for syntactic commonality, I have designed an algorithm called auSearch (anti-unification search). This algorithm is not in fact specific to lambda calculus, and can be used to find syntactic commonality among arbitrary tree-structured terms. Given a set of such terms, auSearch searches the space of all non-trivial anti-instances of two or more of them. An anti-instance is a kind of syntactic generalisation which represents what a set of terms have in common. The total number of possible anti-instances can be exponentially large in the number of terms, therefore auSearch uses a customisable heuristic score function in order to perform an efficient guided beam search through this space. Note that this search process bears some similarity to that used in the propositional inverse resolution system Duce [Muggleton, 1987], although Duce is concerned with propositional clauses rather than tree-structured terms.

The terms fed into auSearch are intended to be subterms of some larger master term. These subterms may overlap one another, and auSearch must take this into account. Indeed, auSearch is designed to discover only non-overlapping syntactic commonality. It does this by making use of a sub-algorithm called mnos (maximum non-overlapping subset), which allows it to detect overlapping syntactic structure early, and avoid it, throughout the search process. This is important because KANDINSKY’s inverse deduction algorithm, which I shall describe in the next chapter, is only able to create an abstraction from non-
overlapping commonality.

6.1 Tree-Structured Terms

As just mentioned, auSearch works with generic tree-structured syntax; it is not specific to lambda calculus. This is advantageous because it simplifies the design of auSearch, freeing it from dealing with the details of typed lambda calculus. In order to represent generic tree-structured syntax, I shall use untyped first-order terms. These first-order terms have no semantics, and are simply a convenient, plain syntactic representation. In the next chapter I will show how typed lambda calculus terms can be translated to these first-order terms.

Thoughout the whole of this chapter, ‘term’ shall be taken to mean ‘first-order term’, not ‘lambda calculus term’ as in the rest of this thesis. I now define (first-order) terms, substitutions, instances, etc. following Nienhuys-Cheng and de Wolf [1997]. The concept of a place comes from Plotkin [1969]. Note that in what follows I state some propositions without proof when their validity is assumed to be obvious to the reader.

Let us assume the existence of a countably-infinite alphabet of variables and another of function symbols. Each function symbol has a non-negative integer, its arity, associated with it.

Definition 6.1 (term, ground, trivial, linear). A term is either a variable, or a syntactic object of the form \( f(t_1, \ldots, t_n) \) where \( f \) is a function symbol of arity \( n \) and \( t_1 \ldots t_n \) are terms. A ground term contains no variables. A trivial term is a variable. A linear term is one in which all variables are distinct.

Definition 6.2 (substitution). A substitution, written \( \{ x_1/t_1, \ldots, x_n/t_n \} \), represents a mapping from distinct variables \( x_i \) to terms \( t_i \). If \( s \) is a term and \( \theta \) is a substitution, then \( s\theta \) is the term obtained from \( s \) by simultaneously replacing each occurrence of \( x_i \) with \( t_i \), for all \( x_i/t_i \in \theta \).

Definition 6.3 (instance, anti-instance). Let \( t_1, t_2 \) be terms. If there exists some substitution \( \theta \) such that \( t_1\theta = t_2 \) then we say that \( t_2 \) is an instance of \( t_1 \) and conversely that \( t_1 \) is an anti-instance of \( t_2 \).

Definition 6.4 (variant). We say that two terms are variants if they are identical up to renaming of variables.

Definition 6.5 (place, subterm). A place is a list of natural numbers marking an occurrence of a subterm within a term. The set of places in a term is defined
as follows:

- \([\ ]\) is a place in any term.  

- A list of natural numbers \(i : p\) is a place in a term \(t\) if and only if \(t\) is of the form \(f(t_1 \ldots t_n)\) where \(n \geq i\), and \(p\) is a place in \(t_i\).  

Each place in a term marks a particular subterm as follows:

- The subterm at place \([\ ]\) in a term \(t\) is equal to \(t\).

- The subterm at a place of the form \(i : p\) in a term \(f(t_1 \ldots t_n)\) is equal to the subterm at place \(p\) in \(t_i\).

**Proposition 6.10.** For terms \(s, t\) where \(s\) is an anti-instance of \(t\), it holds that every place in \(s\) is a place in \(t\).

**Proposition 6.11.** For a term \(t\) of which \(t'\) is a subterm at some place \(a\), it holds for all places \(p\) in \(t'\) that:

- \(a + + p\) is a place in \(t\).

- The subterm at place \(a + + p\) in \(t\) is equal to the subterm at place \(p\) in \(t'\).

**Proof.** By structural induction on \(a\):

1. \(a = [\].\) By Eqn. 6.8 it holds that \(t' = t\), from which the proof then follows.

2. \(a = i : a'\). By Fact 6.7 it holds that \(t\) is of the form \(f(t_1 \ldots t_n)\) where \(n \geq i\), and \(a'\) is a place in \(t_i\). Furthermore by Eqn. 6.9, \(t'\) is equal to the subterm at place \(a'\) in \(t_i\). By the inductive hypothesis, \(a' + + p\) is a place in \(t_i\), and also the subterm at place \(a' + + p\) in \(t_i\) is equal to the subterm at place \(p\) in \(t'\). The two statements that we wish to prove then follow by Fact 6.7 and Eqn. 6.9 respectively.

\[\square\]

**Definition 6.12** (\(\text{varPlaces}\)). For a term \(t\), we define \(\text{varPlaces}(t)\) to be the set of places in \(t\) at which variables occur.

**Definition 6.13** (<\>). For places \(p, q\) we write \(p < q\) if \(p\) comes before \(q\) lexicographically. We also write \(p \leq q\) to mean \((p < q\) or \(p = q\)).

**Proposition 6.14.** The \(\leq\) relation over places is a total order.

**Definition 6.15** (ancestor, descendant). Let \(p, q\) be places. \(p\) is an improper ancestor of \(q\) (\(q\) is an improper descendant of \(p\)) if \(p\) is a prefix of \(q\). \(p\) is an ancestor of \(q\) (\(q\) is a descendant of \(p\)) if \(p\) is an improper ancestor of \(q\) and \(p \neq q\).

**Proposition 6.16.** The relation ‘is an improper ancestor of’ over places is a partial order.
The following concept of \textit{structural similarity} allows one to talk about terms as having the same tree structure, ignoring the details of function symbols and variables:

\textbf{Definition 6.17} (structurally similar). Terms \(s, t\) are \textit{structurally similar} if the set of places in \(s\) is identical to the set of places in \(t\).

\section{The \textit{auSearch} Specification}

In this section I give a high-level specification for the intended behaviour of the \textit{auSearch} algorithm. In the next section I shall then give the algorithm itself, and prove that it meets this specification.

I start with some preliminary definitions. \textit{auSearch} is parametrised by a ground term called the \textit{master term}:

\textbf{Definition 6.18} (master term). The \textit{master term} is some ground term.

The terms provided as input to \textit{auSearch} shall be ground, and furthermore each must be structurally similar to a subterm of the master term. These two properties are summarised in the following definition, \textit{LG-term}:

\textbf{Definition 6.19} (LG-term, location). A \textit{Located Ground term (LG-term)} is a pair \(\langle a, t \rangle\) where \(a\) is a place in the master term, and \(t\) is a ground term that is structurally similar to the subterm at place \(a\) in the master term. We call \(a\) the \textit{location} of the LG-term.

\textbf{Proposition 6.20}. For an LG-term \(\langle a, t \rangle\) and a term \(s\) that is an anti-instance of \(t\), it holds for all places \(p\) in \(s\) that \(a \oplus p\) is a place in the master term.

\textit{Proof}.

\[
\begin{align*}
p & \text{ is a place in } s \\
\implies p & \text{ is a place in } t \quad \text{(by 6.10)} \\
\implies p & \text{ is a place in the subterm at place } a \text{ in the master term} \quad \text{(by 6.19 and 6.17)} \\
\implies a \oplus p & \text{ is a place in the master term} \quad \text{(by 6.11)}
\end{align*}
\]

The output of \textit{auSearch} shall consist of objects called \textit{lassos}\textsuperscript{1}. A lasso delineates or ‘selects’ some piece of tree-structure within the master term. See Fig. 6.1 for an illustration.

\textsuperscript{1}The name \textit{lasso} is intended to be suggestive of the ‘lasso select’ tool sometimes found in image editing software.
Figure 6.1: An example involving an LG-term and a lasso. The three tree-diagrams above represent terms (for example, \( M = f(g(h, h), h) \)); function symbols are shown enclosed in circles and variables without circles. Suppose \( M \) is the master term. It follows that \( \langle [1], t \rangle \) is an LG-term since \( t \) is ground and also structurally similar to \( g(h, h, h) \), the subterm at place \([1]\) in \( M \). Also, \( s \) is an anti-instance of \( t \), and hence the lasso of \( s \) on \( \langle [1], t \rangle \) is \( \langle [1], \{[1, 1],[1, 3]\} \rangle \). This lasso, which contains places \([1]\) and \([1, 2]\), is represented in the diagram by the dashed loop.

**Definition 6.21** (lasso, location). A lasso is a pair \( \langle a, P \rangle \) where \( a \) is a place in the master term and \( P \) is a set of places in the master term, obeying the following conditions:

- Every element of \( P \) is an improper descendant of \( a \).

- No element of \( P \) is a descendant of any other.

We call \( a \) the location of the lasso.

**Definition 6.22** (lasso of \ldots on \ldots). For an LG-term \( \langle a, t \rangle \) and a term \( s \) that is an anti-instance of \( t \), we define the lasso of \( s \) on \( \langle a, t \rangle \) to be equal to:

\[
\langle a, \{ p \in \text{varPlaces}(s) : a \to p \} \rangle
\]

Note that due to Proposition 6.20 this is guaranteed to be a valid lasso.

**Definition 6.23** (contains, overlap, empty). A lasso \( \langle a, P \rangle \) contains a place \( x \) if \( x \) is in the master term, \( x \) is an improper descendant of \( a \), and \( x \) is not an improper descendant of any element of \( P \). Two lassos overlap if there exists a place that is contained in both of them. A lasso is empty if it contains no places.

**Definition 6.24** (non-overlapping, maximum non-overlapping subset). A non-overlapping set of lassos is one such that no two elements overlap. For a set \( A \) of lassos and a non-overlapping set \( B \subseteq A \), we say that \( B \) is a maximum non-overlapping subset of \( A \) if for every non-overlapping set \( C \subseteq A \) it holds that \(|B| \geq |C|\).

Figure 6.2 gives an input-output specification for *auSearch* as well as for its sub-algorithm *mnos*. Below is the specification for the behaviour of *mnos*. Later
on in this chapter I shall give an implementation of $mnos$ and prove that this theorem holds with respect to that implementation.

**Theorem 6.25** (*$mnos$* specification). For a set $X$ of non-empty lassos whose locations are all distinct, it holds that $mnos(X)$ is a maximum non-overlapping subset of $X$.

In order to write down the specification for the behaviour of $auSearch$, I need to define the following ‘pattern matching’ helper function:

**Definition 6.26** (*match*). The function $match$ takes a term and a set of LG-terms, and returns a set of lassos:

$$match(s, T) = \{ \langle a, t \rangle \mid s \text{ is an anti-instance of } t \bullet \text{ the lasso of } s \text{ on } \langle a, t \rangle \}$$

**Proposition 6.27.** For a non-trivial term $s$ and a set $T$ of LG-terms whose locations are all distinct, it holds that $match(s, T)$ is a set of non-empty lassos whose locations are all distinct.

**Proof.** First show that the elements of $match(s, T)$ are non-empty:

$s$ is non-trivial

$$\implies \text{ any lasso of } s \text{ on an LG-term } \langle a, t \rangle \text{ contains } a$$  
(by 6.22 and 6.23)

$$\implies \text{ each element of } match(s, T) \text{ is non-empty}$$  
(by 6.26 and 6.23)

Second, show by contradiction that no two elements of $match(s, T)$ share the same location:

two elements of $match(s, T)$ share the same location

$$\implies \text{ lassos of } s \text{ on two elements of } T \text{ share the same location}$$  
(by 6.26)

$$\implies \text{ two elements of } T \text{ share the same location}$$  
(by 6.22)

$$\implies \text{ false}$$
The specification for the behaviour of \textit{auSearch} consists of the two theorems below. The parameters $\sigma$ and $\tau$ are called \textit{beam size parameters}; these are intended to limit the computational complexity of the search at the expense of completeness, and I shall describe their function in more detail in the next section. When the beam size parameters are very large, \textit{auSearch} is an exhaustive search of the space characterised in Theorem 6.28. When the beam size parameters are small, Theorem 6.29 guarantees that the search is still sound, though it may no longer be complete.

\textbf{Theorem 6.28} (convergence of \textit{auSearch}). Let $T$ be a set of LG-terms whose locations are all distinct. There then exist non-negative integers $\sigma, \tau$ such that for all integers $\sigma', \tau' \geq \sigma$ and $\tau' \geq \tau$, the following holds for all sets $R$ of lassos:

$$R \in \text{auSearch}(\sigma', \tau', T) \iff R = \text{mnos}(\text{match}(s, T)) \text{ for some non-trivial linear term } s, \text{ and } |R| \geq 2$$

Note that Proposition 6.27 guarantees that this assertion is well-defined.

The above theorem implies that $\text{auSearch}(\sigma, \tau, T)$ has a well-defined value in the limit as $\sigma, \tau \to \infty$. Let us denote this limiting value as $\text{auSearch}(\infty, \infty, T)$.

\textbf{Theorem 6.29} (soundness of \textit{auSearch}). Let $T$ be a set of LG-terms whose locations are all distinct, and let $\sigma, \tau$ be non-negative integers. It follows that:

$$\text{auSearch}(\sigma, \tau, T) \subseteq \text{auSearch}(\infty, \infty, T)$$

In the next section I shall give an implementation of \textit{auSearch} and prove that these theorems hold with respect to that implementation. For a visual example illustrating some of the behaviour of \textit{auSearch}, see Fig. 6.3.

\section{The \textit{auSearch} Algorithm}

We saw in Theorem 6.28 that each element of the set of results returned by a call to \textit{auSearch} represents a non-overlapping set of matches of some ‘template’ term $s$ over all the input terms. In the context of abstraction invention, this template term corresponds to a pattern that can be abstracted, and the matches correspond to occurrences of this pattern within the corpus of knowledge. We shall see in the next chapter that, when choosing an \textit{auSearch} result from which to make an abstraction, it is desirable to maximise if possible both the size of the pattern and the number of matches. Indeed, there is often a trade-off to be had between these two desiderata.

In the \textit{auSearch} algorithm, we allow the user to specify a custom \textit{score} function in order to guide the search towards results that maximise such desiderata.
Figure 6.3: An example illustrating the behaviour of auSearch. Suppose $M$ is the master term. Below it in the diagram are some examples $s_i$ of non-trivial linear terms, along with corresponding sets of lassos given by $R_i = \text{mnos}(\text{match}(s_i, T))$ where $T = \{p \prec \text{places in } M \cdot (p, \text{ the subterm at place } p \text{ in } M)\}$. By Theorem 6.28, $R_1$ and $R_2$ are members of $auSearch(\infty, \infty, T)$, whereas $R_3$ is not because it contains only one element. Note that for $i = 1$, $\text{match}(s_i, T)$ has a unique maximum non-overlapping subset, whereas for $i = 2$ there are three different maximum non-overlapping subsets to choose from, and two for $i = 3$. Which subsets actually get chosen depends on the implementation details of $\text{mnos}$; the $R_i$ shown here are those chosen by the $\text{mnos}$ implementation to be described later in this chapter.
From Theorem 6.28, it is clear that the total number of results is in the worst case exponentially large in the size of the input. In order to make the search tractable, it is therefore crucial that guidance is effective, in order to efficiently discover results that maximise the score within this large search space.

In order to achieve guidance, the approach shall be to ‘grow’ template terms incrementally, updating a template’s score each time it is enlarged based on its size and its number of matches against the input terms. The beam parameters \( \tau, \sigma \) have the following meaning. At each step in the search, all but the top \( \tau \) such templates by score shall be discarded. Furthermore, for each place in the master term, all but the top \( \sigma \) templates whose location is equal to that place are discarded.

In order to ‘grow’ the templates, we need the following definitions alongside the definitions of the previous section. An RLG-term is an LG-term that is in the process of being ‘grown’. Likewise, a shoot is a lasso that is in the process of being ‘grown’.

**Definition 6.30** (RLG-term, root, location). A Rooted Located Ground term (RLG-term) is a triple \( \langle a, b, t \rangle \) where \( a \) is a place in the master term, \( \langle b, t \rangle \) is an LG-term, and \( b \) is an improper descendant of \( a \). We call \( a \) the **root**, and \( b \) the **location**, of the RLG-term.

**Definition 6.31** (shoot, root). A shoot is a triple \( \langle a, b, P \rangle \) where \( a \) is a place in the master term, \( \langle b, P \rangle \) is a lasso, and \( b \) is an improper descendant of \( a \). We call \( a \) the **root** of the shoot.

**Definition 6.32** (root-set). The **root-set** of a set \( S \) of shoots is equal to the following set of places:

\[
\{ \langle a, b, P \rangle \mid S \bullet a \}
\]

**Definition 6.33** (contains, overlap, etc.). A shoot \( \langle a, b, P \rangle \) contains a place \( x \) if \( x \) is in the master term and either of the following two conditions hold:

- \( x \) is an improper descendant of \( a \) and a proper ancestor of \( b \);
- \( \langle b, P \rangle \) contains \( x \).

We define overlap, empty, non-overlapping, and maximum non-overlapping subset for shoots in the same way as for lassos (Defns. 6.23 and 6.24).

An algorithm \( \text{mnos}' \) serves the same purpose as \( \text{mnos} \), but for shoots rather than lassos. The type signature is given in Fig. 6.4 and the function specification for \( \text{mnos}' \) is as follows:

**Lemma 6.34** (**mnos'** specification). For a set \( S \) of non-empty shoots whose roots are all distinct, it holds that \( \text{mnos}'(S) \) is a maximum non-overlapping subset of \( S \).
function $mnos'(S)$:

input: $S$, a set of non-empty shoots whose roots are all distinct.

output: a set of shoots.

Figure 6.4: The type signature for $mnos'$.

The $auSearch$ algorithm is parametrised by some $score$ function mapping every set of shoots to a real number:

**Definition 6.35** ($score$). Every set of shoots has a real number, its $score$, associated with it.

In Fig. 6.5 and Fig. 6.6 is given the $auSearch$ algorithm itself. I shall defer giving the $mnos'$ algorithm until the next section. I now give a proof of Theorem 6.28 and Theorem 6.29, showing that the $auSearch$ algorithm meets its specification.

**Definition 6.36** ($match'$). The function $match'$ takes a term and a set of RLG-terms, and returns a set of shoots:

$$match'(s,T) = \{⟨a,b,t⟩ \prec T \mid s \text{ is an anti-instance of } t$$

$$\bullet ⟨a,b',P⟩ \text{ where } ⟨b',P⟩ \text{ is the lasso of } s \text{ on } ⟨b,t⟩\}$$

**Proposition 6.37.** For a non-trivial term $s$ and a set $T$ of RLG-terms whose roots are all distinct, it holds that $match'(s,T)$ is a set of non-empty shoots whose roots are all distinct.

**Proof.** First show that the elements of $match'(s,T)$ are non-empty:

$s$ is non-trivial

$\implies$ any lasso of $s$ on an LG-term $⟨b,t⟩$ contains $b$ (by 6.22 and 6.23)

$\implies$ each element of $match'(s,T)$ is non-empty (by 6.36 and 6.33)

Second, show by contradiction that no two elements of $match'(s,T)$ share the same root:

two elements of $match'(s,T)$ share the same root

$\implies$ two elements of $T$ share the same root (by 6.36)

$\implies$ false

**Lemma 6.38.** If $s$ is a trivial term and $T$ is a set of RLG-terms then:

$$match'(s,T) = \{⟨a,b,t⟩ \prec T \bullet (a,b,\{b\})\}$$

**Proof.** The proof follows by Defn. 6.36 and Defn. 6.22. }
function auSearch(σ, τ, T):
  input:  σ, τ, non-negative integers;
          T, a set of LG-terms whose locations are all distinct.
  output: a set of sets of lassos.
  1. Let $T'$, a set of RLG-terms, equal \{⟨a, t⟩ ▷ T • ⟨a, a, t⟩\}.
  2. Let Beam, a set of sets of shoots, equal
     \{R ◁ findMany(σ, τ, T') • mnos'(R)\}.
  3. Return \{R ◁ Beam • \{(a, b, P) ◁ R • ⟨a, P⟩\}\}.

function findMany(σ, τ, T):
  input:  σ, τ, non-negative integers;
          T, a set of RLG-terms.
  output: a set of sets of shoots.
  1. Let Beam be the set of all possible values of findOne(σ, τ, T).
  2. Partition Beam into non-empty disjoint subsets, each of the form
     \{R ◁ Beam | the root-set of R is equal to X\} for some X. From each of
     these subsets extract the top σ elements by score, and place all the
     extracted elements into a new set Beam'.
  3. Return the top τ elements by score of Beam'.

Figure 6.5: The auSearch function and its helper function findMany. The main part of
the auSearch algorithm is actually contained in the findOne function of Fig.6.6. The
auSearch function itself merely translates LG-terms into RLG-terms at the beginning,
and then translates shoots into lassos at the end. findMany and findOne are mutually
recursive. The findMany function invokes findOne and then discards any intermediate
search results that exceed the limits determined by the beam size parameters σ and τ.
nondeterministic function $\text{findOne}(\sigma, \tau, T)$:

**input:** $\sigma, \tau$, non-negative integers;
$T$, a set of RLG-terms.

**output:** a set of shoots.

1. Let $G$, a set of RLG-terms, be a non-empty subset of $T$ of the form $\{ \langle a, b, t \rangle \triangleleft T \mid$ the principal functor of $t$ is equal to $f \}$ for some function symbol $f$.
   Let $n$ be the arity of $f$.

2. Let $\mathcal{S}$, a set of shoots, equal
   $\{ \langle a, b, t \rangle \triangleleft \mathcal{G} \bullet \langle a, b, \{ b \rightarrow + [1], b \rightarrow + [2], \ldots, b \rightarrow + [n]\} \rangle \}$.

3. If $|\text{mnos}'(\mathcal{S})| < 2$ then fail.

4. For each $i \in \{1 \ldots n\}$:

5. Let $\mathcal{B}_i$, a set of RLG-terms, equal
   $\{ \langle a, b, f(t_1 \ldots t_n) \rangle \triangleleft \mathcal{G} \bullet \langle a, b \rightarrow [i], t_i \rangle \}$.

6. Let $\mathcal{R}_i$, a set of shoots, either equal $\{ \langle a, b, t \rangle \triangleleft \mathcal{B}_i \bullet \langle a, b, \{b\}\rangle \}$ or be an element of $\text{findMany}(\sigma, \tau, \mathcal{B}_i)$.

7. Let $\mathcal{R}$, a set of shoots, equal
   $\{ \langle a_1, b_1, P_1 \rangle \triangleleft \mathcal{R}_1; \langle a_2, b_2, P_2 \rangle \triangleleft \mathcal{R}_2; \ldots; \langle a_n, b_n, P_n \rangle \triangleleft \mathcal{R}_n$
   $\mid a_1 = a_2 = \ldots = a_n \bullet \langle a_1, \text{init}(b_1), P_1 \cup P_2 \cup \ldots \cup P_n \rangle \}$.

8. If $|\text{mnos}'(\mathcal{R})| < 2$ then fail, otherwise return $\mathcal{R}$.

Figure 6.6: The helper function $\text{findOne}$, which forms the core of the auSearch algorithm. In line 1, the set of RLG-terms is partitioned according to the function symbol at the head of each term. In lines 2 and 3, an early check for overlap is made. Then in lines 4, 5, and 6, $\text{findMany}$ is invoked recursively in order to search for commonality among the bodies of the RLG-terms. Line 7 is a kind of ‘direct product’ operation which grows the shoot by joining together the shoots that were derived from the different branches of the RLG-terms below their head function symbol. Finally at line 8, the main check for overlap is made.
Lemma 6.39. Let $s_1, s_2$ be terms and let $T$ be a set of RLG-terms. If $s_1$ and $s_2$ are variants then:

$$\text{match}'(s_1, T) = \text{match}'(s_2, T)$$

Proof. By Defn. 6.4 and Defn. 6.3, it holds for any term $t$ that $s_1$ is an anti-instance of $t$ if and only if $s_2$ is an anti-instance of $t$. It follows by Defn. 6.36 and Defn. 6.22 that $LHS = RHS$. 

Lemma 6.40. For a linear term $f(s_1 \ldots s_n)$ and a ground term $f(t_1 \ldots t_n)$, the following holds:

$$\forall i \in \{1 \ldots n\} (s_i \text{ is an anti-instance of } t_i) \iff f(s_1 \ldots s_n) \text{ is an anti-instance of } f(t_1 \ldots t_n)$$

Proof. Let $s = f(s_1 \ldots s_n)$ and let $t = f(t_1 \ldots t_n)$. If $LHS$ holds, then by Defn. 6.3 there exist substitutions $\theta_1 \ldots \theta_n$ such that $\forall i (s_i \theta_i = t_i)$. Since $s$ is linear, the $\theta_i$ may be chosen such that $s_i \theta_j = s_i$ for all $i,j$ where $j \neq i$. Hence $s \theta_1 \ldots \theta_n = t$, from which $RHS$ follows by Defn. 6.3 and the fact that substitutions can be composed [Nienhuys-Cheng and de Wolf, 1997, Sec. 4.3]. Conversely, if $RHS$ holds then by Defn. 6.3 there exists a substitution $\theta$ such that $s \theta = t$. Therefore $\forall i (s_i \theta = t_i)$, from which $LHS$ follows by Defn. 6.3. 

Lemma 6.41. Let $\sigma, \tau$ be non-negative integers and let $T$ be a set of RLG-terms whose roots are all distinct. Suppose that evaluation of findOne($\sigma, \tau, T$), using thefindOne algorithm of Fig. 6.6, has proceeded successfully along some path as far as the end of line 7. Local variables $G, f, n, B_1 \ldots B_n, R_1 \ldots R_n,$ and $R$ are in scope at this point. Let $s_1 \ldots s_n$ be terms such that $f(s_1 \ldots s_n)$ is linear and $R_i = \text{match}'(s_i, B_i)$ for each $i \in \{1 \ldots n\}$. It follows that:

$$R = \text{match}'(f(s_1 \ldots s_n), T)$$

Proof. By Defn. 6.36 and Defn. 6.22, we derive the following fact $A$ which holds for each $i$:

$$R_i = \{ (a, b, t) \mid B_i \upharpoonright s_i \text{ is an anti-instance of } t \}
\begin{align*}
\bullet (a, b, \{ p \leftarrow \text{varPlaces}(s_i) \bullet b \leftrightarrow p \})
\end{align*}$$
Hence:

\[
\text{LHS} = \{ \langle a_1, b_1, t_1 \rangle \triangleleft B_1; \ldots; \langle a_n, b_n, t_n \rangle \triangleleft B_n \mid a_1 = a_2 = \ldots = a_n \text{ and } \forall i (s_i \text{ is an anti-instance of } t_i) \text{ and } \forall i (s_i \text{ is an anti-instance of } t_i) \}
\]

(by fact \(A\) and the definition of \(R\))

\[
= \{ \langle a, b, f(t_1 \ldots t_n) \rangle \triangleleft G \mid \forall i (s_i \text{ is an anti-instance of } t_i) \}
\]

(by the definition of \(B_i\))

\[
= \text{match}'(f(s_1 \ldots s_n), G)
\]

(by 6.36 and the definition of \(G\))

\[
= \text{RHS}
\]

Lemma 6.42 (convergence of \(\text{findMany}\)). Let \(T\) be a set of RLG-terms whose roots are all distinct. There then exist non-negative integers \(\sigma, \tau\) such that for all integers \(\sigma' > \sigma\) and \(\tau' > \tau\), the following holds for all sets \(R\) of shoots:

\[
R \in \text{findMany}(\sigma', \tau', T) \iff R = \text{match}'(s, T) \text{ for some non-trivial linear term } s, \text{ and } |\text{mnos}'(R)| \geq 2
\]

Note that Proposition 6.37 guarantees that the above assertion is well-defined.

Proof. By induction on \(N(T) = \sum_{\langle a,b,t \rangle \in T} \text{size}(t) \) where \(\text{size}(t)\) is the number of places in \(t\):

1. \(N(T) = 0\), and hence \(T\) is empty. If \(\text{RHS}\) is true then, by Defn. 6.36, \(R\) is empty, which by Lemma 6.34 contradicts the fact that \(|\text{mnos}'(R)| \geq 2\). Therefore \(\text{RHS}\) is false. Furthermore, \(\text{findOne}(\sigma', \tau', T)\) will fail at line 1, therefore \(\text{LHS}\) is also false.

2. \(N(T) \geq 1\), and hence \(T\) is non-empty. We first prove the forward implication, followed by its converse:

(a) Assume \(\text{LHS}\). By lines 1, 2, and 3 of \(\text{findMany}\), \(R\) is a possible value of \(\text{findOne}(\sigma', \tau', T)\). Therefore, evaluation of \(\text{findOne}(\sigma', \tau', T)\) must be able to proceed successfully along some path right up to the end of line 8 of the \(\text{findOne}\) algorithm, where local variables \(f, n, B_1 \ldots B_n, \text{ and } R_1 \ldots R_n\) are in scope. By line 8 of \(\text{findOne}\), \(|\text{mnos}'(R)| \geq 2\).
It now remains to prove that \( R = match'(s, T) \) for some non-trivial linear term \( s \). By the definition of \( R_i \), Lemma 6.38, and the inductive hypothesis, it holds for each \( i \in \{1 \ldots n\} \) that \( R_i = match'(s_i, B_i) \) for some linear term \( s_i \). By Lemma 6.39, the \( s_i \) may be chosen such that \( f(s_1 \ldots s_n) \) is linear. The proof then follows by Lemma 6.41.

(b) Assume RHS. Hence, \( R = match'(s, T) \) where \( s \) is some linear term of the form \( f(s_1 \ldots s_n) \). Also, by Lemma 6.34 it holds that \( |R| \geq 2 \). Now, consider an evaluation path of \( \text{findOne}(\sigma', \tau', T) \). Since \( T \) is non-empty, evaluation will proceed beyond line 1 of the \( \text{findOne} \) algorithm on at least one such path, at which point the local variable \( G \) is in scope. By Defn. 6.36 and the definition of \( G \), we may choose this path such that the local variables \( f, n \) introduced at line 1 are the same as the \( f, n \) defined above, from which it further follows that \( R = match'(s, G) \). Evaluation then proceeds up to line 3, at which point the local variable \( S \) is in scope. The following working shows that evaluation will not fail at line 3:

\[
|mnos'(S)| \geq |mnos'((\{(a,b,t) \triangleleft G\} \setminus \{s \text{ is an anti-instance of } t\})
\bullet (a,b,\{p \triangleleft \{[1],[2],\ldots,[n]\} \bullet b \Join p\})))|
\begin{align*}
\text{(by the definition of } S) \\
\geq |mnos'((\{(a,b,t) \triangleleft G\} \setminus \{s \text{ is an anti-instance of } t\})
\bullet (a,b,\{p \triangleleft \text{varPlaces}(s) \bullet b \Join p\})))|
\text{(by 6.34 and 6.33)} \\
\geq |mnos'(R)|
\text{(by 6.36 and 6.22)} \\
\geq 2
\end{align*}
\]

Next, evaluation proceeds up to the end of line 6, where local variables \( B_1 \ldots B_n \) and \( R_1 \ldots R_n \) are in scope. We derive the following
fact A:

\[ |mnos'(match'(s_i, B_i))| \]

\[ \geq |mnos'(\{(a, b, t) \triangleleft B_i \mid s_i \text{ is an anti-instance of } t \}
\bullet (a, b, \{p \triangleleft \text{varPlaces}(s_i) \bullet b' \triangleleft p\})| \]

(by 6.36 and 6.22)

\[ \geq |mnos'(\{(a, b, f(t_1 \ldots t_n)) \triangleleft G \mid f(s_1 \ldots s_n) \text{ is an anti-instance of } f(t_1 \ldots t_n) \}
\bullet (a, b, \{p \triangleleft \text{varPlaces}(f(s_1 \ldots s_n)) \bullet b' \triangleleft p\})| \]

(by the definition of \(B_i\))

\[ \geq |mnos'(R)| \]

(by 6.36 and 6.22)

\[ \geq 2 \]

By Lemma 6.38, fact A, and the inductive hypothesis, the evaluation path up to the end of line 6 may be chosen such that \(\forall i(R_i = match'(s_i, B_i))\). On this path, when evaluation reaches the end of line 7, it follows by Lemma 6.41 that the local variable \(R\) has the same value as the \(R\) already defined above. Failure will not occur at line 8, therefore \(R\) is a possible value of \(\text{findOne}(\sigma', \tau', T)\). The proof then follows by lines 1, 2, and 3 of \(\text{findMany}\).

\[ \square \]

**Proof of Theorem 6.28.** Let \(T'\) be the value of the local variable \(T'\) at line 1 of the \(auSearch\) algorithm during the evaluation of \(auSearch(\sigma', \tau', T)\). We derive the following fact A which holds for all terms \(s\):

\[
\text{match}'(s, T') = \{(a, t) \triangleleft T' \mid s \text{ is an anti-instance of } t \}
\bullet (a, a', P) \text{ where } (a', P) \text{ is the lasso of } s \text{ on } (a, t) \}

(by 6.36 and the definition of \(T'\))

\[ = \{(a, P) \triangleleft \text{match}(s, T) \bullet (a, a, P)\} \]

Using fact A we derive the following fact B which holds for all terms \(s\):

\[
\{(a, b, P) \triangleleft mnos'(\text{match}'(s, T')) \bullet (a, P)\} = mnos(\text{match}(s, T))
\]

Then using fact B we derive the following fact C which holds for all terms \(s\):

\[
|mnos'(\text{match}'(s, T'))| = |mnos(\text{match}(s, T))|
\]
The proof then follows:

\[ LHS \iff R = \{ \langle a, b, P \rangle \triangleleft\triangleleft mnos'(R') \circ \langle a, P \rangle \} \]

for some \( R' \in \text{findMany}(\sigma', \tau', T') \)

\[ \iff R = \{ \langle a, b, P \rangle \triangleleft\triangleleft mnos'(\text{match}(s, T')) \circ \langle a, P \rangle \} \]

for some non-trivial linear term \( s \),

and \(|mnos'(\text{match}(s, T'))| \geq 2\)

\[ \iff RHS \]

\[ \square \]

**Proof of Theorem 6.29.** By induction on \( N(T) \) as in the proof of Lemma 6.42 it holds that:

\[ \text{findMany}(\sigma, \tau, T) \subseteq \text{findMany}(\infty, \infty, T) \]

The proof then follows straightforwardly by the definition of \( \text{auSearch} \) (Fig. 6.5).

\[ \square \]

### 6.4 The mnos Algorithm

In this section I describe the maximum non-overlapping subset algorithms \( mnos \) and \( mnos' \), which work for lassos and shoots respectively. I prove that both algorithms meet their specifications (Theorem 6.25 and Lemma 6.34).

#### 6.4.1 Gavril’s maximum independent set algorithm

The maximum non-overlapping subset algorithms are based on an algorithm by Gavril [1972] for finding a maximum independent set of a chordal graph. I shall state Gavril’s algorithm and its properties first, and then explain how it can be used to implement \( mnos \) and \( mnos' \).

It is assumed that the reader is familiar with basic graph theory terminology. Graphs are taken to be undirected, finite, and simple (no self-loops or parallel edges) unless otherwise specified. Between two vertices \( a, b \) we denote the existence of an undirected edge by \( a \cdash b \), and a directed edge by \( a \to b \).

Gavril’s algorithm makes use of a particular kind of ordering imposed on the vertices of a chordal graph. In modern graph theory literature, this is usually called a *perfect elimination ordering* [Rose et al., 1976], however to avoid any confusion I shall stick to the terminology used by Gavril in his 1972 paper. Gavril states the following definition and two results:

**Definition 6.43** (R-oriented, R-orientation). A directed graph is *R-oriented* if the following two conditions hold:
• it has no directed cycles;
• for all vertices \(a, b, c\) such that \(a \neq b\): if \(a \rightarrow c\) and \(b \rightarrow c\), then either
  \(a \rightarrow b\) or \(b \rightarrow a\).

An \(R\)-orientation of a graph is an assignment of direction to each of its edges
such that the resulting directed graph is \(R\)-oriented.

**Proposition 6.44.** An undirected graph is chordal if and only if it has an
\(R\)-orientation.

**Proposition 6.45.** Given an \(R\)-oriented directed graph, it is always possible to
number the vertices \(1, 2 \ldots n\), where \(n\) is the total number of vertices, such that
that all edges are directed from lower to higher numbers.

For our purposes, Proposition 6.44 may suffice as a definition of what it
means for a graph to be `chordal`. Proposition 6.45 guarantees the existence of
the ordering on vertices that will be crucial to Gavril’s algorithm.

The following is Gavril’s maximum independent set algorithm:

**Definition 6.46** (Gavril’s algorithm). Let \(G\) be an \(R\)-oriented directed
graph whose vertices are numbered \(1, 2 \ldots n\) such that all edges are directed from low
to high. The function \(gavril\) takes and returns a list of vertices of \(G\):

\[
gavril([1, 2 \ldots n]) = [n_t, n_{t-1} \ldots n_1]
\]

where

• \(n_1 = n\);  \hspace{1cm} (6.47)
• for \(2 \leq k \leq t\): \(n_k\) is the highest vertex smaller than \(n_{k-1}\) that is not
  a direct predecessor of any of \(n_1, n_2 \ldots n_{k-1}\);  \hspace{1cm} (6.48)
• all vertices smaller than \(n_t\) are direct predecessors of at least one of
  \(n_1, n_2 \ldots n_t\).  \hspace{1cm} (6.49)

Finally, the following result, which Gavril proves in his paper, confirms that
this algorithm behaves as desired:

**Proposition 6.50.** Let \(G\) be a chordal graph of which \(G'\) is an \(R\)-orientation,
and let \(1, 2 \ldots n\) be the vertices of \(G'\) arranged such that all edges are directed
from low to high numbers. It follows that the elements of \(gavril([1, 2 \ldots n])\)
constitute a maximum independent set of \(G\).

### 6.4.2 Using Gavril’s algorithm to find maximum non-overlapping subsets

I now show how to implement \(mnos'\) in terms of \(gavril\). Recall that \(mnos'\) takes,
as its input, a set of non-empty shoots whose roots are all distinct (Fig. 6.4 on
**function mnos′(S):**

**input:** \( S \), a set of non-empty shoots whose roots are all distinct.

**output:** a set of shoots.

1. Let \( G \) be the directed overlap graph of \( S \).

2. Let \( \Sigma \) be the list containing the elements of \( S \) ordered lexicographically by root.

3. Return the set of elements of \( gavril_G(\Sigma) \).

**function mnos(X):**

**input:** \( X \), a set of non-empty lassos whose locations are all distinct.

**output:** a set of lassos.

1. Let \( S \), a set of shoots, equal \( \{ (a, P) \bowtie X \cdot (a, a, P) \} \).

2. Return \( \{ (a, b, P) \bowtie mnos'(S) \cdot (a, P) \} \).

Figure 6.7: The mnos’ and mnos algorithms. Note that the subscript \( G \) in \( gavril_G(\Sigma) \) (line 3 of mnos’) clarifies that the elements of \( \Sigma \) are to be interpreted here as vertices of \( G \).

Page 114); we can use the ‘overlaps’ relation to convert any set of shoots to a graph as follows:

**Definition 6.51** (overlap graph). The **overlap graph** of a set \( S \) of shoots is the graph \( G \) such that:

- \( S \) is the set of vertices of \( G \);
- for \( a, b \in S \): \( a \bowtie b \) i.f.f. \( a \) overlaps \( b \).

We can then construct an R-orientation of such a graph like so:

**Definition 6.52** (directed overlap graph). For a set \( S \) of non-empty shoots whose roots are all distinct, the **directed overlap graph** of \( S \) is obtained from its overlap graph by assigning a direction to each edge \( a \bowtie b \) as follows: if the root of \( a \) comes before the root of \( b \) lexicographically then \( a \rightarrow b \), else \( b \rightarrow a \).

**Lemma 6.53.** For a set \( S \) of non-empty shoots whose roots are all distinct, the directed overlap graph of \( S \) is \( R \)-oriented.

**Proof.** Given in Appendix B.

This gives us everything we need. It is clear by Defns. 6.24 and 6.51 that a maximum non-overlapping subset of a set of shoots corresponds to a maximum independent set of its overlap graph. Putting everything together, we obtain the two maximum non-overlapping subset algorithms mnos’ and mnos (Fig. 6.7).
Note that $mnos'$ is implemented directly in terms of $gavril$, while $mnos$ is in turn implemented in terms of $mnos'$. Finally I now prove that both algorithms meet their earlier specifications:

Proof of Lemma 6.34. Define $G$ and $\Sigma$ as in the $mnos'$ algorithm of Fig.6.7. By Lemma 6.53, $G$ is an R-orientation of the overlap graph of $S$. If we number the elements of $\Sigma$ from left to right as $1, 2 \ldots n$, then by Defn.6.52 and the definition of $\Sigma$, all edges of $G$ between these elements are directed from low to high numbers. Hence by Proposition 6.50, $mnos'(S)$ is a maximum independent set of the overlap graph of $S$, from which the proof follows by Defn.6.51 and Defn.6.24.

Proof of Theorem 6.25. Define $S$ as in the $mnos$ algorithm of Fig.6.7. Observe that the elements of $X$ are related to the elements of $S$ by the following one-one mapping:

$$f((a, P)) = (a, a, P)$$

By Defn.6.33, the ‘overlaps’ relationship is preserved under this mapping, i.e. for all $x_1, x_2 \in X$:

$$x_1 \text{ overlaps } x_2 \iff f(x_1) \text{ overlaps } f(x_2)$$

By Defn.6.31 and Defn.6.33, $S$ is a set of non-empty shoots whose roots are all distinct. The proof then follows:

$$mnos'(S) \text{ is a maximum non-overlapping subset of } S \quad \text{(by 6.34)}$$

$$\Rightarrow \{ x < mnos(X) \cdot f(x) \} \text{ is a maximum non-overlapping subset of } \{ x < X \cdot f(x) \} \quad \text{(by lines 1 and 2 in the } mnos \text{ algorithm)}$$

$$\Rightarrow mnos(X) \text{ is a maximum non-overlapping subset of } X \quad \text{(by the preservation of the ‘overlaps’ relation under } f)$$
Chapter 7

Abstraction Invention and KANDINSKY

In this chapter I describe the KANDINSKY abstraction invention system. It works within the setting for learning of Sect. 3.4 and uses the toplevel environment and cumulative learning prior of Sect. 4.3. KANDINSKY discovers syntactic commonality within a lambda calculus term by invoking the anti-unification search algorithm of the previous chapter. Once commonality has been found, it transforms the term so as to create an abstraction from that commonality.

When KANDINSKY transforms a term, it does so in such a way that the value of the term does not change with respect to the toplevel environment, and hence the likelihood of the term does not change. However, the transformation does change the term’s prior probability. The aim is to find a transformation so as to increase the prior probability as much as possible. Since the likelihood stays the same, this also increases the posterior probability as much as possible.

The transformation process that KANDINSKY uses to construct an abstraction is a form of inverse deduction. It involves running the deduction rule of lambda calculus known as beta reduction [Stoy, 1981] in reverse, and is very much a lambda calculus analogue of the inter-construction operator of inverse resolution (Sect. 2.2.2). Both KANDINSKY’s transformation operator and the inter-construction operator work by deriving a novel concept from commonality in existing knowledge.

7.1 The absInv Algorithm

The input to KANDINSKY is a master term, which is a hypothesis for a monolithic cumulative learning problem as in Sect. 4.3:

Definition 7.1 (master term, body place). A master term is a long normal
term of the following form:

\[ T = \text{master} (\lambda a_1 \ldots a_m \rightarrow \text{Tuple } A_1 \ldots A_m B_1 \ldots B_n) \]

whose lambda parameters are all distinctly named. A body place in \( T \) is a place that points inside one of the subterms \( A_i \) or \( B_j \). In other words, a body place in \( T \) is a place in \( T \) of the form:

\[ 2 : \text{replicate}(m, 1) : \text{replicate}(i, 1) : 2 : p \]

where \( 0 \leq i < m + n \) and \( p \) is a place.

In this chapter, unlike in Chap. 4 and Chap. 5, I do not assume Barendregt’s convention. Hence, the following standard notion of \textit{alpha-equivalence} [Stoy, 1981] is distinct from syntactic equality:

**Definition 7.2** (alpha-equivalent). Two terms are \textit{alpha-equivalent} if they are equal up to renaming of lambda parameters and bound variables.

The de-Bruijn index of a bound variable occurrence [de Bruijn, 1972] is equal to the ‘distance’ between that variable occurrence and the lambda abstraction that binds it. This ‘distance’ is measured by counting the number of lambda abstractions that occur between the two:

**Definition 7.3** (de-Bruijn index). The de-Bruijn index \( d \) of a bound variable occurrence within a term \( T \) is defined as follows. Let \( p \) be the place in \( T \) of the bound variable occurrence, and let \( q \) be the place in \( T \) of the lambda abstraction at which it is bound. The de-Bruijn index of the variable occurrence is equal to the number of places \( r \) in \( T \) that satisfy the following conditions:

- \( r \) is both an ancestor of \( p \) and an improper descendant of \( q \);
- the subterm at place \( r \) in \( T \) is a lambda abstraction.

In order to invoke anti-unification search on the master term, KANDINSKY must first encode it in the first-order term representation accepted by the \textit{auSearch} algorithm. This is done as follows. Note that de-Bruijn indices are used to enable syntactic matching of alpha-equivalent subterms:

**Definition 7.4** (encodeTerm). The function \textit{encodeTerm} takes a (lambda) term \( T \) and returns a first-order term \( T' \) such that the following conditions hold:

- The set of places in \( T' \) is equal to the set of places in \( T \) (see Defn. 3.8 and Defn. 6.5).
- At each place \( p \) in \( T \), let \( S, S' \) be the subterms at \( p \) in \( T, T' \) respectively:
  - If \( S \) is a variable whose occurrence at \( p \) in \( T \) is free, then \( S' \) is equal to \text{varFree}_S.
function findAbstractions(T):
    input: T, a master term.
    output: a set of sets of lassos over T.
1. Let \{p_1 \ldots p_N\} be the set of body places in T.
2. Let \{X_1 \ldots X_N\} be disjoint, countably infinite sets of symbol names.
3. For each \(i \in \{1 \ldots N\}:
4. Let \(S_i\) be the subterm at \(p_i\) in T.
5. Let \(Q_i\) be the set of places \(q\) such that a free variable occurs at \(q\) in \(S_i\) and a bound variable occurs at \((p + q)\) in T.
6. Let \(S'_i\) be the term obtained by replacing the variable at each place \(q \in Q_i\) in \(S_i\) with a symbol of appropriate type and a distinct name chosen from \(X_i\).
7. Let \(A_i = (p_i, \text{encodeTerm}(S'_i))\).
8. Return \(auSearch(\text{sigmaParam}, \text{tauParam}, \{A_1 \ldots A_N\})\).

Figure 7.1: The findAbstractions algorithm.

- If \(S\) is a variable whose occurrence at \(p\) in \(T\) is bound, then \(S'\) is equal to \(\text{varBound}_{\alpha d}\) where \(\alpha\) is the type of \(S\) and \(d\) is the \(\text{de-Bruijn}\) index of the variable occurrence at \(p\) in \(T\).
- If \(S\) is an application of the form \((M N)\) then the principal functor of \(S'\) is equal to \(\text{app}_{\alpha\beta}\) where \(\alpha \rightarrow \beta\) is the type of \(M\).
- If \(S\) is a lambda abstraction then the principal functor of \(S'\) is equal to \(\text{abs}_{\alpha\beta}\) where \(\alpha \rightarrow \beta\) is the type of \(S\).

KANDINSKY has two non-negative integers \(\text{sigmaParam}\) and \(\text{tauParam}\) as global parameters, which correspond to the beam size parameters \(\sigma\) and \(\tau\) of \(auSearch\) (Fig. 6.2):

Definition 7.5 (\(\text{sigmaParam}, \text{tauParam}\)). \(\text{sigmaParam}\) and \(\text{tauParam}\) are non-negative integers.

The full search algorithm used to find syntactic commonality within a master term is findAbstractions (Fig. 7.1). The action of this algorithm is to generate all of the subterms of the master term at body places (line 4), then to encode all of these subterms as LG-terms (line 7). Finally, the LG-terms are passed to \(auSearch\) (line 8). Note that the master first-order term of Defn. 6.18 is equal to \(\text{encodeTerm}(T)\) where \(T\) is the master lambda term.

Lines 5 and 6 of findAbstractions deal with something called ‘outside-bound variables’. An outside-bound variable is a variable in a subterm that is bound
The `genSymbols` procedure:

```latex
\textbf{stateful function} \texttt{genSymbols(tys)}:

\begin{itemize}
  \item \textbf{input}: \texttt{tys}, a list of types.
  \item \textbf{output}: a list of symbols.
  \item \textbf{state}: \texttt{G}, a symbol generator.
\end{itemize}

1. Let \([\alpha_1 \ldots \alpha_n] = \texttt{tys}\) and let \([s_1, s_2, \ldots] = \texttt{G}\).
2. \(\texttt{G} \leftarrow [s_{n+1}, s_{n+2}, \ldots]\)
3. Return \([s_1: \alpha_1 \ldots s_n: \alpha_n]\).
```

Figure 7.2: The \texttt{genSymbols} procedure.

within the master term as a whole, but free within that subterm. It is important
that outside-bound variables do not match with one another during the search
for commonality, because an abstraction of commonality between such variables
would have no meaning at global scope. Therefore, all outside-bound variables
are replaced with distinctly named dummy variables that cannot match one
another.

A result returned by \texttt{auSearch} consists of a set of lassos (Defn. 6.21). Since
the encoding process of the master term does not change its tree-structure
(Defn. 7.4), these lassos are perfectly meaningful over the master lambda term.
For clarity, I give here a definition of a lasso over a lambda term:

**Definition 7.6** (lasso, location, contains, overlap, non-overlapping). A \textit{lasso}
over a term \(T\) is a pair \(\langle a, P \rangle\) where \(a\) is a place in \(T\) and \(P\) is a set of places in
\(T\), obeying the following conditions:

- Every element of \(P\) is an improper descendant of \(a\).
- No element of \(P\) is a descendant of any other.

We call \(a\) the \textit{location} of the lasso. \(\langle a, P \rangle\) \textit{contains} a place \(x\) if \(x\) is in \(T\), \(x\) is an
improper descendant of \(a\), and \(x\) is not an improper descendant of any element
of \(P\). Two lassos \textit{overlap} if there exists a place that is contained in both of
them. A \textit{non-overlapping} set of lassos is one such that no two elements overlap.

Once \texttt{auSearch} has been invoked on the master term and returned some re-
sults, then the next step is to transform the master term so as to construct an
abstraction from the commonality specified by one of these results. This trans-
formation process will involve generating fresh lambda parameters, for which
we need a procedure \texttt{genSymbols} (Fig. 7.2) and the following definition:

**Definition 7.7** (symbol generator, fresh). A \textit{symbol generator} is an infinite
sequence of alpha-numeric strings. The predicate \textit{fresh} takes a symbol generator
\(G\) and a set of terms \(X\). \textit{fresh}(\(G, X\)) is satisfied if no element of \(G\) is equal to
the name of a symbol occurring within a term in \(X\).
function reexpressMaster(T, R):
    input: T, a master term;
           R, a set of lassos over T.
    output: a master term.

1. Let g be a symbol that does not occur as a lambda parameter in T, and let G be a symbol generator satisfying fresh(G, {T, g}).

2. Execute makeAbstraction(T, R) with initial state G and let X be the result.

3. Execute reexpress(T, R, g) with initial state G and define a1...am, A1...Am, and B1...Bn by writing the result in the following form:

        master (\ a1 ... am -> Tuple A1 ... Am B1 ... Bn)

4. Return a term that is alpha-equivalent to the following, with all lambda parameters distinctly named:

        master (\ a1 ... am g -> Tuple A1 ... Am X B1 ... Bn)

Figure 7.3: The reexpressMaster algorithm.

When the master term is transformed, it is important to preserve long normal form (Defn. 4.1). During transformation we shall manipulate intermediate terms that are subterms of some long normal term, but are not necessarily themselves long normal. Such terms are in quasi long normal form:

Definition 7.8 (quasi long normal form). A term, of type α1...αm->R where R is a data type, is in quasi long normal form if it is in long normal form, or if it of the form:

        f T1 ... Tn

where:

- f is a variable whose type has arity n + m;
- each Tj is a term in long normal form;
- if f is a hole then n = m = 0.

The transformation algorithm itself is called reexpressMaster (Fig. 7.3). It takes a master term and an auSearch result, and returns a new, transformed master term. reexpressMaster invokes two sub-algorithms, makeAbstraction (Fig. 7.4) and reexpress (Fig. 7.5). makeAbstraction constructs a term that represents the abstraction. On the other hand, reexpress refactors the existing body of the master term so as to invoke the new abstraction by name. The symbol g generated at line 1 of reexpressMaster provides this name.
stateful function \texttt{makeAbstraction}(T, R):

\textbf{input}: \quad T, a term in long normal form;
\hspace{1cm} R, a set of lassos over T.

\textbf{output}: \quad a term in long normal form.

\textbf{state}: \quad a symbol generator.

1. Let \(\langle a, P \rangle\) be the element of \(R\) whose location occurs last lexicographically.

2. Let \(p_1 \ldots p_m\) be the elements of \(P\) arranged in forward lexicographical order.

3. For each \(i \in \{1 \ldots m\}\), let \(\alpha_i\) be the type of the subterm at \(p_i\) in \(T\).

4. Call \texttt{genSymbols}([\(\alpha_1 \ldots \alpha_m\)]) and let \([x_1 \ldots x_m]\) be the result.

5. Let \(S_0 = T\).

6. For each \(i \in \{1 \ldots m\}\):
   
   7. Let \([y_{i1} \ldots y_{ir_i}]\) = \texttt{lostVars}(T, R, 1, i).
   
   8. For each \(j \in \{1 \ldots r_i\}\), call \texttt{etaExpand}(y_{ij}) and let \(Y_{ij}\) be the result.
   
   9. Call \texttt{etaExpandReplace}(p_i, (x_{i1} Y_{i1} \ldots Y_{ir_i}), S_{i-1}) and let \(S_i\) be the result.

10. Let \(A\) be the subterm at \(a\) in \(S_m\).

11. Call \texttt{etaExpand}(A) and let \(A'\) be the result.

12. Return \((\setminus x_1 \ldots x_m \rightarrow A')\).

Figure 7.4: The \texttt{makeAbstraction} algorithm.
stateful function \textit{reexpress}(T, R, g):

input: \( T \), a term in long normal form;
\( R \), a set of lassos over \( T \);
\( g \), a symbol.

output: a term in long normal form.

state: a symbol generator.

1. Let \( \langle a_1, P_1 \rangle \ldots \langle a_n, P_n \rangle \) be the elements of \( R \) arranged in reverse lexicographical order of location.

2. Let \( T_0 = T \).

3. For each \( i \in \{1 \ldots n\} \):

4. Let \( p_{i1} \ldots p_{im} \) be the elements of \( P_i \) arranged in forward lexicographical order.

5. For each \( j \in \{1 \ldots m\} \):

6. Let \( M_{ij} \) be the subterm at \( p_{ij} \) in \( T_{i-1} \).

7. Call \textit{etaExpand}(M_{ij}) and let \( M'_{ij} \) be the result.

8. Let \( [y_{ij1} \ldots y_{ijr_j}] = \text{lostVars}(T, R, i, j) \).

9. Let \( M''_{ij} = (\forall y_{ij1} \ldots y_{ijr_j} \rightarrow M'_{ij}) \).

10. Call \textit{etaExpandReplace}(a_i, (g \ M''_{i1} \ldots M''_{im}), T_{i-1}) and let \( T_i \) be the result.

11. Return \( T_n \).

Figure 7.5: The \textit{reexpress} algorithm.
function lostVars(T, R, \(i', j'\)):

input:  
\(T\), a term in long normal form;  
\(R\), a set of lassos over \(T\);  
\(i'\), an integer;  
\(j'\), an integer.

output: a list of symbols.

1. Let \((a_1, P_1) \ldots (a_n, P_n)\) be the elements of \(R\) arranged in reverse lexicographical order of location.

2. For each \(i \in \{1 \ldots n\}\):

3. Let \(p_{i1} \ldots p_{im}\) be the elements of \(P_i\) in forward lexicographical order.

4. For each \(j \in \{1 \ldots m\}\):

5. Let \(L_{ij}\) be the set of all places \(q\) satisfying the following conditions:

   \(\hat{\overline{a}q}\) is a place in \(T\) and an ancestor of \(p_{ij}\);

   the subterm at \(\hat{\overline{a}q}\) in \(T\) is a lambda abstraction whose parameter binds a variable at some place \(r\) in \(T\) where \(r\) is an improper descendant of \(p_{ij}\).

6. For each \(j \in \{1 \ldots m\}\), let \(q_{j1} \ldots q_{jr_j}\) be the elements of \(\bigcup_{i=1}^{n} L_{ij}\) in forward lexicographical order.

7. For all \(i \in \{1 \ldots n\}, j \in \{1 \ldots m\}, k \in \{1 \ldots r_j\}\), let \(y_{ijk}\) equal the parameter of the lambda abstraction at \(\hat{\overline{a}q_{jk}}\) in \(T\).

8. Return \([y_{i'j'1} \ldots y_{i'j'r_j'}]\).

Figure 7.6: The lostVars algorithm.
stateful function \( \text{etaExpand}(T) \):

input: \( T \), a term in quasi long normal form.

output: a term in long normal form.

state: a symbol generator.

1. If \( T \) is a lambda abstraction then return \( T \).

2. Else if \( T \) is a variable or an application then:

3. Let \( \alpha_1 \rightarrow \ldots \rightarrow \alpha_m \rightarrow R \) be the type of \( T \) where \( R \) is a data type.

4. Call \( \text{genSymbols}([\alpha_1 \ldots \alpha_m]) \) and let \([x_1 \ldots x_m]\) be the result.

5. For each \( i \in \{1 \ldots m\} \), call \( \text{etaExpand}(x_i) \) and let \( X_i \) be the result.

6. Return \( (~(\lambda x_1 \ldots x_m \rightarrow T \ X_1 \ldots X_m) \).

stateful function \( \text{etaExpandReplace}(p,X,T) \):

input: \( p \), a place in \( T \);

\( X \), a term in quasi long normal form that is a variable or an application, of the same type as the subterm at \( p \) in \( T \);

\( T \), a term in long normal form.

output: a term in long normal form.

state: a symbol generator.

1. If the subterm at \( p \) in \( T \) is a variable or an application then return \( T\{p/X\} \).

2. Else if the subterm at \( p \) in \( T \) is a lambda abstraction then:

3. Call \( \text{etaExpand}(X) \) and let \( X' \) be the result.

4. Return \( T\{p/X'\} \).

Figure 7.7: The \( \text{etaExpand} \) and \( \text{etaExpandReplace} \) procedures.
Both \textit{makeAbstraction} and \textit{reexpress} invoke further sub-algorithms called \textit{lostvars} (Fig. 7.6), \textit{etaExpand}, and \textit{etaExpandReplace} (Fig. 7.7). The role of \textit{lostVars} is to detect something called ‘lost variables’. There are variables that occur in the master term outside the part that is to be abstracted out, but that are bound by lambda parameters within the part that is to be abstracted out. Performing an abstraction transformation naively would leave these variables stranded without lambda parameters to bind them (hence ‘lost’) and would therefore invalidate the term as a whole. To solve this problem, extra lambda parameters have to be added into the re-expressed master term at line 9 of \textit{reexpress}, and corresponding extra variables have to be added into the abstraction at line 9 of \textit{makeAbstraction}. This connects up the lost variables with the right lambda parameters within the body of the abstraction that supply them with their meanings.

The role of the procedures \textit{etaExpand} and \textit{etaExpandReplace} is to perform operations known as \textit{eta expansions} which are necessary in order to ensure that the master term is still in long normal form by the end of the \textit{reexpressMaster} transformation. Eta expansion is the reverse of the standard deduction rule of lambda calculus called \textit{eta-reduction} [Stoy, 1981].

At last, connecting together abstraction discovery and abstraction construction, we obtain the following full specification for KANDINSKY’s abstraction invention process, \textit{absInv}. \textit{absInv} returns a set of candidate transformations of the input master term, each of which corresponds to a different result of \textit{auSearch}:

\textbf{Definition 7.9 (absInv).} The function \textit{absInv} takes a master term and returns a set of master terms:

$$\text{absInv}(T) = \{ R \triangleleft \text{findAbstractions}(T) \bullet \text{reexpressMaster}(T,R) \}$$

We must supply a score function for \textit{auSearch} (Defn. 6.35) in order to allow heuristic guidance of the search for candidate abstractions. The following score function is suitable. It calculates the syntactic compression that would be produced if an abstraction were to be made from the given commonality, ignoring eta-expansion and lost variables. This is a reasonable first approximation to the change in log probability under the cumulative learning prior. Hence, candidate abstraction transformations that will yield larger increases in the prior probability of the master term will tend to be favoured during the guided search:

\textbf{Definition 7.10 (score).} Let \( S \) be a non-empty set of shoots. Let \( \langle a, b, P \rangle \) be an element of \( S \). Let \( c \) be the number of places contained in the lasso \( \langle b, P \rangle \). Let \( m = |P| \). Let \( n = |S| \). Then:

$$\text{score}(S) = (n - 1)c - (n + 2)m - n$$
Finally, the following algorithm selects the best abstraction out of the generated candidates according to the true change in prior probability:

**Definition 7.11** (*absInvBest*). The function *absInvBest* takes a master term and returns a master term:

\[
\text{absInvBest}(T) = \arg\max_{T' \in S} \text{clPrior}(T')
\]

where \( S = \{T\} \cup \text{absInv}(T) \)

### 7.1.1 Worked example

Let us look at a concrete example of KANDINSKY’s abstraction invention algorithm in use. Consider the following master term \( T \):

\[
\text{master (Tuple}
\begin{align*}
& (\lambda \ a \ b \ c \to \text{plus}\ (\text{pow}\ a\ \text{two}) \\
& \quad (\text{plus}\ (\text{pow}\ b\ \text{two})\ (\text{pow}\ c\ \text{two}))) \\
& (\lambda \ a \ b \ c \to \text{pow}\ (\text{plus}\ a\ (\text{plus}\ b\ c))\ \text{two})
\end{align*}
\]

If we take \text{plus} and \text{pow} to be numerical addition and ‘to the power of’ operators respectively, we see that \( T \) contains implementations of two functions, one which calculates the sum of the squares of three numbers, and another which calculates the square of their sum.

In order to perform abstraction invention on \( T \), we must first specify an appropriate prior probability distribution. Let this be the cumulative learning prior defined by \text{clPriorProc}(\text{tys}, \Gamma) as follows, where the data type \text{Int} may be taken to represent the integers:

\[
\text{tys} = [\text{Int} \to \text{Int} \to \text{Int} \to \text{Int}, \\
\text{Int} \to \text{Int} \to \text{Int} \to \text{Int}]
\]

\[
\Gamma = \{\text{plus}:(\text{Int} \to \text{Int} \to \text{Int}), \\
\text{times}:(\text{Int} \to \text{Int} \to \text{Int}), \\
\text{pow}:(\text{Int} \to \text{Int} \to \text{Int}), \\
\text{zero}:\text{Int}, \\
\text{one}:\text{Int}, \\
\text{two}:\text{Int}\}
\]

Furthermore:

\[
\text{npPriorHyperParam}_1 = 8 \\
\text{npPriorHyperParam}_2 = 16
\]
Also, as we did in the example of Sect. 4.3.1, let us assume for simplicity that \(numAbstractionsDist\) and \(abstractionTypeDist\) have been chosen such that the following holds for any relevant type \(\tau\):

\[
P(m = 0) = P(m = 1 \text{ and } A_1 \text{ has type } \tau) = k
\]

where \(m\) is the number of abstractions in a master term and \(A_1\) is its first abstraction; \(k\) is an arbitrary constant. Under these conditions, the prior probability of \(T\) is equal to \(k \times 1.11 \times 10^{-18}\).

We must specify some reasonable values for the \(auSearch\) beam size parameters:

\[
\begin{align*}
\sigmaParam &= 1 \\
\tauParam &= 50
\end{align*}
\]

\(absInvBest(T)\) is then equal to the following:

\[
\text{master } (\lambda f \to \text{Tuple} \\
\quad (\lambda a \to \text{pow } a \text{ two}) \\
\quad (\lambda a b c \to \text{plus } (f a) (\text{plus } (f b) (f c))) \\
\quad (\lambda a b c \to f (\text{plus } a (\text{plus } b c))))
\]

Observe that \(T\) and \(absInvBest(T)\) are semantically equivalent. The prior probability of \(absInvBest(T)\) is equal to \(k \times 2.77 \times 10^{-17}\), about 25 times higher than that of \(T\). This increase in probability has been brought about by the introduction of an abstraction bound to the symbol \(f\), identifiable as the ‘square’ function. This result was tested using a prototype implementation of KANDINSKY.

## 7.2 Proof of Correctness of \(absInv\)

I now prove that KANDINSKY’s abstraction invention transformation operator preserves the semantic denotation of the master term with respect to any toplevel environment that includes the declarations of \texttt{master} and \texttt{Tuple} as given in Sect. 4.3. For the main result see Theorem 7.31.

This proof takes the form of a deduction process in which one starts with the output of \(absInv\) and transforms it using deduction rules including beta- and eta-reduction so as to obtain the input. Hence, \(absInv\) is shown to be an inverse deduction process.

### 7.2.1 Preliminaries

The propositions in this subsection are stated without proof. They are straightforward consequences of the definition of the lambda calculus language.
Definition 7.12 (params, vars, freeVars). For a term $T$ the following functions are defined:

- $\text{params}(T)$ is the set of lambda parameters that occur in $T$;
- $\text{vars}(T)$ is the set of variables that occur in $T$;
- $\text{freeVars}(T)$ is the set of variables that occur free in $T$.

Definition 7.13 (subtermAt). For a term $T$ and a place $p$ in $T$, we define $\text{subtermAt}(p, T)$ to be the subterm at place $p$ in $T$.

Proposition 7.14. For terms $A, B_1 \ldots B_n$ and places $p, q_1 \ldots q_n$ it holds that:

$$\text{subtermAt}(p, A)\{q_1/B_1 \ldots q_n/B_n\} = \text{subtermAt}(p, A\{(p + + q_1)/B_1 \ldots (p + + q_n)/B_n\})$$

The following 'simple' substitution operator is different from the substitution operator that is conventionally defined for lambda calculus [Stoy, 1981]. The simple substitution is a straight syntactic substitution, allowing variable capture to occur:

Definition 7.15 (simple substitution). A simple substitution on terms, written $[x_1/T_1, \ldots, x_n/T_n]$, represents a mapping from distinct symbols $x_i$ to terms $T_i$. If $A$ is a term and $\sigma$ is a simple substitution, then $A\sigma$ is the term obtained from $A$ by simultaneously replacing each occurrence of the variable $x_i$ with $T_i$, for all $i \in \{1 \ldots n\}$.

Proposition 7.16. Let $T$ be a term, let $p_1 \ldots p_n$ be places in $T$, let $x_1 \ldots x_n$ be symbols, and let $M_1 \ldots M_n$ be terms, such that for each $i$ the types of $x_i$, $M_i$, and $\text{subtermAt}(p_i, T)$ are equal. Given the precondition that $x_i \notin \text{vars}(T)$ for all $i$, it holds that:

$$T[x_1=M_1 \ldots x_n=M_n] = T[p_1/M_1 \ldots p_n/M_n]$$

The following two definitions are respectively lambda calculus analogues of logical equivalence relative to background knowledge, and logical equivalence:

Definition 7.17 (relative $\lambda$-equivalence). For terms $A, B$ and an environment $E$, we write $A \sim_E B$ if:

$$\text{eval}(A, E) = \text{eval}(B, E)$$

Definition 7.18 ($\lambda$-equivalence). For terms $A, B$, we write $A \sim B$ if for all environments $E$ it holds that $A \sim_E B$.

Proposition 7.19. Let $T$ be a term, let $p$ be a place in $T$, and let $X$ be a term of the same type as $\text{subtermAt}(p, T)$. It holds that:

$$X \sim \text{subtermAt}(p, T) \implies T[p/X] \sim T$$
**Definition 7.20** *(binding)*. The predicate *binding* takes two places and a term. *binding*(p,q,T) is satisfied if all of the following conditions hold:

- p and q are both places in T;
- p is an ancestor of q;
- *subtermAt*(p,T) is a lambda abstraction with some parameter x;
- *subtermAt*(q,T) is a variable equal to x;
- there does not exist any place p’ that is both a descendant of p and an ancestor of q such that *subtermAt*(p’,T) is a lambda abstraction with parameter x.

**Proposition 7.21.** Let T be a term, let p be a place in T, let X be a term of the same type as *subtermAt*(p,T), and let E be an environment. Assert the following precondition: there do not exist places a,b where a is an ancestor of p and b is an improper descendant of p such that either of the following hold:

\[ binding(a,b,T) \]
\[ binding(a,b,T\{p/X\}) \]

It then holds that:

\[ X \sim_E \text{subtermAt}(p,T) \implies T\{p/X\} \sim_E T \]

**Proposition 7.22.** If terms A,B are alpha-equivalent then A ∼ B.

The following two propositions respectively describe the beta- and eta-reduction rules of lambda calculus:

**Proposition 7.23** *(beta-reduction)*. Let x₁...xₙ be distinct symbols and let M and N₁...Nₙ be terms such that each xᵢ has the same type as Nᵢ. Given the precondition that xᵢ ∉ params(M) and freeVars(Nᵢ) is disjoint from params(M), for all i, it holds that:

\[ (\\backslash \ x₁ \ldots xₙ \rightarrow \ M) \ N₁ \ldots Nₙ \sim M[x₁/N₁ \ldots xₙ/Nₙ] \]

**Proposition 7.24** *(eta-reduction)*. Let T be a term of type α₁ → ... → αₘ → R where R is a data type. Let x₁...xₘ be distinct symbols where the type of each xᵢ is equal to αᵢ. Given the precondition that xᵢ ∉ vars(T) for all i, it holds that:

\[ (\\backslash \ x₁ \ldots xₘ \rightarrow T \ x₁ \ldots xₘ) \sim T \]
7.2.2 Eta-expansion

**Lemma 7.25.** Let $T$ be a term in quasi long normal form, let $G$ be a symbol generator satisfying $\text{fresh}(G, \{T\})$, and let $T'$ be the result of executing $\text{etaExpand}(T)$ with initial state $G$. It follows that $T' \sim T$.

**Proof.** We proceed by structural induction on the type of $T$. If $T$ is a lambda abstraction then the proof is trivial. Otherwise, define $x_1 \ldots x_m$ and $X_1 \ldots X_m$ as in the $\text{etaExpand}$ algorithm (Fig. 7.7) and proceed as follows:

$$T' \sim (\lambda x_1 \ldots x_m \rightarrow T X_1 \ldots X_m)$$
$$\sim (\lambda x_1 \ldots x_m \rightarrow T x_1 \ldots x_m)$$
(by the inductive hypothesis and 7.19)
$$\sim T$$
(by 7.24)

**Lemma 7.26.** Let $p, X, T$ be valid input to $\text{etaExpandReplace}$ as in Fig. 7.7, let $G$ be a symbol generator satisfying $\text{fresh}(G, \{X\})$, and let $T'$ be the result of executing $\text{etaExpandReplace}(p, X, T)$ with initial state $G$. It follows that $T' \sim T\{p/X\}$.

**Proof.** If $T$ is a variable or an application then the proof is trivial. Otherwise, define $X'$ as in the $\text{etaExpandReplace}$ algorithm (Fig. 7.7) and proceed as follows:

$$T' \sim T\{p/X'\}$$
$$\sim T\{p/X\}$$
(by 7.25 and 7.19)

7.2.3 Refactoring the master term

**Definition 7.27 (abstractionCondition).** Let $T$ be a term and let $R$ be a set of lassos over $T$. The predicate $\text{abstractionCondition}(T, R)$ is satisfied when the following three conditions hold:

1. Let $\{L_1 \ldots L_n\} = R$, and for each $i \in \{1 \ldots n\}$ let:

   $$\langle a_i, \{a_i + q_{i1}, \ldots, a_i + q_{im_i}\}\rangle = L_i$$

   It holds that all terms as follows are $\alpha$-equivalent for $i \in \{1 \ldots n\}$:

   $$\text{subtermAt}(a_i, T)\{q_{i1}/# \ldots q_{im_i}/#\}$$

2. $R$ is a non-overlapping set of lassos.
3. For all places \( p, q \) satisfying \( \text{binding}(p, q, T) \) and for all \( L \in R \), it holds that:
\[
L \text{ contains } q \implies L \text{ contains } p
\]

**Lemma 7.28.** Let \( T \) be a master term. Let \( R \), a set of lassos, be an element of \( \text{findAbstractions}(T) \). It follows that abstractionsCondition\((T, R)\) is satisfied.

**Proof.** Let \( \{L_1 \ldots L_n\} = R \). Define \( p_1 \ldots p_N, S_1 \ldots S_N, S'_1 \ldots S'_N \) and \( A_1 \ldots A_N \) as in the \( \text{findAbstractions} \) algorithm (Fig. 7.1). By Theorem 6.28 and Theorem 6.29 it holds that:
\[
R = \text{mnos}(\text{match}(s, \{A_1 \ldots A_N\}))
\]
for some first-order term \( s \). The second condition in Defn. 7.27 then follows by Theorem 6.25. By Theorem 6.25 and Defn. 6.26 it holds for each \( i \in \{1 \ldots n\} \) that \( L_i \) is the lasso of \( s \) on \( \langle p_k, \text{encodeTerm}(S'_{k_i}) \rangle \) for some \( k_{i} \in \{1 \ldots N\} \) where \( s \) is an anti-instance of \( \text{encodeTerm}(S'_{k_i}) \). By Defn. 6.22 it also holds that each \( L_i \) is equal to:
\[
\langle p_k, \{p_k ++ q_1, \ldots, p_k ++ q_m\} \rangle
\]
where \( \{q_1 \ldots q_m\} = \text{varPlaces}(s) \). Hence, if \( X_1 \ldots X_m \) are arbitrary first-order variables then all first-order terms as follows are equal for \( i \in \{1 \ldots n\} \):
\[
\text{encodeTerm}(S'_{k_i})\{q_1/X_1 \ldots q_m/X_m\}
\]
By Defn. 7.4 it then holds that all lambda terms as follows are alpha-equivalent for \( i \in \{1 \ldots n\} \):
\[
S'_{k_i}\{q_1/# \ldots q_m/#\}
\]
Finally, the first and third conditions in Defn. 7.27 follow by the definition of \( \text{findAbstractions} \).

**Lemma 7.29.** Let \( T, R, g \) be valid input to \( \text{reexpress} \) as in Fig. 7.5 such that the lambda parameters of \( T \) are all distinctly named, \( g \notin \text{params}(T) \), and abstractionsCondition\((T, R)\) holds. Let \( G \) be a symbol generator satisfying fresh\((G, \{T, g\})\), let \( E \) be the result of executing \( \text{makeAbstraction}(T, R) \) with initial state \( G \), let \( E \) be an environment such that \( g \sim_E X \), let \( G' \) be another symbol generator satisfying fresh\((G', \{T\})\), and define \( T_0 \ldots T_n \) as in the \( \text{reexpress} \) algorithm (Fig. 7.5) during the execution of \( \text{reexpress}(T, R, g) \) from initial state \( G' \). It then follows for all \( i \in \{0 \ldots n\} \) that:
\[
T_i \sim_E T
\]

**Proof.** For \( i \in \{1 \ldots n\} \), define \( a_i, p_{1i} \ldots p_{mi}, M_{1i} \ldots M_{mi}, [y_1, \ldots y_{1r}], \ldots [y_{im}, \ldots y_{imr}], \) and \( M''_{1i} \ldots M''_{im} \) as in Fig. 7.5 during the execution of \( \text{reexpress}(T, R, g) \) from initial state \( G' \). Define \( a, p_1 \ldots p_m, x_1 \ldots x_m, \)
$S_m$, $A$, and $A'$ as in Fig. 7.4 during the execution of $makeAbstraction(T,R)$ from initial state $G$. By the definition of $etaExpandReplace$ in Fig. 7.7, we can write $S_m$ in the form:

$$T\{p_1/X_1 \ldots p_m/X_m\}$$

for some terms $X_1 \ldots X_m$. By Lemma 7.25 and Proposition 7.19 it holds that $X_j \sim (x_j y_{1j1} \ldots y_{1jr_j})$ for all $j \in \{1 \ldots m\}$. Also, by Defn. 7.6 we can write each $p_j$ as $(a ++ q_j)$ for some place $q_j$ and each $p_{ij}$ as $(a_i ++ q_{ij})$ for some place $q_{ij}$. By induction on $i$:

1. $i = 0$:

$$LHS \sim_E RHS$$

(by line 2 of $reexpress$)

2. $i \in \{1 \ldots n\}$. First let us prove the following preliminary fact:

$$A' \sim A$$

(by 7.25)

$$\sim subtermAt(a,T\{(a ++ q_1)/X_1 \ldots (a ++ q_m)/X_m\})$$

(by 7.14)

$$\sim subtermAt(a,T)\{q_1/X_1 \ldots q_m/X_m\}$$

$$\sim subtermAt(a,T)\{q_1/(x_1 y_{111} \ldots y_{1r_1})\}$$

$$\ldots$$

$$q_m/(x_m y_{m1} \ldots y_{mr_m})\}$$

(by 7.19)

$$\sim subtermAt(a_i,T)\{q_{i1}/(x_1 y_{i11} \ldots y_{ir_1})\}$$

$$\ldots$$

$$q_{im}/(x_m y_{i1} \ldots y_{imr_m})\}$$

(by the first condition in 7.27)

$$\sim subtermAt(a_i,T_{i-1})\{q_{i1}/(x_1 y_{i11} \ldots y_{i1r_1})\}$$

$$\ldots$$

$$q_{im}/(x_m y_{i1} \ldots y_{imr_m})\}$$

(by the second condition in 7.27)
Then proceed as follows:

\[ \text{LHS} \sim_E \{ a_i / (g \ M_i'^n \ldots M_{im}^n) \} \]

\[ \sim_E \{ a_i / (\pre M_i \ldots y_i1r \to M_i) \} \]

\[ \ldots \]

\[ \{ \pre y_{im1} \ldots y_{imr_m} \to M_{im} \} \}

(by 7.25 and 7.19)

\[ \sim_E \{ a_i / \langle X \ (\pre y_{i11} \ldots y_{i1r_1} \to M_i) \} \]

\[ \ldots \]

\[ \{ \pre y_{im1} \ldots y_{imr_m} \to M_{im} \} \}

(by 7.21 and the third condition in 7.27)

\[ \sim_E \{ a_i / \text{subtermAt}(a_i, T_{i-1}) \} \]

\[ \{ q_{i1} / (x_1 y_{i11} \ldots y_{i1r_1}) \} \]

\[ \ldots \]

\[ q_{im} / (x_m y_{im1} \ldots y_{imr_m}) \}

\[ [x_1 / \langle y_{i11} \ldots y_{i1r_1} \to M_i \} \]

\[ \ldots \]

\[ x_m / \langle y_{im1} \ldots y_{imr_m} \to M_{im} ] \}

(by the preliminary fact, 7.19, and 7.23)

\[ \sim_E \{ a_i / \text{subtermAt}(a_i, T_{i-1}) \} \]

\[ \{ q_{i1} / \langle \pre y_{i11} \ldots y_{i1r_1} \to M_i \ \rangle y_{i11} \ldots y_{i1r_1} \} \]

\[ \ldots \]

\[ q_{im} / \langle \pre y_{im1} \ldots y_{imr_m} \to M_{im} \ y_{im1} \ldots y_{imr_m} \} \}

(by 7.16)

\[ \sim_E \{ a_i / \text{subtermAt}(a_i, T_{i-1})\{ q_{i1}/M_i \ldots q_{im}/M_{im} \} \}

(by 7.23 and 7.19)

\[ \sim_E \{ p_{i1}/M_{i1} \ldots p_{im}/M_{im} \}

\[ \sim_E \{ T_{i-1} \}

\[ \sim_E \{ \text{RHS} \} \]

(by the inductive hypothesis)

\[ \square \]

**Lemma 7.30.** Let \( E \) be a toplevel environment, and let \( X, Y, X', \) and \( Y' \) be
terms defined as follows:

\[ X = (\ \lambda \ t \to \text{caseTuple} \ t \ (\ \lambda \ a_1 \ldots a_m \ b_1 \ldots b_n \to \text{Tuple} \ b_1 \ldots b_n)) \]

\[ Y = (\ \lambda \ a_1 \ldots a_m \ b_1 \ldots b_n \to \text{Tuple} \ A_1 \ldots A_m \ B_1 \ldots B_n) \]

\[ X' = (\ \lambda \ t \to \text{caseTuple} \ t \ (\ \lambda \ a_1 \ldots a_m \ g \ b_1 \ldots b_n \to \text{Tuple} \ b_1 \ldots b_n)) \]

\[ Y' = (\ \lambda \ a_1 \ldots a_m \ g \ b_1 \ldots b_n \to \text{Tuple} \ A_1 \ldots A_m \ G \ B_1 \ldots B_n) \]

It follows that:

\[ \text{eval}(X \ (\text{fixTuple} \ Y), E) = \text{eval}(X' \ (\text{fixTuple} \ Y'), E) \]

Proof. The proof follows by the definitions of the \text{Tuple} data type and the \text{fixTuple} value as given in Sect. 4.3. \hfill \Box

**Theorem 7.31.** Let \( T \) be a master term, let \( T' \) be an element of \( \text{absInv}(T) \), and let \( E \) be a top-level environment derived from a set of declarations that includes those for \text{master} and \text{Tuple} as given in Sect. 4.3. It follows that:

\[ T' \sim_E T \]

Proof. By Defn. 7.9 it holds that:

\[ T' = \text{reexpressMaster}(T, R) \]

for some \( R \in \text{findAbstractions}(T) \). Let us then define \( g, X, a_1 \ldots a_m, A_1 \ldots A_m, \) and \( B_1 \ldots B_n \) as in Fig. 7.3. For each \( i \in \{1 \ldots m\} \), let \( A'_i = A_i[g/X] \). Let \( b_1 \ldots b_n \) be symbols of appropriate type whose names are distinct from each other and from those all other symbols occurring within this proof. Now derive
the following preliminary fact:

\[
\begin{align*}
(\forall g \to master (\forall a_1 \ldots a_m \to Tuple A_1 \ldots A_m B_1 \ldots B_n)) \ X \\
\sim_E (\forall a_1 \ldots a_m \to Tuple A_1 \ldots A_m B_1 \ldots B_n)[g/X] \\
& \quad \text{(by 7.23)} \\
\sim_E master (\forall a_1 \ldots a_m \to Tuple A'_1 \ldots A'_m B'_1 \ldots B'_n) \\
\sim_E caseTuple (fixTuple (\forall a_1 \ldots a_m b_1 \ldots b_n \\
\quad \to Tuple A'_1 \ldots A'_m B'_1 \ldots B'_n)) \\
& \quad \text{(by the value declaration for master and 7.23)} \\
\sim_E caseTuple (fixTuple (\forall a_1 \ldots a_m b_1 \ldots b_n \\
\quad \to Tuple A'_1 \ldots A'_m X B'_1 \ldots B'_n)) \\
& \quad \text{(by 7.30)} \\
\sim_E caseTuple (fixTuple (\forall a_1 \ldots a_m b_1 \ldots b_n \\
\quad \to Tuple A_1 \ldots A_m X B_1 \ldots B_n)) \\
& \quad \text{(by the value declaration for fixTuple)} \\
\sim_E master (\forall a_1 \ldots a_m g \to Tuple A_1 \ldots A_m X B_1 \ldots B_n) \\
& \quad \text{(by the value declaration for master and 7.23)} \\
\sim_E T' \\
\end{align*}
\]

Then proceed as follows:

\[
\begin{align*}
\text{eval}(T', E) \\
& = \text{eval}(\{ (\forall g \to master (\forall a_1 \ldots a_m \\
\quad \to Tuple A_1 \ldots A_m B_1 \ldots B_n)) X \}, E) \\
& \quad \text{(by the preliminary fact)} \\
& = \text{eval}(master (\forall a_1 \ldots a_m \to Tuple A_1 \ldots A_m B_1 \ldots B_n), \\
\quad E\{g/eval(X, E)\}) \\
& \quad \text{(by 3.26 and 3.27)} \\
& = \text{eval}(T, E\{g/eval(X, E)\}) \\
& \quad \text{(by 7.29 and 7.28)} \\
& = \text{eval}(T, E) \\
\end{align*}
\]

\[\square\]
Chapter 8

Discussion

8.1 A ‘Proof-of-Concept’ Demonstration of RUFINSKY

In order to show how all the ideas of the previous chapters fit together, let us look at a worked example of how RUFINSKY behaves in a simple cumulative learning scenario. The chosen scenario demonstrates cumulative learning over a sequence of three program synthesis problems, and it also demonstrates invention of the map abstraction which was mentioned in Sect. 1.2.

In this example we work within the setting for learning of Sect. 3.4 and we follow the cumulative learning scheme of Sect. 4.3. First, let us introduce the following data type declarations for natural numbers and lists, as well as value declarations for arithmetic predecessor, addition, and multiplication operators:

\[
\begin{align*}
\text{data Nat} &= \text{Zero} \mid \text{Succ Nat} \\
\text{data List} &= \text{Nil} \mid \text{Cons Nat List} \\
\text{pred} &= \backslash a \rightarrow \text{caseNat } a \ \text{error } (\backslash a2 \rightarrow a2) \\
\text{plus} &= \backslash a \ b \rightarrow \text{caseNat } a \ b (\backslash a2 \rightarrow \text{Succ } (\text{plus } a2 \ b)) \\
\text{times} &= \backslash a \ b \rightarrow \text{caseNat } a \ \text{Zero } (\backslash a2 \rightarrow \text{plus } b \ (\text{times } a2 \ b))
\end{align*}
\]

The background knowledge \( B \) shall be the toplevel environment (Defn. 3.29) defined with respect to the above declarations plus the declarations for \( \text{Tuple}_{\alpha_1 \ldots \alpha_m} \), \( \text{fixTuple} \), and \( \text{master} \) given in Sect. 4.3. The target datum \( D \) shall be that shown in Fig. 8.1. It consists of input-output examples for three sub-problems, aggregated together into ‘monolithic’ form using a \( \text{Tuple} \) constructor. The type of \( D \) is equal to:

\[
\text{Tuple}_{\alpha_a} \text{ where } \alpha = \text{List } \rightarrow \text{List}
\]

The hypothesis space for this cumulative learning problem consists of master
$D = \text{Tuple incElems decElems cubeElems}$

where $\text{incElems}$, $\text{decElems}$, and $\text{cubeElems}$ are the least values under $\sqsubseteq$ such that the following constraints hold:

\[
\begin{align*}
\text{incElems}([], []) &= [] \\
\text{incElems}([4]) &= [5] \\
\text{incElems}([8, 1, 4, 1, 3]) &= [9, 2, 5, 2, 4] \\
\text{decElems}([], []) &= [] \\
\text{decElems}([4]) &= [3] \\
\text{decElems}([8, 1, 4, 1, 3]) &= [7, 0, 3, 0, 2] \\
\text{cubeElems}([], []) &= [] \\
\text{cubeElems}([2]) &= [8] \\
\text{cubeElems}([4, 2, 3, 0, 2]) &= [64, 8, 27, 0, 8]
\end{align*}
\]

Figure 8.1: Target datum in the example cumulative learning scenario. A shorthand notation for values of type $\text{List}$ has been used for convenience: $\[0, 2\]$ represents the value $\text{Cons Zero (Cons (Succ (Succ Zero)) Nil)}$, etc. The monolithic target datum $D$ contains input-output specifications for three functions: $\text{incElems}$ increments the elements of a list, $\text{decElems}$ decrements the elements of a list, and $\text{cubeElems}$ raises the elements of a list to the third power.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Succ</td>
<td>Nat $\rightarrow$ Nat</td>
</tr>
<tr>
<td>pred</td>
<td>Nat $\rightarrow$ Nat</td>
</tr>
<tr>
<td>plus</td>
<td>Nat $\rightarrow$ Nat $\rightarrow$ Nat</td>
</tr>
<tr>
<td>times</td>
<td>Nat $\rightarrow$ Nat $\rightarrow$ Nat</td>
</tr>
<tr>
<td>Nil</td>
<td>List</td>
</tr>
<tr>
<td>Cons</td>
<td>Nat $\rightarrow$ List $\rightarrow$ List</td>
</tr>
<tr>
<td>caseList</td>
<td>List $\rightarrow$ List $\rightarrow$ (Nat $\rightarrow$ List $\rightarrow$ List) $\rightarrow$ List</td>
</tr>
<tr>
<td>fix</td>
<td>((List $\rightarrow$ List) $\rightarrow$ List $\rightarrow$ List) $\rightarrow$ List $\rightarrow$ List</td>
</tr>
</tbody>
</table>

Figure 8.2: Specification for the ‘pool’ $\Gamma$ of symbols in the example cumulative learning scenario. It consists of some primitive arithmetic operators (top group), the constructors and destructor for the $\text{List}$ datatype (middle group), and the fixed point operator for recursion (bottom group).
terms \( T \) of the following form:

\[
T = \text{master} (\land a_1 \ldots a_m \rightarrow \\
\text{Tuple} \ A_1 \ldots A_m \ \text{IncElems DecElems CubeElems})
\]

where \( \text{IncElems} \), \( \text{DecElems} \), and \( \text{CubeElems} \) are hypotheses for the three sub-problems, and \( A_1 \ldots A_m \) are abstractions with \( m \geq 0 \). The prior probability distribution over master terms \( T \) shall be the cumulative learning prior (Sect. 4.3) as defined by \( \text{clPriorProc}(\text{tys,}\Gamma) \) for \( \text{tys} \) equal to:

\[
[\alpha,\alpha,\alpha] \text{ where } \alpha = \text{List} \rightarrow \text{List}
\]

and for \( \Gamma \) equal to the set of symbols whose names and types are given in Fig. 8.2. As in the example of Sect. 4.3.1, let \( \text{clPrior} \) be the distribution form of \( \text{clPriorProc} \):

\[
\text{clPrior}(T,\Gamma) = P(\text{clPriorProc}(\text{tys},\Gamma) \text{ returns } T)
\]

We also again assume for convenience that \( \text{numAbstractionsDist} \) and \( \text{abstractionTypeDist} \) (Defn. 4.37) have been chosen such that the following holds for any relevant type \( \tau \):

\[
P(m = 0) = P(m = 1 \text{ and } A_1 \text{ has type } \tau)
\]

where \( m \) is the number of abstractions in \( T \) and \( A_1 \) is the first abstraction.

Finally, throughout this example let us assume the following values for RUFINSKY’s global parameters:

\[
\text{npPriorHyperParam}_1 = 8 \quad \text{(Defn. 4.29)}
\]
\[
\text{npPriorHyperParam}_2 = 16 \quad \text{(Defn. 4.29)}
\]
\[
\text{evalLimit} = 50 \quad \text{(Defn. 5.12)}
\]
\[
\text{conWeight}(x) = \frac{1}{2} \text{ for } x \in \{\text{Zero, Succ, Nil, Cons}\} \quad \text{(Defn. 5.24)}
\]
\[
\text{stepCountOffset} = 1 \quad \text{(Defn. 5.27)}
\]
\[
\text{sigmaParam} = 1 \quad \text{(Defn. 7.5)}
\]
\[
\text{tauParam} = 50 \quad \text{(Defn. 7.5)}
\]

The policy by which we shall run RUFINSKY in this scenario is as follows. We apply RUFUS to each sub-problem in turn, using the \( \text{runRufus} \) procedure of Defn. 5.29. In between successive invocations of RUFUS, we invoke KANDINSKY by repeatedly applying the \( \text{absInvBest} \) procedure of Defn. 7.11 on the monolithic hypothesis for as many times as continue to produce an increase in its prior probability. Note that in order to apply RUFUS to a sub-problem, a conditional prior probability distribution for that sub-problem will
be constructed using the technique described in Sect. 4.3. We shall see that an appropriate ‘local’ target datum and background knowledge are also needed within the context of a sub-problem. When a sub-problem is solved, its solution may be substituted back into the monolithic hypothesis. Note that KANDINSKY always gets invoked on the monolithic hypothesis as a whole, so it uses the master prior given by \( P(T) = clPrior(T, \Gamma) \).

We can now walk through the action of RUFINSKY step-by-step. RUFINSKY starts with an initial ‘empty’ monolithic hypothesis \( T_0 \) (Fig. 8.3). \( T_0 \) contains holes as placeholder hypotheses for each of the three sub-problems, and the number of abstractions is set to zero.

In the first step, RUFUS is applied to the first sub-problem, for which the target datum is \( incElems \) as in Fig. 8.1. The hypothesis space for this sub-problem is equal to \( L \) of Defn. 5.3 defined with respect to the target type \( \text{List} \rightarrow \text{List} \) (the type of \( incElems \)) and the set of symbols \( \Gamma \). The prior over hypotheses \( IncElems \) for the sub-problem is given by:

\[
P(IncElems) = \frac{clPrior(master (Tuple IncElems # #), \Gamma)}{clPrior(master (Tuple # # #), \Gamma)}
\]

RUFUS is invoked using \( \text{runRufus}(B, incElems) \). The output of RUFUS under these conditions is \( \langle \text{“goal”}, IncElems \rangle \) where \( IncElems \) is given in Fig. 8.3. The hypothesis \( IncElems \) is substituted in at the appropriate place in the master term \( T_0 \) to produce a new master term \( T_1 \) (Fig. 8.3).

Next, KANDINSKY is invoked on the master term:

\[
absInvBest(T_1) = T_1
\]

We see that KANDINSKY fails to find any opportunity for abstraction in \( T_1 \) that would result in an increase in its prior probability, so it leaves \( T_1 \) untransformed.

RUFUS is now applied to the second sub-problem. The target datum is \( decElems \) (Fig.8.1), and the hypothesis space \( L \) is the same as it was for the first sub-problem. The prior over hypotheses \( DecElems \) is given by:

\[
P(DecElems|IncElems) = \frac{clPrior(master (Tuple IncElems DecElems #), \Gamma)}{clPrior(master (Tuple IncElems # #), \Gamma)}
\]

RUFUS is invoked using \( \text{runRufus}(B, decElems) \) which returns \( \langle \text{“goal”}, DecElems \rangle \) where \( DecElems \) is given in Fig. 8.3. \( DecElems \) is then substituted into the master term \( T_1 \) to produce \( T_2 \) (Fig. 8.3).

KANDINSKY is invoked on the master term once again:

\[
absInvBest(T_2) = T_3
\]

This time we see that KANDINSKY succeeds in finding a useful opportunity for abstraction. The syntactic similarity between the two hypotheses \( IncElems \) and \( DecElems \) allows KANDINSKY to abstract out a higher-order function.
\[ T_0 = \text{master (Tuple \# \# \#)} \]
\[ \downarrow \text{refinement (5.0 s)} \]

\[ T_1 = \text{master (Tuple IncElems \# \#)} \]
where \( \text{IncElems} = \lambda a1 \rightarrow \text{fix} (\lambda a2 \ a3 \rightarrow \text{caseList} a3 a3 \ (\lambda a4 \ a5 \rightarrow \text{Cons} (\text{Succ} a4) (a2 a5))) \ a1 \)
\[ \downarrow \text{refinement (6.1 s)} \]

\[ T_2 = \text{master (Tuple IncElems DecElems \#)} \]
where \( \text{IncElems} = \lambda a1 \rightarrow \text{fix} (\lambda a2 \ a3 \rightarrow \text{caseList} a3 a3 \ (\lambda a4 \ a5 \rightarrow \text{Cons} (\text{Succ} a4) (a2 a5))) \ a1 \)
where \( \text{DecElems} = \lambda a1 \rightarrow \text{fix} (\lambda a2 \ a3 \rightarrow \text{caseList} a3 a3 \ (\lambda a4 \ a5 \rightarrow \text{Cons} (\text{pred} a4) (a2 a5))) \ a1 \)
\[ \downarrow \text{abstraction invention (0.01 s)} \]

\[ T_3 = \text{master (\ map \rightarrow Tuple Map IncElems' DecElems' \#)} \]
where \( \text{Map} = \lambda a1 \ a2 \rightarrow \text{fix} (\lambda a3 \ a4 \rightarrow \text{caseList} a4 a4 \ (\lambda a5 \ a6 \rightarrow \text{Cons} (a1 \ a5) (a3 a6))) \ a2 \)
\( \text{IncElems}' = \lambda a1 \rightarrow \text{map} (\lambda a2 \rightarrow \text{Succ} a2) \ a1 \)
\( \text{DecElems}' = \lambda a1 \rightarrow \text{map} (\lambda a2 \rightarrow \text{pred} a2) \ a1 \)
\[ \downarrow \text{refinement (1.1 s)} \]

\[ T_4 = \text{master (\ map \rightarrow Tuple Map IncElems' DecElems' CubeElems)} \]
where \( \text{Map} = \lambda a1 \ a2 \rightarrow \text{fix} (\lambda a3 \ a4 \rightarrow \text{caseList} a4 a4 \ (\lambda a5 \ a6 \rightarrow \text{Cons} (a1 \ a5) (a3 a6))) \ a2 \)
\( \text{IncElems}' = \lambda a1 \rightarrow \text{map} (\lambda a2 \rightarrow \text{Succ} a2) \ a1 \)
\( \text{DecElems}' = \lambda a1 \rightarrow \text{map} (\lambda a2 \rightarrow \text{pred} a2) \ a1 \)
\( \text{CubeElems} = \lambda a1 \rightarrow \text{map} (\lambda a2 \rightarrow \text{times} a2 (\text{times} a2 a2)) \ a1 \)

Figure 8.3: Demonstration of RUFINSKY doing cumulative learning. Each refinement stage is performed by the RUFUS subsystem, and each abstraction invention stage is performed by the KANDINSKY subsystem. With each stage is shown the time taken by a prototype implementation of RUFINSKY running on a desktop PC.
encapsulating what those two hypotheses have in common. I have given this abstraction the name map (Fig. 8.3) because it happens to be an instance of the map operation that is standard in functional programming languages. The type of map here is:

\[(\text{Nat} \to \text{Nat}) \to \text{List} \to \text{List}\]

The factor by which the prior probability of the master term increases as a result of introducing this abstraction is equal to:

\[
\frac{\text{clPrior}(T_3, \Gamma)}{\text{clPrior}(T_2, \Gamma)} = 83.3
\]

KANDINSKY is then invoked once more on \(T_3\) in order to check for additional opportunity for abstraction, however no more is found at this point:

\[\text{absInvBest}(T_3) = T_3\]

In the next step, RUFUS is applied to the third sub-problem whose target datum is \(\text{cubeElems}\) (Fig. 8.1). The hypothesis space for this sub-problem is affected by the presence of the new map abstraction. Thus, \(\mathcal{L}\) (Defn. 5.3) is to be defined in this context with respect to the target type \(\text{List} \to \text{List}\) and the set of symbols \(\Gamma'\) where:

\[\Gamma' = \Gamma \cup \{\text{map}\}\]

A modified ‘local’ background knowledge \(B'\) for the sub-problem is also required due to the presence of map. \(B'\) is the toplevel environment (Defn. 3.29) defined with respect to the same data type and value declarations as \(B\), plus the following additional value declaration:

\[\text{map} = \text{Map}\]

where the term \(\text{Map}\) is given in Fig. 8.3. The prior over hypotheses \(\text{CubeElems}\) for this sub-problem is then equal to:

\[
P(\text{CubeElems}|\text{Map}, \text{IncElems}', \text{DecElems}') = \\
\frac{\text{clPrior}(\text{master (\map \to \text{Tuple \text{Map} IncElems' DecElems'} CubeElems}), \Gamma)}{\text{clPrior}(\text{master (\map \to \text{Tuple \text{Map} IncElems' DecElems'} #)), \Gamma)}
\]

RUFUS is invoked as \(\text{runRufus}(B', \text{cubeElems})\), and returns \(\langle\text{"goal"}, \text{CubeElems}\rangle\) where \(\text{CubeElems}\) is given in Fig. 8.3. Substituting \(\text{CubeElems}\) into the master term yields \(T_4\) (Fig. 8.3).

KANDINSKY is now invoked one last time, but finds no further opportunity for abstraction:

\[\text{absInvBest}(T_4) = T_4\]

At this point all of the sub-problems have been solved. RUFINSKY stops here and outputs \(T_4\) as the final hypothesis.
I tested the above process using a prototype implementation of RUFINSKY running on a desktop PC. The times taken to run each stage are shown in Fig. 8.3.

It is illuminating to see what happens when the prototype implementation of RUFINSKY is applied to the same learning scenario without abstraction invention, i.e. just applying RUFUS to each of the sub-problems in turn and not using KANDINSKY. The result on the first two problems is much the same: RUFUS again takes about five or six seconds to solve each. However, for the third problem, RUFUS runs for 150 seconds without finding a solution at all. This reflects the fact that the third problem, cubeElems, is in fact significantly more difficult for RUFUS to solve than the first two problems when using the initial background knowledge. This is due to the extra complexity of expressing the compound ‘cube’ operation rather than the simple primitive ‘succ’ or ‘pred’ operations. The exponentially large nature of RUFUS’ search space means that a small increase in the complexity of a solution can mean a large increase in the time taken to find it.

With the help of the invented map abstraction, we see that the time taken for RUFUS to solve cubeElems falls to only 1.1 seconds. This illustrates the basic principle of cumulative learning: progression to more difficult problems is made possible by the accumulation of new knowledge.

8.2 How RUFINSKY could be Experimentally Evaluated

The worked example of the previous section serves as a simple demonstration of cumulative learning. In this section I suggest an experiment that would more rigorously evaluate the effectiveness of RUFINSKY’s cumulative learning mechanism. This experiment is designed to test the hypothesis that ‘RUFINSKY’s accumulation of knowledge facilitates its progression to more difficult problems’.

In order to demonstrate progression from easy to hard problems, it is necessary to choose a test problem domain in which there is a ready source of related inductive inference problems with a range of difficulties. A suitable choice is the domain of integer sequence prediction. These are puzzles, commonly found in IQ tests or recreational puzzle books, in which one is asked to spot the pattern in a sequence of numbers and hence predict what comes next, for example:

\[ 1, 4, 9, 16 \ldots \]

Here, one should spot that this is a sequence of square numbers, and that the next number is 25. Here is a much more difficult example:
This is fibonacci sequence interleaved with a geometric sequence (the next two numbers are 16 and 8). Integer sequence puzzles are ideal as a test domain because they are easy to obtain, they come in sufficient variety and range of difficulties, and they are also interesting from an AI point of view due to their association with human intelligence tests.

A corpus of integer sequence prediction problems of varying difficulties could be collected from published sources such as puzzle books [Mullins, 2008; Nasser, 2005; Tolley and Thomas, 2006]. A few hundred problems would be a suitable size for such a corpus, large enough so that a variety of different sequence types could be represented. Each integer sequence would need to be partitioned into ‘training’ and ‘test’ parts. The ‘training’ parts would be used to construct a target datum, and the ‘test’ parts would be kept aside to evaluate predictive accuracy.

An outline of the method for this experiment is as follows. Runs of RUFINSKY would be performed over the corpus in four different modes:

1. adaptive weights OFF and abstraction invention OFF;
2. adaptive weights ON and abstraction invention OFF;
3. adaptive weights OFF and abstraction invention ON;
4. adaptive weights ON and abstraction invention ON.

Here, ‘adaptive weights’ refers to the use of the weights learning mechanism of the non-parametric prior (Sect. 4.2). ‘Abstraction invention’ refers to the use of KANDINSKY. Both of these features can be turned on or off independently within RUFINSKY.

RUFINSKY’s initial background knowledge should be as minimal as possible, consisting of only a few primitives such as arithmetic operators, a sequence data type, and the fixed point operator. Thus, RUFINSKY would not start with a strong bias towards the problem domain: it would not be given any prior knowledge about standard classes of sequence such as iterative, arithmetic, geometric, fibonacci, polynomial etc. Instead, RUFINSKY would have the opportunity to cumulatively learn such a bias, and if the experiment were successful one might expect it to invent abstractions that corresponded to such sequence classes of its own accord.

In order to achieve a convincing test of autonomous cumulative learning it would be desirable for the corpus of sequence problems to be provided to RUFINSKY in a randomized order, rather than in a pre-specified order of ‘difficulty’. However, this would require RUFINSKY to use a different learning policy at
the high level from the one used in the worked example of Sect. 8.1. Recall that in that simple policy, RUFINSKY simply worked through the problems in the order given, alternately applying RUFUS and KANDINSKY. In order to handle randomly ordered problems, better results would likely be achieved with a policy that applied RUFUS to many problems in parallel. This would allow RUFUS to discover solutions to the easiest of the problems without getting stuck for large amounts of time on the hard ones. On the other hand, KANDINSKY could still be invoked in the usual way, performing abstraction invention over the monolithic hypothesis immediately following each successful solution of a problem by RUFUS.

If this experiment were successful I would expect the following pattern of results. With abstraction invention OFF, I would expect RUFINSKY to be capable of correctly solving only a small fraction of the problems within a reasonable time limit (say, less than 20% of the problems after running for 100 CPU-hours). These solved problems would correspond to the ‘easy’ problems of the corpus. On the other hand, with abstraction invention ON I would expect RUFINSKY to correctly solve the majority of the problems within the same time limit (say, more than 80% of the problems). Such an increase in the proportion of problems solved is expected if cumulative learning does indeed facilitate ‘progression to more difficult problems’.

I would also expect that turning adaptive weights ON would have a positive effect on reducing the time taken for many of the problems to be solved, and for increasing the total number of problems solved. The reason for this is that the weights adaptation mechanism allows RUFINSKY to retrospectively adjust the probability weights of learned abstractions according to how useful each one turns out to be for future learning. This can mitigate a potential slowdown due to an increase in the branching factor of RUFUS’ search space as the number of learned abstractions increases.

In order to be suitable for an experiment like this, the prototype RUFINSKY implementation used in the worked example of the previous section would need to be extended somewhat. In particular, a different search policy other than the simple best-first search of Sect. 5.2.4 would have to be devised for RUFUS. The reason for this is a practical one of memory usage. Best-first search, like breadth-first search, uses an amount of memory that increases linearly with the amount of time for which the search process runs. In the prototype implementation, RUFUS typically uses up memory at a rate of about 30 MB/s. That means that of the order of 100 GB of memory would be needed in order to run RUFUS’ search procedure for one hour. A sensible way to solve this problem is to design a better policy than breadth-first search, which could either be a stochastic search
or some form of iterative deepening policy. A stochastic search would probably
be simpler to implement, and might also have other advantages such as being
able to cope better with semantic redundancy in the search space. On the other
hand, the results produced by a stochastic search might contain structural noise
that could reduce the ability of KANDINSKY to perform abstraction invention
effectively.

8.3 Limitations of RUFINSKY’s Cumulative
Learning Mechanism

RUFINSKY does have some limitations that restrict its capacity for cumulative
learning, which I shall now describe. Note that each of these limitations could
in principle be overcome by extending the RUFINSKY system: they are not
limitations of the lambda calculus language itself.

Firstly, KANDINSKY is restricted in its ability to recognise commonality be-
cause it operates at a purely syntactic level: auSearch knows nothing about the
semantics of lambda calculus terms. This means that KANDINSKY will miss
opportunities for abstraction that would require taking the meaning of back-
ground knowledge into account. For example, given appropriate background
knowledge, the terms (times 2 x) and (plus x x) have the same meaning. If
they were both to be present as subterms in some larger term then they would
present an opportunity for abstraction that KANDINSKY would be unable to
take advantage of.

Secondly, RUFINSKY has no mechanism for inventing new data types.
Though it can invent the equivalent of new value declarations through KANDIN-
SKY’s abstraction invention mechanism, the data type declarations that RUFIN-
SKY has access to are fixed a priori and must be set by a human user. Since
the set of available data types is an important component of inductive bias,
this is a significant limitation on the extent to which RUFINSKY can adapt its
inductive bias through cumulative learning.

Finally, the restriction to simply typed lambda calculus that is made through-
out this thesis limits what kinds of abstractions can be made by RUFINSKY.
As an example, consider a variation on the demonstration example of Sect. 8.1
in which instead of incElems and decElems, the first two sub-problems are
incElems and negateElems with the following types (given in Haskell-style no-
tation):

\[
\text{incElems :: [Int] -> [Int]}
\]
\[
\text{negateElems :: [Bool] -> [Bool]}
\]

Here, negateElems is the function that applies the boolean ‘not’ operator to
each element of a list. Under the constraints of simply typed lambda calculus, it is impossible to abstract out the map operation from commonality in these two programs, because it would require map to have a polymorphic type:

\[
\text{map :: forall } a. \ (a \rightarrow a) \rightarrow \ [a] \rightarrow \ [a]
\]

To overcome this limitation it would be necessary to use a formulation of lambda calculus that has a more powerful type system supporting polymorphism, such as a Hindley-Milner type system or System F [Pierce, 2002].

### 8.4 Issues Raised about the Design of Inductive Programming Systems

#### 8.4.1 First order logic and lambda calculus

In this thesis I adapted a number of concepts and techniques from first-order logic based ILP to a lambda calculus setting for learning. In particular, I adapted the idea of semantic and syntactic generality orderings (entailment and subsumption), as well as the techniques of refinement, proof-directed search, and inverse deduction. Actually, I have been surprised at how readily these techniques could be transferred across representation languages. It is evidence that the fundamental ideas of ILP transcend first-order logic, and that they may be regarded as inductive programming techniques in a more general sense. It is likely that other ILP techniques too besides those investigated in this thesis, such as inverse entailment [Muggleton, 1995], could also be adapted to lambda calculus.

In this transfer of techniques from first-order logic to lambda calculus, it is interesting to ask what things have changed and what things have stayed the same. In the setting for learning, the concept of background knowledge was retained. On the other hand, positive and negative examples were replaced in the lambda calculus setting by a ‘target datum’. Actually, the target datum concept can be seen as a generalisation of the the idea of positive and negative examples, providing more flexibility in how we may specify observed data. Indeed, when the target type is a function type with return type \(\text{Bool}\) as defined by the following data type declaration:

\[
data \text{Bool} = \text{True} \mid \text{False}
\]

we see that the target datum becomes equivalent to a set of positive and negative examples for a predicate.

However, unlike in the first-order logic setting, one may also have a target datum with a non-boolean return type. Thus, in the lambda calculus setting one
can directly model a function that returns, say, a three-valued class, or a list of integers. This gives improved flexibility in what constraints may be imposed over the hypothesis space in a learning problem. It eliminates the need for negative examples whose only purpose is to enforce a functional dependency as in Fig. 2.1; instead, such a functional dependency can be an automatic consequence of the choice of target type.

We saw in Sect. 3.3 that one can use the semantic approximation order, i.e. the concept of ‘definedness’, as a suitable definition of generality in lambda calculus. At first sight, this might seem quite different from the standard ILP notion of generality based on implication. However, both notions of generality can be seen to express the same basic idea, that of narrowing the set of ‘possible worlds’ that may be true. The lambda calculus definedness ordering is one of partial information: a hypothesis specifies a partially defined value in the domain of some type. We can think of each fully defined value of that type as a ‘possible world’. When a hypothesis becomes more general, it becomes more defined, and hence is consistent with fewer possible worlds.

In the case of first-order logic, a ‘possible world’ corresponds to a Herbrand interpretation, i.e. an assignment of truth or falsity to all possible atoms, where an atom is a predicate symbol applied to ground arguments. When a hypothesis becomes more general, it implies stronger constraints on the allowed truth values of these atoms, and hence also is consistent with fewer possible worlds (Fig. 8.4).

On the other hand, there are some clear differences between the first-order and the lambda calculus versions of generality. The most significant concerns the existence of a ‘most general’ or ‘top’ hypothesis. Now, both settings feature a ‘least general’ / ‘bottom’ hypothesis that is consistent with all possible worlds, and hence contains no information. In the case of first-order logic, a tautology is such a bottom hypothesis, and in the case of lambda calculus it is a term.

Figure 8.4: An illustration of the concept of generality in terms of ‘possible worlds’, applicable to both the first-order logic and the lambda calculus settings. In each of the Venn diagrams, the large rectangle represents the set of all possible worlds, and each circle represents the set of possible worlds consistent with a particular hypothesis. In a), hypothesis Y is more general than hypothesis X. In b), hypotheses X and Y are neither more nor less general than one another, however they do possess a common generalisation. In c), hypotheses X and Y are neither more nor less general than one another, and they do not possess a (consistent) common generalisation.

156
representing a completely undefined value. In first-order logic we also have a top hypothesis, i.e. a logical contradiction, consistent with no possible worlds.

By contrast, in the lambda calculus setting it is built into the structure of the ‘definedness’ order that every hypothesis is consistent with at least one possible world.\(^1\) Therefore, there is no top hypothesis, i.e. no hypothesis exists that is a generalisation of all others. One might see this as a disadvantage, because it restricts us to ‘bottom-up’ learning techniques, i.e. searching the hypothesis space by starting with the least general hypothesis and then attempting to generalise; in the first-order logic setting one has a choice between ‘top-down’ and ‘bottom-up’. However, the nonexistence of contradictory hypotheses in the lambda calculus setting can be seen as an advantage because it means that hypotheses do not need to be checked for inconsistency: self-consistency is always guaranteed.

8.4.2 Bayesian inference

In Sect. 3.4 I formulated a lambda calculus setting for learning explicitly in terms of Bayesian inference. In Chap. 4 I then defined exact, normalised prior probability distributions over a hypothesis space of lambda calculus terms. This manifestly Bayesian approach has a number of advantages. As discussed in Chap. 4, it makes it possible to do inference by Bayes prediction rather than being limited to Bayes MAP. Note that although the best-first search strategy for RUFUS that was outlined in Sect. 5.2.4 is a Bayes MAP strategy, the rest of the design of RUFUS is entirely agnostic as to whether Bayes MAP or Bayes prediction is being used. It therefore would be straightforward to replace this best-first search strategy with, say, a Monte Carlo sampling policy for Bayes prediction.

By making prior probability distributions explicit, it was possible to get significantly more flexibility out of these priors than is possible when one is limited to using an ILP-style compression measure. It enabled the introduction of a ‘soft’ form of declarative bias via weights on elements of background knowledge. It also enabled the introduction of a means for automatically adapting these weights with experience, using the non-parametric extension to the base prior. Finally, it enabled the formulation of cumulative learning scenarios using the ‘monolithic hypothesis’ technique and the cumulative learning prior. Crucially, these extra features were all added without making any change to the setting for learning itself: only the prior needed to be modified.

Overall, a fully Bayesian setting for learning has many advantages and helps

\(^1\)Except in the trivial case of an uninhabited data type, i.e. a data type with zero constructors.
to bring inductive programming onto a firmer statistical footing alongside conventional Bayesian machine learning techniques such as Bayesian networks, Gaussian processes, etc.

8.5 Related Work

In this section I discuss in retrospect some alternative techniques from the literature which are related to the work in this thesis.

8.5.1 Search strategies

For the RUFUS inductive programming system I purposefully chose a simple, systematic search strategy consisting of refinement coupled with best-first search. My reason for not using a more complex or ‘intelligent’ search strategy was that, when doing cumulative learning, it is desirable to avoid where possible having inductive bias implicitly encoded in the search procedure. As explained in Sect. 2.3.1, cumulative learning requires that inductive bias be transparently specified and easy to modify automatically, therefore it is preferable to encode inductive bias in forms such as background knowledge and type declarations. However, it is worth mentioning that many other approaches to inductive programming do use advanced or domain-specific search strategies. Two notable examples are the so-called ‘analytical’ approach and the ‘evolutionary’ approach.

In the ‘analytical’ approach, the idea is to scan the data or input-output examples so as to detect specific kinds of patterns. Once a pattern is detected, a hypothesis describing it can be synthesised directly [Summers, 1977; Biermann, 1978]. The advantage of the approach is that it does not involve a combinatorial search, however it does entail that a strong domain-specific bias towards certain classes of patterns is built in to the system. The Igor2 system [Kitzelmann, 2009b] is a more recent attempt to improve the flexibility of the analytical approach by combining it with combinatorial search.

Certain deductive program synthesis techniques such as that of Manna and Waldinger [1980] can also be understood as instances of the analytical approach to inductive programming. Manna and Waldinger used theorem-proving in order to produce a constructive proof of the existence of a program satisfying a given specification. ‘Constructive’ means that a proof comes with a procedure for constructing one program satisfying the specification. The specification need not be complete, and different proofs may yield different programs; hence the technique can be used to generate multiple alternative hypotheses.

In the ‘evolutionary’ approach of ADATE [Olsson, 1995], a population of programs is evolved by repeatedly applying various transformation operators to
those programs. In between transformations the programs are tested for fitness against a specification. Heuristics are employed in order to choose the most appropriate transformations so as improve the efficiency of the search.

8.5.2 Hybrid functional logic languages

In this thesis I have compared the first-order logic (Prolog-style) and lambda calculus (Haskell-style) programming paradigms in terms of their suitability for inductive inference and cumulative learning. Although the vast majority of the literature in inductive programming sits within one of these two paradigms, there have been some attempts to merge the two together, i.e. to use a hybrid functional logic programming paradigm. Functional logic programming languages are characterised by their support for first-class functions as well as features typically associated with Prolog such as unification and multimodality. Examples are the experimental languages Curry [Antoy and Hanus, 2010] and Escher [Lloyd, 1999]. The use of such hybrid languages for inductive programming is known as Inductive Functional Logic Programming (IFLP). An IFLP framework has been advocated by Bowers et al. [1997], who defined a setting for learning for the Escher language and outlined a simple IFLP learning algorithm based on heuristic search. Also, Hernandez-Orallo and Ramirez-Quintana [1999] formalised an analogue of inverse resolution called inverse narrowing within an IFLP setting.

The main drawback of the IFLP approach is that it necessitates using a programming paradigm that is itself experimental, and much less well understood than the Prolog or Haskell paradigms. From the relatively small amount of work in IFLP done so far, it is not entirely clear whether the extra expressivity of hybrid languages such as Curry or Escher is an overall benefit or drawback. These are without doubt powerful languages, but with increased power tends to come increased complexity and this could make it difficult to develop practical inductive programming algorithms.

8.5.3 Program transformation

I showed in Sect. 7.2 that KANDINSKY’s abstraction construction mechanism can be understood as a transformation process involving multiple applications of inverse beta- and eta- reduction rules. Related processes have been studied in work on program transformation such as that of Burstall and Darlington [1977]. Burstall and Darlington implemented a semi-automatic system that could be used to transform a functional program into a more computationally efficient form without changing its meaning (for example by altering its recursion structure to avoid the recomputation of intermediate values). Their system
used a set of transformation rules which could be applied in sequence, and indeed one of these rules was an ‘abstract’ rule very similar to inverse beta-reduction. However, in their case the aim in applying this rule was not to abstract out commonality, but rather it was to enable the subsequent ‘folding’ of a program by matching the abstracted expression against the body of some existing function declaration.

8.5.4 Analogue reasoning

A field related to abstraction invention is analogue reasoning [Bartha, 2013]. This is the branch of philosophy and cognitive science that attempts to characterise what it means to think in terms of analogies, i.e. similarity arguments between objects or domains. Authors such as Weller and Schmid [2006] have argued that abstraction should be seen as the principal basis of analogue reasoning. Weller and Schmid advocate anti-unification as a mechanism for generating abstractions, as others have done (see Sect. 2.4.1) and as I do in this thesis. Additionally, they demonstrate how a process called E-generalisation [Burghardt, 2005] can be used to take background knowledge into account during the abstraction process. At present E-generalisation has only been formalised in the context of first-order logic, however a lambda calculus version of E-generalisation could potentially be used to solve the first limitation of KANDINSKY that I described in Sect. 8.3.
Chapter 9

Conclusions and Future Work

9.1 Summary of Thesis and Main Results

In this PhD project I designed a software system to perform cumulative learning. Cumulative learning means solving problems by inductive inference, and also incrementally accumulating knowledge so as to enable progression to more difficult problems. I identified the software engineering technique of abstraction as a suitable means for acquiring knowledge. I designed an algorithm to perform abstraction invention, which automates the process of searching for syntactic patterns within a corpus of knowledge and then abstracting out those patterns into re-usable units. For the system’s knowledge representation language I considered both lambda calculus and first-order logic, and argued that lambda calculus is more suitable in principle because it better supports the technique of abstraction. However, more effective generic inductive inference algorithms were known to exist for first-order logic. To resolve this dilemma I adapted the inference techniques of refinement, proof-directed search, and inverse deduction from first-order logic to lambda calculus.

The main results of this thesis are as follows:

- The semantic approximation ordering is a suitable notion of ‘generality’ for lambda calculus.

- Appropriate prior probability distributions can be defined over hypothesis spaces of lambda calculus terms, allowing the use of a fully Bayesian setting for inductive inference. This also improves the flexibility with which prior assumptions can be specified versus using a more standard ‘compression’ measure.
Refinement and proof-directed search can readily be adapted from first-order logic to lambda calculus, yielding an inductive inference algorithm that features guided search and pruning. Levin search can also be incorporated into the same algorithm in order to handle non-terminating hypothesis programs with ease.

The technique of anti-unification can be extended to act over the exponentially large space of all combinations of subterms of a lambda calculus term, and made feasible via a heuristic beam search. This allows for discovery of syntactic commonality between arbitrary combinations of parts of a term.

A form of inverse deduction, based on inverting beta- and eta-reduction rules, can be used to automate the construction of abstractions in lambda calculus.

9.2 Directions for Future Work

A number of avenues of future work follow from this thesis. The most obvious next step is to do an experimental evaluation of RUFINSKY, as outlined in Sect. 8.2, so as to measure how effective RUFINSKY’s cumulative learning mechanism is in practice. A direct experimental comparison between RUFUS and existing ILP/IFP systems would also be worthwhile. For example, a comparison between RUFUS and MagicHaskeller [Katayama, 2007] would reveal to what extent refinement is an improvement over brute-force search in the lambda calculus setting. Furthermore, a comparison between RUFUS and refinement-based ILP systems such as FOIL [Quinlan, 1990], MC-TopLog [Muggleton et al., 2012b], or HYPER [Bratko, 1999] would shed more light on the relative strengths and weaknesses of first-order logic versus lambda calculus.

On the theoretical side, it would be useful to derive asymptotic time and space complexity results for the algorithms presented in this thesis, particularly auSearch and absInv. Such results would help us to judge how well these algorithms can scale up to large datasets.

A major theme of this thesis was the notion that techniques from first-order ILP can be adapted to the lambda calculus setting. It would be interesting to investigate the transfer of additional techniques other than the ones looked at here. I mentioned in Sect. 8.4.1 that the technique of inverse entailment may well have a useful analog in the lambda calculus setting. Other techniques worth investigating are relative least-general-generalisation [Muggleton and Feng, 1990] and abduction [Denecker and Kakas, 2002].
In Sect. 8.3 I described three fundamental limitations of RUFINSKY’s existing design. All three limitations could potentially be addressed by extending or generalising the algorithms presented here. KANDINSKY could be extended so as to take semantics and background knowledge into account by incorporating a technique like E-generalisation [Burghardt, 2005]. Regarding invention of new data types, from a theoretical point of view it would be straightforward to extend the cumulative learning prior of this thesis so as to act additionally over a space of data type declarations. It may then be feasible to design a search algorithm, perhaps along analogous lines to the mechanism of KANDINSKY, that can discover opportunities to usefully re-express a program in terms of new data types. Finally, extending RUFINSKY to support a polymorphic type system such as Hindley-Milner polymorphism would be a useful improvement. It would require significant changes mainly to the prior probability distributions and to RUFUS’ refinement operator. Some guidance could be taken from the design of MagicHaskeller [Katayama, 2007], an IFP system which already supports type polymorphism.

As I discussed in Sect. 5.2.4, a natural extension to RUFUS would be to add a new search strategy in place of best-first search so as to support Bayes prediction effectively. Such a search strategy might be based on Monte Carlo sampling. KANDINSKY could also potentially be made Bayes prediction friendly by replacing the greedy \texttt{absInvBest} strategy with an appropriate stochastic function that samples from the results returned by \texttt{absInv}. Some theoretical work would likely need to be done to ensure that hypotheses generated by a Monte Carlo version of RUFINSKY are sampled fairly from the Bayes posterior.

Some of the techniques developed in this thesis may have practical applications outside of inductive programming. A KANDINSKY-like system could potentially be used to help automate the design of software libraries by searching for useful opportunity for abstraction in code written by human programmers. Another application of a KANDINSKY-like system is to automated theorem proving, where abstraction invention could be used to invent re-usable lemmas from commonality in mathematical proofs. The fact that modern theorem provers typically use lambda calculus as a representation language for proofs makes this a natural extension. Finally, prior probability distributions over programs such as those introduced in this thesis may be relevant to the design of predictive software development aids such as autocompletion or a ‘Dasher’-like system [Ward et al., 2000] for programming.
9.3 Final Thoughts

What are the future possibilities for cumulative learning? In the introduction at the very beginning of this thesis I gave three sources of motivation for this research:

- improving our understanding of the nature of human learning;
- developing better practical machine learning systems;
- making progress towards the development of strong AI.

Such goals are of course long term aims of the field of artificial intelligence as a whole, and cumulative learning is one approach complementary to other ideas. However, perhaps this thesis has convinced you that research into cumulative learning shows promise as a route towards such goals. In its favour, cumulative learning is clearly a relatively untapped field in which there are many interesting avenues still to explore.
Appendix A

Mathematical Conventions

I use the following miscellaneous conventions in this thesis:

- A natural number is an integer greater than or equal to one.
- \( |X| \) denotes the cardinality of a set \( X \).
- The operator \( \cdot \) applies a function to an argument, i.e. \( f \cdot x \equiv f(x) \).
- The operator \( : \) is the ‘cons’ operator for lists. It prepends an element to the front of a list.
- The operator \( +\!+\) appends two lists together.
- \( \text{product}(x) \) is the product of the elements of a list \( x \).
- \( \text{replicate}(n,x) \) is the list of \( n \) instances of \( x \).

A.1 Set Comprehension Notation

Throughout this thesis I use set comprehension notation in the style of the formal language Z [Woodcock and Davies, 1996, Sect. 5.2]. Traditional mathematical set-builder notation is often difficult to use in a way that is both concise and unambiguous at the same time, and I have found Z’s set comprehension notation to be a vast improvement in this respect. Other authors such as Fokkinga [2006] have also advocated this use of Z-style set comprehension notation for mathematical writing.

The set comprehension notation in this thesis is the same as Z’s apart from a minor difference in use of symbols. I use \( \triangleleft \) instead of \( : \) in order to declare a variable, and I use \( \mid \) instead of \( | \) to indicate the start of the ‘predicate part’ of an expression. This is because the symbols \( : \) and \( | \) are already in use in this thesis as the list ‘cons’ operator and the set cardinality operator respectively.
The basic form of a set comprehension expression is:

\[ \{ x \triangleleft X \mid p(x) \bullet f(x) \} \]

The broken vertical bar and the bullet divide the expression into three parts, a *declaration part* on the left, a *predicate part* in the middle, and a *term part* on the right. In the declaration part, the \( \triangleleft \) symbol declares the variable \( x \) to be a member of the set \( X \). The whole expression has the following meaning:

“The smallest set such that, for each \( x \in X \) where \( p(x) \) holds, \( f(x) \) is a member of the set.”

As an abbreviation, either the predicate part or the term part may be omitted, in which case the meaning is as follows:

\[ \{ x \triangleleft X \mid f(x) \} = \{ x \triangleleft X \mid \text{true} \bullet f(x) \} \]
\[ \{ x \triangleleft X \mid p(x) \} = \{ x \triangleleft X \mid p(x) \bullet x \} \]

One may use pattern matching, or multiple declarations separated by semicolons, inside the declaration part. Here are a few examples:

\[ \{ \langle x, y \rangle \triangleleft \{ (1, 3), (2, 4), (5, 2) \} \bullet x + y \} = \{ 4, 6, 7 \} \]
\[ \{ x \triangleleft \mathbb{N}; \ y \triangleleft \mathbb{N} \mid x + y = 4 \bullet (x, y) \} = \{ (1, 3), (2, 2), (3, 1) \} \]
Appendix B

Proof of Lemma 6.53

Lemma B.1. Let \( a, b \) be places and let \( t \) be a term such that \( b \) is a place in \( t \) and \( a \) is an improper ancestor of \( b \). It follows that \( a \) is a place in \( t \).

Proof. By structural induction on \( a \):

1. \( a = [] \). The proof follows from Fact 6.6.

2. \( a = i : a' \). By Defn. 6.15, \( b \) is of the form \( i : b' \) where \( a' \) is an improper ancestor of \( b' \). By Fact 6.7 it holds that \( t \) is of the form \( f(t_1 \ldots t_n) \) where \( n \geq i \), and \( b' \) is a place in \( t_i \). By the inductive hypothesis, \( a' \) is a place in \( t_i \), and hence the proof follows by Fact 6.7.

\( \square \)

Lemma B.2. For places \( a, b, c \) such that \( a \) and \( b \) are both improper ancestors of \( c \), it holds that:

\[ a \leq b \implies a \text{ is an improper ancestor of } b \]

Proof. Assume \( LHS \). By Defn. 6.15, \( a \) and \( b \) are both prefixes of \( c \). Hence by Defn. 6.13, \( a \) is a prefix of \( b \). \( RHS \) follows by Defn. 6.15.

\( \square \)

Lemma B.3. If a shoot \( \langle a, b, P \rangle \) contains a place \( x \), then \( x \) is an improper descendant of \( a \).

Proof. Consider each of the two cases of Defn. 6.33:

1. It is given that \( x \) is an improper descendant of \( a \).

2. By Defn. 6.23, \( x \) is an improper descendant of \( b \). It follows by Defn. 6.31 and Proposition 6.16 that \( x \) is also an improper descendant of \( a \).

\( \square \)

Proof. Let \( \langle a, b, P \rangle \) be a non-empty shoot. By Defn. 6.31, there are two cases to consider:

1. \( a = b \). Since the shoot is non-empty, then it must contain some place \( x \) which, by Defn. 6.33, is contained in the lasso \( \langle a, P \rangle \). By Defn. 6.23, \( x \) is not an improper descendant of any element of \( P \). Hence, by Lemma B.3 and Proposition 6.16, \( a \) is not an improper descendant of any element of \( P \), and the proof then follows from Defn. 6.23 and Defn. 6.33.

2. \( a \) is a proper ancestor of \( b \). By Proposition 6.16, \( a \) is an improper descendant of itself. Hence the proof follows from Defn. 6.33.

\[ \square \]

**Lemma B.5.** Given a shoot \( \psi = \langle a, b, P \rangle \) and places \( x, y \) such that \( \psi \) contains \( y \), \( a \) is an improper ancestor of \( x \), and \( x \) is an improper ancestor of \( y \), it follows that \( \psi \) contains \( x \).

Proof. By Defn. 6.33 and Defn. 6.23, there are two cases to consider:

1. \( y \) is a proper ancestor of \( b \). Therefore, by Proposition 6.16, \( x \) is a proper ancestor of \( b \). Hence the proof follows by Defn. 6.33 and Lemma B.1.

2. \( y \) is an improper descendant of \( b \) and not an improper descendant of any element of \( P \). By Proposition 6.16, it holds that \( x \) also is not an improper descendant of any element of \( P \). Furthermore, by Lemma B.2, \( x \) is either an ancestor or an improper descendant of \( b \). Therefore the proof follows from Defn. 6.33, Lemma B.1, and Defn. 6.23.

\[ \square \]

**Lemma B.6.** Given two shoots \( \psi_1, \psi_2 \) that overlap one another but do not share the same root, it follows that the root of \( \psi_1 \) is either an ancestor or a descendant of the root of \( \psi_2 \).

Proof. By Defn. 6.33, there exists some node \( x \) contained in both \( \psi_1 \) and \( \psi_2 \). Furthermore, by Lemma B.3, the roots of \( \psi_1 \) and \( \psi_2 \) are both improper ancestors of \( x \). Hence by Lemma B.2, either the root of \( \psi_1 \) is an ancestor of the root of \( \psi_2 \) or vice-versa, depending on which comes first lexicographically.

\[ \square \]

**Lemma B.7.** Given two non-empty shoots \( \psi_1, \psi_2 \) that do not share the same root, it holds that \( \psi_1 \) contains the root of \( \psi_2 \) i.f.f. \( \psi_1 \) overlaps \( \psi_2 \) and the root of \( \psi_1 \) is an ancestor of the root of \( \psi_2 \).

Proof. We shall consider the forward implication followed by its converse. Let \( a \) be the root of \( \psi_1 \) and let \( x \) be the root of \( \psi_2 \), then proceed:
1. Suppose that $\psi_1$ contains $x$. By Lemma B.4, $\psi_2$ also contains $x$. By Defn. 6.33, $\psi_1$ and $\psi_2$ therefore overlap. Also, given that $a \neq x$, it follows from Lemma B.3 that $a$ is an ancestor of $x$.

2. Suppose that $\psi_1$ and $\psi_2$ overlap and $a$ is an ancestor of $x$. By Defn. 6.33 there must exist some place $y$ that is contained in both $\psi_1$ and $\psi_2$. By Lemma B.3, $x$ is an improper ancestor of $y$. Given also that $a$ is an ancestor of $x$, it follows by Lemma B.5 that $\psi_1$ contains $x$.

Lemma B.8. For non-empty shoots $\psi_1, \psi_2, \psi_3$ such that the root of $\psi_1$ is an ancestor of the root of $\psi_2$, the root of $\psi_2$ is an ancestor of the root of $\psi_3$, and $\psi_1$ overlaps $\psi_3$, it follows that $\psi_1$ overlaps $\psi_2$.

Proof. By Proposition 6.16 and Lemma B.7, $\psi_1$ contains the root of $\psi_3$. Therefore by Lemma B.5, $\psi_1$ contains the root of $\psi_2$. It follows by Lemma B.7 that $\psi_1$ overlaps $\psi_2$.

Proof of Lemma 6.53. Letting $G$ be the directed overlap graph of $S$, consider in the turn the two conditions of Defn. 6.43:

1. By Defn. 6.52, it holds for every vertex $x$ in $G$ that, lexicographically, the root of $x$ comes strictly before the root of every successor of $x$. If a directed cycle exists in $G$, then there exists a vertex that is its own successor, and hence whose root comes strictly before itself lexicographically. This is impossible, therefore $G$ has no directed cycles.

2. Suppose that there exist vertices $a, b, c$ in $G$ such that $a \neq b$, $a \rightarrow c$, and $b \rightarrow c$. Without loss of generality, assume that the root of $a$ comes before the root of $b$ lexicographically. By Defn. 6.52 and Lemma B.6, the roots of $a$ and $b$ are both ancestors of the root of $c$. Therefore, by Lemma B.2, the root of $a$ is an ancestor of the root of $b$. Also, by Lemma B.7, $a$ contains the root of $c$, and furthermore by Lemma B.5, $a$ contains the root of $b$. Therefore, by Lemma B.7, $a$ overlaps $b$, so it follows from Defn. 6.52 that $a \rightarrow b$. 

\[\square\]
Bibliography


