Synthesis of Recursive Functional Programs from Examples

Bjarte M. Østvold
Norwegian University of Science and Technology

October, 1999
Abstract

An approach to synthesis of recursive functional programs from examples is presented. The problem: Given a finite set $E$, called examples, of equations corresponding to applications of a function $f$ to constant expressions, $f t_0 \cdots t_n = t_{n+1}$, a type $\tau$ for $f$, and information about the complexity of $f$. Find a recursive definition $D$ of $f$ that assigns type $\tau$ to $f$ and that computes results from arguments as specified by $E$. Definition $D$ is then called a solution. An algorithm solving such problems is the contribution of the thesis. The language used for synthesised definitions is a subset of the statically typed non-strict functional language programming Haskell.

An important idea of the approach is the extension of the language of definitions with special variables, called templates, representing unknown parts of a definition during synthesis. It is assumed that recursion in definitions can be realised using certain recursive higher-order definitions called schemas.

A state $(E, D)$ is a pair consisting of a set of equations and a definition; it represents a partially solved synthesis problem. The algorithm finds solutions by transforming states, starting from an initial state of the given equations $E$ and an initial definition $D$ defining $f$ as a template. The algorithm simplifies $E$, conjectures definitions for templates, and adds them to $D$. It makes recursive definitions by using a schema with template arguments. Synthesis is driven by the equations. If $E$ is unsatisfiable, the corresponding $D$ is ignored, if $E$ is satisfied by $D$ then $D$ is a solution; otherwise the state is transformed further.

Conjecturing definitions for templates is the crux of synthesis, and here the synthesis algorithm applies two auxiliary syntactic algorithms for certain polymorphic functions. Generalisation, an extension of Plotkin’s least general generalisation algorithm, constructs a non-recursive definition of a function from a set of examples. Factoring is a heuristic algorithm that transforms some recursive equations into a set of examples, thereby decomposing a synthesis problem into partly independent sub-problems. Factoring enables generalisation to be applied when it otherwise could not.

The synthesis algorithm is presented partly as a functional program and partly as semi-formal text with examples. Practical experience with an implementation of the algorithm is reported. Amongst the functions that the algorithm can synthesise is list sort, including an auxiliary list insert, when staring from a single example. It can also synthesise various higher-order functions.
Acknowledgements

I pursued a long and rather indirect path to arrive at the end result that is this thesis, and I am grateful to a number of people.

My supervisor Øystein Nystro participated in many long discussions, especially on logic program synthesis, and provided feedback on numerous drafts of the thesis. Being a headstrong and stubborn student I did not make his task an easy one. The colleagues in the Software Engineering Group provided a friendly working environment during the years in Trondheim.

From August to December 1996 I visited the Department of Computer Science at the University of Bristol, U.K. This stay was a turning point in my work, and I benefited greatly from talking to people I met there: John Gallagher took care of the administrative matters of my visit to Bristol and answered my questions on deductive synthesis. Tony Bowers, Christophe Giraud-Carrier and John Lloyd discussed machine learning and declarative programming with me. John Lloyd first made me consider Haskell and functional programming as the way forward. Chris Dorman and Ian Holey answered many of my questions on Haskell. Also, the social atmosphere in Bristol was very good.

I thank my friends and family for encouragement and inspiration.

I could not have carried out my research without access to free software provided by researchers and idealist hackers: Hugs and GHC for compiling Haskell were essential, and without GNU Emacs and LATEX work would have been much less pleasant.

Financial support for the major part of my work was a research fellowship from the Department of Computer and Information Science at the Norwegian University of Science and Technology. The department provided an office and other practical matters throughout the time I worked on this thesis, also after my fellowship ended, and gave me a flexible part-time job at short notice. My stay at Bristol was supported by a travel grant from Det Norske Veritas Research. I am especially grateful to my current employer the Norwegian Computing Center and to Steinar Kristoffersen for allowing me to spend part of my working hours on the final write-up.
# Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.2 Illustration</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1.3 Overview</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>Program representation</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>2.1 Programming languages</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>2.1.1 Functional programming</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>2.1.2 Higher-order functions for recursion</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>2.1.3 Meta-programming</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>2.2 Programs</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>2.2.1 Kinds of programs</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>2.2.2 Program properties</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>Survey of program synthesis research</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>3.1 Program synthesis</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>3.2 Deductive synthesis</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>3.2.1 The proofs-as-programs approach</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>3.2.2 The transformational approach</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>3.2.3 Miscellaneous work</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>3.3 Inductive synthesis</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>3.3.1 Generalisation and Plotkin’s algorithm</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>3.3.2 Logic program synthesis</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>3.3.3 Functional program synthesis</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>3.4 Discussion</td>
<td>31</td>
</tr>
<tr>
<td>4</td>
<td>The synthesis problem</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>4.1 The object-language</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>4.2 Schemas as higher-order functions</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>4.2.1 The list schemas</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>4.2.2 Discussion</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>4.3 Problem formulation</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>4.4 Anatomy of solutions</td>
<td>40</td>
</tr>
</tbody>
</table>
5 Synthesis by transformation 41
5.1 Template expressions 41
5.2 The meta-language 43
5.3 Representing equations and definitions 45
5.4 Overview of the synthesis algorithm 46
5.5 Main logic 47
5.6 Refinement and simplification 48
5.7 Conjecturing definitions 51
5.7.1 Recursive definitions using schemas 52
5.7.2 Enumerating non-recursive definitions 53
5.8 Canonical expressions 54

6 Syntactic methods 57
6.1 Restricted polymorphism 57
6.2 Motivation for generalisation 60
6.2.1 Reversing computations 60
6.2.2 Further examples of generalisation 62
6.3 Generalising example equations into definitions 66
6.4 Motivation for factoring 69
6.4.1 Constant matching and rewriting 69
6.4.2 Further examples of factoring 71
6.5 Factoring recursive equations into example equations 76
6.5.1 Factoring standard fold equations 76
6.5.2 Factoring unfold equations 78

7 Examples and initial analysis 82
7.1 Examples of definition synthesis 82
7.2 Towards formal analysis of the algorithm 86

8 Practical experience 89
8.1 Implementation 89
8.2 The system at work 91

9 Conclusion 94
9.1 Contributions 94
9.2 Related work 95
9.3 Applications 99
9.4 Critique 100
9.5 Further work 101

A Example sets and synthesised definitions 104

References 112
# List of figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Recursive higher-order functions involving lists.</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Plotkin's least general generalisation algorithm.</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Assumptions of ILP.</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>Feher's divide-and-conquer algorithm schema.</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>Questions and answers in DIALOGS session synthesising sort.</td>
<td>27</td>
</tr>
<tr>
<td>3.5</td>
<td>Clause set program for intersection predicate.</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>Zhu and Jin's schema.</td>
<td>31</td>
</tr>
<tr>
<td>4.1</td>
<td>Object-language syntax.</td>
<td>34</td>
</tr>
<tr>
<td>4.2</td>
<td>Functions defined by object-language Prelude.</td>
<td>35</td>
</tr>
<tr>
<td>4.3</td>
<td>List schemas.</td>
<td>36</td>
</tr>
<tr>
<td>4.4</td>
<td>Miscellaneous schemas.</td>
<td>36</td>
</tr>
<tr>
<td>4.5</td>
<td>Schema hierarchy.</td>
<td>37</td>
</tr>
<tr>
<td>4.6</td>
<td>Reductions for lists and natural numbers.</td>
<td>37</td>
</tr>
<tr>
<td>4.7</td>
<td>Bird and Wadler's 38 standard functions defined with the list schemas.</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>Syntax of template expressions.</td>
<td>42</td>
</tr>
<tr>
<td>5.2</td>
<td>Functions synthesise and trans.</td>
<td>48</td>
</tr>
<tr>
<td>5.3</td>
<td>Functions simplify and refine.</td>
<td>49</td>
</tr>
<tr>
<td>5.4</td>
<td>Function conjecture.</td>
<td>51</td>
</tr>
<tr>
<td>5.5</td>
<td>Equivalent non-canonical and canonical expressions.</td>
<td>56</td>
</tr>
<tr>
<td>6.1</td>
<td>Classification of sample functions.</td>
<td>59</td>
</tr>
<tr>
<td>6.2</td>
<td>Generalisation algorithm.</td>
<td>67</td>
</tr>
<tr>
<td>6.3</td>
<td>Factoring standard fold equations.</td>
<td>77</td>
</tr>
<tr>
<td>6.4</td>
<td>Factoring of unfold equations with reduction tail.</td>
<td>79</td>
</tr>
<tr>
<td>6.5</td>
<td>Factoring of gunfold equations with reduction tail.</td>
<td>79</td>
</tr>
<tr>
<td>8.1</td>
<td>Core equation simplification in Haskell.</td>
<td>91</td>
</tr>
<tr>
<td>8.2</td>
<td>The system at work: identifiers, problem data, and statistics.</td>
<td>92</td>
</tr>
<tr>
<td>9.1</td>
<td>Examples and synthesised definitions using Zhu and Jin's schema.</td>
<td>96</td>
</tr>
<tr>
<td>9.2</td>
<td>Hutchinson's directly recursive definition and a version using gunfold.</td>
<td>97</td>
</tr>
<tr>
<td>9.3</td>
<td>Hamfelt and Nilsson's schema, synthesis of append.</td>
<td>98</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The goal of computation is the emulation of our synthetic abilities, not the understanding of our analytic ones.
—ALAN J. PERLIS

The essence of my thesis is this: simple recursive functions, defined as functional programs, can be constructed automatically from a few examples of applications of the functions, that is, examples of arguments and corresponding results and a type declaration. For the list sort function an example could be this:

\[
\text{sort } [3, 1, 4, 0, 2] = [0, 1, 2, 3, 4]
\]

The contribution of the thesis is an algorithm for doing such automatic construction, in particular it builds a definition of list sort from the above example and a type declaration. Certain assumptions are made regarding the structure of recursion.

The functional programming language Haskell is used informally in the introduction. I refer readers not familiar with statically typed functional programming to §2.1.

1.1 Motivation

Here is a set of examples partially specifying the list insert function: insertion of a new element into a sorted list of elements.

\[
E_0: \begin{cases} 
\text{insert } 0 \; [] & \equiv [0] \\
\text{insert } 4 \; [3, 5, 6, 7] & \equiv [3, 4, 5, 6, 7]
\end{cases}
\]

Each example is an equation relating one combination of arguments to a result for function \text{insert}. Roughly, the problem of synthesis from examples is to automatically find a recursive definition

\[
\text{insert } y \; x \; s = \ldots
\]
such that left-hand sides equals right-hand sides in \( E_0 \). This definition is called a solution to the problem. Solutions can use standard pre-defined functions and normal programming constructs such as application, variable binding, recursion, and if-statements. Clearly, synthesis involves searching some kind of program space. To ensure that synthesis terminates one can assume that this space is limited in size, for example, by some syntactic restriction on program size and a finite alphabet. Depending on the program space, the problem formulation, and the way search is performed a solution may not exist in the space or it may not be found if it exists. The synthesis from examples problem has a simple formulation, yet it is non-trivial; this makes it an interesting problem.

There are several reasons to study synthesis from examples. Consider the examples \( E_0 \) above, but imagine \texttt{insert} replaced with \texttt{foobar}. It is likely that a programmer looking at the examples for \texttt{foobar} would guess that the function is the list insert function, given the premise that the function is recursive. As programmers we have intuitions about and make assumptions regarding recursive functions. Synthesis from examples is an attempt to partially capture such knowledge, and thereby automating its application, either to assist the programmer or to use the knowledge in an autonomous system.

Here is an alternative specification \( S \) of list insert, this time written in predicate logic.

\[
S : \quad \forall y, xs : \exists xs' : \begin{cases} 
(sorted(xs) \Rightarrow insert(y, xs, xs') \wedge sorted(xs')) \\
\wedge \forall z : (member(z, xs) \Rightarrow member(z, xs')) \\
\wedge member(y, xs') \\
\wedge length(xs, n) \wedge length(xs', n + 1)
\end{cases}
\]

where \( y \) is an integer, and \( xs \) and \( xs' \) are integer lists and I assume that that predicates \textit{sorted}, \textit{member} and \textit{length} are defined using a suitable theory. Specification \( S \) is an attempt at an exact description of \texttt{insert}.\footnote{While I believe \( S \) to be an exact description of \texttt{insert}, this is not crucial to my argumentation as long as an exact definition is more complicated to write than the examples \( E_0 \).} The following is a definition of the \texttt{insert} function written in Haskell.

\[
D_0 : \\
\begin{array}{l}
insert \ y \ [1 = [y]] \quad \text{-- Used when list argument is empty list.} \\
insert \ y \ (x:xs) \quad \text{-- Used with non-empty list arguments.} \\
\quad = \text{ if } y \leq x \text{ then} \\
\quad \quad y:x:xs \quad \text{-- } \texttt{'}:\texttt{'} \text{ is (infix) list cons.} \\
\quad \text{ else} \\
\quad \quad x:insert \ y \ xs \\
\end{array}
\]

Predicate \texttt{insert}(x, xs, xs') from specification \( S \) defines a function where \( x \) and \( xs \) are the arguments and \( xs' \) is the result, and the same function \texttt{insert} is defined by \( D_0 \), though it can be implemented differently from \( D_0 \). Writing the complete specification \( S \) is comparable in difficulty to writing the definition \( D_0 \), and both are significantly more complicated that specification \( E_0 \).
1.2 Illustration

For certain kinds of specifications there are techniques to automatically synthesise a corresponding definition realising the specifications. For specifications like $S$ above this is a difficult problem; for specifications like $E_0$ the synthesis machinery is simpler, though the synthesised solution may not define the function the specification writer had in mind. A system synthesising definition $D_0$ from example $E_0$ can be built using the synthesis algorithm presented in this thesis.

1.2 Illustration

In this section I discuss the use of a hypothetical program synthesis system to solve a synthesis problem. I show how different formulations of the problem and restrictions on the language of definitions affect the outcome, and this is used to argue for a set of requirements to the synthesis system. Design of the internal workings of the system is not discussed here; this is the subject of later chapters.

Consider the set of equations given before:

$$E_0: \begin{align*}
\text{insert } 0 & \; [\] \quad = \; [0] \\
\text{insert } 4 & \; [3, 5, 6, 7] \quad = \; [3, 4, 5, 6, 7]
\end{align*}$$

It is trivial to come up with definitions of \texttt{insert} that compute these examples. Here are three:

$$D_1: \begin{align*}
\text{insert } 0 & \; [\] \quad = \; [0] \\
\text{insert } 4 & \; [3, 5, 6, 7] \quad = \; [3, 4, 5, 6, 7] \\
\text{-- Equivalent to} \\
\text{-- insert } x & \; [\] \; \mid \; x = 0 \quad = \; [x] \\
\text{-- insert } ... 
\end{align*}$$

$$D_2: \begin{align*}
\text{insert } y & \; [\] \quad = \; [y] \\
\text{insert } y & \; [x_1, x_2, x_3, x_4] \quad = \; [x_1, y, x_2, x_3, x_4]
\end{align*}$$

$$D_3: \begin{align*}
\text{insert } y & \; [\] \quad = \; [y] \\
\text{insert } y & \; (x:xs) \quad = \; x:y:xs
\end{align*}$$

There are problems with these definitions. Definition $D_1$ works only for the two specific cases given in $E_0$, and fails otherwise. Definition $D_2$ defines \texttt{insert} correctly for all empty lists and also when the list is of length four and $y \leq x_4$; otherwise it fails or gives the wrong result. The left-hand side patterns of the definitions are worth a closer look. The first definition hard-codes the values of the argument through integers and the second hard-codes the length of the list arguments. Definitions $D_1$ and $D_2$ partial; they are only defined when the list argument has length 0 or 4—an unlikely kind of definition. Definition $D_3$ is a total function, but even though it never fails it may also give a wrong result. All three definitions could be trivially rejected if \texttt{insert} is required to be recursive.

Writing recursive definitions requires special care. The comments about patterns apply also to recursive definitions. Furthermore, recursive functions must have a certain structure—base-cases and recursive cases—to work correctly. Termination can be
achieved by ensuring that the recursion argument is reduced to a strictly 'smaller' value for each recursive call, for example, a shorter list like in \( D_0 \). Here are a few invalid versions of recursive cases (\( t \) and \( s \) denote expressions).

\[
\begin{align*}
\text{insert } y \hspace{1pt} xs &= \ldots \text{insert } t \hspace{1pt} xs \ldots \\
\text{insert } y \hspace{1pt} xs &= \ldots \text{insert } t \hspace{1pt} (\ldots \hspace{1pt} xs) \ldots \\
\text{insert } y \hspace{1pt} xs &= \ldots \text{insert } t \hspace{1pt} [] \ldots \\
\text{insert } y \hspace{1pt} xs &= \ldots \text{if} \hspace{1pt} \text{false} \hspace{1pt} \text{then} \hspace{1pt} \text{insert } t \hspace{1pt} s \hspace{1pt} \text{else} \hspace{1pt} []
\end{align*}
\]

In the first two recursion goes on forever and the last two are trivial in the sense that they have equivalent non-recursive versions. The conclusion is that recursion must have a certain structure with base- and recursive cases, a reduced recursion argument, and recursion must be non-trivial. These requirements can partly be ensured by using special higher-order definitions to take care of the recursion; more on this in later chapters.

An operation like \text{insert} also makes sense for other datatypes, for example, lists of characters.

\[
\text{insert 'b' ['a', 'c']} = ['a', 'b', 'c']
\]

With this example in addition to the examples in \( E_0 \) definitions like \( D_0 \) are no longer valid since they are specific to integer lists. When the examples use several datatypes solutions must be general enough to work for all given datatypes. An alternative to giving examples involving several datatypes is to give a type declaration. I assume that a correct type declaration is available as part of the examples. Using Haskell syntax this could look as follows.

\[
E'_0: \quad \begin{array}{l}
\text{insert :: Ord } a \Rightarrow a \rightarrow [a] \rightarrow [a] \\
\text{insert } 0 \hspace{1pt} [] &= \hspace{1pt} [0] \\
\text{insert } 4 \hspace{1pt} [3, 5, 6, 7] &= \hspace{1pt} [3, 4, 5, 6, 7]
\end{array}
\]

The first line is the type declaration, or just the type, and it consists of two parts. Informally the first part, \text{Ord } a \Rightarrow a \rightarrow [a] \rightarrow [a], specifies that \( a \) is a datatype where a total order \( '<' \) is defined on all pairs of data elements—it is similar to a quantifier on \( a \). The second part, \( a \rightarrow [a] \rightarrow [a] \), specifies that \text{insert} takes an element of type \( a \), and a list of type \( a \), and returns a list of the same type.

Definition \( D_0 \) is not a valid solution to \( E'_0 \) above since \( D_0 \) defines a function with type, \text{Int } \rightarrow [\text{Int}] \rightarrow [\text{Int}], which is too specific. Consider the following untyped examples.

\[
E_1: \quad \begin{array}{l}
\text{insert } 0 \hspace{1pt} [] &= \hspace{1pt} [0] \\
\text{insert } 8 \hspace{1pt} [3, 5, 6, 7] &= \hspace{1pt} [3, 5, 6, 7, 8]
\end{array}
\]

A valid solution to the synthesis problem \( E_1 \) is this definition,

\[
D_4: \quad \begin{array}{l}
\text{insert :: } a \rightarrow [a] \rightarrow [a] \\
\text{insert } x \hspace{1pt} [] &= \hspace{1pt} [x] \\
\text{insert } x \hspace{1pt} (y:ys) &= y: \text{insert } x \hspace{1pt} ys
\end{array}
\]

\section*{CHAPTER 1. INTRODUCTION}
that always inserts an element at the end of the list. Note that the type given in $D_4$

does not place restrictions on the kind of elements involved. In sum, providing a type

like that of $E_0'$ prevents some unwanted solutions.

Last, consider again the examples $E_0'$ and the following valid solution.

$D_5$:

\[
\begin{align*}
\text{insert} & : \text{Ord } a \Rightarrow a \rightarrow [a] \rightarrow [a] \\
\text{insert} \ y \ x & = \text{sort } (y:xs) \\
\text{sort} & : \text{Ord } a \Rightarrow [a] \rightarrow [a] \\
\text{sort} \ x & = \ldots \ -- \ \text{Some definition of list sort.}
\end{align*}
\]

Definition $D_5$ is a valid solution, but it is too complicated with its auxiliary function

sort when in fact a simpler solution like $D_0$ would do. One possible heuristic a

synthesis system can use is to prefer simpler solutions over complex solutions, for

example, preferring syntactically small definitions with few recursive calls. I require

that the number of recursive calls be given; alternatively this number could be found

by the system.

Note that there may not be any solutions to a synthesis problem, for example, this

set of examples:

$E_2$:

\[
\begin{align*}
\text{bogus} & : [a] \rightarrow [a] \\
\text{bogus} \ [1] & = [1] \\
\text{bogus} \ [2] & = [2, 2]
\end{align*}
\]

From its type function $\text{bogus}$ is known to be polymorphic, that is, it does not inspect

the actual element values of its list argument, only the list structure. This means

that it is impossible to define a function $\text{bogus}$ so that it distinguishes between the

arguments in the examples. (There is, however, a solution for a relation computing the

examples.) This means that writing examples requires special care. To simplify matters

the synthesis algorithm ignores the possibility of erroneous examples, and makes no

attempt at second-guessing the specification writer or ruling out impossible cases like

$E_2$.

The synthesis algorithm I present in later chapters can synthesise the definition of

$\text{insert}$ using only one example from $E_0$:

\[
\begin{align*}
\text{insert} & : \text{Ord } a \Rightarrow a \rightarrow [a] \rightarrow [a] \\
\text{insert} \ 4 \ [3, 5, 6, 7] & = [3, 4, 5, 6, 7]
\end{align*}
\]

though this requires more search. An important part of the algorithm is a syntactic

method that tries to construct a general computation by replacing constants with

variables:

\[
\text{insert} \ 4 \ [5, 6, 7] \rightarrow [4, 5, 6, 7] \mid 4<5
\Rightarrow \text{insert } x \ (y:ys) \rightarrow (x:y:ys) \mid x<y
\]

In this first look at synthesis from example we have seen that specification of recursive

functions by examples is ambiguous, but that some reasonable assumptions about type,

the language used and recursion increase the chance of finding a satisfactory

solution.
1.3 Overview

§2 discusses representation of programs, properties of programs, and kinds of programs, preparing later discussion of program synthesis. An introduction to functional programming and Haskell is also given. §3 surveys current research in program synthesis and points out lacking work. It defines the term 'program synthesis', and surveys several kinds of such synthesis.

In §4 I give a formulation of the problem of synthesis from examples based on a certain language and a discussion of schemas. The core of the thesis is §5 and §6 where I present my algorithm for synthesis of recursive functional programs from examples. The former chapter describes the main synthesis algorithm, based on transforming equation sets and definitions, together with material on language and representation. Two specialised auxiliary algorithms related to conjecturing definitions—generalisation and factoring—appear in the latter chapter. §7 consists of several examples of definition synthesis and an initial discussion of the algorithm's formal properties. Practical experience with an implementation of the synthesis algorithm is reported in §8, with details in the appendix, showing that implementation is possible. In §9 I review my results and give directions for future work.
Chapter 2

Program representation

This chapter is about how programs are represented and about different kinds of programs and their properties. Representation is about language and §2.1 discusses functional programming and recursive higher-order functions. In §2.2 programs and other formalised knowledge for synthesis such as examples and algorithm schemas are introduced.

2.1 Programming languages

Programming languages exist in a wide variety, motivated by theoretical properties or practical applications. While the diversity and complexity of languages make it difficult to classify them one, loosely knit group of languages, called declarative languages, have the following common characteristics.

- They are relatively close to mathematical or logical language making formal reasoning about the languages and their properties simpler.

- Their expressiveness and powerful abstraction mechanisms make them good implementation languages for problems with complicated data structures.

Here are some examples of declarative programming languages. Logic programming languages: Prolog, Gödel [48, 30]; functional programming languages: Haskell, Miranda, Standard ML, Scheme [13, 81, 69, 82, 57, 2]; and functional logic programming languages: Escher, Curry [49, 28]. Non-declarative languages include C and Java. In later chapters I use Haskell so the following sections focuses on it and functional programming.

2.1.1 Functional programming

Haskell is a pure, statically typed, and non-strict functional programming language [69]. Pure means that the language does not have observable side-effects; more on the other
adjectives below.\footnote{Haskell has undergone a number of revisions and at the time of writing the current version is Haskell 98. All code shown in this thesis is valid in both Haskell 1.4 and Haskell 98.} I assume that the reader is familiar with the basics of lambda-calculus [70, §2]; the purpose of this section is to enable the reader to understand Haskell programs.

In Haskell, like in most other functional languages, every value can be seen as a function (of zero or more arguments). Some quite standard terminology to be used throughout this thesis: Programs define functions, name them using identifiers and bind them to variables. A function returning a Boolean is a predicate. A definition is a part of a program defining one function. For example here is a definition of the Fibonacci function using identifier \texttt{fib}.

\begin{verbatim}
fib :: Int \rightarrow Int
f fib 0 = 1
f fib 1 = 1
f fib n = fib (n-1) + fib (n-2)
\end{verbatim}

This definition consists of a type-declaration, on the first line, and three (program) equations where the part on the right-hand side of '=' is called an expression. Haskell has first-class functions so lambda-functions are also expressions, for example, \((\lambda n.n+1)\) is written as \((\langle n \rightarrow n+1 \rangle)\).

Consider a function \texttt{take} with type \texttt{Int \rightarrow [a] \rightarrow [a]}. The constant type constructor \texttt{Int} stands for the integers. Symbol \texttt{a} is a type variable and \texttt{[a]} is the type of a list of elements of type \texttt{a}, made using the list type constructor \texttt{[ ]}. The function type constructor is \texttt{-} and it is right-associative. This means that \texttt{take} takes an integer argument and returns a new function which again takes a list argument and returns a list argument.

A function \(f\) with a (most general) type containing type variables is called a polymorphic function, otherwise \(f\) is monomorphic. Arguments corresponding to type variables, like the list elements in \texttt{take}'s second argument, are abstract and may not be inspected by polymorphic functions. Here is the definition of \texttt{take}:

\begin{verbatim}
take :: Int \rightarrow [a] \rightarrow [a]
take 0 _ = []
take _ [] = []
take n (x:xs) | n>0 = x : take (n-1) xs
              | _   = error "Prelude.take: negative argument"
\end{verbatim}

Let the arguments to \texttt{take} be \(n\) and \(xs\). Then \((\texttt{take n xs})\) evaluates to a list consisting of the \(n\) first elements of \(xs\), or all of \(xs\) if \(n\) is larger than its length. A technique called pattern matching is used to restrict the arguments of individual equations in the definition. The first equation applies to all cases where the integer is 0 and the list can be any list; pattern \_ matches anything. The third equation applies to cases where the list argument is non-empty; \(x\) and \(xs\) are bound to head and tail of the list and can be used on the right-hand side. This equation also uses a guard, \(n>0\), to further restrict the use of the equation.
2.1. PROGRAMMING LANGUAGES

The datatypes Int and list are built-in types in Haskell together with other types and a number of functions, including take. They are part of a standard module visible to all Haskell programs, called the Haskell Prelude. In addition Haskell has a number of other library modules.

The list datatype has two value constructors, or just constructors: the empty list, ‘[]’ of type [a], and list cons, ‘:’ of type a -> [a] -> [a]. Haskell has syntactic sugar for lists: 1:2:[] is equivalent to [1, 2]. Variables and functions start with lower-case letters and constructors start with upper-case letters, except infix operators like ‘+’. This holds both for regular expressions and type expressions. Some more useful Haskell syntax: If f is a regular prefix identifier then ‘f’ is an infix version of f. Likewise, (+) is a prefix version of +, and (0:) is a prefix function adding 0 at the start of an integer list.

Haskell has a non-strict or lazy semantics which roughly means that functions only evaluate the part of the arguments they need to compute the result.\(^2\) In the case of take above the integer argument is always evaluated since its value must be compared to 0, but only the n first elements of the list argument are evaluated. This makes working with infinite lists simple:

\[
\text{take } 5 \text{ [0 ..]} \Rightarrow [0, 1, 2, 3, 4]
\]

where [0 ..] is a Haskell notation for the list of all natural numbers, an infinite list. An example of a function completely evaluates its arguments is integer addition; addition is said to be strict in its arguments.

Haskell and the strict functional language Standard ML have near-identical type regimes based on the Hindley-Milner type system [56]. The type-inference algorithm used in Haskell can automatically deduce the type of most definitions, including all definitions shown in this thesis. The system guarantees that type-errors cannot happen at run-time; this regime is called static typing. Haskell extends the Hindley-Milner system with a form of polymorphism called type classes or ad-hoc polymorphism [26, 69]. Here is an example of a definition using type classes:

\[
\begin{align*}
\text{elem} :: \text{Eq } a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool} \\
\text{elem} y [ ] & = \text{False} \\
\text{elem} y (x : xs) & = y == x \mid \mid \text{elem} y xs \quad \text{-- ‘|’ is logical or.}
\end{align*}
\]

Eq a means that values supplied for a must have the predicate ‘==’ defined on them. Another example is in §1.2. To distinguish between ad-hoc polymorphism and regular polymorphism the latter is sometimes called parametric polymorphism. I only use a restricted form of type classes in this thesis.

2.1.2 Higher-order functions for recursion

In Haskell iteration is implemented using using recursion. Since functions are first-class a recursive function \(f\) can be defined once and then used with different arguments to

\(^2\)A non-strict semantics is sometimes called lazy, but this is technically incorrect; lazy evaluation is a technique which implements non-strict semantics [70, §2.5.4].
implement different recursive functions. In this case \( f \) is called a higher-order function, and it can be seen as an abstraction of the structure of recursion over a datatype.

In Haskell and other functional languages the existence of standard definitions for recursive higher-order functions, especially involving list data, is essential for practical programming.\footnote{Meijer et al. \[53\] compare direct recursion to the goto of imperative programming and argue in favour of ‘structured functional programming’ using recursive higher-order functions.} Fig. 2.1 shows six definitions of such functions. The first five definitions are from the Haskell Prelude. The sixth, \texttt{unfold}, is a less common, but nevertheless useful definition \[54,33,24\]. To understand what these functions do it is instructive to look at the following equalities (inspired by Bird and Wadler’s discussion \[13, \S 3.5\]).

\[
\begin{align*}
\text{map} \ f \ [x_1, x_2, \ldots, x_n] & = [f \ x_1, f \ x_2, \ldots, f \ x_n] \\
\text{foldr} \ f \ a \ [x_1, x_2, \ldots, x_n] & = x_1 \ \text{foldr} \ f \ a \ (x_2 \ \text{foldr} \ f \ a \ (\cdots (x_n \ \text{foldr} \ f \ a \ a) \cdots)) \\
\text{foldl} \ f \ a \ [x_1, x_2, \ldots, x_n] & = (\cdots ((a \ f \ x_1) \ f \ x_2) \cdots) \ f \ x_n
\end{align*}
\]

Function \texttt{map} applies a function to every element in a list. The two functions \texttt{foldr} and \texttt{foldl} are best explained by inspecting the equalities. The name \texttt{foldr} stands for ‘fold right’, \texttt{foldl} for ‘fold left’; note how the brackets group to the right and to the left on the right-hand side for these two functions. Both return \( a \) if given the

---

\( \texttt{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\texttt{map} \ f \ [] = [] \\
\texttt{map} \ f \ (x:xs) = f \ x \ : \ \texttt{map} \ f \ x \ $s
\]

\( \texttt{foldr} :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a \\
\texttt{foldr} \ f \ [x] = x \\
\texttt{foldr} \ f \ (x:xs) = f \ x \ (\texttt{foldr} \ f \ x \ $s)
\]

\( \texttt{foldl} :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a \\
\texttt{foldl} \ f \ [x] = x \\
\texttt{foldl} \ f \ (x:xs) = \texttt{foldl} \ f \ (f \ x \ x \ $s)
\]

\( \texttt{unfold} :: (a \rightarrow b) \rightarrow (a \rightarrow \text{Bool}) \rightarrow (a \rightarrow a) \rightarrow a \rightarrow [b] \\
\texttt{unfold} \ f \ p \ g \ x \\
| \ p \ x = [] \\
| \texttt{otherwise} = f \ x \ : \ \texttt{unfold} \ f \ p \ g \ (g \ x) \quad -- \texttt{otherwise} = \text{True}
\]
empty list, and in this case $f$ is never evaluated. Functions \texttt{foldr1} and \texttt{foldl1} are as \texttt{foldr} and \texttt{foldl}, except that they do not have the second argument and they fail on the empty list. The first five definitions always terminate, provided that $f$ and $a$ (if present) terminate and the list is finite.

Function \texttt{unfold} stands out as having a recursion argument not specific to lists. It uses a function $g$ to reduce the recursion argument to a ‘smaller’ value for each recursive call and a predicate $p$ to determine if a base-case has been reached; $g$ is called a \textit{reduction} and $p$ a \textit{condition}. Both the reduction and the condition have to be chosen carefully to ensure termination. Function \texttt{unfold} always returns a list, but, unlike the other functions, it can recurse over non-list data.

\textbf{Ex. 2.1} Here is how some standard Prelude functions could be defined using the definitions of Fig. 2.1.

\begin{verbatim}
reverse :: [a] -> [a]
reverse = foldl (\xs x -> x:xs) []

length xs :: [a] -> Int
length xs = foldr (\_ n -> n+1) 0 xs
\end{verbatim}

More examples of such definitions appear in later chapters, especially §7 and in the appendix. ◊

Of the first three functions in Fig. 2.1 only \texttt{foldr} is needed; functions \texttt{mapr} and \texttt{foldl} are just useful special cases of \texttt{foldr}. The reason that the special cases are included is that it is simpler to use the special cases than to use \texttt{foldr} all the time; Ex. 2.2 below shows the kind of higher-order ‘trick’ that must be used to define \texttt{foldl} using \texttt{foldr}.

\textbf{Ex. 2.2} Meijer and Jeuring show how to define the Prelude functions \texttt{take} and \texttt{foldl} using \texttt{foldr} by exploiting the power of higher-order functions; this example is taken from their lecture notes [55]. Since the first argument of \texttt{take} :: \texttt{Int} -> [a] -> [a] is not fixed, \texttt{take} cannot be defined in terms of \texttt{foldr} directly. The authors’ trick is to swap the arguments:

\begin{verbatim}
take' :: [a] -> Int -> [a]
take' [] = \_ -> []
take' (x:xs) = \n -> case n of 0 -> []; n+1 -> a : take' xs n
\end{verbatim}

This leads to a \texttt{foldr}-based definition.

\begin{verbatim}
take :: Int -> [a] -> [a]
take n xs
    = foldr (\x h -> \n -> case n of 0 -> []; n+1 -> x : h n)
           (\_ -> []) xs n
\end{verbatim}

Here is the definition at work. Application is left-associative, and I introduce abbreviations for the first two arguments to \texttt{foldr}:
take \( m \ (y; ys) \)
\[ \rightarrow \operatorname{foldr} \ f \ a \ (y; ys) \ m \]
\[ = \operatorname{foldr} \ ((\_ \ h \rightarrow \_ \rightarrow \text{case } n \rightarrow 0 \rightarrow []; \ n+1 \rightarrow x : h \ n) \ (\_ \rightarrow [])) \ (y; ys) \ m \]
\[ \rightarrow ((\_ \ h \rightarrow \_ \rightarrow \text{case } n \rightarrow 0 \rightarrow []; \ n+1 \rightarrow x : h \ n) \ y \ (\operatorname{foldr} \ f \ a \ ys) \ m \]
\[ \rightarrow (\text{case } n \rightarrow 0 \rightarrow []; \ n+1 \rightarrow y : h \ n) \ (\operatorname{foldr} \ f \ a \ ys) \ m \]
\[ \rightarrow \text{case } m \rightarrow 0 \rightarrow []; \ n+1 \rightarrow y : (\operatorname{foldr} \ f \ a \ ys) \ n \]

The same method is used to define \( \text{foldl} \).

\[
\text{foldl} :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b
\]
\[
\text{foldl} \ f \ b \ xs
\]
\[
= \operatorname{foldr} \ ((a \ h \rightarrow b \rightarrow h \ (f \ b \ a)) \ (b \rightarrow b) \ xs \ b
\]

The opposite, defining \( \operatorname{foldr} \) using \( \text{foldl} \), is impossible since \( \text{foldl} \) is strict in the tail of its list argument [34].

Ex. 2.3 Function \( \text{map} \) can be implemented using \( \text{unfold} \) as follows.

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]
\[
\text{map} \ f \ xs = \text{unfold} \ (f, \text{head}) \ \text{null} \ \text{tail} \ xs
\]

When defining other functions alternative reductions to \( \text{tail} \) are \( \text{tail} \cdot \text{tail} \), etc. Choosing \( \text{id} \) for the reduction leads to non-termination since \( \text{id} \) does not reduce the argument. Function \( \text{map} \) can also be defined using \( \text{foldr} \). Observe that \( \text{foldr} \) cannot define \( \text{unfold} \): the latter allows recursion to be selected in a programmer-defined way using the condition argument \( p \).

All recursive functions defined so far have been linear, but more complicated functions can be defined by using several recursive higher-order functions. The indirect recursion introduced by recursive higher-order functions can be eliminated using, for example, partial evaluation [40]. On the other hand this form of structured recursion lead to program that are amenable to certain optimisations [55].

2.1.3 Meta-programming

The characteristics of declarative languages mentioned at the start of §2.1 means that they are particularly suited for representing and analysing a fundamentally important kind of data, namely programs. The analysis and manipulation of program-data by other programs is called meta-programming.

Meta-programming involves a representation of program-data, the object-level, and modifying and evaluating program-data, the meta-level. In languages without static type checking it is possible to convert values from the object-level into expressions at
the meta-level and evaluate them using the built-in evaluator. This makes it simple to
do meta-programming in for example Lisp and Prolog. How to do this conveniently
and efficiently in a statically type language without compromising type-safety is a
research issue. Gödel [30] is the strongly typed language that best addresses the meta-
programming issue, providing the programmer with a full library and datatypes for
building and running object-programs.

2.2 Programs

This section is about programs and specifications, and relations between them. The
terminology I introduce below allows me be more precise than in the introduction
(§1), but it is adapted to my needs and I do not make claims about its general usefulness.
Turski and Maibaum [83] take an advanced and thorough approach to the same
material.

2.2.1 Kinds of programs

An algorithm is a finite and terminating procedure in mathematical language for com-
puting a non-trivial relation or function.⁴ A program is a formal language description
of one or more algorithms, executable by some machine. A specification is a description
in a formal but not necessarily executable language of an algorithm, possibly including
a background theory defining concepts used in the description. Note that a specification
may describe an algorithm exactly or only partly. There are many specification
languages in use, from predicate logic to more complicated special-purpose languages.

Ex. 2.4 Here is a specification of list sort.

\[(\forall u, v : \text{perm}(u, v) \land \text{sorted}(v) \Rightarrow \text{sort}(u, v))\]

This specification is written in predicate logic, where types have been left out. The
sorting relation \text{sort}(u, v) is defined here such that list \(v\) is the sorted version of list
\(u\). The background theory of the specification is assumed to contain definitions of
the relations \text{perm}(u, v), true if \(u\) is a permutation of \(v\), and \text{sorted}(v), true if the
elements of \(v\) are in ascending order. Observe that minor syntactic changes can make
the definition into a logic program that implements an inefficient sorting algorithm. ◊

An example is a simple kind of specification that describes (non-)membership in a
relation, or function seen as a relation. The example language allows only simple
statements of membership involving constants, (quantified) variables and the relation
itself. In particular no other relations can occur in an example. A positive example
states membership, a negative example states non-membership.

---

⁴Informally a non-trivial relation or function is one whose implementation requires recursion or
some kind of loop.
Ex. 2.5 A negative example for ternary relation insert (assume that \( z \) ranges over some element kind, and \( u \) over lists of such elements):  
\[ \forall z, u : (x, u, u) \not\in \text{insert} \]
which can be written as  
\[ \forall z, u : \neg \text{insert}(x, u, u). \]
Positive example for binary function \( \text{gcd} \):  
\[ \text{gcd}(15, 6) = 3 \]
or in the notation of the introduction,  
\[ \text{gcd} \ 15 \ 6 \ = \ 3. \]
The positive example describes the returned value for one argument-combination, whereas the negative example describes many non-instances of the relation. 

Specifications describe the effects of computations, but, not necessarily how computations are done. 
An algorithm schema, or just schema, is a description of a pattern of computation common to a class of algorithms.

Ex. 2.6 The following three-step procedure defines the divide-and-conquer algorithm schema. It is taken from Flener [21] (who calls it a design strategy).  
1. Divide a problem into sub-problems, unless it can be trivially solved.  
2. Conquer the sub-problems by solving them recursively.  
3. Combine the solutions to the sub-problems into a solution to the initial problem. 
An example of a divide-and-conquer algorithm is ‘size of binary tree’ where the three steps are:  
1. If the tree is a non-leaf node split it into its left and right sub-trees.  
2. Compute the size of each non-leaf sub-tree by a recursive call; leaves have size 1.  
3. Add the computed sizes. 

Well-written algorithm schemas help in the implementation of algorithms:  
- They restrict the pattern, or high-level structure, of algorithms to patterns that are sensible.  
- They decompose the implementation of an algorithm into smaller restricted parts. 
The usefulness of a set of schemas hinges on the assumption that they are relevant to the kinds of algorithms we want to implement. §4.2.1 presents a set of schemas for recursion over lists, implemented as recursive higher-order Haskell definitions. One of these schemas is \texttt{foldr}. 
2.2. PROGRAMS

2.2.2 Program properties

The properties of programs can be divided in two: Properties of programs-as-such, called internal properties, and properties of a program given its relation to another program or specification, called external properties. The latter is discussed in §3.1. Of internal properties semantics, syntax, and pragmatics are central. These properties are not independent.

Semantics of formal languages is a technical and well developed subject [25, 88]. A semantics of a program $P$ written in language $\mathcal{L}$ is an assignment of meaning to $P$, for example, in the form of a function, a set, or another mathematical concept amenable to formal analysis. Such an assignment should exist for all well-formed $\mathcal{L}$ programs. A semantics can be more or less abstract and emphasise different aspects of programs, for example, operational aspects. Semantics can be defined for other formal languages too, such as specification languages.

The syntax of programs describes their grammatical structure and concrete textual representation. Two programs with the same semantics can have different syntactic structure. There are several approaches to quantifying the syntactic structure of programs, sometimes called their structural complexity. Examples of such approaches, inspired by information theory, are the minimum description length principle and Kolmogorov complexity [47, 64]. Kolmogorov complexity gives a theoretical lower limit on the structural complexity of a program independent of the representation used.

Pragmatics is concerned with the efficiency of execution of programs on some formal machine. Efficiency, or computational complexity, can be measured as time or space usage of a program as function of the size of the problem it solves. The way recursion is used in a program affects its time complexity. Improving the time complexity of programs without changing their semantics is the subject of program transformation [66].

When synthesising definitions from examples syntactic methods like Plotkin’s least general generalisation (discussed in the next chapter) are more useful than semantic methods. Such synthesis generally has a number of alternative solutions, and here a form of structural complexity measure, for example, syntactic size, helps in selection. Efficiency of programs is not a concern in the following.
Chapter 3

Survey of program synthesis research

This chapter surveys program synthesis research and assesses the state of the art. A representative selection of approaches to program synthesis are studied, each with its problem formulation and synthesis algorithm.

Program synthesis can roughly be divided in two categories, deductive synthesis (§3.2) and inductive synthesis (§3.3) and this is reflected in this chapter. The survey focuses on inductive synthesis, especially inductive synthesis when the problem specification consists of examples as this is the kind of synthesis I study in later chapters (§4 and §5). I round off the survey with a discussion where I point to lacking work in current research (§3.4).

The research I survey almost exclusively involves declarative programs: functional programs or logic programs. The reasons for this are explained at the start of §2.1. There exists a number of other surveys using various labels for research related to that in this survey: A survey on automatic programming [76, 11, 6], on program synthesis [21, §1], and surveys of logic program synthesis [19, 61]. My survey, in addition to being more recent, includes systems with typed languages. A lot of research in artificial intelligence is relevant to program synthesis, but are not included in the survey: search, planning, knowledge representation, knowledge relevance.

3.1 Program synthesis

The terminology I introduce in this section serve to structure and focus the survey. This is useful since the surveyed research is quite diverse with idiosyncratic definitions and concepts.

The program synthesis problem is defined as follows.

Given

• a specification $S$ of some algorithm $A$ in the form of a relation or function;
3.1. **PROGRAM SYNTHESIS**

- a language $\mathcal{L}$;
- a background program library $B$.

Find a recursive program $P$ such that

- $P$ computes all of the relation or function described by $S$—called $P$ satisfies $S$;
- $P$ is in $\mathcal{L}$ and may refer to definitions in $B$.

The program $P$ is called a *solution* to the synthesis problem. Solutions are found by searching a space of programs. A *hypothesis* $H$ is any candidate for a solution during search. $H$ may or may not be a solution, and depending on the search technique it may not even be a program. Implicitly problem formulations must include the relevant languages and semantics of specifications and programs as they are needed to define satisfaction.

By assumption $S$ is *sound*, that is, it must either describe the same relation or function as $A$ or a subset of it (considering functions as relations). If $S$ completely describes $A$ it is called a *complete* specification, otherwise it is a *partial specification*. The first case is called a *deductive* synthesis problem, and the second case an *inductive* synthesis problem. When $S$ is a complete specification then program $P$ implements $A$. If $S$ is incomplete then $P$ implements *some* algorithm, but not necessarily $A$, and $A$ is called the *unknown relation* or *function*. Since $S$ is assumed to be sound issues like noise, uncertainty and specification errors are not considered here.

I refer to the process of solving program synthesis problems automatically as *program synthesis*. An important special case of the inductive synthesis problem is when the specification is in the form of a (finite) set of examples. This case of inductive synthesis is called *synthesis from examples*.

Consider a program that is the solution to a deductive synthesis problem. This program can be proved correct with respect to its specification using deductive reasoning. A program that in addition to satisfying a specification implements more, can only be obtained by inductive reasoning. In the latter case the program cannot be proved correct in classical logic, but inductive reasoning can still be justified and formalised.

Induction—going from particular observations to a general conclusion—is a well-known concept in science and philosophy,\footnote{Here induction is not to be confused with the mathematical technique of *proof by induction*.} and it serves as one model for explaining scientific reasoning\[18\]. (That this is a controversial model is not a major concern for the idealised kind of reasoning I consider here.) Formal reasoning techniques have been studied at great length in logic and mathematics, but so far deductive techniques have received far more attention than inductive techniques. Flach\[20\] investigates the logic of induction and argues that a formal logical foundation for induction can be constructed by replacing the conventional satisfaction-preserving semantics by a more general semantics that preserve other properties. One such property is explanatory power, for which Flach (p. 74) gives the intuition “the set of explanations of observed phenomena provided by a logical theory.” There is nothing informal about this kind of synthesis—it is just not classical in a logical sense.
It is desirable for a program synthesis system to operate autonomously, or with a minimum of human intervention. Writing just two simple and specific equations, like $E_i$ in the introduction (§1.2), and automatically getting a general working program, $D_0$, from such a system seems attractive, but at the same time it sounds like a dangerously unconstrained problem. The constructed program must perform the computations described in the specification, but how can we make sure the program returns the value we expect for argument combinations not described by the specification? We cannot—this is the price we pay for using an incomplete specification.

### 3.2 Deductive synthesis

Deductive synthesis is characterised by powerful specification languages since the specification and its background theory must provide a complete description of the algorithm in question. One example of such a specification language is predicate logic. Deductive synthesis systems mostly produce declarative programs, but some also produce imperative programs. If one considers compilation and optimisation to be deductive synthesis many different languages can be listed here.

Tradeoffs in deductive synthesis include the one between autonomy of the synthesis algorithm, that is, the degree of automation, and the complexity of the algorithm itself; trading time for quality of solutions; and conflicts between time and space complexity of solutions. In general deductive synthesis can be as hard as proving theorems.

#### 3.2.1 The proofs-as-programs approach

The proofs-as-programs approach is characterised by application of theorem-proving methods to find solution programs.

**Problem formulation** The specification typically defines an input-output relation in the form of an existence theorem; the background theory of the specification defines the predicates used in the theorem. This can be abstractly formulated as

$$\forall x : \exists z : (C(x) \Rightarrow Q(x, z))$$

where $C(x)$ is true only for acceptable inputs and $Q(x, z)$ holds only if $C(x)$ holds and output $z$ is acceptable for input $x$.

In general the input-output relation $Q(x, z)$ does not tell us how $z$ is computed, it only specifies what properties a solution must have. A program $P$ satisfies a specification $S$ if the theorem holds for the input-output relation $P$ computes.

**Algorithm** Synthesis proceeds by constructing a proof of the existence theorem and extracts a program from the proof, or, if useful, it does the two tasks in parallel. The proof must be carried out in a (sufficiently) constructive manner for a program to be obtainable from the proof.
Manna and Waldinger [51, 50] present a method for performing such proofs called the deductive tableau method which they use to synthesise functional programs. Armando et al. [4] seek to increase the level of automation in synthesis of recursive functional programs; they achieve this by using plans for how to carry out proofs.

Pauline-Mohring and Werner [67] take a different approach, based on type theory, that allows them to synthesise ML programs, that is, functional programs with polymorphic types and references.

The work on proofs-as-programs synthesis has mostly synthesised functional programs but Bundy et al. [16] adapt the method to synthesise pure logic programs.

### 3.2.2 The transformational approach

Transformational synthesis, as its name suggests, is concerned with methods for transforming a specification or program into another specification or program. Transformations preserve certain properties of programs, their semantics say, and advance towards a measurable goal such as increased execution efficiency of programs.

**Problem formulation** The specification, again an input-output relation, is an inefficient or non-operational description of an algorithm and its auxiliary definitions. The specification could even be an inefficient program, like the one in Ex. 24, that is to be transformed into an efficient one. This is attractive: one can write a program without regard to efficiency, but instead emphasising readability and ease of construction, and then have a synthesis system come up with an efficient version.

**Algorithms** A synthesis algorithm in the transformational approach applies a set of synthesis rules to a program or specification. The application of the rules in done in systematic way in order to advance towards the goal. Planning how to reach the goal and modifying programs to enable rule usage are part of the algorithm. Techniques for theorem-proving are applicable here.

Abstractly a transformation rule can be described as follows.

\[
\begin{align*}
\text{pat} & \xrightarrow{\text{cond}} \text{repl} \\
\end{align*}
\]

Here *pat* is a pattern that must match an expression—a part of a program—to be transformed, *cond* is a condition that must be satisfied for the transformation to take place, and *repl* is an expression replacing the expression matched by *pat* before introduction of bindings from the matching.

There are two kinds of transformation rules: vertical rules and lateral rules. The former rules transform a program from one abstraction level (or language) to a lower abstraction level. The latter rules transform a program within one level of abstraction.
Ex. 3.1 A simple lateral transformation rule that eliminates useless if-statements:

\[
\begin{align*}
\text{if } b \text{ then } e \text{ else } e & \Downarrow \text{ defined}(b) \\
& e
\end{align*}
\]

Here \text{defined}(b) is true if \text{b} is a defined Boolean expression. \diamond

An early example of this approach is Burstall and Darlington’s unfold/fold transformation system [17]. It involves transforming an initial inefficient functional program into an efficient final program by applying rules that modify the recursion of the program.

Broy and Pepper [15] and Partsch [66] study formalised program development using program transformations. Partsch treats both functional and procedural programs, and deals with the issue of transformation of data structures.

3.2.3 Miscellaneous work

A growing body of work is concerned with ‘calculating programs’ [12]. Based on category theory Bird and de Moor provide techniques to derive efficient function definitions in Haskell\(^2\) from inefficient ones, or to derive a definition of a function \(f\) given its inverse function \(f^{-1}\). There is also an initial attempt at automating such program calculations using a theorem prover [52].

Extended ML [41] is a framework for formal development of Standard ML programs from specifications, with emphasis on modularity and treating all features of a ‘real’ programming language. Its specification language EML, having SML as a sub-language, allows results from all stages of program development to be expressed in a single formalism. The long-term goal of this research is to provide a completely formalised development setting with development tool support in order to obtain correct programs. Indeed, for real-life applications this seems more realistic than synthesis from a correct specification with minimal guidance.

3.3 Inductive synthesis

In inductive synthesis the specification is incomplete. The approaches I survey here all have very restricted specification languages: a set of examples, possibly augmented other simple statements about the unknown relation or function. The rationale for this choice is twofold: Specifications in a more complex language are difficult to reason about, and simple specifications are simpler to come up with for the specification writer. I return to some of the work presented here when discussing related work in §9.2.

\(^2\)The language used is Gofer, a language very close to Haskell.
3.3 INDUCTIVE SYNTHESIS

3.3.1 Generalisation and Plotkin’s algorithm

Before going into paradigm-specific synthesis approaches I look at one important algorithm for syntactic generalisation due to Plotkin, together with some related concepts for logical clauses. Plotkin’s algorithm is used in a variety of synthesis algorithms.

Applying a substitution $\theta$ to a term $t$ yields another term $t\theta$ that is an instance $t$; substitution define an order on terms, called the term generalisation order. Material implication is an important order on logical clauses, and it has a decidable approximation in subsumption: Let $C_0$ and $C_1$ be clauses. $C_0$ subsumes $C_1$ if

$$C_0 \theta \subseteq C_1$$

for some substitution $\theta$ [71]. (Here clauses are considered to be sets.) Muggleton and De Raedt [61] attribute the following observation to Plotkin. If $C_0$ subsumes $C_1$, then $C_0 \models C_1$. The converse does not hold: let $C_0 = p(f(X)) \leftarrow p(X), C_1 = p(f(f(Y))) \leftarrow p(Y)$, then $C_0 \models C_1$ but $C_0$ does not subsume $C_1$.

Logical clauses are the building-blocks of logic programs. Subsumption between clauses, while not identical to material implication, is a handy approximation; its syntactic characterisation makes it simple to implement in synthesis systems. There are several approaches to defining and traversing lattices of clauses [61, 84]. Synthesis algorithms representing solutions as clausal programs, that is, programs consisting of Horn clauses, can use such a lattice to guide search. In the case of functional programs there is no order with the same nice properties as subsumption or implication for logical clauses. The reason is that comparing functions, as opposed to comparing relations, is problematic.

Plotkin’s algorithm provides a way to generalise a pair of terms into a new term—called the least general generalisation ($\text{lgg}$)—using simple syntactic matching [71, 32, 61]. In the lattice of terms this algorithm is the dual operation of unification: the $\text{lgg}$ corresponds to a most specific unifier (modulo variable renaming).

Considering a term to represent those below it in the lattice, Plotkin’s algorithm realises induction on terms by replacing a set of terms with a larger set including the first set. The basic version of the algorithm takes two terms $w_0$ and $w_1$, and outputs their least general generalisation ($\text{lgg}$) $w$, another term, and two substitutions $\phi_0$ and $\phi_1$ such that

$$w\phi_0 = w_0, \quad w\phi_1 = w_1.$$  

This is denoted as $w = \text{lgg}(w_0, w_1)$. Any other generalisation of $w_0$ and $w_1$ also generalises $w$. The algorithm always terminates successfully; a version computing only $w$ is given in Fig. 3.1 (adapted from Hutchinson [32, §6.2.3]).

Plotkin extends his algorithm to pairs of literals as follows ($p$ is a predicate symbol, $t_i$ and $s_j$ are terms, and the following assumes that variables from $\text{lgg}$ computations are reused):

$$\text{lgg}(p(t_0, \ldots, t_n), p(s_0, \ldots, s_n)) = p(\text{lgg}(t_0, s_0), \ldots, \text{lgg}(t_n, s_n))$$

$$\text{lgg}(-p(t_0, \ldots, t_n), -p(s_0, \ldots, s_n)) = -p(\text{lgg}(t_0, s_0), \ldots, \text{lgg}(t_n, s_n))$$
Given two terms $t_0$ and $t_1$ with no common variables. Let $t[p]$ denote $t$'s subterm at position $p$.
The $lgg$ $w$ of $t_0, t_1$ is computed as follows:
While $t_0 \neq t_1$ do:

Let $p$ be the a position where the two terms differ in their top-level symbol.
Let $s_0 = t_0[p], s_1 = t_1[p]$, and let $u$ be new variable not occurring in $t_0, t_1$.
Replace all occurrences of $s_0, s_1$ at same position (including $p$) in $t_0, t_1$ with $u$.

Let $w = t_0$.

and the $lgg$ is undefined if the literals' predicate symbols or signs differ. The $lgg$ of two clauses $C_0$ and $C_1$ is defined as follows.

$$lgg(C_0, C_1) = \{ lgg(L_0, L_1) | L_0 \in C_0, L_1 \in C_1, lgg(L_0, L_1) \text{ defined} \}.$$ 

Note that $lgg(C_0, C_1)$ subsumes both $C_0$ and $C_1$. The $lgg$ is the least upper bound on the lattice of clauses (actually, equivalence classes of clauses have to be introduced) ordered by subsumption. Plotkin also extends the algorithm to generalise pairs of clauses relative to a clausal theory [72], called the relative least general generalisation (rlgg).

3.3.2 Logic program synthesis

Recently there has been an surge of interest in combining machine learning [58]—especially inductive reasoning techniques—with logic programming. This work has been termed inductive logic programming [59, 61, 45, 63], or ILP for short. ILP is not primarily concerned with program synthesis. Its most well-known applications are in automatic learning of non-recursive definitions describing properties of chemicals [14], but, given its use of programs as a representation formalism, it is still relevant to program synthesis. Recursive predicates are frequently used as examples in ILP papers.

ILP is a very elegantly formulated and general approach. I present a formulation of the ILP approach first, and I present two important ILP systems. Then I look at other work that focus on inductive synthesis of recursive logic programs specifically. Note that logic programming is a quite attractive for program synthesis, at least from a language point of view: Specifications, programs, and synthesis algorithms can all be described and implemented as logic programs; this makes it simpler to relate specifications and programs.

3.3.2.1 The inductive logic programming approach

The terminology used in the ILP literature is influenced by its relation to machine learning, for example, solutions (also called hypotheses in ILP) are "learned from problem definitions". I stick to the terminology of §2.
### 3.3 INDUCTIVE SYNTHESIS

<table>
<thead>
<tr>
<th>Prior consistency</th>
<th>All $e^- \in E^-$ are false in $\mathcal{M}^+(B)$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Posterior consistency</td>
<td>All $e^- \in E^-$ are false in $\mathcal{M}^+(B \land P)$.</td>
</tr>
<tr>
<td>Prior necessity</td>
<td>Some $e^+ \in E^+$ are false in $\mathcal{M}^+(B)$.</td>
</tr>
<tr>
<td>Posterior sufficiency</td>
<td>All $e^+ \in E^+$ are true in $\mathcal{M}^+(B \land P)$.</td>
</tr>
</tbody>
</table>

$\mathcal{M}^+(P)$ is the minimal Herbrand model of program $P$.

**Fig. 3.2:** Assumptions of ILP.

The ILP approach can be formulated in several ways. With the specification language and solution language used in this section, the testing of hypotheses can be done efficiently using SLD-resolution. This, together with its simple formulation has made this formulation the most common for ILP systems, and it is also the most relevant for program synthesis.

**Problem formulation** The specification $S$ consists of two sets, one for positive example and one for negative examples, $S = (E^+, E^-)$. Both kinds of examples are represented as ground atoms. The language for solutions and the program library are definite clause programs.

As before $B$ is the background library and $P$ a solution program. To avoid trivial or redundant synthesis problem formulations, one assumes that certain conditions hold initially, see Fig. 3.2, adapted from Muggleton and De Raedt [61].

Observe that negative examples $e^-$ and $B$ can be considered written in the same language; prior consistency prevents these from being in conflict. Prior necessity ensures that none of the positive examples $e^+$ are already present in $B$. The two posterior conditions together ensure that solution $P$ satisfies specification $S$.

**Algorithms** Both ILP and program synthesis can be formulated as search problems over a program space. Given a specification the synthesis algorithm makes an initial hypothesis program, and then refines the hypothesis, that is, traverses the space, until a program satisfying the specification is found.

Searching is expensive so synthesis algorithms must be designed carefully. In the case of clausal programs one reasonable approach is to search for one clause in the solution program at a time.

The lattice of clauses ordered by subsumption can be searched using two main search strategies, leading to two main categories of ILP algorithms. **General-to-specific algorithms** start with the most general clause (Prolog syntax),

$$p(X_1, \ldots, X_n) :- \text{true}. \quad \text{(Prolog syntax)}$$

and refine it by adding literals on the right-hand side in order to avoid entailing negative examples, and taking care to retain entailment of positive examples. Entailing positive example can also be achieved by adding more clauses. When all examples are entailed,
that is, the specification is satisfied, the set of clauses is a solution program. Specific-to-
general algorithms start from the most specific hypothesis or hypotheses, for example, 
the positive examples themselves, and then generalise the clauses. Generalisation can 
use Plotkin's algorithm to compute clauses subsuming a given clause $C$ and that are 
close to $C$ in the lattice. This process, called clause refinement, is an subject in its 
own right [85, 84].

In the examples below I study two prominent ILP systems; these systems are not 
tailored to solving program synthesis problems, but they are still relevant as they 
can to some extent find recursive programs and their authors use recursive predicates 
extensively as examples. The language used in the systems varies and is commented 
below.

**Ex. 3.2** The system FOIL synthesises clausal programs with a restricted form of 
negated literals [73, 75]. Terms in programs must be either constants, variables or 
lists. The examples $E^+, E^-$ as well as the background library $B$ are required to be all 
ground atoms.

FOIL uses a general-to-specific algorithm. It starts from the most general clause 
and adds literals in a greedy manner with simple backtracking. The addition of lit-
erals to a hypothesis clause is guided by a heuristic 'gain computation', an measure 
based on information-theory. Clauses that correctly classifies examples as positive and 
negative receive higher gain than incorrect clauses. FOIL restrict solutions' clause size 
and variable usage. There are also rules that ensures soundness of recursion by requir-
ing clauses to obey a certain order on literals involving the unknown predicate and 
variables.

Quinlan and Cameron-Jones [75] report an experiment in which FOIL synthesises 
many (first-order) recursive predicates on lists, for example member, append, last and 
insert (but not reverse), given a sufficient background library and a large number of 
positive and negative examples: all positive examples of lists of length up to 4, 
involving 4 different constants, and between 2% and 80% of all negative examples that 
can be constructed similarly.

FOIL is capable of synthesising many recursive predicates, but it requires a large 
number of examples and a background library consisting only of ground atoms. The 
latter can be generated automatically from a regular program, but if $B$ contains a re-
cursive predicate a sample must be selected for inclusion in $B$ with the risk of excluding 
atoms required to find a solution.

FFOIL [74] is a derivative of FOIL specialised to synthesise definitions of (first-
order) functions. This is realised by synthesising predicates where one variable is 
designated as an output variable and the other variables as input variables. FFOIL 
 Improves on FOIL for such predicates by reduced running times and by synthesising 
some predicates that FOIL could not, like reverse. The main drawbacks of FOIL, 
requiring many examples and extensional background knowledge, are still present in 
FFOIL.

**Ex. 3.3** The greatest advantage of Muggleton's Progol system [60] over FOIL is that 
Progol does not require an extensional background library. Also, Progol's examples
can contain variables, they need not be ground atoms. The system uses a combination
of specific-to-general and general-to-specific search. Let $B$ be the background library,
including previously found hypotheses clauses. Inverse entailment is a technique that
given $B$ and an example $e$, finds a bounded lattice containing all clauses $h$ such that
$B \land h \models e$ (with some syntactic restrictions). A single $h$ is picked by searching this
lattice from general-to-specific and Progol goes on to the next example. Hypotheses
are restricted through a 'mode language' which includes simple constant types and
predicate modes. This, together with a restriction to clausal programs, determines
Progol's language for solutions. Inverse entailment generalises previous work on inverse
resolution. Progol employs an information-compression measure to pick hypotheses,
and prefer hypotheses that are simple according to this measure.

Progol synthesises a reverse predicate from 10–15 positive and negative examples,
a suitable background library $B$ and the following definite mode language declarations
(declared predicates other than reverse are in $B$).

\begin{verbatim}
% mode/type declarations for head literal of h
:- modeh(1,reverse([+const]+clist],[-clist])?
:- modeh(1,reverse(+clist,-clist))?

% declarations for body literals of h
:- modeb(1,append(+clist,+clist,-clist))?
:- modeb(1,reverse(+clist,-clist))?
:- modeb(1,unitlist(+const,-clist))?
\end{verbatim}

The 1s here state that all mode usages have exactly one answer substitution, const and
clist declares constant atomic and constant list arguments. Note that the two mode
declarations for the head literal correspond to the recursive and base case. They give
rise to one solution clause each. If we restrict the number of body literals in solution
clauses to 3 Progol quickly finds the following solution.

\begin{verbatim}
reverse([],[]).
reverse([A|B],C) :- reverse(B,D), unitlist(A,E), append(D,E,C).
\end{verbatim}

Predicate unitlist makes a constant into a list of 1 element, and append is the usual
list append predicate. This is a correct definition of reverse, in spite of the mode and
type declarations that restrict the usage to the functional sub-case.

In general, making Progol synthesise recursive predicates requires great care in
choosing the positive and negative examples and in designing mode declarations.

There are some problems common to both FOIL and Progol: The presence of
append in $B$ is crucial, otherwise reverse cannot be synthesised. The reason is that
neither system has the ability to introduce auxiliary predicates.

### 3.3.2.2 Flener's approach

Flener presents a system for inductive synthesis of logic programs called SYNAPSE [21,
23]. A general framework for such synthesis is also presented, but I do not consider
\[ R(X,Y) \Leftrightarrow \]

\[
\text{Minimal}(X) \land \text{Solve}(X,Y) \\
\lor \quad \forall 1 \leq k \leq c \quad \text{NonMinimal}(X) \land \text{Decompose}(X,HX,TX) \\
\quad \land \text{Discriminate}_k(HX,TX,Y) \\
\quad \land ( \text{SolveNonMin}_k(HX,TX,Y) ) \\
\quad \lor ( \text{R}(TX,TY) \\
\quad \land \text{Process}_k(HX,HY) \\
\quad \land \text{Compose}_k(HY,TY,Y) )
\]

where \( R(TX,TY) \) denotes a conjunction of recursive atoms and \(|\cdot|\) denotes the exclusive-or of the schema-language.

**Fig. 3.3:** Flener’s divide-and-conquer algorithm schema.

This here. The key idea of Flener’s approach is the use of algorithm schemas to guide synthesis. The synthesis algorithm also has a tool box of methods available for different parts of the synthesis problem, including Plotkin’s algorithm.

The specification consists of positive examples of an unknown predicate \( p \) and what Flener calls properties, which are non-recursive definite clauses whose heads are atoms of predicate \( p \). Properties are motivated as a compromise between ease of specification and power of expression.

**Ex. 3.4** Specification of the sum-of-integer-list predicate by examples and properties [23]:

\[ E = \{ \text{sum}([],0), \text{sum}([1],1), \text{sum}([3,2],5), \text{sum}([2,6,4],12) \} \]

\[ Pr = \{ \text{sum}([X],X), \text{sum}([X,Y],S) \Leftrightarrow \text{add}(X,Y,S) \} \]

where \( \text{add}(X,Y,S) \) holds if and only if integer \( S \) is the sum of integers \( X \) and \( Y \).

**Problem formulation** From a specification of examples and properties SYNAPSE synthesises ‘logic algorithms’—a formalisms that can be translated into Prolog. Flener requires that synthesised logic algorithm be instances of a given divide-and-conquer algorithm schema. Fig. 3.3 shows the schema in Flener’s notation [23] (there are also more general versions of the schema). The schema can be seen as a second-order logic algorithm where \( \text{Minimal} \), etc. are predicate variables, or schema operators. \( \text{Minimal}(X) \), etc. must be replaced by one or more literals involving regular first-order predicates in order to obtain a logic algorithm. Variable \( X \) is the input argument, \( Y \) is the output argument. If \( X \) is minimal \( Y \) can be found by a direct step. Otherwise \( X \) is non-minimal and must be decomposed into a vector \( HX \) of heads of \( X \) and a vector
3.3 INDUCTIVE SYNTHESIS

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer (Prolog syntax)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate declaration?</td>
<td><code>sort([l:list(int),s:list(int)])</code></td>
</tr>
<tr>
<td>Base case?</td>
<td><code>S=[]</code></td>
</tr>
<tr>
<td>When does <code>sort([1],S)</code> hold?</td>
<td><code>S= []</code></td>
</tr>
<tr>
<td>When does <code>sort([A],S)</code> hold?</td>
<td><code>S=[[A],B],A&lt;=B; S=[[B],A],A&gt;B</code></td>
</tr>
</tbody>
</table>

Fig. 3.4: Questions and answers in DIALOGS session synthesising sort.

\( \mathcal{T} \) of tails of \( X \). Tails are the same kind of data as \( X \), but smaller according to some criterion that ensures termination of recursion. The tails are recursively processed into \( \mathcal{T} \) and the heads \( \mathcal{H} \) are processed to \( \mathcal{H} \), and then \( Y \) is composed from these. Sub-cases can have different process and compose operators, and a non-minimal sub-case may be solvable directly.

**Algorithm** The SYNAPSE algorithm proceeds in a number of steps corresponding to one or more schema operators. Most steps use simple deduction, syntactic transformation, or lookup in databases of predicates. The Process and Compose operators are partly obtained using Plotkin’s algorithm to match and generalise based on syntactic similarities. Plotkin’s algorithm and a simple proofs-as-programs technique is used to find the Discriminate operators. The synthesis done by SYNAPSE is non-incremental, meaning that given a specification the system proceeds once through all synthesis steps.

DIALOGS [22] is an incremental version of SYNAPSE where the specification consists of only examples, and where queries are posed to the specifier or user in order to get information corresponding to that of SYNAPSE’s properties. DIALOGS takes care to pose queries that should be answerable by a naive user, and does not query the user unnecessarily.

SYNAPSE and DIALOGS use general-to-specific and specific-to-general methods in their search, and both systems are capable of inventing new predicates, though SYNAPSE is still not able add an auxiliary parameter to the reverse predicate in order to get the linear version. There are no empirical data on SYNAPSE yet [21, p. 213].

DIALOGS is capable of synthesising recursive logic programs for Insert-sort, Quick-sort and Merge-sort. The specifier must interactively answers the queries shown in Fig. 3.4. There are also some other questions, but the defaults are correct. In the synthesis of Insert-sort DIALOGS comes up with **insert** on its own, but the syntheses of Quick-sort and Merge-sort requires **partition** and **halves** to partition and split lists to be known and mentioned by the specifier. DIALOGS comes up with an auxiliary definition of an **append**-like combination operator for Quick-sort and a merging function for Merge-sort. DIALOGS also synthesises a predicate `filterOddInts(L,R)` where integer list \( R \) is integer list \( L \) without odd integer elements. Again the declaration and the following cases must be provided.

`filterOddInts([],[]).`
filterOddInts([A],R) :- odd(A),R=[ ]; not(odd(A)),R=[A].

where odd is a known predicate.

Flener’s systems are flexible and powerful. The main drawbacks are the necessity to provide base-cases and some other simple cases. In addition Flener’s systems are quite complex.

3.3.2.3 Schema-related work

Here I present other work on schemas or related formalisms in logic program synthesis.

Flener uses second-order programs to represent schemas. Sadahara and Haraguchi [78] define a notion of syntactic similarity between a hypothesis program and a schema program based on subsumption. This allows them to instead use regular programs as schemas.

Clause sets is a formalisms that describes a program space using a Prolog-like program with some notational extensions [10, §6.5]. Known clauses are written as usual while optional clauses are written in braces. Optional clauses are interpreted roughly as a set of possible clauses; there are also optional literals and terms. From a set of clauses or literals any subset may be taken to form clauses, from a set of terms a single term must be selected. An example showing the specification of the program space for the intersection predicate is in Fig. 3.5 [10]. Additional constraints on clauses are used when extracting programs from clause sets. These constraints are based on modes, variable usage and explicitly forbidden clauses or literal conjunctions. The clause set formalisms specifies at once the language (known and optional clauses) and
3.3 INDUCTIVE SYNTHESIS

background library (known clauses). Designing the clause set for the former is similar to designing a schema.

While the formalism is simple to read or write for a programmer—this is part of its motivation—it is not as powerful as schemas. For example, there is no concept of recursion, but it can be specified indirectly by repeating the same predicate in head and body.

The formalism is used by the TRACY system [9] to quickly synthesise predicates like intersection, union, and reverse given the appropriate background knowledge and 5 examples, both positive and negative, on average. The search space is however very restricted. TRACY uses an efficient way of testing programs from the search space.

Several authors have studied the use of higher-order predicates, either as schemas to restrict hypothesis [27] or in order to introduce special predicates that can be used in hypotheses [62].

The algorithm of Kirschenbaum and Sterling [42] builds a number of simple skeletons based on knowledge of recursion and datatypes (lists). These skeletons—a form of schemas—are simple programs with basic recursive structure, and other literals are added later. The LOPSTER system [43] assumes that solutions are in a certain class of recursive programs and it uses a special mechanism that spots syntactic similarities between terms. For the class of programs in question it is a quite powerful method, allowing for example append to be synthesised from just 2 positive examples.

3.3.3 Functional program synthesis

Research on what I have defined as inductive program synthesis started in the late 1960s with work on building imperative programs and Lisp programs, and reached a peak in the mid 1980s [11]. Since then the focus of inductive synthesis appears to have moved to logic programming.

Bauer [7, 8] describes an algorithm that synthesises simple imperative programs from sequences of instructions. The instructions involve variables, constants, branches and they compute concrete values. The sequences are converted to trees, constants to variables, nodes generalised and grouped using Plotkin’s algorithm, and the algorithm uses knowledge of variables, parameters and loops to form a program. The algorithm synthesises sort from a single example computation, but note that instruction sequences show how to compute a value, whereas input-output values show just what to compute.

3.3.3.1 Summers’s approach

Much work has been done on synthesis of list-manipulating Lisp programs from computation traces [79]. Summers [80] presents an early and rigorous approach to the problem.
Problem formulation The specification consists of traces, a restricted form of examples of list operations where some correlation between individual examples is supposed: the examples must correspond to a sequence of recursive calls, and this fact is exploited by the synthesis algorithm.

Ex. 3.5 Traces describing the Lisp function for reverse [79]:

\[(\) \rightarrow (\)\]
\[(a) \rightarrow (a)\]
\[(a \; b) \rightarrow (b \; a)\]
\[(a \; b \; c) \rightarrow (c \; b \; a)\]

Multiple occurrences of constants on the left-hand sides above are forbidden. This restriction means that this specification also implicitly contains example

\[(b \; c) \rightarrow (c \; b)\]

and so on. \hfill \diamond

Solution programs are recursive Lisp programs that must follow certain patterns of recursion. The program library consists of primitive list operations like \texttt{car}, \texttt{cdr}, \texttt{()}, and tests for the empty list.

Algorithm For each example Summers’s algorithm makes a definition \(f_i\) computing it using only the primitives. The algorithm also makes a guard definition \(p_i\) distinguishing that example from the others. Then it examines all pairs \(f_i, f_{i+1}\) and tries to find a recurrence relation, and the same for the \(p_i\)’s. If finding such a relation the algorithm conjectures that the relations also holds for values not given in the specification. The algorithm then applies a so-called synthesis theorem to obtain a solution program that will compute the examples. A special heuristic is used to introduce an auxiliary argument in some functions. The algorithm uses this heuristic to synthesise efficient list reversal without using list append.

3.3.3.2 Recent work

Several authors have extended Summers’s work [79]; two recent works are notable.

Summers’s example and programming languages are untyped. Ishino and Yamamoto [37] extend Summers’s approach as follows: The specification contains a type context, roughly a set of type declarations in a language with polymorphism, and a type declaration for the unknown function. The examples can contain constants of different types and have repetitions, but they must be well-typed in the type language. The examples are grouped and generalised to get examples in Summers’s language, and Summers’s algorithm is applied. This grouping and generalisation is the crux, and here the fact that the types of the examples and the solution must match guides the process, preventing too specific solutions.

Zhu and Jin’s approach [90] takes a specification similar to that of Summers’s approach. Solutions are programs in Backus’s FP [32, §6.3.1]—an untyped functional
language. Solutions are required to follow the schema in Fig. 3.6 [32], translated from the authors' FP notation. The schema allows linear recursion and synthesis consists of finding the non-recursive definitions \( \text{test, base, } \eta, \epsilon, \text{ and } d \). The synthesis algorithm uses the examples together with algebraic properties of functions, list operations in particular, in order to find these definitions. For example, the algorithm includes a procedure to find solutions \( d \) to equations on the form \( R \circ d = S \) where \( S \) and \( R \) are expressions and \( \circ \) is functional composition. The authors claim that Summers's approach is a special case of their approach. The system can synthesise list reverse given append.

Last I look at an approach related to genetic algorithms [58, 89] and genetic programming [3]. These methods are concerned with evolving descriptions—simple classifiers or simple functional programs—by simulating natural selection in a populations of such descriptions. The simulation uses the genetic operators crossover and mutation of individuals; the operators are inspired by their counterparts in biological systems. Such a genetic system is given a fitness measure on individuals, and an initial population of more-or-less random individuals, that is, no ‘well-chosen examples’ are required, only a lot of examples. Then the hard part, searching for individuals with high fitness (solutions), is done automatically.

Olsson's ADATE approach [65] synthesises typed functional programs in a simple ML-like language. The specification consists of examples of arguments to an unknown function, but no return values, and an ML definition that evaluates return value of the unknown function, a kind of fitness measure. The algorithm breeds a small number of candidate definitions of the unknown function, and, instead of the regular genetic operators, refines the definitions by special program transformations. The transformations seek to increase fitness in accordance with the fitness measure. The fact that the return values need not be given directly is an interesting idea making the job of the specification writer easier. The described implementation of ADATE synthesises a number of list-manipulation programs from scratch, including \texttt{sort, delete} and balancing of binary trees, but there appears to be some efficiency problems.

3.4 Discussion

Inductive synthesis of recursive programs has not been a particularly active field of research in recent years, and this is reflected in the state of the art. First, there is little work involving typed languages. This is surprising given developments in programming language research over the last 20 years. It may be explained by the observation that most synthesis work has been carried out by researchers working

\begin{verbatim}
prog in = if test in
    then base in
    else h (cons (e in) (prog (d in)))
\end{verbatim}

\textbf{Fig. 3.6}: Zhu and Jin's schema.
in artificial intelligence. ML and Haskell programmers know how useful static type checking is in eliminating errors and enforcing invariants. It seems likely that synthesis algorithms could benefit from type-checking too. Second, there seems to be no work on synthesis of higher-order functions, and little on synthesis of higher-order predicates. Since higher-order functions are essential to functional programming this represents a crucial omission. Last, synthesising recursive programs is a hard problem and as an answer to this some researchers use algorithm schemas. While it is well-known, at least in functional programming, that higher-order functions trivially implement schemas, no-one in inductive synthesis seems to have done the ‘obvious’: Instead of ad-hoc algorithms with implicit schemas, one can have a background library of ordinary higher-order functions available as schemas.

The synthesis algorithm I present in the next section uses type-information and represents schemas using higher-order functions. It can synthesise definitions of higher-order functions, but it works more efficiently for first-order functions. An important part of the synthesis algorithm is an extension of Plotkin’s least general generalisation algorithm from terms (expressions) to definitions, that is, to expressions with lambda-bound variables.
Chapter 4

The synthesis problem

In this chapter I formulate a version of the problem of synthesis from examples and give the rationale for this way of specifying the problem. How to solve the problem is addressed in the next two chapters. The chapter starts with a section, §4.1, where I define the language to be used for synthesised definitions. This language, called the object-language, is a subset of the functional language Haskell. In §4.2 I describe how to define schemas as higher-order functions. Following this I give a formulation of the synthesis problem in §4.3; the unknown function is given as a set of examples, a type declaration, and information about the function's complexity. In §4.4 I analyse the characteristics of solutions.

4.1 The object-language

The language used to represent synthesised definitions and schemas is called the object-language and is a simplified version of Haskell [69]. As a typographical convention object-language code is written in teletype font, foobar. The language is a statically typed non-strict functional programming language with the syntax in Fig. 4.1. Some syntactic categories are not defined in the figure: \texttt{var}, variable identifiers including function names and special operator symbols like ‘==’; and \texttt{typevar}, type variable identifiers. The concrete syntax for these categories is identical to that of Haskell. The object-language has built-in datatypes for integers, Booleans, and lists; it does not allow user-defined datatypes. The language is minimal to simplify the automatic manipulation of it. For this reason the object-language does not have tuples.

To be valid all object-programs must also be Haskell programs, in particular they must type-check. The semantics of an object-program is identical to the Haskell semantics of the same program. The object-language prelude—a program of definitions visible to all object-programs—contains a small subset of the definitions in the Haskell Prelude. From here on ‘the Prelude’ refers to the object-language Prelude. The functions defined by the Prelude and their types are in Fig. 4.2, with fixity as in Haskell. As in Haskell there is a special definition \texttt{undefined} whose value is indistinguishable from
that of a run-time error in a program. I sometimes use additional Haskell notation and conventions: pattern matching, guards and syntactic sugar for lists. These constructs all have translations into the kernel syntax in Fig. 4.1, similar to those in the Haskell Report [69].

I use the shorthand notation \( v :: \tau \) to state that a variable \( v \) has type \( \tau \). The mathematical meta-notation is standard: \( t \rightarrow_{P} t' \) means that \( t \) reduces, in one or more steps, to \( t' \) in the context of program \( P \) (may be left implicit), and then one may write \( t = t' \).

### 4.2 Schemas as higher-order functions

Functional languages have first-class functions so realising schemas is trivial: they are simply defined as recursive higher-order functions. This is a simple and elegant solution: applying a schema is equivalent to applying any other function and evaluation of definitions using schemas comes for free.

Below I first define a set of schemas for the list datatype and then I discuss schemas
4.2 Schemas as Higher-Order Functions

<table>
<thead>
<tr>
<th>Identifier(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>==, /=</td>
<td>Eq a =&gt; a -&gt; a -&gt; Bool</td>
</tr>
<tr>
<td>&lt;, &gt;, &lt;:=</td>
<td>Ord a =&gt; a -&gt; a -&gt; Bool</td>
</tr>
<tr>
<td>+, *, div</td>
<td>Int -&gt; Int -&gt; Int</td>
</tr>
<tr>
<td>succ, pred</td>
<td>Int -&gt; Int</td>
</tr>
<tr>
<td>not</td>
<td>Bool -&gt; Bool</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>null</td>
<td>[a] -&gt; Bool</td>
</tr>
<tr>
<td>head</td>
<td>[a] -&gt; a</td>
</tr>
<tr>
<td>tail</td>
<td>[a] -&gt; [a]</td>
</tr>
<tr>
<td>.</td>
<td>(a -&gt; b) -&gt; (c -&gt; a) -&gt; c -&gt; b</td>
</tr>
<tr>
<td>undefined</td>
<td>a</td>
</tr>
</tbody>
</table>

Fig. 4.2: Functions defined by object-language prelude.

in functional languages in general.

4.2.1 The list schemas

Here I present a set of concrete schemas for the list datatype, written using the object-language. The set of schemas is called the list schemas and consists of the 8 definitions shown in Fig. 4.3. The list schemas are based on the recursive higher-order functions of §2.1.2 and definitions foldr1, foldl1, foldr, foldl, and unfold are discussed there.

The other list schemas are specialisations or generalisations of these schemas or map. A schema scm1 is more general than another schema scm2 if scm2 is equivalent to scm1 instantiated with some argument. The order of generality is indicated with straight lines from more general down to less general in Fig. 4.5 (I comment the dashed lines later). For example the arrow from foldr down to mapr follows from

\[ \text{foldr} \ ((\cdot) \cdot f) \ as \ xs \ \equiv \ \text{mapr} \ f \ as \ xs. \]

§5.8 shows the details. Note how foldr and foldl are in two different unconnected parts in the figure. This happens since I do not permit object-definitions to use the higher-order trick in Ex. 2.2. Definition mapr is the Haskell Prelude definition map, except that it has an additional list argument that is appended to the result map returns. Definition mapl is foldl, what mapr is to foldl.

I divide the list schemas into three groups of schemas. The standard folds consist of foldr, foldl, foldr1, and foldl1; the unfolds are unfold and gunfold; and mapr and mapl are the myopic folds, so named because they only ‘see’ one element at a time from the recursion argument. The miscellaneous schemas in Fig. 4.4 are generalisations of the list schemas and included here only to show possible generalisations of the list schemas; they are not used in the following. The miscellaneous schemas’ relations to the list schemas are shown by dashed lines in Fig. 4.5.

By design convention all schemas shown here have their recursion argument as the last argument. Of the list schemas mapr, mapl, foldr1, foldl1, foldr, and foldl
mapr :: (a -> b) -> [b] -> [a] -> [b]
mapr f as [] = as
mapr f as (x:xs) = f x : mapr f as xs

mapl :: (a -> b) -> [b] -> [a] -> [b]
mapl f as [] = as
mapl f as (x:xs) = mapl f (f x:as) xs

foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)

foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs) = foldl f x xs

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f a [] = a
foldl f a (x:xs) = foldl f (f a x) xs

unfold :: (a -> b) -> (a -> Bool) -> (a -> a) -> a -> [b]
unfold f p g x
| p x = []
| otherwise = f x : unfold f p g (g x)

gunfold :: (a -> b) -> (a -> Bool) -> (a -> a) -> (a -> [b]) -> a -> [b]
gunfold f p g h x
| p x = h x
| otherwise = f x : unfold f p g h (g x)

Fig. 4.3: List schemas.

gfoldr :: (a -> b -> b) -> (a -> Bool) -> (a -> a) -> b -> a -> b
gfoldr f p g a x
| p x = a
| otherwise = f x (gfoldr f p g a (g x))

sfoldr :: (a -> b -> b) -> (a -> Bool) -> (a -> a) -> (a -> b) -> a -> b
sfoldr f p g h x
| p x = h x
| otherwise = f x (sfoldr f p g h (g x))

Fig. 4.4: Miscellaneous schemas.
always terminate provided that their arguments terminate.\footnote{This is not true for infinite lists, but I consider only finite lists.} Schema gunfold has, like unfold, reduction and condition arguments for controlling recursion. For recursion on lists and natural numbers one can guarantee termination by picking \( g \) as in Fig. 4.6; failure to reduce the base cases prevents non-termination. All the definitions in Figs. 4.3 and 4.4 are valid both in Haskell—except for name clashes with the Haskell Prelude—and in the object-language if appropriately desugared.

The list schemas are used by the synthesis algorithm of §5 as the only means to realise recursive definitions. The list schemas also found in §2.1.2 are used frequently by Haskell programmers, and they are also used repeatedly in the Haskell Prelude. As an informal argument of the usefulness of the list schemas consider the selection of standard definitions in Bird and Wadler’s classic textbook [13].\footnote{It is in Appendix B, Some Standard Functions; I have translated the authors’ Miranda-like names into the corresponding Haskell names.} The authors give 38 functions that they call “the most commonly used” (p. 279). Of these 29 can be defined using the list schemas, see Fig. 4.7, if defining insert-sort in place of quick-sort.\footnote{The reason that quick-sort cannot be defined is that it requires two recursive calls, something that the schemas here cannot provide.}

Looking at the remaining 9 definitions:

- \texttt{fst}, \texttt{snd} and \texttt{zip} have signatures involving tuples;

- \texttt{drop}, \texttt{!!} and \texttt{take} need a recursion argument of type \((\text{Int}, [\text{a}])\), or the trick in Ex. 2.2.
\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
7 & \&\&,(k \to k), head,(x \to x), not, \|, tail. \\
& Non-recursive, in Prelude or definable. \\
\hline
22 & and, ++ (list append), concat, filter, foldl, foldl1, foldr, foldr1, init, \\
& iterate, last, length, \setminus (list difference), map, max, min, or, product, \\
& reverse, sort, sum, takeWhile. \\
& Definable using list schemas. \\
\hline
9 & drop, dropWhile, fst, \_ (list indexing), scanl, snd, take, until, zip. \\
& Requires one of tuple datatype, breaking usage rules or using miscellaneous \\
& schemas. \\
\hline
\end{tabular}
\caption{Fig. 4.7: Bird and Wadler's 38 standard functions defined with the list schemas.}
\end{table}

- dropWhile and until needs a different schema, \texttt{mfold}, see Fig. 4.4.

- \texttt{scanl} can be defined using multiple list schemas, but, it requires a form of usage 
  that I rule out in §4.4—therefore I do not count it as definable by the list schemas.

In sum the list schemas successfully define all but 3 of Bird and Wadler's standard 
functions, if not counting definitions that require the tuple datatype or higher-order 
tricks.

Furthermore, an automatic system, just like a human programmer, benefits from 
having several schemas: specialised schemas means less complicated usage and result 
in simpler definitions.

\subsection{4.2.2 Discussion}

The object-language is somewhat limited in that it has lists as the only structured 
datatype, but schemas can also be defined for other datatypes like trees [55]. One can 
even define schemas for numbers, but this leads to a quite unfamiliar programming 
style.

There is a theoretical literature on the properties of \texttt{foldr} and \texttt{unfold}-like schemas 
for various datatypes [53, 54] and on the expressiveness of higher-order functions [34, 
12]. Finding a complete or minimal set of list schemas from this seems difficult, though 
it appears that tuples are useful.\footnote{According to Hutton [34] "every primitive recursive function on lists can be redefined in terms of [foldr]" when using a tuple construction presented in the same paper.} There is also some work studying \texttt{gunfold} as a 
generalisation of \texttt{unfold} [86]. Jansson and Jeuring [38] define a technique they call 
polytypism. Polytypic programs can be used unchanged on a certain class of datatypes, 
and the authors apply the technique to write a unification algorithm parametrised on 
term types [39].
4.3 Problem formulation

Here I formulate a kind of synthesis problem where the objective is to define some unknown function. The main part of a problem formulation is a set of example equations describing the function whose definition is sought. I require the expression on the right-hand side of each example to be a first-order value; otherwise one would need to compare functional values and thereby break the abstraction of functions. Consequently all unknown functions must return first-order values, and this is not a valid example:

\[ f \square 3 = (-) \]

The *synthesis problem* is as follows. Given

- \( f :: \tau \), a new identifier with arity \( n \), \( n \geq 0 \);
- \( E \), a set of example equations for \( f \);
- \( Scm \), a set of schemas, and \( L \), a library of definitions;
- \( c \), the number of times \( f \) traverses its recursion argument (wholly or in part), if it has one; otherwise \( c = 0 \);
- \( s \), the maximal size of a definition of \( f \).

Find a definition \( D \) such that

- \( D \) together with \( L \) and \( Scm \) define a function \( f \) with exactly type \( \tau \);
- \( D \) respects numbers \( c \) and \( s \), and does not use undefined;
- recursion in \( D \) is only through schemas in \( Scm \);
- for all \( l = r \) in \( E \) the expressions \( l \) and \( r \) have the same value when using \( D \) for \( f \).

Such a definition \( D \) is called a *solution* to the synthesis problem.

Some remarks:

- The problem formulation is independent of concrete languages, but when I address such problems in later chapters the object-language will be used to write definitions, \( Scm \) will be the list schemas, and \( L \) the Prelude.
- When using schemas recursion must be real—definitions cannot ignore results of a recursive call (no short-circuited recursion).
- The size of a definition is measured as the number of symbol occurrences in its definition; other measures are possible.
- Examples do not specify the laziness of solutions.
4.4 Anatomy of solutions

Here I analyse the anatomy of definitions that are solutions to the synthesis problem in the previous section. Assume that the unknown function \( f \) has arity \( n \). The type \( \tau \) of \( f \) has the following structure:

\[
\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_{n+1}.
\]

Using these names the \( i \)-th argument to \( f \) has type \( \tau_i \), and the return value has type \( \tau_{n+1} \). Assume the existence of one solution \( D \). By the given requirements it must traverse its recursion argument \( c \) times. Since every list schema traverses its argument once this means that \( D \) must use \( c \) schemas (possibly the same schema more than once). The form of \( D \) depends upon \( c \).

If \( c = 0 \) the definition looks like the following.

\[
f \; x_1 \cdots x_n = g
\]

where \( g : : \tau_{n+1} \) is an expression not containing a schema, though it may reference \( x_1, \ldots, x_n \).

If \( c = 1 \) then \( D \) has the form\(^5\)

\[
f \; x_1 \cdots x_n = \text{scm}_1 \; g_1 \cdots g_k \; x_r
\]

which is the same as

\[
f = \lambda x_1 \cdots x_n \rightarrow \text{scm}_1 \; g_1 \cdots g_k \; x_r
\]

where \( x_r, 1 \leq r \leq n, \) is the selected recursion argument of \( f \), \( \text{scm}_1 \in \text{Scm} \), \( \text{scm}_1 \) has arity \( k+1 \) and each \( g_i \) is an expression not containing a schema. Each \( g_i \) may reference any \( x_j \). Note that there may be several alternatives for the recursion argument \( x_r \).

For \( c > 1 \) the definition \( D \) is as the previous except that one \( g_i \) is instead replaced with a schema-defined lambda-expression,

\[
f \; x_1 \cdots x_n = \text{scm}_1 \; g_1 \cdots (\lambda y_1 \cdots y_m \rightarrow \text{scm}_2 \; h_1 \cdots h_l \; y_p) \cdots g_k \; x_r
\]

where \( y_p, 1 \leq p \leq m, \) is the chosen recursion argument for the function defined by the lambda-expression. Here expressions \( h_1, \ldots, h_l \) and \( g_i, i \neq j \) are defined recursively such that they together use \( c = k - 2 \) schemas. Note that each \( h_i \) may both refer to any \( y_m \) and any \( x_k \) since all these are in scope.

\(^5\)In fact, the following slightly more complicated form could be used.

\[
f \; x_1 \cdots x_n = g_0 \; (\text{scm}_1 \; g_1 \cdots g_k \; x_r)
\]

See discussion in \S 9.5.
Chapter 5

Synthesis by transformation

In this chapter and the next I present an algorithm for synthesis from examples. The algorithm synthesises solutions in the form of object-language definitions. An advantage of using a functional language is that solutions have small and elegant definitions. Furthermore there is little previous work on synthesis from examples using languages with static type-checking like Haskell.

§5.1 extends the object-language with templates, a special kind of variables that represent unknown parts of expressions. The synthesis algorithm is presented partly as a program in a functional meta-language, introduced in §5.2, and partly as semi-formal text with examples. The central concept ‘equation’ is presented in §5.3. Starting with an overview section, the details of the algorithm appear in §5.4 through §5.8.

The description of the algorithm can with little effort be translated into a program in Haskell or a similar language and form the core of an implementation. Practical experience with such an implementation is reported in §8. The choice of a Haskell-like language for the synthesis algorithm itself is easy: functional languages excel when used for writing interpreters and compilers for other functional languages, and they are generally well-suited for tasks involving symbolic reasoning.

5.1 Template expressions

The main technical idea in my approach to synthesis is the introduction of a new kind of expression with special variables—called template variables or just templates—representing unknown parts of expressions. This leads to a new language, a minor variation on the object-language, with conservative and easily defined extensions in syntax and semantics. I refer to this extended language as the template language.

The syntax of template expressions is given in Fig 5.1. Template variables have a ‘bar’, for example, \( \tau \), to distinguish them from regular variables. Note that template expressions also include expressions annotated with substitutions, and this requires a corresponding syntactic category \( \text{subst} \).
The following is an informal description of the operational semantics of template expressions. All templates, and all expression whose immediate reduction requires the value of a template, are considered atomic and not reduced. Such expressions are said to be suspended, and I use the notation $t^*$ to indicate that expression $t$ is suspended because of template $\bar{t}$. Since reduction is deterministic $\bar{t}$ is unique in $t$, and the notation is well-defined (even though there may be several occurrences of $\bar{t}$). Suspended expressions are on normal form in the new language. Substitution on template expressions is the same as on ordinary expressions, except that applying a substitution $\theta$ to a suspended expression $t$ results in a new suspended expression $t\theta$. Here $\theta$ is an explicit substitution. In the following I consider the object-language to be extended with templates.

Here are some concrete template expressions.

**Ex. 5.1** Suspended expressions:

- $1 + \bar{t}$, $\bar{t} \times y$, tail $\bar{t}$, if $\bar{t}$ then $i$ else 0

Expressions with templates need not be suspended:

- if True then $t$ else $\bar{t}$ → $t$
- $\text{False} \&\& \bar{t}$ → False
- tail ($\bar{x}: t$) → $t$
- $(\forall v \rightarrow t) \bar{t}$ | (v does not occur in t) → $t$

In the last two cases non-strictness prevents suspension: in each case the expression represented by the template is not needed to perform the reduction. Conversely, expressions in normal form need not be suspended when containing templates:

- $(\forall x \rightarrow 1 + \bar{t})$, $[\forall, 3]$

Since the language is non-strict functions need not be fully evaluated when they are applied. This means that it is possible to apply lambda-abstractions with templates in the body, that is, to apply (partly) unknown functions. This is an important property for the synthesis algorithm.
5.2 THE META-LANGUAGE

Ex. 5.2 Consider this reduction.

\[(\lambda x \rightarrow 1 + x + \tilde{n})(\text{f \ [1,2]})\]
\[\rightarrow 1 + \text{f \ [1,2]} + \tilde{n}(\text{f \ [1,2]}/x)\]

Note the resulting explicit substitution.

When templates already have substitutions glued to them the substitution is updated:

\[(\lambda a \ b \rightarrow \tilde{a}) \ 1 \ 2 \rightarrow (\lambda b \rightarrow \tilde{a}(1/a)) \ 2 \rightarrow \tilde{a}(1/a, \ 2/b)\]

\[\diamond\]

Ex. 5.3 Let \(f\) be an abbreviation for \((\lambda x \ xs \rightarrow \text{if } \tilde{b} \ \text{then } \tilde{e}_1 \ \text{else } \tilde{e}_2)\) and consider the following expression using \text{foldr}.

\[\text{foldr} \ f \ \tilde{a} \ [1,2,3] \rightarrow f \ 1 \ (\text{foldr} \ f \ \tilde{a} \ [2,3])\]
\[\rightarrow \text{if } \tilde{b} \ \text{then } \tilde{e}_1 \tilde{\theta} \ \text{else } \tilde{e}_2 \tilde{\theta}\]

where \(\tilde{\theta}\) abbreviates \((1/x, \ (\text{foldr} \ f \ \tilde{a} \ [2,3])/\text{xs})\). The expression is suspended on \(\tilde{b}\). Note that there are several occurrences of the templates; for instance, \(\tilde{b}\) occurs in the \text{if}-statement and in \(f\) inside each \(\tilde{\theta}\).  

\[\diamond\]

A remark on types. The object-language is statically typed. To have the same type-system after the introduction of template expressions I require that all templates be annotated with their type. Determining the type of templates during synthesis is not difficult; more on that in §5.7.1. Annotations are assumed present in the following.

5.2 The meta-language

Here I introduce a language that I use to formalise parts of my synthesis algorithm. The language, called simply the meta-language, is inspired by Haskell, but it has been tailored particularly for meta-programming in the sense of §2.1.3. There is nothing original in terms of language design in this section—the meta-language is a trivial variation on Haskell. Since the language is quite close to Haskell I do not give a precise definition of the meta-language, but present it informally.

Both the meta-language and regular mathematical language are used to make statements about object-language definitions. The following typographical convention is used: With the exception of arguments, the meta-language is written in sans serif font, \texttt{foobar}. Meta-language arguments and mathematical variables denote object-language entities such as expressions or definitions. Such meta-variables are written in italics, \(x, X\). Convention for naming meta-variables: \(v, x, f, g, h\); template variables: \(\bar{v}\); types: \(\tau\); type variables: \(a, b\); expressions: \(s, t\); substitutions: Greek letters.

Compared with Haskell the following properties of the meta-language are notable:

- All syntactic categories of the object-language—\texttt{def}, \texttt{expr}, \texttt{type}, etc.—are data-types in the meta-language.
• It has lazy sets as datatype and pattern matching on sets.
• It has set comprehensions: \( \{ f(x) \mid x \in X \} \).
• Type language: \( \{ a \} \) is the type of a set of elements of type \( a \); \times \) separates function arguments; \( \to \) separates function arguments and result; \( \times \) binds tighter than \( \to \);
• Pattern matching on sets, \( \{ x \} \cup X \), binds variables to disjoint sets, \( a \notin A \).
• The language has no currying.
• There is no layout rule; semicolon is used in a Pascal-like manner.

Ex. 5.4 The following definition corresponds to the usual definition of polymorphic
map for lists, but the datatype here is sets:

\[
\begin{align*}
\text{map} & :: (a \to b) \times \{a\} \to \{b\} \\
\text{map} & (\_ , \emptyset) \Rightarrow \emptyset; \\
\text{map} & (f , \{x\} \cup X) \Rightarrow f(x) \cup \text{map}(f , X);
\end{align*}
\]

An equivalent definition using a set comprehension:\(^1\)

\[
\text{map}(f , X) \Rightarrow \{ f(x) \mid x \in X \};
\]

The raison d'etre of the meta-language is its support for meta-programming. The
language has the following meta-programming operations on object-language data.

• The evaluation function, \( \text{eval} :: \text{expr} \times \text{def} \to \text{expr} \), takes an expression \( t \) and a
definition \( D \), and it evaluates \( t \) in the context of \( D \) and the Prelude.

• Replacement, \( \text{replace} :: \text{expr} \times \text{expr} \times \text{expr} \to \text{expr} \), modifies expressions. The
expression \( \text{replace}(s,s',t) \) is the same as expression \( t \), except that every occurrence of
sub-expression \( s \) in \( t \) has been replaced by \( s' \).

Ex. 5.5 Consider the following expression:

\[
\begin{align*}
\text{let} & f = \text{eval}(\text{head} \ [\] , D) \\
\text{in} & [f,f]
\end{align*}
\]

The value of the let-statement is the list \([\text{undefined,undefined}]\) since \text{undefined} is a
regular value of the meta-language datatype \text{expr}.

Ex. 5.6 The replacement operation must be used with care as it can cause unbound
variables to be captured.

\[
\text{replace}(y,x\ast z,(\backslash x \to y) \ 42) \ \to \ (\backslash x \to x\ast z) \ 42
\]

\(^1\)Haskell has list comprehensions and \text{map} can be written like this:

\[
\text{map} f \ \text{xs} \ \Rightarrow [f \ x \mid x \leftarrow \text{xs}]
\]
5.3 Representing Equations and Definitions

Variable capture may be wanted as part of the operations to construct or manipulate an expression.

The meta-programming operation must be defined on template expressions. It extends trivially to template variables, but explicit substitutions requires some care.

\[
\text{replace}(s, s', \theta) \Rightarrow \text{replace}(s, s', \theta)(\text{replace}(s, s', u) \mid (u/v) \in \theta);
\]

For notational convenience the definition consider the substitution to be a set and uses a comprehension:

Ex. 5.7 Returning to the suspended expression of Ex. 5.3 assume that \( b \) has somehow determined to be \( x \neq 0 \). Evaluation then proceeds as follows.

\[
\text{replace}(b, x \neq 0, \text{if } b \theta \text{ then } b_1 \theta \text{ else } b_2 \theta)
\rightarrow \text{if } (x \neq 0) \theta' \text{ then } b_1 \theta' \text{ else } b_2 \theta'
\]

where \( \theta' = \{t/x, (\text{foldr } f' \ a [2, 3]) /xs\} \) and \( f' \) is \( \lambda x \cdot \text{if } x \neq 0 \text{ then } 1 \text{ else } a \). Note how the replacement causes the free variable \( x \) in \( x \neq 0 \) to be captured inside the abstraction \( f' \). Variable \( x \) appears to be free in the condition of the if-statement, but note explicit substitution \( \theta' \) attached to it. Applying the substitution to the condition gives:

\[
\text{if } (x \neq 0) \theta' \text{ then } b_1 \theta' \text{ else } b_2 \theta'
\rightarrow \text{if } 1 \neq 0 \text{ then } b_1 \theta' \text{ else } b_2 \theta'
\rightarrow b_1 \theta'
\]

Observe how one explicit substitution is eliminated by evaluation after template \( b \) is replaced. \( \diamond \)

5.3 Representing Equations and Definitions

This section discusses representation of information used by the synthesis algorithm. The synthesis problem specification is initially represented as example equations, but general equations are needed for synthesis. For this I define a meta-language datatype for equations:

\[
data \text{eq} \quad = \quad \text{expr} \cdot \text{expr}; \quad \text{-- '\cdot' is an infix constructor}
\]

Typographic convention for synthesis information: equations: \( e \); sets of equations: \( E \); sets of equation sets: \( E \); definitions and programs: \( D, P \); sets of definitions: \( D \); sets of pairs of equations and definitions: \( ED \).

I also use some terminology. Consider an equation \( e, l = r \) and a program \( P \). If \( l \) and \( r \) have the same value in the context of \( P \) then \( e \) is satisfied by \( P \); otherwise \( e \) is unsatisfied; the latter is sometimes called inconsistent. These notions extends to sets of equations in the standard way. Program \( P \) or parts of it, for example, the Prelude, may be left implicit if clear from the context.

If an equation has form
\[ f \; t_1 \cdots t_n = t_{n+1} \]

where the \( t_i \)'s are expressions fully defined by the Prelude then it is called an \textit{example equation for} \( f \) or just an \textit{example of} \( f \). When the difference is important I refer to non-example equations as \textit{general equations} to disambiguate. When a \textit{meta-expression of type eq} is known to be an example equation I use the identifier \textit{ex} to indicate this. Examples of an unknown function make up the main part of synthesis problem specifications.

A remark on language: Template expressions extend normal object-expressions, and likewise I introduce \textit{template equations} with a template expression on the left-hand side and a normal expression on the right-hand side (the latter is always given in synthesis problems). The definition of suspension extends to equations: an equation is suspended if its left-hand side is. I extend the definition of example equations to include applied templates, possibly with explicit substitutions, in place of applied regular variables.

The synthesis algorithm also needs to represent the definition it is building. Assume the unknown function \( f :: \tau \) we are looking for is recursive. Following the analysis in §4.4 we know that the definition of \( f \) has the following form.

\[ f \; x_1 \cdots x_n = \text{scm}_1 \; g_1 \cdots g_k \; x_r \]

where \( x_r \) is the selected recursion argument of \( f \), \( \text{scm}_1 \in \text{Scm} \), and the \( g_i \)'s are unknown expressions about which we only known the type, and, if \( c > 0 \), that one or more of them use schemas. Based on \( \tau \) and types of schemas in \( \text{Scm} \) there are a small number of possible choices for \( \text{scm}_1 \) and \( x_r \), but for the \( g_i \)'s there are generally a large number of possibilities depending on the maximal size \( s \). Assume further that \( \text{scm}_1 \) and \( x_r \) are known (or that the same reasoning is done for every legal combination of the two) and that instead of the unknown expressions \( g_1, \ldots, g_k \) template variables are used.

\[ f \; x_1 \cdots x_n = \text{scm}_1 \; \tilde{v}_1 \cdots \tilde{v}_k \; x_r \]

The next sections show how equations and this kind of template definitions are used for synthesis.

## 5.4 Overview of the synthesis algorithm

The main idea of the synthesis algorithm is stepwise transformation of a set of problem states, or just states. Each state is a pair consisting of:

- \( E \), an equation set representing the unsolved part of the problem;
- \( D \), a definition representing the known part of the solution.

The algorithm starts from a certain initial state. Assume the synthesis problem is given as a function \( f :: \tau \) of arity \( m \) with examples \( E_0 \) as in §4.3. This leads to an initial state \( (E_0, \Box) \) where ‘\( \Box \)’ is the following initial and trivial definition for \( f \),

\[ f \; x_1 \cdots x_m = \Box \; x_1 \cdots x_m \]
and where $\bar{u}$ is a fresh template variable. The state $(E_0, \square)$ can be seen as the specification of the synthesis problem.

A transformation step

$$(E, D) \rightarrow (E', D')$$

adds a definition of a template from $E$ to $D$ yielding $D'$ and simplifies $E$ into $E'$ using the new definition and equational reasoning. For the algorithm to advance towards a solution it must hold that the set of solutions to $(E', D')$ is included in the set of solutions to $(E, D)$ (see also §7.2).

Synthesis stops transforming a state $(E_n, D_n)$ when either $E_n = \emptyset$ or $E_n$ is inconsistent, the latter denoted as $\perp^{eq}$. In the first case $D_n$ is a solution to the synthesis problem, in the second a dead-end has been reached. This can be depicted as follows.

$$(E_0, \square) \longrightarrow (E_1, D_1) \longrightarrow \cdots \longrightarrow (\emptyset, D_n) \text{ solution}$$

or

$$\cdots \longrightarrow (\perp^{eq}, D_n) \text{ non-solution}$$

So there is a tree of possible states that the algorithm must search through.

The rest of this chapter is structured as follows. The main synthesis logic is explained in §5.5, which also briefly discusses search control. §5.6 shows the details of how states are transformed, including equation simplification. Conjecturing new definitions from templates is the subject of §5.7. Last, §5.8 discusses how to avoid redundant definitions, based on the concept of equivalent expressions.

### 5.5 Main logic

The initiation of synthesis, and the main logic, is shown in Fig. 5.2. Function `synthesise` launches synthesis from the initial state, transforms the state using function `trans`, post-processes the output and returns the result as a set of solutions to the synthesis problem; more explanation below. The function `trans`, in the same figure, transforms a set of states into a set of definitions. If the state set is empty the empty definition set is returned; otherwise `trans` chooses a state $(E, D)$ from `ED` using some function `select`. Then $E$ is simplified using function `simplify`, into $E_s$. What to do with $D$ is decided based on $E_s$: if $E_s$ is inconsistent then $D$ is discarded. If $E_s = \emptyset$ then $E_s$ is satisfied and $D$ is made part of the returned set of solutions. Otherwise the state $(E_s, D)$ is refined, using function `refine`, and then transformed further. The other states in `ED` are also transformed further. Functions `simplify` and `refine` are explained in the next section.

A note on search strategy. The code for `trans` does not say anything about the order in which states are generated or indeed if they are generated at all. This depends upon `select` and, since the meta-language is non-strict, how the returned set is used. How `select` is defined determines the search strategy of the algorithm. I do not consider this issue here; an example of a simple strategy appears in §8.
synthesise :: \{ex\} \rightarrow \{def\}
synthesise(E) \Rightarrow
let D = \emptyset;
  D = trans(\{(E, D)\})  -- \{ex\} is a special case of \{eq\}
in
    postProcess(D, E);

trans :: \{eq\} \times \{def\} \rightarrow \{def\}
trans(\emptyset) \Rightarrow \emptyset;
trans(ED) \Rightarrow
  let (E, D) = select(ED);
  ED' = ED \setminus \{(E, D)\};
  E_s = simplify(E, D)
in
    case E_s of
    \perp \Rightarrow trans(ED');
    \emptyset \Rightarrow \{D\} \cup trans(ED');
    _ \Rightarrow trans(refine(E_s, D) \cup ED')
end;

Fig. 5.2: Functions synthesise and trans.

Function trans terminates if simplify and select terminate; refine must not generate
an infinite set of states, nor must it generate infinite chains of states. Later sections
explain what the set of returned definitions looks like.

The post-processing step—function postProcess in the definition of synthesise—is
trivial. Formally a solution is a definition in the object-language, so definitions
returned by function synthesise in Fig. 5.2 are not solutions if they contain templates.
The following is done by the post-processing function for each definition in D: Each
template in the definition is replaced with the simplest possible definition that can be
enumerated for it. Note that this does not affect the correctness of resulting solutions
since these templates were never evaluated and they are therefore unnecessary in com-
puting return values for the examples. An example is an if-statement where one of
the cases is never used.

5.6 Refinement and simplification

This section is about refinement, that is, how to construct the set of possible successor
states to a given state; I use the word refinement to indicate similarity to refinement
operators in ILP (§3.3.2). This section is also about simplification of equation sets:
Taking a set of equations \(E\) and a definition \(D\) and returning a simplified version \(E_s\) of
\(E\), such that \(E\) and \(E_s\) are equivalent given \(D\). Simplification resembles normalisation
in lambda-calculus.

Function refine is given in Fig. 5.3. Consider an \((E', D') \in refine(E, D)\).
5.6. **REFINEMENT AND SIMPLIFICATION**

\[
\text{Simplify: } \{eq\} \times def \rightarrow \{eq\}
\]

\[
\text{Simplify}(\emptyset) = \emptyset;
\]

\[
\text{Simplify}(\{l \Rightarrow r\} \cup E, D) \Rightarrow \quad \text{-- select arbitrary equation } l \Rightarrow r
\]

let \( e = \text{eval}(l, D) \Rightarrow r; \)

\[
E_s = \text{Simplify}(E, D)
\]

end;

\[
\text{Refine: } \{eq\} \times def \rightarrow \{eq\} \times def
\]

\[
\text{Refine}(E, D) \Rightarrow
\]

let \( E_f = \text{factor}(E) \)

end;

```
((replace(\( \bar{v} \), D_v, E_f), replace(\( \bar{v} \), D_v, D))
 | E_f \in E_f, \( \bar{v} \) = selectTemplate(E_f), D_v \in conjecture(\( \bar{v} \), E_f));
```

**Fig. 5.3: Functions simplify and refine.**

- \( D' \) is \( D \) except that template \( \bar{v} \) has been converted into a fresh regular variable \( v \) and a definition \( D_v \) for \( v \) has been added. The selection of \( \bar{v} \) by function selectTemplate is also a search strategy issue; a simple definition appears in §8.

- \( E' \) is \( E \) but with \( \bar{v} \) replaced by its new definition \( D_v \). In addition function factor rewrites \( E \) into a number of alternative equation sets. This is explained in §6.5; here I only note that factor(\( E \)) returns \( \{E\} \), that is, it has no effect, unless \( E \) has certain properties.

Every refined state \( (E', D') \) differs from \( (E, D) \). A state can only be reached once since the refined states always add a part to the definition. This means that the synthesis algorithm cannot get caught in a loop, forever revisiting the same state. If all expressions and definitions are canonical (§5.8) then the algorithm never visits a state equivalent to one it has visited before. Function refine depends upon function conjecture to generate actual definitions of \( v \), and refine uses a set comprehension to access the returned set. Function conjecture is discussed in the next section.

Fig. 5.3 also shows function simplify that works as follows. The empty equation set simplifies to itself. If the set of equations is non-empty an arbitrary equation \( l \Rightarrow r \) is selected and the rest of the equations are simplified recursively into \( E_s \). The left-hand side \( l \) of the selected equation is evaluated using \( D \) (and implicitly the Prelude), and the then obtained equation \( e \) is examined. If the left-hand side of \( e \) is suspended then \( e \) is returned as part of the simplified equation set. If the left-hand side and right-hand side of \( e \) is headed by the same expression constructor \( c \) then a recursive call is made.
to simplify equations made from corresponding sub-expressions on the two sides. If
the left-hand side and right-hand side of \( e \) are headed by different constructors \( c \) and
\( c' \) then the equation set is inconsistent. If the left-hand side and the right-hand side of
\( e \) are identical then this equation is satisfied and it is redundant. The patterns in the
case-statement are exhaustive. The equation set returned by simplify is either \( \emptyset \), \( \perp \),
or a set where all equations are suspended. The function terminates if \( \text{eval} \) terminates,
which again follows from proper schema usage.

**Ex. 5.8** Some examples of the properties of simplification.

\[
\text{simplify}\{\ldots,[1] \equiv [1], \ldots\}, D\} = \perp
\]
\[
\text{simplify}\{\{g \, 2 \equiv 4\} \cup E, g \, x = 2 \times x\} = \text{simplify}(E, g \, x = 2 \times x)
\]
\[
\text{simplify}\{\{\bar{I} \, 2 \equiv 4\}, D\} = \{\bar{I} \, 2 \equiv 4\}
\]

\[\diamondsuit\]

The next example illustrates both refinement and simplification.

**Ex. 5.9** Assume given the following example set for the recursive function \( \text{append} \).

\[
E_0 : \begin{cases}
\text{append} : [a] \rightarrow [a] \rightarrow [a] \\
\text{append} \left[1, 2\right] \left[3\right] \equiv \left[1, 2, 3\right]
\end{cases}
\]
The trivial initial definition of \( \text{append} \).

\[
D_0 : \begin{cases}
\text{append} : [a] \rightarrow [a] \rightarrow [a] \\
\text{append} \, x \, s \, y = \bar{u} \, x \, s \, y
\end{cases}
\]
Refining \( (E_0, D_0) \) means finding a definition of identifier \( \bar{u} \) to replace template \( \bar{u} \); this
notational convention for naming fresh variables replacing templates is used in the
following. Since \( \text{append} \) is recursive a schema must be used, and here is one possibility:

\[
D_1 : \begin{cases}
\bar{u} : [a] \rightarrow [a] \rightarrow [a] \\
\bar{u} \, x \, s \, y = \text{mapr} \, \bar{I} \, \bar{a} \, \bar{s} \, x
\end{cases}
\]
This gives

\[
E_0' : \begin{cases}
\bar{u} : [a] \rightarrow [a] \rightarrow [a] \\
\bar{u} \left[1, 2\right] \left[3\right] \equiv \left[1, 2, 3\right]
\end{cases}
\]
which is transformed by simplify as follows:

\[
\bar{u} \left[1, 2\right] \left[3\right] \equiv \left[1, 2, 3\right]
\Rightarrow \text{mapr} \, \bar{I} \, \bar{\theta}_1 \, \bar{a} \, \bar{\theta}_1 \left[1, 2\right] \equiv \left[1, 2, 3\right]
\Rightarrow \ldots
\Rightarrow \bar{I} \, \bar{\theta}_1 \, 1 : \bar{I} \, \bar{\theta}_1 \, 2 : \bar{a} \, \bar{\theta}_1 = \left[1, 2, 3\right]
\Rightarrow \bar{I} \, \bar{\theta}_1 \, 1 = 1
\Rightarrow \bar{I} \, \bar{\theta}_1 \, 2 = 2
\Rightarrow \bar{a} \, \bar{\theta}_1 = \left[3\right]
\]
5.7. CONJECTURING DEFINITIONS

\[
\text{conjecture} :: \text{var} \times \{\text{eq}\} \rightarrow \{\text{def}\}
\]

\[
\text{conjecture}(\bar{v}, E) \Rightarrow
\]

\[
\text{case kind}(\bar{v}) \text{ of}
\]

\[
\text{Reduction} \rightarrow \ldots \ -- \text{use reductions}
\]

\[
\text{Recursive e} \rightarrow \ldots \ -- \text{use e schemas from \textit{Scm} (e > 0)}
\]

\[
\rightarrow \text{generalise}(\bar{v}, E) \cup \text{enum}(\bar{v})
\]

Fig. 5.4: Function conjecture.

where \(\theta = \{[1, 2, 3]/xs, [3]/ys\}\). Here and in the following \(\Rightarrow\) is used to denote any step taken by the synthesis algorithm. To simplify notation I sometimes write, inspired by Haskell's layout rule, equation sets as above: one equation per line without braces and commas. This gives

\[
\begin{align*}
E_1 : & \\
& a : a \rightarrow a \\
& a\theta_1 1 = 1 \\
& a\theta_1 2 = 2 \\
& a\tilde{a} : [a] \\
& \tilde{a}\theta_1 = [3]
\end{align*}
\]

The transformation of \(E_0\) into \(E_1\) using \(D_1\) is an example of how simplify rewrites equations into a less complicated but equivalent form. \(\diamond\)

Unlike for the initial state and in the example above, the equation set of an arbitrary state is not always an example equation set.

Some remarks on implementation issues: An implementation should recognise that \(E\) is inconsistent as early as possible since this means that state \((E, D)\) cannot lead to a solution. The environment of suspended expressions must be stored away and later used when such expressions are evaluated further after refinement. Note how the equation set \(E_f\) is transformed in the definition of \textit{refine} in Fig. 5.3: replace\((\bar{v}, D_v, E_f)\), that is, the template is replaced by the \textit{definition} of \(v\) (as a lambda-expression), \textit{not} with \(v\) itself. The reason is that \(\bar{v}\) may have an explicit substitution \(\theta\) attached where it occurs in \(E_f\), and \(\theta\) may bind variables occurring free in \(D_v\), see template \(\bar{v}\) in Ex. 6.11 in \S6.4.

5.7 Conjecturing definitions

Function \textit{conjecture} takes a template \(\bar{v}\) and a set of equations \(E\), and conjectures a set of alternative definitions of \(v\); Fig. 5.4 gives a high-level definition. Conjecturing this set of definitions is the main inductive or ‘creative’ part of the synthesis process. The algorithm inspects the template using a function \textit{kind} to determine what kind of unknown definition of \(v\) template \(\bar{v}\) represents, and then acts accordingly.
• If \( v \) is a reduction function the algorithm simply returns the reductions corresponding to its type from Fig. 4.6. The number of reductions returned is limited by the size restriction \( s \).

• If \( v \) is recursive definitions using a schema from \( Scm \) are returned, more on this below.

• Otherwise the algorithm tries both syntactic generalisation of definitions from equations (§6.3), and enumeration (§5.7.2), to construct the definitions to return. Syntactic generalisation is better as it returns few and probable definitions, but when it fails to produce definitions enumeration, that is, blind guessing of well-typed canonical definitions, is used as a fallback.

Note that the set of conjectured definitions is limited by the size \( s \) from the problem formulation and the finiteness of \( Scm \).

During synthesis certain information about templates must be maintained by the synthesis algorithm. This includes the type of templates and the information provided by kind in function conjecture, that is, whether the template represents a recursive function, a reduction, or another non-recursive function. Observe that type is not necessarily a local piece of information: the type one template is assigned may affect the possible types for another template, as the following example shows.

**Ex. 5.10** Consider

\[
\text{inclist} ::= [\text{Int}] \rightarrow [\text{Int}]
\]
\[
\text{inclist \( \text{xs} = \text{mapr \( \tilde{f} \) \( \tilde{a} \) \( \text{xs} \)\)}\]

Given the provided type for \( \text{inclist} \). If \( \tilde{f} \) is determined to be \( \text{succ} :: \text{Int} \rightarrow \text{Int} \) then \( \tilde{a} \) can be chosen as \([1 :: [\text{a}]\), and the overall type of \( \text{inclist} \) is still correct. ◊

In general the book-keeping is not difficult to implement; here I just list what information it includes for each template: Type \( \tau \), kind, size \( s \), and number of schemas used, \( c \).

### 5.7.1 Recursive definitions using schemas

When the synthesis algorithm conjectures recursive definitions for \( \tilde{v} \) it uses a schema to implement the recursion. For such functions \( \text{kind}(\tilde{v}) \) computed in Fig. 5.4 includes the number \( c \) of schemas to be used in definitions, but only one schema is introduced at a time.

Assume that the unknown definition has type

\[
v :: : \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_{n+1}.
\]

The algorithm tries every schema in \( Scm \) when defining \( v \) and every possible recursion argument \( r \). Let \( \text{scm} \) be one such schema,

\[
scm :: : \rho_1 \rightarrow \cdots \rightarrow \rho_k \rightarrow \rho_{\text{rec}} \rightarrow \rho_{\text{range}}.
\]

To make a definition using \( \text{scm} \) the algorithm must:
5.7. CONJECTURING DEFINITIONS

- Find \( n \) fresh variables \( x_1, \ldots, x_n \) to represent the formal arguments of \( v \), and select a recursion argument \( x_r \) from these.

The recursion argument could be any \( x_i \). With a more complicated type-language combinations of original argument must be considered, for example, pairs of lists and integers.

- Verify that the constructed definition is well-typed. This is the case if the following type unification does not fail, but returns a type substitution.

\[
\sigma = \text{unify}(\tau_r \rightarrow \tau_{r+1}, \rho_{rec} \rightarrow \rho_{range}).
\] (5.1)

The algorithm treats the \( \tau_i \)'s as constant types (sometimes called generic types) in order to avoid synthesising a definition with a too specific type.

- Find \( k \) fresh templates \( \tilde{g}_1, \ldots, \tilde{g}_k \) and annotate them with types \( \rho_1\sigma, \ldots, \rho_k\sigma \).

The definitions of \( v \) made have this form:

\[
v \ x_1 \cdots x_n = \text{sem}_i \ \tilde{g}_i \cdots \tilde{g}_k \ x_r
\]

Following this the problem of finding a definition for \( \tilde{f} \) has been transformed into the problem of finding definitions for \( \tilde{g}_1, \ldots, \tilde{g}_k \). This, together with the introduction of if-statements, is the only time during synthesis that the algorithm introduces new templates.

\textbf{Ex. 5.11} The type of the unknown function can exclude certain schemas from being used, for example, the type \([a] \rightarrow \text{Int} \) means the unfolds are out of the question since they both return lists, and given \((a \rightarrow a) \rightarrow a \rightarrow [a]\) only the unfolds make sense since the standard and myopic folds require a list recursion argument. For such inapplicable schemas the type unification in Eq. 5.1 fails.

\textbf{5.7.2 Enumerating non-recursive definitions}

As the name suggests enumeration naively tries all definitions that can take the given type \( \tau \) and have the given size \( s \). Enumeration is best illustrated with an example.

\textbf{Ex. 5.12} For a template \( \tilde{f} :: a \rightarrow [a] \rightarrow [a] \) the set of enumerated definitions, \( \text{enum}(\tilde{f}) \), includes:

\begin{verbatim}
f x xs = [ ]       -- type a -> b -> [c]
f x xs = xs       -- type a -> b -> b
f x xs = tail xs  -- type a -> [b] -> [b]
f x xs = [x]       -- type a -> b -> [a]
f x xs = x:xs     -- type a -> [a] -> [a]
f x xs = x:x:xs
f x xs = [head xs]
f x xs = head xs:[x]
\end{verbatim}
\[
f \times xs = x : [x] \\
f \times xs = x : \text{tail} \ xs \\
f \times xs = x : [\text{head} \ xs] \\
f \times xs = x : x : \text{tail} \ xs
\]

\[\text{\Diamond}\]

Enumeration also takes care of the introduction of if-statements. In addition to the alternatives shown in the example above function \texttt{enum} also tries this,

\[
f \times xs = \text{if } \delta \text{ then } \delta_1 \ \text{else } \delta_2
\]

where template \(\delta::\text{Bool}\) and the types of templates \(\delta_1\) and \(\delta_2\) are the same as the return type of \(f\). The templates must be fresh and they are replaced with definitions later, just like templates introduced with schemas. In a non-strict language like the object-language the \textit{if-then-else} construct is not special and could be replaced with a regular higher-order function:

\[
\text{ifThenElse} :: \text{Bool} \rightarrow a \rightarrow a \rightarrow a \\
\text{ifThenElse} b e1 e2 = \text{if } b \text{ then } e1 \text{ else } e2
\]

Therefore using an \textit{if}-statement in a definition is similar to using a schema, except that the \textit{if}-statement does not introduce recursion. Note that types can help the synthesis algorithm to introduce an \textit{if}-statement. Assume that a function \(f\) has a predicate argument \(p\), that is, \(p\) produces a Boolean, but that \(f\) itself does not return a Boolean. This means that the definition of \(f\) must consume the Boolean value itself (or ignore it). Consumption can only happen in the condition of an \textit{if}-statement.

A last point regarding \textit{if}-statements. Operationally the following two definitions are equivalent,

\[
f \times xs = \text{tail} \ xs \\
f \times (y:ys) = ys
\]

but the they introduce a different set of bindings. When an \textit{if}-statement with templates is introduced in a definition the argument bindings of the definition affects the set of 'building blocks' available to construct unknown definitions replacing the templates. Enumeration must therefore include \textit{if}-statement definitions with refined argument patterns; Ex. 7.1 in §7.1 shows this.

### 5.8 Canonical expressions

As programmers we strive to write correct and concise programs. A correct function definition computes the right value for all argument combinations. A concise definition does not have unnecessary repetitions or redundant parts. Redundancy can help humans understand programs more easily, but for machines manipulating programs it is a problem: Transforming a redundant part of a program or evaluating two identical expressions means wasted computational resources.
5.8 CANONICAL EXPRESSIONS

**Ex. 5.13** Here are some redundant definitions.

```haskell
foo n = if (n < 0) && False then 0 else n*n
bar arg = (\x -> [x]) arg
baz x xs = tail (x:xs)
```

Due to the short-circuiting of the condition in the if-statement ‘foo n’ evaluates to ‘n^2’ for all n. The next two definitions are also clumsy attempts at defining functions that have simpler equivalent definitions. These are as follows.

```haskell
foo n = n*n
bar = \x -> [x]
baz x xs = x:xs
```

\diamond

In mathematical terms the synthesis algorithm should enumerate only one representative of each equivalence class of expressions: the smallest one, in accordance with the requirements in the synthesis problem definition. I call such minimal expressions **canonical expressions**. All expressions and sub-expressions used in definitions should be canonical, extending the concept to **canonical definitions**. Recursion is only present in object-programs through schemas and therefore checking if a definition is canonical is simpler than for a language with unrestricted use of recursion.

Each equivalence in Fig. 5.5 has a non-canonical expression on the left-hand side and the corresponding canonical expression on the right-hand side, assuming that meta-variables—like x in the second equivalence—denote canonical sub-expressions of the appropriate type. The first group of equivalences follows trivially from the properties of datatypes or programming constructs; the last equivalence in this group is \(\eta\)-conversion from lambda-calculus [70, p. 19], it serves to avoid spurious abstractions. The last group of equivalences shows redundant definitions caused by using a too general schema. More such equivalences can be constructed, for example, involving integers. Expressions with canonical form **undefined** should not be enumerated.
<table>
<thead>
<tr>
<th>head []</th>
<th>\equiv\ undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>head (x:xs)</td>
<td>\equiv\ x</td>
</tr>
<tr>
<td>tail []</td>
<td>\equiv\ undefined</td>
</tr>
<tr>
<td>tail (x:xs)</td>
<td>\equiv\ xs</td>
</tr>
<tr>
<td>head xs:tail xs</td>
<td>\equiv\ xs</td>
</tr>
<tr>
<td></td>
<td>(xs \neq [])</td>
</tr>
<tr>
<td>False &amp;&amp; b</td>
<td>\equiv\ False</td>
</tr>
<tr>
<td>True &amp;&amp; b</td>
<td>\equiv\ b</td>
</tr>
<tr>
<td>b &amp;&amp; b</td>
<td>\equiv\ b</td>
</tr>
<tr>
<td>False \mid b</td>
<td>\equiv\ b</td>
</tr>
<tr>
<td>True \mid b</td>
<td>\equiv\ True</td>
</tr>
<tr>
<td>b \mid b</td>
<td>\equiv\ b</td>
</tr>
<tr>
<td>if True then c₁ else c₂</td>
<td>\equiv\ c₁</td>
</tr>
<tr>
<td>if False then c₁ else c₂</td>
<td>\equiv\ c₂</td>
</tr>
<tr>
<td>if b then e else e</td>
<td>\equiv\ e</td>
</tr>
<tr>
<td>(\forall v \to f \nu\equiv f)</td>
<td>(\nu\ \text{not free in}\ f)</td>
</tr>
</tbody>
</table>

foldr \((\cdot).f\) as xs \equiv\ mapr\ f\ as\ xs
foldl (flip \((\cdot).f\)) as xs \equiv\ mapl\ f\ as\ xs
gunfold \((f,\text{head})\) null tail \((\_ \to y)\) xs \equiv\ mapr\ f\ ys\ xs
gunfold \((f,p)\) \((\_ \to [])\) x \equiv\ unfold\ f\ p\ g\ x

Fig. 5.5: Equivalent non-canonical and canonical expressions.
Chapter 6

Syntactic methods

In this chapter I present two algorithms that help the synthesis algorithm conjecture definitions more efficiently than enumeration—in the case of certain polymorphic functions. These algorithms employ syntactic methods, meaning that they match parts of expressions (or equations) against each other, and use the results to construct new expressions or definitions. The advantage of the two algorithms come from the fact that they use the equations to construct definitions, while enumeration does not.

The two algorithms are called generalisation and factoring. Generalisation constructs a definition that performs the computations described by a set of example equations. This is a useful technique but it is not always applicable: the synthesis algorithm needs to have example equations of the unknown function available to use it. In the general case equations may have more than one template on the left-hand side, and recursive equations, that is, equations where the left-hand side has more than one occurrence of a template, occur frequently. This is where synthesis uses the factoring algorithm. Factoring rewrites general equations into example equations, thereby making room for generalisation. Factoring is the essential companion to generalisation and it is also a way to decompose certain synthesis problems.

§6.1 defines two restricted classes of polymorphic functions having the property that functions in the classes can be subject to generalisation and factoring. Then §6.2 and §6.3 present the motivation for and definition of generalisation, and §6.4 and §6.5 do the same for factoring.

6.1 Restricted polymorphism

Polymorphic functions have the property that they can be applied to arguments of more than one type. In this section I use type-based criteria, that is, syntactic criteria, to define two restricted classes of polymorphic functions. The motivation for introducing the classes is that the generalisation algorithm only applies to function in the first class and the factoring algorithm only applies to function in the second class. The second class includes the first class.
I start by defining some concepts related to polymorphic functions. In the following the type of a function is always assumed to be its most general type, called its principal type in the literature. Let \( f : \tau \) be a function. If \( \tau \) contains no constant type constructors then \( f \) is called a completely polymorphic function. Examples of such functions are `foldr` and `append`, but not `length` as the latter's type uses \( \text{Int} \). This class is the counterpart of monomorphic functions. Abusing terminology somewhat I use these terms also to characterise types, for example, 'a completely polymorphic type'.

I define two operators on object-language types using the meta-language.

\[
\begin{align*}
\text{range} &: \text{type} \rightarrow \text{type} \\
\text{domain} &: \text{type} \rightarrow \{\text{type}\} \\
\text{range}(\cdot \rightarrow \tau) &= \text{range}(\tau) \\
\text{domain}(\tau \rightarrow \tau') &= \{\tau\} \cup \text{domain}(\tau') \\
\text{domain}(\cdot) &= \emptyset \\
\end{align*}
\]

Now I can define the new classes. Let \( f : \tau \) be a function. If

- \( \text{range}(\tau) \) is completely polymorphic; and
- \( \forall \tau' \in \text{domain}(\tau) : \text{if} \ \text{domain}(\tau') = \emptyset \ \text{then} \ \tau' \ \text{is completely polymorphic else} \ \text{range}(\tau') \) is monomorphic;

then \( f \) is a range-polymorphic function. If \( f : \tau \) is a range-polymorphic function and in addition \( \tau' \in \text{domain}(\tau) \) implies \( \text{domain}(\tau') = \emptyset \) then \( f \) is a first-order range-polymorphic function. Note that all first-order range-polymorphic functions are completely polymorphic, but not vice-versa; `map` is a counterexample. Ad-hoc polymorphic functions, like `sort :: \text{Ord} \ a \rightarrow [a] \rightarrow [a]`, can be seen as having an implicit predicate argument of type \( a \rightarrow a \rightarrow \text{Bool} \) responsible for doing the comparison required by the type context.\(^1\) Following this I also consider ad-hoc polymorphic functions to be range-polymorphic, provided that the non-context part of their type is range-polymorphic. Ad-hoc polymorphic functions are not first-order.

With the simple type-system of the object-language the following holds for any range-polymorphic function \( f : \tau \):

1. \( \text{range}(\tau) \) is one of \( a, [a], [[a]] \), etc. where \( a \) is a type variable;

2. \( \tau' \in \text{domain}(\tau) \) means \( \tau' \) has one of these forms:

   (a) \( a, [a], [[a]], \ldots \)

   (b) \( \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow c, \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow [c], \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow [[c]], \ldots \)

   where \( a \) is a type variable, \( c \) is \( \text{Int} \) or \( \text{Bool} \), \( \sigma_i \) is any type, and \( n \geq 1 \).

---

\(^1\)A technique for implementing ad-hoc polymorphism in Haskell compilers involves passing a dictionary of such implicit definitions to ad-hoc polymorphic definitions at run-time.
6.1. \textit{RESTRICTED POLYMORPHISM}

\begin{center}
\begin{tikzpicture}

\node at (0,0) {box{\begin{tabular}{l}
+\hspace{1em} length\hspace{1em} map\hspace{1em} all functions \\
filter, sort \hspace{2em} range-polymorphic functions \\
\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \hspace{1em} first-order \hspace{2em} range-polymorphic functions
\end{tabular}}};

\end{tikzpicture}
\end{center}

\textbf{Fig. 6.1: Classification of sample functions.}

Case 2a corresponds to first-order range-polymorphic functions. Such functions have only first-order arguments and return a first-order value at uncurried application, which motivates the name. Definitions of functions in this class have the property that the structure of their arguments completely determines how computation proceeds. Lists are the only kind of structured data in the object-language, meaning that only difference in list structure can affect the computation of such definitions.

\textbf{Ex. 6.1} Consider the following sample functions.

\begin{verbatim}
(+): Int -> Int -> Int
length : [a] -> Int
map : (a -> b) -> [a] -> [b]
filter : (a->Bool) -> [a] -> [a]
sort : Ord a => [a] -> [a]
(++) : [a] -> [a] -> [a]
\end{verbatim}

Fig. 6.1 shows these functions classified. Note that \texttt{length} is polymorphic, but not completely polymorphic like \texttt{map} and \texttt{++} (list append in the Haskell Prelude). \hfill \ding{53}

Consider an example of a range-polymorphic function \( f :: \tau, \)
\[
f t_1 \ldots t_n = t_{n+1}.
\]

Given the type of \( f, \) the computation of \( f \) considers any constant \( c \) occurring in \( t_{n+1} \) to be polymorphic. Constant \( c \) must therefore occur in one of the \( t_i \)'s on the left-hand side; \( c \) may not stem from the Prelude since it does not contain constants of (polymorphic) type \( a, \) where \( a \) is type variable,\(^2\) nor may \( c \) be the result of applying one of \( t_1, \ldots, t_n \) to a set of arguments, since these, if higher-order, return monomorphic constants. I call this property of range-polymorphic functions to be \textit{constant-bounded}. This is an important property for syntactic methods on equations: it makes it possible to match constants on the right-hand side to constants on the left-hand side.

\textbf{Ex. 6.2} Returning to the previous example, here are some computations using definitions of the functions:

\(^2\)I ignore \texttt{undefined} since this case is not relevant to generalisation.
1 + 2 \rightarrow 3
\text{map} \ (1+) \ [1, \ 2] \rightarrow [2, \ 3]
\text{filter} \ (<3) \ [1, \ 2, \ 3, \ 4] \rightarrow [1, \ 2]
[1, \ 2] ++ [3] \rightarrow [1, \ 2, \ 3]

In the first two cases the application of non-constant-bounded functions resulted in a ‘new’ monomorphic constant 3 on the right-hand side. Note also that functions can be constant-bounded without being range-polymorphic:

\begin{verbatim}
bump42 :: [Int] \rightarrow [Int]
bump42 \ (42:xs) \ = \ xs ++ [42]
bump42 \ xs \ = \ xs
\end{verbatim}

This function moves the first element to the end of the list if it is 42, otherwise the list is unchanged.

Consider again Bird and Wadler’s collection of standard functions in Fig. 4.7. Of their 38 functions 23 are range-polymorphic functions and 16 of these again are first-order. This suggests that membership in the classes is not unusual and that it makes sense to define synthesis operations that only work for functions in these classes. This is what I do in the next sections.

\section{Motivation for generalisation}

Here I present an informal account of and motivation for generalisation; a formal definition appears in \S 6.3. The informal part starts by introducing the idea of generalisation in \S 6.2.1 and then goes on with a series of examples in \S 6.2.2.

Unfortunately there is a problem with the presentation of the generalisation and factoring algorithms. The most interesting synthesis problems to which generalisation applies require using factoring first. On the other hand understanding generalisation and what it can and cannot be applied is the main motivation for factoring. The lack of a good order of presentation for these algorithms leads to some unfortunate cross referencing between the next sections.

\subsection{Reversing computations}

Consider a set of example equations $E$ for a template $f$ with $c = 0$, that is, $f$ is not recursive. Template $f$ represents a definition of an unknown function that, when applied as on the left-hand side of an example in $E$, computes the value on the right-hand side. The idea of generalisation is to attempt to reverse the computations specified by $E$ to construct definitions from them. This includes matching constants occurring on the right-hand side to constants in the arguments of $f$, and thereby reconstructing the substitution for the arguments that lead to the result on the right-hand side. This matching of constants can fail for functions that are not constant-bounded, and therefore generalisation is only attempted for first-order range-polymorphic functions.
6.2 MOTIVATION FOR GENERALISATION

Generalisation is a meta-programming operation computing a definition, that is, data of type \( \textit{def} \), by using a set of examples, data of type \( \{\textit{ex}\} \). In general there may be more than one definition satisfying the examples, and the generalisation algorithm must choose one.

Assume given the following example and (partial) definition for \texttt{append} of type \( \tau \).

\[
\begin{align*}
\text{append} &:: \ [a] \to [a] \to [a] \quad -- \tau \\
\text{append} &\quad [1, 2] \quad [3] \Rightarrow [1, 2, 3] \\
\text{append} &\quad [a] \to [a] \to [a] \\
\text{append} &\quad xs \quad ys \equiv \text{mapr } \tilde{f} \ a \tilde{s} \ xs
\end{align*}
\]

Using the definition of \texttt{append} the synthesis algorithm evaluates the equation, simplifies it, and finds three new equations:

\[
\begin{align*}
\text{append} &\quad [1, 2] \quad [3] \Rightarrow [1, 2, 3] \\
\Rightarrow &\quad \tilde{f}\theta \quad 1 \equiv 1 \\
\Rightarrow &\quad \tilde{f}\theta \quad 2 \equiv 2 \\
\Rightarrow &\quad a\tilde{s}\theta \equiv [3]
\end{align*}
\]

where \( \theta = \{ [1, 2] \mapsto xs, [3] \mapsto ys \} \). First consider the third example equation above. The template has type \( a\tilde{s} :\ [a] \) where the type variable \( a \) comes from \( \tau \). The template has no arguments to match constants on the right-hand side against, but an explicit substitution \( \theta \) is attached to \( a\tilde{s} \), resulting from the application of \texttt{append}.

The first step of generalisation is to reconstruct examples’ right-hand sides as they were before the substitution \( \theta \) was applied; the left-hand sides are not affected. Here reverting the effect of \( \theta \) on the right-hand side of the equation for \( a\tilde{s} \) means to replace occurrences of \( [1, 2] \) with \( xs \) and occurrences of \( [3] \) with \( ys \).

\[
\begin{align*}
a\tilde{s} \equiv [3]\theta^{-1} \Rightarrow a\tilde{s} \equiv ys
\end{align*}
\]

I call this operation the inverse substitution of \( \theta \), or just inverting \( \theta \), denoted \( \theta^{-1} \) for an expression \( t \); a definition appears in the next section. At the same time the explicit substitution \( \theta \) is removed from the template on the left-hand side since its effect is undone by the inverse substitution on the right-hand side. Now, replacing \( ‘=’ \) with \( ‘\equiv’ \) gives the definition \( a\tilde{s} = ys \). Note that \( ys \) is not a free variable, it is bound as an argument of \texttt{append}. For the two examples for template \( \tilde{f} : a \to a \) the same procedure does not lead to a valid definition yet:

\[
\begin{align*}
\tilde{f} \quad 1 \equiv 1\theta^{-1} \Rightarrow \tilde{f} \quad 1 \equiv 1 \\
\tilde{f} \quad 2 \equiv 2\theta^{-1} \Rightarrow \tilde{f} \quad 2 \equiv 2
\end{align*}
\]

The problem with interpreting these examples as a definition is the too specific type \( \textit{Int} \to \textit{Int} \) it would assign to \( \tilde{f} \).

In the following I take the liberty to sometimes consider example equations as part of definitions. While technically incorrect it allows terminology like pattern matching
on (the left-hand side of) equations, for example, \( f \ (x : \mathcal{S}) = \tau \), and allows to consider the type an equation gives a template.

The second step of generalisation is to make a new left-hand side consisting of arguments, variables or patterns, and then reconstruct the right-hand side as it was before constants from the arguments were substituted into it. This step too uses inverse substitution. The definition for \( \tilde{f} \) above has a single argument of type \( \alpha \). Naming the argument \( x \) the substitution to invert is \( \{1/x\} \) in the case of the first example and \( \{2/x\} \) in the case of the second. This gives:

\[
\begin{align*}
\tilde{f} \ x & = 1 \{1/x\}^{-1} \Rightarrow \tilde{f} \ x = x \\
\tilde{f} \ x & = 2 \{2/x\}^{-1} \Rightarrow \tilde{f} \ x = x
\end{align*}
\]

The result is two identical equations, leading to the definition \( \tilde{f} \ x = x \). If not all equations are equal a more refined technique must be used. Together the definitions of \( \alpha \) and \( f \) give the same definition for \textit{append} as seen in §6.1, which is equivalent to the standard definition in the Haskell Prelude. Remark that synthesis found a solution for \textit{append} without using enumeration. In this case the sub-definitions found for the templates were trivial and would be found immediately by enumeration, but with a more complex problem using generalisation can save a lot of work.

6.2.2 Further examples of generalisation

The examples in this section illustrate various aspects of generalisation. Ex. 6.3 explains why generalising higher-order values is difficult, Ex. 6.4 makes the case that having more than one example is useful, Ex. 6.5 shows the role of argument pattern-matching, and the section ends with some examples showing the problems of generalising functions that are not range-polymorphic.

**Ex. 6.3** Why is generalisation difficult for range-polymorphic functions that are not first-order? Consider this definition.

\[
\begin{align*}
\text{baz} :: (\alpha \rightarrow \text{Bool}) \rightarrow \alpha \rightarrow [\alpha] \\
\text{baz} \ p \ x &= \text{if } p \ x \ \text{then } [x] \ \text{else } []
\end{align*}
\]

What \text{baz} returns may depend only on argument \( p \):

\[
\begin{align*}
\text{baz} \ (> 0) \ 1 &= [1] \\
\text{baz} \ (> 1) \ 1 &= []
\end{align*}
\]

Therefore finding a definition by matching constants from these two equations is futile. The semantics of the predicates must be taken into account, and this is difficult in the general case. \( \diamond \)

Having more than one example is an advantage as it can prevent generalisation from drawing too specific conclusions.

**Ex. 6.4** Given the following generalisation problem involving a first-order range-polymorphic function represented by template \( \tilde{f} \).
\[ h : a \rightarrow [a] \]
\[ \tilde{h}\theta_1 \ 3 \ 3 = [9, 9, 3] \]
\[ \tilde{h}\theta_2 \ 2 \ 2 = [2, 7, 2] \]

where \( \theta_1 \) is \( \{z_1/9, \ z_2/9\} \), \( \theta_2 \) is \( \{z_1/2, \ z_2/7\} \), and each example of \( \tilde{h} \) is obtained from a different original example in a synthesis problem. Inverting the explicit substitution of each example, naming \( \tilde{h} \)'s argument \( y \) and then inverting the substitution of each argument gives:

\[ \tilde{h} \ y \ \hat{=} \ [z_1, \ z_1, \ y] \]
\[ \tilde{h} \ y \ \hat{=} \ [z_1, \ z_2, \ z_1] \]

Here a too specific example has resulted in a too specific equation. Note that the equations have equal left-hand sides, but different right-hand sides—they are invalid. These equations do not correspond to the same computation—but they should, given the type of \( \tilde{h} \). The answer is to perform the inverse substitution so that sub-expressions at the same positions in different equations are always replaced with the same variable. Said differently inverting substitution on equations should always make the same changes to each equation. With this refinement both examples generalise to the same equation,

\[ \tilde{h} \ y \ \hat{=} \ [z_1, \ z_2, \ y] \]
and this yields a solution.

Sometimes generalisation may need to be a little smarter.

**Ex. 6.5** Given this example and partial definition:

\[
\begin{align*}
\text{init} & : [a] \rightarrow [a] \\
\text{init} \ [1, 2, 3] & = [1, 2] \\
\text{init} \ & : [a] \rightarrow [a] \\
\text{init} \ xs \\
& = \text{unfold} \ \tilde{f} \ \tilde{p} \ \tilde{g} \ xs \\
\end{align*}
\]

Here it is possible to obtain examples for the templates, but it requires the factoring algorithm. The application of factoring is illustrated in Ex. 6.13 in §6.4; here the result of that example is used:

\[
\begin{align*}
\tilde{f} & : [a] \rightarrow a \\
\tilde{f}\theta_1 & = 1 \\
\tilde{f}\theta_2 & = 2 \\
\tilde{p} & : [a] \rightarrow \text{Bool} \\
\tilde{p}\theta & = \text{False} \\
\tilde{p}\theta & = \text{False} \\
\tilde{p}\theta & = \text{True} \\
\end{align*}
\]
where \( \theta \) is not relevant to the synthesis of template definitions and can be ignored, and the reduction \( \xi \) is tail. Consider first the equations for \( \xi \). Calling \( \xi \)'s argument \( ys \) and inverting substitutions \( \{1, 2, 3\}/ys \) and \( \{2, 3\}/ys \) yields

\[
\begin{align*}
\xi (1) &= 1 \\
\xi (2) &= 2
\end{align*}
\]

There were no matches for the value of \( ys \) and there remains a constant of type Int in both equations, which gives a too specific type \( \xi : [a] \to \text{Int} \). The first attempt at generalisation fails and these equations are discarded. As a refinement the algorithm hypothesises that the argument is a pattern \( (y:ys) \). The patterns create one binding for each variable, and inverting \( \{1/y, [2, 3]/ys\} \) and \( \{2/y, [3]/ys\} \) results in two identical equations,

\[
\begin{align*}
\xi (y:ys) &= y \\
\xi (y:ys) &= y
\end{align*}
\]

Following this the solution uses pattern matching, \( \xi (y:ys) = y \), which is equivalent to the \( \xi = \text{head} \).³

The examples for \( \phi \) prove more difficult. Introducing an argument \( ys \) in each case and inverting the corresponding substitutions yields these three examples:

\[
\begin{align*}
\phi (\text{False}) &= \text{False} \\
\phi (\text{False}) &= \text{False} \\
\phi (\text{True}) &= \text{True}
\end{align*}
\]

Note the identical left-hand sides and different right-hand sides: no function \( \phi \) can satisfy these equations. Introducing a pattern \( (y:ys) \) and inverting \( \{1/y, [2, 3]/ys\} \), \( \{2/y, [3]/ys\} \) and \( \{3/y, [\text{[]}]/ys\} \) gives

\[
\begin{align*}
\phi (y:ys) &= \text{False} \\
\phi (y:ys) &= \text{False} \\
\phi (y:ys) &= \text{True}
\end{align*}
\]

The equations still do not define a function. Note that \( (y:ys) \) is bound to \( 3:\text{[]} \) in the third equation. This means that the more specific pattern \( (y:[]) \), or \( [y] \), can be used instead.

\[
\begin{align*}
\phi (y:ys) &= \text{False} \\
\phi (y:ys) &= \text{False} \\
\phi (y:[]) &= \text{True}
\end{align*}
\]

Since \( ys \) is bound to non-empty lists in the first equations there is no overlap. This gives a definition that has the right type.

³Transforming definitions to remove patterns is a standard task for functional language compilers [70, §4].
6.2 MOTIVATION FOR GENERALISATION

\[
p :: [a] \to \text{Bool}
p (y:[]) = \text{True}
p (y:ys) = \text{False}
\]
and this can be transformed into \( p = \text{null . \text{tail} }. \) Note that the equation with the most specific left-hand side has been placed at the start—otherwise it would never be selected during evaluation. Observe that \( p \) is never applied to the empty list. The synthesis algorithm has found the following definition:

\[
\begin{align*}
\text{init} & :: [a] \to [a] \\
\text{init} \ \text{x} & = \text{unfold head (null . \text{tail}) \text{tail} x}\n\end{align*}
\]
which again is equivalent to a Haskell Prelude definition of the same name.

Here are some examples illustrating generalisation failing on functions that are not range-polymorphic:

**Ex. 6.6** Consider again the \texttt{map} function with its argument \( f :: a \to b. \)

\[
\begin{align*}
\text{map} & :: (a \to b) \to [a] \to [b] \\
\text{map \ (2+2)} [2, 3] & = [4, 5] \\
\text{map} \ f \ \text{x} & = \text{map} \ \bar{\text{g}} \ \bar{\text{a}}\bar{\text{s}} \ \text{x} \\
\end{align*}
\]

With the given partial definition this leads to these examples:

\[
\begin{align*}
\bar{\text{g}} & :: a \to b \\
\bar{\text{g}} \ 2 & = 4 \\
\bar{\text{g}} \ 3 & = 5 \\
\bar{\text{a}}\bar{\text{s}} & :: [b] \\
\bar{\text{a}}\bar{\text{a}}\bar{\text{a}} & = []
\end{align*}
\]

Generalising the examples for \( \bar{\text{g}} \) fails since there is no syntactic connection between constants 2 and 4, nor between 3 and 5. In this particular case, however, enumeration immediately finds argument \( f \) of \texttt{map} as the simplest solution for \( \bar{\text{g}} \) based on type-information. Template \( \bar{\text{a}}\bar{\text{s}} \) can be generalised as before; it represents a first-order range-polymorphic sub-problem.\(^\Diamond\)

**Ex. 6.7** Assume the following is part of some larger synthesis problem.

\[
\begin{align*}
\bar{\text{h}} & :: \text{Int} \to \text{Int} \to \text{Int} \\
\bar{\text{h}} \ 2 \ 3 & = 1 \\
\bar{\text{h}} \ 3 \ 1 & = 8
\end{align*}
\]
Again the template is not range-polymorphic and generalisation is again futile. The synthesis algorithm must enumerate definitions involving arithmetic operators from the Prelude if the answer is not found using \( \theta \).\(^\Diamond\)
6.3 Generalising example equations into definitions

Given a set \( E \) of examples of a function \( f \). A non-recursive solution to \( E \) is called a generalisation of \( E \). This terminology is motivated by Plotkin’s term ‘least general generalisation’. In this section I give an algorithm that can compute a definition for \( f \) that is a generalisation of \( E \), provided that \( f \) is a first-order range-polymorphic function. The algorithm uses syntactic matching and can be seen as an extension of Plotkin’s algorithm. As a preparation for this I first define some concepts.

I use a simple form of patterns in object-language definitions. It suffices to say that a pattern \( p \) matches values and creates bindings as follows: If \( p \) is a variable \( x \) then \( p \) can match any value \( t \) and \( x \) is bound to \( t \). If \( p \) is \( [] \) then it can only match the empty list and creates no bindings; if \( p \) is \( (p_1 : p_2) \) then it can match any non-empty list \( t_1 : t_2 \) provided that \( p_1 \) can match \( t_1 \) and \( p_2 \) can match \( t_2 \); the total number of bindings created is the sum of the bindings created by the two patterns. A pattern may not bind the same variable twice, and the expression where the bindings are used must be well-typed considering the bindings.

§5.1 introduces a syntactic category \( \text{subst} \), consisting of a sequence of bindings, in order to represent explicit substitutions. The generalisation algorithm constructs substitutions as part of its operation, and these are also represented using \( \text{subst} \). I use the notation \((t/x) \in \theta\) to refer to a binding \( t/x \) in a substitution \( \theta \). A pattern substitution is a sequence \( \{ \text{expr}_1 / \text{pat}_1, \ldots, \text{expr}_n / \text{pat}_n \} \), \( n \geq 0 \), where the syntactic category \( \text{pat} \) is a pattern of the kind described above. Application of a pattern substitution \( \theta \) to an expression \( t \) is defined as for ordinary substitution except that for \((t_i/p_i) \in \theta\) pattern \( p_i \) matches against \( t_i \) first and then the bound variables from \( p_i \) are substituted into \( t \). If one such matching fails during substitution, for example, matching \( [] \) against a non-empty list, then the substitution is not well-formed, and the result is undefined. In the following all pattern substitutions are assumed to be well-formed, and all constructed ones are. I use the notation \((t/x) \in \theta\) also if \( \theta \) is a pattern substitution and \( x \) is bound by a pattern binding in \( \theta \). Pattern substitution generalises ordinary substitution and gives a simpler formulation of generalisation. In the following I refer to both just as substitution.

Let \( \theta \) be the substitution \( \{t_1 / p_1, \ldots, t_n / p_n\} \) where for each \( i \) it holds that \( \{t_i / p_i\} \) has the same effect on expressions as the substitution \( \{t_1 / x_1, \ldots, t_n / x_n\} \). Then the inverse application of \( \theta \) to an expression \( t \), denoted \( t^{\theta^{-1}} \), is a new expression where, for \( i = 1, \ldots, n \), and for \( j = 1, \ldots, m_i \) each occurrence of \( t_i \) in \( t \) is replaced with \( x_i^j \).

Now I can define how to compute generalisations. A non-deterministic algorithm for computing generalisations of example equations—corresponding to the function \text{generalise} in Fig. 5.4—can be found in Fig. 6.2. This formulation assumes that all examples involve the same template \( t \), otherwise it must be applied to the examples of each template separately. Function \text{instPat}(p, \alpha)\ returns a pattern equal to \( p \) except that all variables \( x \) in \( p \) such that \((t/x) \in \alpha\) have been replaced with pattern \( [] \).

\textit{Splitting} a variable \( x \) occurring in a pattern \( p \) means modifying \( p \) by replacing \( x \) with \((u_1 : u_2) \) where \( u_1 \) and \( u_2 \) are fresh variables.

Comments to the steps of the algorithm and references to relevant examples:
6.3 GENERALISING EXAMPLE EQUATIONS INTO DEFINITIONS

Given a non-empty set $E$ of example equations of a template $\bar{v} :: \tau$ of a non-recursive function,$\quad E = \{ t_1 \bar{v} t_2 \ldots t_n \bar{v} t_{n+1} | i = 1, \ldots, k \}.$

If $\bar{v}$ is not first-order range-polymorphic then return $\emptyset$.

1. Let $p_1, \ldots, p_n$ be patterns in the form of distinct fresh variables.

2. Let $s_i$ be $t_{i(n+1)} \theta_i^{-1}$ for $i = 1, \ldots, k$, but always replacing expressions at identical positions in the $s_i$'s with the same variable.

3. Let $\alpha_i$ be the substitution $\{ t_{i(1)} / p_1, \ldots, t_{i(k)} / p_n \}$ for $i = 1, \ldots, k$.

4. Let $l_i$ be $\bar{v} \text{instPat}(p_1, \alpha_i) \ldots \text{instPat}(p_n, \alpha_i)$ and let $r_i$ be $s_i \alpha_i^{-1}$, for $i = 1, \ldots, k$.

5. Let $E'$ be $\{ l_i \bar{v} r_i | i = 1, \ldots, k \}$. If there is an equation $l_j \bar{v} r_j \in E'$ that assigns a type more specific than $\tau$ to $\bar{v}$ due to a constant $c$ occurring in $r_j$ then select a variable $x$ occurring in $l_j$, such that $(t/x) \in \alpha_j$ and $c$ occurs in $t$, split $x$ and goto 3.

6. If there is an equation $l \bar{v} r \in E'$ such that the set $E_i \neq \emptyset$,

$$E_i = \{ (|l'| \bar{v} r' \in E', l', r' \neq r) \},$$

then select a variable $x$ occurring in $l$, split $x$ and goto 3.

7. Convert each distinct equation $\bar{v} p_1 \ldots p_n \bar{v} r$ in $E'$ into a definition equation $v p_1 \ldots p_n = r$, placing those with more specific left-hand sides at the start. Calling the resulting definition $D$, return $\{ D \}$.

Backtrack to different choices if needed. If there are no more backtracking possibilities return $\emptyset$.

**Fig. 6.2**: Generalisation algorithm.
• Step 2: The motivation for the 'same variable' restriction can be found in Ex. 6.4.

• Steps 3 and 4: See the comments for templates $\bar{f}$ and $\bar{p}$ in Ex. 6.5. The purpose of function $\text{instPat}$ is to simplify the patterns of equations matching on the empty list. If a pattern variable $x$ matches the empty list, it is pointless to introduce $x$ on the right-hand side or to split it.

• Step 5 modifies a pattern to introduce a binding for the constant $c$ that makes the type too specific. If before the pattern is modified $(t/x) \in \alpha_j$ and $t$ is $[c, \ldots]$ then $c$ inside $r_j$ will be replaced with a fresh variable $u_1$ by inverse substitution of $\alpha_j$ since, after the splitting of $x$ into $(u_1 : u_2)$ there is a binding for $c$, $(c/u_1) \in \alpha_j$. If instead $c$ occurs deeper inside $t$ then several splittings are needed before $c$ can be eliminated from $r_j$.

• In step 6 variable $x$ must be chosen such that the left-hand sides of the conflicting equations bind $x$ to expressions with different list structure, if possible. Function $\text{instPat}$ then gives such equations distinct left-hand sides.

• Step 7: See Ex. 6.5, template $\bar{p}$.

Remark that the generalisation algorithm only replaces constants with variables on the right-hand side, it never replaces expressions containing variables. Initially all $l_i$’s are the same and equal to $\bar{f} x_1 \cdots x_n$ where the $x_i$’s are variables. The left-hand sides only differ if a variable is split. Splitting causes the algorithm to iterate, but patterns cannot be split indefinitely since at some point a binding for every right-hand side constant is created on the left-hand side, and then the equations must be distinct since the original equations $E$ were distinct. The variable patterns of the explicit substitutions cannot be split since the definition does not introduce these bindings.

**Ex. 6.8** Note that the algorithm cannot find a solution if the examples specify an impossible problem:

\[
\bar{f} :: a \rightarrow a
\]

\[
\bar{f} \ 1 = 2
\]

\[
\bar{f} \ 2 = 3
\]

The definition of inverse substitution affects what kind of solution the algorithm computes. Given:

\[
\bar{f} :: a \rightarrow a \rightarrow a
\]

\[
\bar{f} \ 1 \ 1 = 1
\]

both these definitions are solutions:

\[
f \ x \ y = x
\]

\[
f \ x \ y = y
\]

The algorithm currently returns the first.
6.4 Motivation for factoring

It is possible, at the expense of clarity, to write deterministic version of generalise where all choices and priorities are made explicit. I believe that the algorithm fails only if there are no solutions with the given type for the examples, but I have no proof (see §7.2). The generalisation algorithm should be allowed to reject unlikely definitions when further computation yields more promising ones; criteria for this are further work. Note that generalised definitions are always canonical since their right-hand sides only consist of constructors and variables.

Generalisation is related to Plotkin's lgg algorithm. Let $E = \{e_1, e_2, \ldots, e_k\}$ and assume that there are no explicit substitutions. Then

$$\text{generalise}(\bar{v}, E) = \text{lgg}(e_1, \text{lgg}(e_2, \ldots \text{lgg}(e_{k-1}, e_k) \ldots))$$

provided that lgg is required to introduce the same distinct variables on every left-hand side, that pattern matching is not used by generalise and that generalise does not fail. The generalisation algorithm can be seen as grouping the examples in $E$ and computing one lgg for each group. This is the reverse of pattern matching in functional programming—constructing patterns and bindings from instances.

6.4.1 Constant matching and rewriting

Like the generalisation algorithm the factoring algorithm is based on syntactic matching of constants occurring in an equation. Therefore, given an equation recursive in a template $\bar{v}$, the algorithm only applies factoring to the equation when $\bar{v}$ represents a range-polymorphic function. Template $\bar{v}$ need not necessarily represent a first-order function.

Ex. 6.9 Given the following example and partial definition of a function revdouble.

```
revdouble :: [a] -> [a]
revdouble [1, 2] = [2, 2, 1, 1]

revdouble :: [a] -> [a]
revdouble xs = foldl h \ c xs
```
The synthesis algorithm computes
\[
\text{revdouble} \ [1, 2] \equiv [2, 2, 1, 1] \\
\Rightarrow (\lambda x \cdot \text{fold1} \ \tilde{h} \ \tilde{c} \ x) \ [1, 2] \equiv [2, 2, 1, 1] \\
\Rightarrow \text{fold1} \ \tilde{h} \ \tilde{c} \ [1, 2] \equiv [2, 2, 1, 1] \\
\Rightarrow \tilde{h} \ (\tilde{h} \ \tilde{c} \ 1) \ 2 \equiv [2, 2, 1, 1]
\]
where \( \theta = [(1, 2)/x] \). The equation is suspended on template \( \tilde{h} :: [a] \to a \to [a] \), it also involves another template \( \tilde{c} :: [a] \), and it is recursive in \( \tilde{h} \). Such equations are not amenable to generalisation: matching of constants is unlikely to find a definition from this equation that relate \( \tilde{h} \) to itself.

The purpose of factoring is to obtain example equations for the templates \( \tilde{h} \) and \( \tilde{c} \). Knowing that \text{revdouble} is range-polymorphic the equation can be made simpler using the following heuristic assumption: All occurrences of a constant \( c :: a \) on the right-hand side result from one application \( \tilde{h} \ t \ c \) on the left-hand side; note that \( \tilde{h} \)'s first argument is the recursion argument of \text{revdouble}. This heuristic, which I refine later, allows two new equations to be deduced from the above equation:
\[
\tilde{h} \ (\tilde{h} \ \tilde{c} \ 1) \ 2 \equiv [2, 2, 1, 1] \\
\Rightarrow \tilde{h} \ \tilde{c} \ 1 \equiv [1, 1] \\
\Rightarrow \tilde{c} = []
\]
The third equation is already an example for \( \tilde{c} \), and using this example as a rewrite rule by ordering it left to right, the second equation can be made into an example and a rewrite rule, and used in turn on the first equation. The original recursive equation has been factored into three example equations for two templates,
\[
\tilde{h} \ [1, 1] \ 2 \equiv [2, 2, 1, 1] \\
\tilde{h} \ [1] \ 1 \equiv [1, 1] \\
\tilde{c} = []
\]
and definitions for the templates can be found using generalisation. Let \( \alpha_1 \) be \{2/y, [1, 1]/ys\} and \( \alpha_2 \) be \{1/y, []/ys\}.
\[
\tilde{h} \ ys \ y \equiv [2, 2, 1, 1] \theta^{-1} \alpha_1^{-1} \\
\tilde{h} \ ys \ y \equiv [1, 1] \theta^{-1} \alpha_2^{-1} \\
\Rightarrow \\
\tilde{h} \ ys \ y \equiv y:y:ys \\
\tilde{h} \ ys \ y \equiv y:y:ys \\
\tilde{c} \equiv [] \theta^{-1} \Rightarrow \tilde{c} \equiv []
\]
Together this leads to the following definition.
\[
\text{revdouble} :: [a] \to [a] \\
\text{revdouble} \ x s \\
= \text{let} \ h \ y s \ y = y:y:ys \\
c = []
\]
This function reverses its list argument while duplicating each individual element.

### 6.4.2 Further examples of factoring

Overview of the examples in this section: Ex. 6.10 shows that schemas must sometimes be explicitly eliminated to enable factoring. Ex. 6.11 illustrates the interplay between if-statements and factoring while Ex. 6.12 shows a case where the unfolding of schema recursion requires special attention. The last example, Ex. 6.13, gives a factoring used previously in §6.2.2.

**Ex. 6.10** This example is similar to Ex. 6.9; the function described by the example is `double` which repeats each element in a list. This time the partial definition uses the schema `foldr` and this complicates matters.

```haskell
double :: [a] -> [a]
double [1, 2] = [1, 1, 2, 2]

double :: [a] -> [a]
double xs = foldr g (g xs)

double [1, 2] = [1, 1, 2, 2] 
\Rightarrow \text{g} 1 (\text{foldr} \text{g} \text{g} [2]) = [1, 1, 2, 2]
```

where \( \theta = \{[1, 2]/\text{xs}\} \). The recursive equation is suspended on \( \text{g} \) and it also uses `foldr`. The schema is not eliminated by evaluation since \( \text{g} \) is assumed to be non-strict by default. A refinement of the heuristic used by factoring is to evaluate arguments eagerly to eliminate schemas.

```haskell
\Rightarrow \text{g} 1 (\text{foldr} \text{g} \text{g} \text{g} [2]) = [1, 1, 2, 2] 
\Rightarrow \text{g} 1 (\text{g} 2 (\text{foldr} \text{g} \text{g} \text{g} [1])) = [1, 1, 2, 2] 
\Rightarrow \text{g} 1 (\text{g} 2 \text{g} \text{g}) = [1, 1, 2, 2]
```

As in the previous example the new equation can be rewritten into example equations.

```haskell
\text{g} 1 [2, 2] = [1, 1, 2, 2] 
\text{g} 2 [1] = [2, 2] 
\text{g} \text{g} = []
```

where \( \theta = \{[1, 2]/\text{xs}\} \). Generalisation now gives the definitions \( g \ y \ y s = y:y:y s \) and \( b = [1] \), and a correct definition of `double`.

**Ex. 6.11** Here is a problem involving the familiar `filter` function from the Haskell Prelude.
filter :: (a -> Bool) -> [a] -> [a]
filter (\x -> x /= 0) [3, 0, 1, 0, 2] \= [3, 1, 2]

filter :: (a -> Bool) -> [a] -> [a]
filter p xs = foldr \x \x

This gives
\[ \tilde{f} 3 (\tilde{f} 0 (\tilde{f} 1 (\tilde{f} 2 \tilde{a}))) \] \= [3, 1, 2]

where \( \theta = \{ (\lambda x \rightarrow x /= 0)/p, [3, 0, 1, 0, 2]/xs \} \). Proceeding as before gives the equation set
\[
\begin{align*}
\tilde{f} & : a \rightarrow [a] \rightarrow [a] \\
\tilde{f} 3 [1, 2] & \equiv [3, 1, 2] \quad \text{-- } e_1 \\
\tilde{f} 0 [1, 2] & \equiv [1, 2] \quad \text{-- } e_2 \\
\tilde{f} 1 [2] & \equiv [1, 2] \\
\tilde{f} 0 [2] & \equiv [2] \\
\tilde{f} 2 [] & \equiv [2] \\
\tilde{a} & : [a] \\
\tilde{a} \theta & \equiv []
\end{align*}
\]

Template \( \tilde{a} \) is no problem, but generalising the example equations for \( \tilde{f} \) fails. Note that equations \( e_1 \) and \( e_2 \) above have identical structure on the left-hand side and different structure on the right-hand side. But, given that \( \tilde{f} \) is first-order range-polymorphic it cannot distinguish the two left-hand sides.

The enumerations for \( \tilde{f} \) include a definition with an if-statement.

\[ f \ z \ zs = \text{if } \tilde{b} \text{ then } \tilde{e}_1 \text{ else } \tilde{e}_2 \]

With this definition the first example equation for \( \tilde{f} \) becomes:

\[
\begin{align*}
f \theta 3 [1, 2] & \equiv [3, 1, 2] \\
\Rightarrow (\lambda z \ z s \rightarrow \text{if } \tilde{b} \text{ then } \tilde{e}_1 \text{ else } \tilde{e}_2) \theta 3 [1, 2] & \equiv [3, 1, 2] \\
\Rightarrow \text{if } \tilde{b} \phi \text{ then } \tilde{e}_1 \phi \text{ else } \tilde{e}_2 \phi & \equiv [3, 1, 2]
\end{align*}
\]

where \( \phi = \theta \cdot \{3/z, [1, 2]/zs\} \). The equation is suspended on \( \tilde{b} \) and the algorithm enumerates definitions of \( \tilde{b} \) using \( p :: a \rightarrow \text{Bool} \):

\[ \begin{align*}
b & = p \ z \\
b & = \text{not } (p \ z)
\end{align*} \]

Using the first (simplest) solution allows the equation to be simplified into an example of \( \tilde{e}_1 \).
6.4 Motivation for Factoring

if (p z) \( \phi \) then \( \bar{\phi}_1 \) else \( \bar{\phi}_2 \) = [3, 1, 2]
\[ \Rightarrow \] if ((\( \lambda x \rightarrow x /= 0 \)) z) \( \phi \) then \( \bar{\phi}_1 \) else \( \bar{\phi}_2 \) = [3, 1, 2]
\[ \Rightarrow \] if ((\( \lambda x \rightarrow x /= 0 \)) 3) then \( \bar{\phi}_1 \) else \( \bar{\phi}_2 \) = [3, 1, 2]
\[ \Rightarrow \] \( \bar{\phi}_1 \) = [3, 1, 2]

This is an equation for the then-part of the if-statement. Doing the same for all other equations for \( \bar{f} \) gives the following equation set.

\[ \bar{\phi}_1(\theta\{3/z, \{1, 2\}/zs\}) = [3, 1, 2] \]
\[ \bar{\phi}_2(\theta\{0/z, \{1, 2\}/zs\}) = [1, 2] \]
\[ \bar{\phi}_1(\theta\{1/z, \{2\}/zs\}) = [1, 2] \]
\[ \bar{\phi}_2(\theta\{0/z, \{2\}/zs\}) = [2] \]
\[ \bar{\phi}_1(\theta\{2/z, \{\}\}/zs\}) = [2] \]

Generalising \( \bar{\phi}_1, \bar{\phi}_2, \) and \( \bar{a} \) gives

\begin{verbatim}
filter p xs
  = let f z zs = if p z then z:zs else zs
      a = []
in
    foldr f a xs
\end{verbatim}

which is equivalent to the Haskell Prelude definition of filter.

Had the original example for filter instead been

\begin{verbatim}
filter :: (a -> Bool) -> [a] -> [a]
\end{verbatim}

the factoring heuristic gets the algorithm into trouble. Here the two occurrences of 1 on the right-hand side both get deleted upon seeing the first 1 on the left-hand side:

\[ \bar{f}\theta 1 (\bar{f}\theta 2 (\bar{f}\theta 1 \bar{a}\theta)) = [1, 2, 1] \]
\[ \Rightarrow \bar{f}\theta 2 (\bar{f}\theta 1 \bar{a}\theta) = [2] \]
\[ \Rightarrow \bar{f}\theta 1 \bar{a}\theta = [1] \]
\[ \Rightarrow \bar{a}\theta = [] \]

where \( \theta = \{(\lambda x \rightarrow x /= 0)/p, \{1, 2, 1\}/xs\} \). This leads to the following bogus equation set:

\begin{verbatim}
\bar{f}\theta 1 [2] = [1, 2, 1]
\bar{f}\theta 2 [] = [2] -- e_3
\bar{f}\theta 1 [1] = [] -- e_4
\bar{a}\theta = []
\end{verbatim}

Introducing an if-statement does not help here since \( e_3 \) and \( e_4 \) cannot be distinguished by \( p \). Therefore, given a template applied to a constant \( c \) on the left-hand side, the factoring heuristic must, as an alternative, delete only one occurrence of \( c \) from the right-hand side, or in general, delete a certain selection of such occurrences. \( \diamond \)
The functions involved in the examples seen so far have something in common: they always walk over all of the recursion argument \( \text{xs} \). When this is not the case the factoring algorithm must be refined. The next example shows why.

**Ex. 6.12** Example equations and partial definition for another Haskell Prelude function:

\[
\text{takeWhile } :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]
\]

\[
\text{takeWhile } (\lambda x \rightarrow x \not= 0) \ [3, 1, 2, 0, 2] \ = \ [3, 1, 2]
\]

\[
\text{takeWhile } (\lambda x \rightarrow x \leq 10) \ [7, 8, 9] \ = \ [7, 8, 9]
\]

\[
\text{takeWhile } :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]
\]

\[
\text{takeWhile } p \text{ xs} = \text{foldr } \mathfrak{I} \ \& \text{ xs}
\]

This gives

\[
\begin{align*}
\mathfrak{I} &\ 3 \ [1, 2] \ = \ [3, 1, 2] \\
\mathfrak{I} &\ 1 \ [2] \ = \ [1, 2] \\
\mathfrak{I} &\ 2 \ [] \ = \ [2] \quad -- \ e_1 \\
\mathfrak{I} &\ 0 \ (\mathfrak{I} \ 2 \ \& \mathfrak{I}) \ = \ [] \quad -- \ e_2 \\
\mathfrak{I} &\ 7 \ [8, 9] \ = \ [7, 8, 9] \\
\mathfrak{I} &\ 8 \ [9] \ = \ [8, 9] \\
\mathfrak{I} &\ 9 \ [] \ = \ [9] \\
\& \mathfrak{I} &\ 2 \ = \ [] \quad -- \ e_3
\end{align*}
\]

where \( \theta_1 = \{ (\lambda x \rightarrow x \not= 0)/p, [3, 1, 2, 0, 2]/\text{xs} \} \) and \( \theta_2 = \{ (\lambda x \rightarrow x \leq 0)/p, [7, 8, 9]/\text{xs} \} \). These equations are example equations, except the one marked \( e_2 \). Equation \( e_2 \) has not been simplified further since the right-hand side is the empty list, and this value is less likely to be the result of a recursive computation than other lists. Stopping the factoring at this point is a heuristic, but in this particular case fully justified since further simplification of \( e_2 \), together with \( e_3 \) results in the following equations,

\[
\begin{align*}
\mathfrak{I} &\ 0 \ [] \ = \ [] \\
\mathfrak{I} &\ 2 \ [] \ = \ []
\end{align*}
\]

where the latter example contradicts \( e_1 \) and prevents solutions. Ignoring \( e_2 \) and enumerating a definition of \( \mathfrak{I} \) using an if-statement, leads to the expected solution for \( \text{takeWhile} \).

\( \diamond \)

So far the factoring examples have all involved the standard folds. The next example uses the unfold schema, and it shows how to compute the factoring previously used to motivate generalisation in Ex. 6.5.
6.4  MOTIVATION FOR FACTORING

Ex. 6.13  Given

\[ \text{init :: } [a] \to [a] \]
\[ \text{init } [1, 2, 3] = [1, 2] \]
\[ \text{init :: } [a] \to [a] \]
\[ \text{init } \text{xs} \]
\[ = \text{unfold } \tilde{f} \tilde{p} \tilde{g} \text{ xs} \]

Evaluating the example:
\[ \text{init } [1, 2, 3] = [1, 2] \]
\[ \Rightarrow \]
\[ \text{if } \tilde{p} [1, 2, 3] \]
\[ \text{then } [] \]
\[ \text{else } \tilde{f} [1, 2, 3]: (\text{unfold } \tilde{f} \tilde{p} \tilde{g} \theta (\tilde{g} \theta [2, 3])) = [1, 2] \]

where \( \theta \) is \([1, 2, 3]/\text{xs} \) and the if-statement is introduced by desugaring the guard in the definition of unfold from Fig 4.3. This equation is suspended on \( \tilde{p} \). Note that the then-part of the if-statement is \([1\), so here \( \tilde{p} \) \( \text{xs} \) can only be \text{True} when the right-hand side is the empty list too. Template \( \tilde{g} \) is a reduction, and the algorithm first tries the simplest list reduction, \text{tail}, from Fig 4.6. Simplifying and reasoning the same way:

\[ \tilde{p} \theta [1, 2, 3] = \text{False} \]
\[ \tilde{f} \theta [1, 2, 3] = 1 \]
\[ \text{if } \tilde{p} [2, 3] \]
\[ \text{then } [] \]
\[ \text{else } \tilde{f} [2, 3] : (\text{unfold } \tilde{f} \tilde{p} \text{ tail (tail [2, 3]))} = [2] \]
\[ \Rightarrow \]
\[ \tilde{p} \theta [1, 2, 3] = \text{False} \]
\[ \tilde{f} \theta [1, 2, 3] = 1 \]
\[ \tilde{p} \theta [2, 3] = \text{False} \]
\[ \tilde{f} \theta [2, 3] = 2 \]
\[ \text{if } \tilde{p} [3] \]
\[ \text{then } [] \]
\[ \text{else } \tilde{f} [3] : (\text{unfold } \tilde{f} \tilde{p} \text{ tail (tail [3]))} = [] \]

Here the right-hand side is the empty list so recursion must terminate. The result is the set of examples seen in Ex 6.5.

\[ \tilde{p} \theta [1, 2, 3] = \text{False} \]
\[ \tilde{f} \theta [1, 2, 3] = 1 \]
\[ \tilde{p} \theta [2, 3] = \text{False} \]
\[ \tilde{f} \theta [2, 3] = 2 \]
\[ \tilde{p} \theta [3] = \text{True} \]
6.5 Factoring recursive equations into example equations

Here I describe an algorithm corresponding to the function factor in Fig. 5.4. The description is split in several parts, depending on the schema used in the recursive equation to be factored. §6.5.1 contains a factoring algorithm for equations involving the standard folds, or no schemas, and §6.5.2 has rules for how to factor equations involving the unfolds. Equations involving the myopic folds are factored trivially by equation simplification.

Regardless of schema usage the factoring algorithm has these applicability conditions:

1. The recursion argument and the return value must be lists, but neither can be a list of lists.

2. The template in which the equation is recursive must be range-polymorphic.

If these conditions are not met then factor(E) returns \(\{E\}\). Factoring with other argument types is further work.

Let \(S(E)\) denote the set of solutions of an equation set \(E\). Let \(e_r\) be a general equation, for example, a recursive equation, and let \(E\) be a set of example equations such that \(S(E) \neq \emptyset\). If \(S(E) \subseteq S(\{e_r\})\) then \(E\) is a factoring of \(e_r\). A good factoring is one that preserves many solutions from the general equation. The definition extends to several general equations by taking the union of their solutions.

6.5.1 Factoring standard fold equations

Given an equation involving one of the standard folds, fold1, foldl1, foldr or foldl, or an equation involving no schema. The factoring algorithm for such equations is given in Fig. 6.3; the \(q\)-th argument is the recursion argument. Function reduceEqn(\(l = r, q\)) used in the figure computes a simpler equation \(l' = r'\) from a given equation \(l = r\) as follows: if \(l = \bar{v}_t_1 \cdots t_n\) then \(l' = t_q; r' = r\) except that for each constant \(c\) in \(t_i\) for all \(i \neq q\), a selection of occurrences of \(c\)'s has been deleted from \(r\). The same occurrences must be selected each time if \(\bar{v}\) is first-order range-polymorphic, otherwise several possibilities must be tried; see Ex. 6.10 and Ex. 6.11. Each selection tried represents an alternative choice for \(r\) and leads to an alternative equation set and an alternative factoring. The maintenance of a set of alternative factorings originating from different selections has been left out of Fig. 6.3.

Comments to the algorithm:

- Step 1: Consider the equation after eliminating any schema present. Writing applied templates \(\bar{v}\) as nodes and their arguments \(t_i\) as leaves shows that the
6.5. FACTORING RECURSIVE EQUATIONS

Given an equation \( l = r \) recursive in a range-polymorphic template \( \bar{v} \), directly or through one of the standard folds, and such that \( \bar{v} \)'s recursion argument is a list and it returns a list (but neither is a list of lists).

1. If \( l \) has form \( scm \ldots \) then let \( l \) be \( \text{eval}(l, \_\_\_) \) and the same for the new \( l \), recursively eliminating schemas from the left-hand side. Now the equation has form

\[
\bar{v}\theta \ t_1 \ldots \ t_n \ = \ [s_1, \ldots, s_m]
\]

where \( s_1, \ldots, s_m \) are constants, \( t_i \)'s except \( t_q \) are constants or other templates, \( \theta \) is an explicit substitution, and \( \bar{v} \) may only occur in \( t_q \), and, if it does, \( t_q \) has the same structure as \( l \), and so on recursively for the \( q \)-th sub-expression of \( t_q \).

2. \( E := \emptyset \).

3. If \( r = [\ldots] \) then goto 6.

4. \( E := E \cup \{ l = r \} \).

5. If \( l \) contains more than one occurrence of \( \bar{v} \) then let \( l = r \) be \( \text{reduceEqn}(l = r, q) \) and goto 3.

6. Let \( E' \) be the set of all equations in \( E \), but where each left-hand side is rewritten as far as possible using as rules the other equations in \( E \) ordered left to right.

**Fig. 6.3:** Factoring standard fold equations.
left-hand side has the structure of a tree with a chain of \( \bar{v} \)'s at the trunk,

\[
\bar{\theta} \quad \doteq \quad [s_1, \ldots, s_m].
\]

The trunk is at argument number \( q \); I use superscripts to distinguish the arguments of the \( k \) different \( \bar{v} \)'s. The effect of \texttt{reduceEqn} on the left-hand side of an equation is to cut off the top node of the tree together with its leaves.

- Step 3: See Ex. 6.12 for the rationale behind this test.
- Step 6: See, for example, Ex. 6.9.

Each standard fold has an argument called \( f \), and the arguments to \( f \) includes the value computed by the recursive call of the schema. This can lead to recursive equations for \( f \), and thereby to solutions using schemas. See Ex. 7.1 in §7.1.

### 6.5.2 Factoring unfold equations

Before showing how to factor equations using unfolds it is instructive to look at the case of the myopic fold \texttt{mapr} (\texttt{map1} is similar). Consider the following equation.

\[
\text{mapr } f \bar{\theta} \ a \bar{\theta} \ [x_1, \ldots, x_n] \doteq [y_1, \ldots, y_m]
\]

Such an equation could result from an example equation and a \texttt{mapr}-based partial definition of an unknown function. Using just the function \texttt{simplify} from Fig. 5.3 yields the following set of equations.

\[
\begin{align*}
\bar{f} \theta & \ x_1 \doteq y_1 \\
\vdots & \\
\bar{f} \theta & \ x_n \doteq y_n \\
\bar{a} \bar{\theta} \doteq [y_{n+1}, \ldots, y_m]
\end{align*}
\]

provided that \( n < m \). If \( n = m \) the last equation becomes

\[
\bar{a} \bar{\theta} \doteq [\].
\]

The original equation is reduced to a set of equivalent equations, and these represent a factoring obtained ‘for free’. If \( n > m \) the following unsatisfiable equation results,

\[
[\bar{f} \ x_{m+1}, \ldots, \bar{f} \ x_n] \doteq [\].
\]
6.5. FACTORING RECURSIVE EQUATIONS

Given equation

\[ \text{unfold } \tilde{f} \theta \bar{p} \theta \text{ tail } [x_1, \ldots, x_n] \doteq [y_1, \ldots, y_m] \]

for some explicit substitution \( \theta \). By definition \( 1 \leq k \leq n+1 \) and the following equation holds:

\[ \bar{p} \theta \ [x_k, \ldots, x_n] \doteq \text{True} \]

The following analysis cover all possibilities for \( k \).

Case \( k \leq m \): Regardless of \( n \) the following equation must be satisfied.

\[ [\ ] \doteq [y_k, \ldots, y_m] \]

The right-hand side is non-empty and the equation is unsatisfiable. This case has no solutions, and therefore no factorings.

Case \( k = m+1 \): Here \( n \geq m \). Equations:

\[ F_{k-1} \cup \{ \bar{p} \theta \ [x_k, \ldots, x_n] = \text{true} \} \]

Case \( k > m+1 \): Here \( n > m \). This also gives an unsatisfiable equation:

\[ [\tilde{f} \ [x_m, \ldots, x_n], \ldots, \tilde{f} \ [x_{k-1}, \ldots, x_n]] = \square. \]

Fig. 6.4: Factoring of unfold equations with reduction tail.

Given equation

\[ \text{gunfold } \tilde{f} \theta \bar{p} \theta \text{ tail } \tilde{h} \theta \ [x_1, \ldots, x_n] \doteq [y_1, \ldots, y_m] \]

for some explicit substitution \( \theta \). The following equations must be satisfied:

\[ F_{k-1} \cup \{ \bar{p} \theta \ [x_k, \ldots, x_n] = \text{true}, \ \tilde{h} \theta \ [x_k, \ldots, x_n] = [y_k, \ldots, y_m] \}. \]

Fig. 6.5: Factoring of gunfold equations with reduction tail.

the consequence being that there are no solutions to the set of equations, and therefore no factoring.

As shown in §5.8 mapr is a special case of gunfold, and gunfold also generalises unfold. Equations involving unfolds can be factorized in a way that generalises the above trivial factoring of equations involving mapr. For convenience in the following discussion I introduce this definition:

\[ F_k \doteq \left\{ \begin{array}{l}
\bar{p} \theta \ [x_1, \ldots, x_n] = \text{false}, \ldots, \bar{p} \theta \ [x_k, \ldots, x_n] = \text{false}, \\
\tilde{f} \theta \ [x_1, \ldots, x_n] = y_1, \ldots, \tilde{f} \theta \ [x_k, \ldots, x_n] = y_k
\end{array} \right\} \]

for \( 0 < k \leq n \) and \( F_0 = \emptyset \) otherwise. Symbols \( \tilde{f}, \tilde{g}, \bar{p} \) and \( \theta \) will be clear from the context. The notation \([x_1, \ldots, x_n]\) means \( [\ ]\) when \( l_1 > l_2 \). Fig. 6.4 shows how recursive equations using unfold with \( g = \text{tail} \) can be factored and Fig. 6.5 shows the same for gunfold, also with reduction tail. In both figures \( k, 1 \leq k \leq n+1 \), is defined by \( \bar{p} \) returning \( \text{true} \) for the first time at the list starting with \( x_k \), or \( k = n+1 \).
if \( p \) returns True the first time on \([\ ]\). The two cases shown here assume that \( g = \text{tail} \); similar analyses can be carried out for unfolds using other reductions. Factorings in the two figures implicitly includes the following equation.

\[ p \theta \ [\ ] = \text{True} \]

That is, the schema does not walk past the end of the list to return undefined. Following this, \( \hat{p} \) must have a definition on the form \( p \text{xs} = \text{null xs} \mid \mid q \) for some \( q : \text{Bool} \).

Template \( p \) and the number \( k \) are the key to factoring an equation. For unfold the choice \( k = m + 1 \) is the only possibility since the base-case returns \([\ ]\), and this means there can be at most one factoring. In the case of gunfold there can be up to \( n + 1 \) factorings corresponding to the range of \( k \). Valid factorings are those where the equations in Fig. 6.4 or Fig. 6.5 are satisfiable. Since there is overlap between the equations for cases \( l \) and \( l+1 \) an implementation should use an incremental computation when factoring gunfold equations.

**Ex. 6.14** The below synthesis problem involves the function tails from the standard Haskell module List, a partial definition using gunfold and reduction tail.\(^4\)

\[
\text{tails} :: [a] \rightarrow [[a]]
\]

\[
\text{tails} [1, 2, 3] = [[1, 2, 3], [2, 3], [3], []]
\]

\[
\text{tails} :: [a] \rightarrow [[a]]
\]

\[
\text{tails xs} = \text{gunfold } \hat{f} \hat{p} \hat{g} \hat{h} \text{ xs}
\]

\[
\text{tails } [1, 2, 3] = [[1, 2, 3], [2, 3], [3], []]
\]

\[
\Rightarrow \text{gunfold } \hat{f} \theta \hat{p} \theta \text{ tail } \hat{h} \theta [1, 2, 3] = [[1, 2, 3], [2, 3], [3], []]
\]

There are 4 factorings; let \( E_k \) denote the equation set for \( k = l \):

\[
E_1: \begin{cases} 
\hat{p} \theta [1, 2, 3] = \text{True} & \quad \Rightarrow F_0 = \emptyset \\
\hat{h} \theta [1, 2, 3] = [[1, 2, 3], [2, 3], [3], []] 
\end{cases}
\]

\[
E_2: \begin{cases} 
\hat{p} \theta [1, 2, 3] = \text{False} \\
\hat{f} \theta [1, 2, 3] = [1, 2, 3] \\
\hat{p} \theta [2, 3] = \text{True} \\
\hat{h} \theta [2, 3] = [[2, 3], [3], []]
\end{cases}
\]

\[
E_3: \begin{cases} 
\hat{p} \theta [1, 2, 3] = \text{False} \\
\hat{f} \theta [1, 2, 3] = [1, 2, 3] \\
\hat{p} \theta [2, 3] = \text{False} \\
\hat{f} \theta [2, 3] = [2, 3] \\
\hat{p} \theta [3] = \text{True} \\
\hat{h} \theta [3] = [[3], []]
\end{cases}
\]

\(^4\)The range of tails is a list of lists, but this case is solvable by factoring.
6.5. FACTORING RECURSIVE EQUATIONS

$$E_4:$$

$$\begin{align*}
\bar{p} \ [1, 2, 3] &:= \text{False} \\
\bar{t} \ [1, 2, 3] &:= [1, 2, 3] \\
\bar{p} \ [2, 3] &:= \text{False} \\
\bar{t} \ [2, 3] &:= [2, 3] \\
\bar{p} \ [3] &:= \text{False} \\
\bar{t} \ [3] &:= [3] \\
\bar{p} \ [] &:= \text{True} \\
\bar{t} \ [] &:= [[]]
\end{align*}$$

where $$\theta = \{(1, 2, 3) / x \}$$. All 4 factorings lead to solutions, but the simplest solution is obtained by using $$E_4$$:

```haskell
tails :: [a] -> [[a]]
tails xs = let f = id
            p = null
            g = tail
            h = (:) []
            in gunfold f p g h xs
```

This is the expected definition. In the other solutions the sub-definitions for $$\bar{p}$$ and $$\bar{t}$$ found by generalisation introduce more complicated pattern matching.

In general, consider a function $$f$$ that results in a set of equations using $$\text{gunfold}$$. If $$f$$ is first-order range-polymorphic then the condition $$\bar{p}$$ can only do structural matching on its argument and therefore $$k$$ must be the same for all equations where recursion terminates with $$k < n + 1$$.\[\square]
Chapter 7

Examples and initial analysis

The purpose of this chapter is to further illustrate the synthesis algorithm using examples in §7.1 and to present some initial thoughts on formal analysis of the algorithm in §7.2.

7.1 Examples of definition synthesis

This section contains one big example involving the synthesis of sorting algorithms, and two smaller examples illustrating specific problems.

Ex. 7.1 Given the following example and partial definition and the knowledge that sort uses two schemas.

\[
\begin{align*}
\text{sort} & : \text{Ord a} \rightarrow \text{[a]} \rightarrow \text{[a]} \\
\text{sort} [3, 1, 2, 0, 4] & \equiv [0, 1, 2, 3, 4] \\
\text{sort} & : \text{Ord a} \rightarrow \text{[a]} \rightarrow \text{[a]} \\
\text{sort} \ x s & = \text{foldr} \ \tilde{f} \ \tilde{a} \ x s \\
\text{sort} [3, 1, 4, 0, 2] & \equiv [0, 1, 2, 3, 4] \\
\Rightarrow & \text{foldr} \ \tilde{f} \ \tilde{a} \ [3, 1, 4, 0, 2] \equiv [0, 1, 2, 3, 4] \\
\Rightarrow & \tilde{f} \ : \text{Ord a} \rightarrow \text{a} \rightarrow \text{[a]} \rightarrow \text{[a]} \\
\tilde{f}_{\theta_1} 3 & [0, 1, 2, 4] \equiv [0, 1, 2, 3, 4] \\
\tilde{f}_{\theta_1} 1 & [0, 2, 4] \equiv [0, 1, 2, 4] \\
\tilde{f}_{\theta_1} 4 & [0, 2] \equiv [0, 2, 4] \\
\tilde{f}_{\theta_1} 0 & [2] \equiv [0, 2] \\
\tilde{f}_{\theta_1} 2 & [] \equiv [2] \\
\tilde{a} & : \text{Ord a} \rightarrow \text{[a]} \\
\tilde{a}_{\theta_1} & \equiv []
\end{align*}
\]
7.1. EXAMPLES OF DEFINITION SYNTHESIS

where \( \theta_1 = \{3, 1, 4, 0, 2\}/xs\). Factoring gives templates \( \bar{f} \) and \( \bar{a} \); the equations for \( \bar{f} \) correspond to list insertion. One of the templates must use a schema. Assume template \( \bar{f} \) does, and that the algorithm uses \texttt{foldr} to define \( f \) also.

\[
f :: \text{Ord} \ a \Rightarrow a \rightarrow [a] \rightarrow [a]
f \ z \ zs = \text{foldr} \ \bar{g} \ \bar{b} \ zs
\]

\[
f \theta_1 \ 3 \ [0, 1, 2, 4] \Downarrow [0, 1, 2, 3, 4]
\Rightarrow \text{foldr} \ \bar{g} \theta_2 \ \bar{b} \theta_2 \ [0, 1, 2, 4] \Downarrow [0, 1, 2, 3, 4]
\Rightarrow \bar{g} \theta_2 \ 0 \ (\bar{g} \theta_2 \ 1 \ (\bar{g} \theta_2 \ 2 \ (\bar{g} \theta_2 \ 4 \ \bar{b} \theta_2))) \Downarrow [0, 1, 2, 3, 4]
\Rightarrow \bar{g} :: \text{Ord} \ a \Rightarrow a \rightarrow [a] \rightarrow [a]
\bar{g} \theta_2 \ 1 \ [2, 3, 4] \Downarrow [1, 2, 3, 4]
\bar{g} \theta_2 \ 2 \ [3, 4] \Downarrow [2, 3, 4]
\bar{g} \theta_2 \ 4 \ [3] \Downarrow [3, 4]
\bar{b} :: \text{Ord} \ a \Rightarrow [a]
\bar{b} \theta_2 \Downarrow [3]
\]

\[
\bar{f} \theta_1 \ 1 \ [0, 2, 4] \Downarrow [0, 1, 2, 4]
\Rightarrow \text{foldr} \ \bar{g} \theta_3 \ \bar{b} \theta_3 \ [0, 2, 4] \Downarrow [0, 1, 2, 4]
\Rightarrow \bar{g} \theta_3 \ 0 \ (\bar{g} \theta_3 \ 2 \ (\bar{g} \theta_3 \ 4 \ \bar{b} \theta_3)) \Downarrow [0, 1, 2, 4]
\Rightarrow \bar{g} :: \text{Ord} \ a \Rightarrow a \rightarrow [a] \rightarrow [a]
\bar{g} \theta_3 \ 0 \ [1, 2, 4] \Downarrow [0, 1, 2, 4]
\bar{g} \theta_3 \ 1 \ [2, 4] \Downarrow [1, 2, 4]
\bar{g} \theta_3 \ 2 \ [4] \Downarrow [2, 4]
\bar{b} :: \text{Ord} \ a \Rightarrow [a]
\bar{b} \theta_3 \Downarrow [4]
\]

\[
\vdots
\]

where \( \theta_2 = \theta_1 \cdot \{3/z, [0, 1, 2, 4]/zs\} \) and \( \theta_3 = \theta_1 \cdot \{1/z, [0, 2, 4]/zs\} \). Only the equations from the first two examples for \( \bar{f} \) are shown. Template \( \bar{g} \) is not first-order so generalisation cannot be used; compare for example the last equation for \( \bar{g} \) from each group of examples to see that generalisation is futile. The algorithm enumerates definitions for \( \bar{g} \), including

\[
\bar{g} \ z \ zs = \text{if} \ \bar{c} \ \text{then} \ \bar{e}_1 \ \text{else} \ \bar{e}_2
\]

\[
\bar{g} \ z \ (u:us) = \text{if} \ \bar{c} \ \text{then} \ \bar{e}_1 \ \text{else} \ \bar{e}_2
\]

\[
\vdots
\]

The first definition above fails to give examples that can be generalised; why will become clear below. Using the second the algorithm enumerates definitions for \( \bar{c} :: \text{Bool} \) using the implicit predicate \( (<) :: \text{Bool} \) from the \text{Ord} context.
\[
c = x < z \\
c = x < u \\
\vdots \\
c = z > u
\]

With the last definition for \( \tilde{c} \) the following examples are obtained from \( \tilde{g} \)'s first 4 examples:

\[
\begin{align*}
\tilde{e}_2(\theta_1, \{0/z, 1/u, [2, 3, 4]/us\}) &\triangleq [0, 1, 2, 3, 4] \\
\tilde{e}_2(\theta_1, \{1/z, 2/u, [3, 4]/us\}) &\triangleq [1, 2, 3, 4] \\
\tilde{e}_2(\theta_1, \{2/z, 3/u, [4]/us\}) &\triangleq [2, 3, 4] \\
\tilde{e}_1(\theta_1, \{4/z, 3/u, [1]/us\}) &\triangleq [3, 4]
\end{align*}
\]

Note that the bindings in the substitution must be modified to account for the new pattern. Generalising \( \tilde{a}_1 \) and \( \tilde{e}_2 \), together with the definition enumerated for \( \tilde{c} \), gives this definition of \( g \):

\[
g z (u:us) = \text{if } z > u \text{ then } u:z:us \text{ else } z:u:us
\]

The definition of \( b \) in \( g \) is found immediately by generalisation, \( b = [z] \). This gives the following solution for \( f \)—a correct definition of list insertion, but note that it is strict in the list argument.

\[
f :: \text{Ord} \ a \Rightarrow \ a \rightarrow [\ a \] \rightarrow [\ a \]
f x xs
  = \text{let } g z (u:us) = \text{if } z > u \text{ then } u:z:us \text{ else } z:u:us
      b = [x]
     \text{ in }
     \text{foldr } g \ b \ x s
\]

Returning to the definition \( \text{sort} \) using \( f \), the other template \( \tilde{a} :: \text{Ord} \ a \Rightarrow [\ a \] is generalised immediately, \( a = [1] \), giving a solution for \( \text{sort} \).

\[
\text{sort} :: \text{Ord} \ a \Rightarrow [\ a \] \rightarrow [\ a \]
\text{sort} \ xs
  = \text{let } f = ... \\
a = [1] \\
\text{ in }
\text{foldr } f \ a \ x s
\]

This is a correct but inefficient definition of \( \text{sort} \), similar to (imperative) Bobble-sort. Alternatively the algorithm could try \( \text{gunfold} \) in the partial definition of \( f \):

\[
f :: \text{Ord} \ a \Rightarrow \ a \rightarrow [\ a \] \rightarrow [\ a \]
f z zs = \text{gunfold} \ k \ p \ g \ h \ zs
\]

1Here a sub-\( f \)ion with a more general type \([a]\) than that stated for \( b, \text{Ord} \ a \Rightarrow [\ a ] \), is acceptable since the definition of \( c \) ensures that the overall type is restricted to the right one.
7.1. EXAMPLES OF DEFINITION SYNTHESIS

\[ f \theta_1 3 [0, 1, 2, 4] \equiv [0, 1, 2, 3, 4] \]
\[ \Rightarrow \text{gunfold } \tilde{\theta}_4 \tilde{\theta}_4 \tilde{\theta}_4 [0, 1, 2, 4] \equiv [0, 1, 2, 3, 4] \]

where \( \theta_4 = \theta_1 \cdot \{3/z, [0, 1, 2, 4]/zs\} \). One of the equation sets found by factoring:

\[
\begin{align*}
\tilde{\theta}_4 [0, 1, 2, 4] & \equiv \text{False} \\
\tilde{\theta}_4 [0, 1, 2, 4] & \equiv 0 \\
\tilde{\theta}_4 [1, 2, 4] & \equiv \text{False} \\
\tilde{\theta}_4 [1, 2, 4] & \equiv 1 \\
\tilde{\theta}_4 [2, 4] & \equiv \text{False} \\
\tilde{\theta}_4 [2, 4] & \equiv 2 \\
\tilde{\theta}_4 [4] & \equiv \text{True} \\
\tilde{\theta}_4 [4] & \equiv [3, 4]
\end{align*}
\]

Here recursion does not proceed to the end of the list. Doing that produces an equation set that also includes the example

\[ \tilde{\theta}_4 [4] \equiv 3 \]

which has no solutions since \( \tilde{\theta} \) represents a range-polymorphic function.

Generalisation of the equations in \( E \) gives

\[ \tilde{\theta} (x:xs) = \text{x} \]
\[ \tilde{\theta}xs = z:xs \]

The first definition is equivalent to \( k = \text{head} \). A definition for \( p \) must be found using enumeration. This gives a solution that is lazy in \( xs \) as opposed to the previous solution for \( f \).

\[ f :: \text{Ord a} \Rightarrow a \rightarrow [a] \rightarrow [a] \]
\[ f \text{xxs} \]
\[ \text{= let } k = \text{head} \]
\[ p \text{xs} = \text{null xs || x < head xs} \]
\[ g = \text{tail} \]
\[ h = (x:) \]
\[ \text{in} \]
\[ \text{gunfold f p g h xs} \]

Template \( \tilde{a} :: \text{Ord a} \Rightarrow [a] \text{ in sort is again generalised immediately, a = [\text{.}]}. \) Together these definitions of \( f \) and a implement Insert-sort. \( \Diamond \)

Ex. 7.2 This definition deletes elements at non-odd positions in a list.

\[ \text{oddl :: [a] \rightarrow [a]} \]
\[ \text{oddl xs} \]
\[ \text{= let } f = \text{head, tail} \]
\[ p \text{ys} = \text{null ys || null (tail ys)} \]
\[ g = \text{tail, tail} \]
\[ \text{in} \]
\[ \text{unfold f p g xs} \]
To synthesise this the algorithm must introduce a more complicated reduction than in previous examples.

Ex. 7.3 Consider the following examples.

\[\begin{align*}
\text{deleteAll} &:: \text{Eq } a \Rightarrow a \rightarrow [a] \rightarrow [a] \\
\text{deleteAll } 1 &[[1, 2, 1]] = [2] \\
\text{deleteAll } 3 &[[1, 2, 3]] = [1, 2] \quad \text{-- } e_1
\end{align*}\]

This is a solution:

\[\begin{align*}
\text{deleteAll} &:: \text{Eq } a \Rightarrow a \rightarrow [a] \rightarrow [a] \\
\text{deleteAll } y &xs \\
&= \text{let } f z \text{ zs }= \text{if } y == z \text{ then zs else z} : \text{zs} \\
a &= [] \\
&\text{ in } \\
&\text{foldr } f \ a \ \text{xs}
\end{align*}\]

It deletes all occurrences of an element from a list. Had the example set only consisted of \(e_1\) above then the following definition would also have been a solution:

\[\begin{align*}
\text{delete} &:: \text{Eq } a \Rightarrow a \rightarrow [a] \rightarrow [a] \\
\text{delete } y &xs \\
&= \text{let } f = \text{head} \\
&\quad p \ \text{xs }= \text{null} \ \text{xs }\text{ || head} \ \text{xs }= y \\
&\quad g = \text{tail} \\
&\quad h \ \text{xs }= \text{if null} \ \text{xs }\text{ then [] else tail} \ \text{xs} \\
&\text{ in } \\
&\text{gunfold } f \ p \ g \ h \ \text{xs}
\end{align*}\]

This last definition is equivalent to \text{delete} from the standard Haskell module \text{List}. This demonstrates that further criteria for discriminating between solutions would be useful.

7.2 Towards formal analysis of the algorithm

In this section I discuss how formal analysis of the synthesis algorithm could proceed and I speculate what conclusions such an analysis may give. The lack of formal results about my algorithm is one of its main weaknesses, something I return to in §9.4.

The purpose of a formal analysis is to prove that certain properties hold for the synthesis algorithm. Formulating such properties, or theorems, is non-trivial since they are of no use if they cannot be proved. Here are some properties that, if proved, would make useful statements about the algorithm.

- Soundness: Given a set of examples, if the algorithm synthesises a definition then that definition is a solution.
7.2 TOWARDS FORMAL ANALYSIS OF THE ALGORITHM

- Completeness: Given a set of examples, the algorithm synthesises every solution.
- Termination: The synthesis algorithm always halts in finite time.
- Complexity: The algorithm’s time and space complexity is ... on average or in the worst case.

The combination of the properties soundness and completeness is called correctness. When a correct algorithm terminates it always returns the set of all solutions. These properties are based on similar properties formulated by Bauer [8]. The completeness property, with its universal quantification, can be hard to achieve or to prove for a given algorithm. A compromise is to settle for a proof that an algorithm returns at least one of the solutions, for example, the simplest one for some measure of simplicity.

My synthesis algorithm is defined by the preceding chapters; I use names of functions from Fig. 5.2 and Fig. 5.3 in §5. Part of a formal analysis is to define precisely the set of all valid definitions, \( \mathcal{L}_D \), and the set of all valid example sets, \( \mathcal{L}_E \). Assuming this is done and letting \( D \models E \) denote that definition \( D \) satisfies example set \( E \) in the sense of §5.3, the set of solutions to \( E \) is defined as follows.

\[
sol(E) \equiv \{D \mid D \in \mathcal{L}_D, D \models E\}, \quad E \neq \emptyset.
\]

Correctness of the synthesis algorithm is then stated by

\[
synthesise(E) = \sol(E)
\]

which is equivalent to

\[
\begin{align*}
\text{synthesise}(E) &\subseteq \sol(E), \quad \text{(soundness)} \\
\text{synthesise}(E) &\supseteq \sol(E), \quad \text{(completeness)}
\end{align*}
\]

Eq. 7.1 states a property of the whole synthesis process, starting from a set of examples \( E \), obtaining a set of solutions. Function \( \text{synthesise} \) is defined in terms of a function \( \text{trans} \) that synthesises a set of solutions given a set of states \( \text{ED} \), and a post-processing function. Except for post-processing, which I ignore in the following, \( \text{trans} \) is a generalisation of \( \text{synthesise} \). \( \text{trans} \) solves synthesis problems involving general equations and partial definitions instead of problems involving only examples. Given a suitable definition of \( \sol(\cdot) \) for a set of states Eq. 7.1 is equivalent to a new equation:

\[
\text{trans}(\{E, \square\}) = \sol(\{E, \square\})
\]

which is implied by

\[
\text{trans}(\text{ED}) = \sol(\text{ED}). \quad \text{(7.2)}
\]

This equation says that the algorithm is correct no matter what set of states it starts from. A proof of Eq. 7.2 would likely proceed by induction and require the following propositions about decomposing the problem:

\[
\sol(\{(E, D) \cup \text{ED}\}) = \sol(\{(E, D)\}) \cup \sol(\text{ED}) \quad \text{(7.3)}
\]
\[ \text{trans}(\{ (E, D) \} \cup ED) = \text{trans}(\{ (E, D) \}) \cup \text{trans}(ED) \] (7.4)

Eq. 7.3 should be a consequence of a natural definition of \( \text{sol}(\cdot) \) on states. Eq. 7.4 follows if the order in which states are selected, using function \( \text{select} \), can be shown to be immaterial to soundness and completeness. As part of the inductive case in the proof of Eq. 7.2 a result for a single state have to be proved, probably this equation:

\[
\text{sol}(\{ (E, D) \}) = \text{trans}(\text{refine}(\text{simplify}(E, D), D)),
\]

\[
\text{simplify}(E, D) \notin \{ \bot, \emptyset \}
\] (7.5)

This equation formalises the statement that the solutions attainable from a state \((E, D)\) should be the same as the definitions obtained by transforming the refinements of state \((\text{simplify}(E, D), D)\). Proving Eq. 7.5 involves proving that functions \( \text{refine} \) and \( \text{simplify} \) preserve correctness:

\[
\text{sol}(\{ (E, D) \}) = \text{sol}(\text{refine}(E, D)), \quad E \notin \{ \bot, \emptyset \}
\] (7.6)

\[
E = \text{simplify}(E, D)
\] (7.7)

where `=` in the latter equation denotes equivalence modulo \( D \) and the Prelude. Proving Eq. 7.6 and Eq. 7.7 would involve proving that evaluation of template expressions have natural properties, like those of reduction in lambda-calculus, and proving that a concept similar to normal form is well-defined for sets of equations. Furthermore, as part of the proof for Eq. 7.6 functions \( \text{generalise} \) and \( \text{factor} \) must be proved correct under certain conditions, perhaps that functions be first-order range-polymorphic for \( \text{generalise} \) and just range-polymorphic for \( \text{factor} \). Literature relevant to formal analysis: The semantics of lambda-calculus with explicit substitution is studied in some technical papers [1, 46]; Lassez et al. [44] investigates the properties of inverse substitution.

Regarding computational complexity the algorithm must in general try every well-typed canonical definition having the right number of schemas. The cost associated with the size of \( E \) is less significant than the unrelated cost associated with the number of definitions to try. In general the latter number grows exponentially with the size of definitions.
Chapter 8

Practical experience

In this chapter I describe my Haskell implementation, realising most of the synthesis algorithm of §5 and §6, and I report on the implementations performance on a selection of synthesis problems. This chapter is not meant to be the last word on how to implement my algorithm, but is serves as a proof of concept: It can be done, and I claim it is not hard given an understanding of the material in this thesis. The current implementation, in the following referred to as ‘the system’, has some limitations and was not written with efficiency in mind; still the results reported here are encouraging.

8.1 Implementation

The design of the system corresponds to the meta-language code, remarks and algorithms in §5 and §6. Here I comment on implementation techniques and choices.

The representation of expressions, types and environments is quite standard. Evaluation of the core expression language—including templates, explicit substitutions and suspended expressions—is done by a monadic call-by-name interpreter [87]. The structure was influenced by the meta-programming support of the language Gödel [30]. The parser was built using Hutton and Meijer’s monadic parsing combinators [36, 35]. The pretty-printer is ad-hoc (but should have been written using a combinator library).

The core of the system is the representation and manipulation of synthesis structures. Important structures are problem specifications, definitions and programs, sets of equations, and the search space. In addition the system maintains information about templates, that is, about unknown definitions, and various book-keeping information related to the progress of synthesis. The main logic of the synthesis algorithm is a translation of the meta-language functions synthesise, trans, simplify and refine.

The search strategy used in the system is designed with implementation simplicity in mind. The search space is represented as a list of states (pairs of equation sets and definitions).

- The list is maintained sorted using properties of the definition part of synthesis
states, ignoring equation sets. Order is determined by first schema simplicity, then definition size, but generalised definitions always precede enumerated definitions obtained from the same state. This means that schemas are tried in the order they are given in Fig. 4.3.

- The list is only accessed at the start; function select simply takes the head of the list. Each state is visited only once so there is no need to tag states with status information, such as whether generalisation has been attempted on the set of equations or not.

- Function selectTemplate selects the template causing the first equation in the equation set to become suspended.

The search strategy resulting from these choices is depth-first search starting at the head of the state list. Search exhausts all possibilities for one schema before trying the next, including enumerating definitions.

The system has some limitations with respect to the synthesis algorithm. Conjecturing is a translation of meta-language function conjecture including conjecturing recursive definitions using schemas, but the system does not attempt to keep recursive expressions on canonical form. Generalisation does not introduce patterns, it only allows variables on the left-hand side, and it is implemented in a convoluted way closely resembling Plotkin's lgg algorithm with inverse substitution; the comment regarding the relation between generalisation and lgg at the end of §6.3 indicates the design. Factoring is only attempted on standard folds, or equations with no folds, not on those using the unfolds. Only one factoring of an equation set is returned, corresponding to a heuristic matching one occurrence of a constant on the right-hand side to each constant on the left-hand side. Ad-hoc polymorphism, or type classes, is ignored and instead primitive equality and less-than operators are used. These primitive operators are defined only for values where such relations make sense. As a consequence types are represented without type contexts. Last, there is no post-processing of definitions.

Haskell is the obvious implementation language for the system, given its close correspondence to the meta-language. As an example of simple translation from the meta-language into Haskell compare the definition of simplify in Fig. 5.3 with the corresponding Haskell definition simplifyEqns given in Fig. 8.1 (from the source code, renamed and formatted for clarity). In the code equation sets are represented as lists. The standard datatype Maybe is used to represent simplified equation sets, Nothing being an unsatisfiable set and Just [...] a satisfiable one. The meta-language constructor ‘::’ is written as ‘:=’. Function maybeAppend is append for values of type Maybe [a]: it returns Nothing if either argument is Nothing, otherwise it appends the two lists. Observe that the code for simplifyEqns is just the meta-language function rewritten with a concrete representation for sets and with meta-language pattern matching on constructors replaced with all possible combinations of object-language constructors.

Last, a note on the elegance of non-strictness. The system has the ability to find n solutions to a synthesis problem instead of just one. In the top-level synthesis function,
simplifyEqns :: [Eqn] -> Env -> Def -> Maybe [Eqn]
simplifyEqns [] _ _ = Just []
simplifyEqns ((1 :=> r):eqns) env def
  = let e = eval 1 env :=> r
      eqns_s = simplifyEqns eqns env def
  in
case e of
  Suspended _ _ :=> _ -> maybeAppend (Just [e]) eqns_s
  Error _ :=> Error _ -> eqns_s  -- fails when it should
  Error _ :=> _ -> Nothing     -- fails when it shouldn’t
  CBool b1 :=> CBool b2 -> if b1 == b2 then eqns_s else Nothing
  CInt i1 :=> CInt i2 -> if i1 == i2 then eqns_s else Nothing
  Nil :=> Nil   -> eqns_s
  Nil :=> Cons _ _ -> Nothing
  Cons _ _ :=> Nil   -> Nothing
  Cons x xs :=> Cons y ys ->
      maybeAppend (simplifyEqns [x :=> y, xs :=> ys] env def) eqns_s

Fig. 8.1: Core equation simplification in Haskell.

corresponding to the meta-language function synthesise, \( n \) solutions are taken from the
(lazy) list returned by synthesise as follows:

... take \( n \) (synthesise ...)

No special low-level logic is needed—the problem of extracting \( n \) solutions is solved
independently at the top-level.

In total my system consists of 3,900 lines or 130 KB of Haskell 1.4 code including
not-too-verbose comments.

8.2 The system at work

The sources were compiled using the Glasgow Haskell Compiler version 2.10, and GHC
applied the GNU C Compiler version 2.7.2.1 to generate native code. The executable
was run on a Sun Ultra Sparc 2. Fig. 8.2 gives the results of running the system on a
selection of problem specifications. Explanation of the columns:

- identifier and type: Name of the synthesised definition and the type assigned to
  it by Haskell.
- |\( E \)|: Number of examples given.
- |\( Scm \)|: Schemas available, ‘*’ means all list schemas.
- |\( n_g/n \)|: \( n_g \) is number of templates whose definitions were found by generalisation,
  or ‘\( \ll \)’ if generalisation is disabled; \( n \) is total number of templates used. Note
<table>
<thead>
<tr>
<th>identifier and type</th>
<th>( E )</th>
<th>Scan</th>
<th>( n_2/n )</th>
<th>( f )</th>
<th>( t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>and’ :: [Bool] -&gt; Bool</td>
<td>2</td>
<td>*</td>
<td>0/2</td>
<td>0.06</td>
<td></td>
</tr>
<tr>
<td>append :: [a] -&gt; [a] -&gt; [a]</td>
<td>1</td>
<td>*</td>
<td>2/2</td>
<td>0.05</td>
<td></td>
</tr>
<tr>
<td>bobble :: [rd a -&gt; a] -&gt; [a] -&gt; [a]</td>
<td>2</td>
<td>*</td>
<td>2/5</td>
<td>√</td>
<td>6.64</td>
</tr>
<tr>
<td>deleted :: a -&gt; [a] -&gt; [a]</td>
<td>1</td>
<td>*</td>
<td>2/5</td>
<td>√</td>
<td>32.31</td>
</tr>
<tr>
<td>dist :: [Int] -&gt; [Int]</td>
<td>1</td>
<td>*</td>
<td>0/2</td>
<td></td>
<td>0.14</td>
</tr>
<tr>
<td>double :: [a] -&gt; [a]</td>
<td>1</td>
<td>*</td>
<td>2/2</td>
<td>√</td>
<td>0.08</td>
</tr>
<tr>
<td>dupl :: [a] -&gt; [a]</td>
<td>1</td>
<td>*</td>
<td>2/2</td>
<td></td>
<td>0.03</td>
</tr>
<tr>
<td>elem’ :: Eq a -&gt; a -&gt; [a] -&gt; Bool</td>
<td>3</td>
<td>*</td>
<td>1/5</td>
<td></td>
<td>0.15</td>
</tr>
<tr>
<td>embed :: [a] -&gt; [[a]]</td>
<td>1</td>
<td>*</td>
<td>2/2</td>
<td></td>
<td>0.04</td>
</tr>
<tr>
<td>evenl :: [a] -&gt; [a]</td>
<td>2</td>
<td>*</td>
<td>0/7</td>
<td>×</td>
<td>1.16</td>
</tr>
<tr>
<td>filter’ :: (a -&gt; Bool) -&gt; [a] -&gt; [a]</td>
<td>1</td>
<td>*</td>
<td>3/5</td>
<td>√</td>
<td>0.24</td>
</tr>
<tr>
<td>idist :: [a] -&gt; [a]</td>
<td>1</td>
<td>*</td>
<td>2/2</td>
<td></td>
<td>0.03</td>
</tr>
<tr>
<td>init’ :: [a] -&gt; [a]</td>
<td>1</td>
<td>*</td>
<td>0/3</td>
<td>×</td>
<td>0.28</td>
</tr>
<tr>
<td>last’ :: [a] -&gt; a</td>
<td>1</td>
<td>*</td>
<td>0/1</td>
<td></td>
<td>0.04</td>
</tr>
<tr>
<td>length’ :: [a] -&gt; Int</td>
<td>2</td>
<td>*</td>
<td>0/2</td>
<td></td>
<td>0.10</td>
</tr>
<tr>
<td>map’ :: (a -&gt; b) -&gt; [a] -&gt; [b]</td>
<td>1</td>
<td>*</td>
<td>1/2</td>
<td></td>
<td>0.05</td>
</tr>
<tr>
<td>oddl :: [a] -&gt; [a]</td>
<td>2</td>
<td>*</td>
<td>0/6</td>
<td>×</td>
<td>0.50</td>
</tr>
<tr>
<td>parity1 :: [a] -&gt; Bool</td>
<td>2</td>
<td>*</td>
<td>0/2</td>
<td></td>
<td>0.09</td>
</tr>
<tr>
<td>reverse’ :: [a] -&gt; [a]</td>
<td>1</td>
<td>*</td>
<td>2/2</td>
<td></td>
<td>0.06</td>
</tr>
<tr>
<td>snoc :: [a] -&gt; [a]</td>
<td>1</td>
<td>*</td>
<td>2/2</td>
<td></td>
<td>0.04</td>
</tr>
<tr>
<td>sort :: [rd a -&gt; [a] -&gt; [a]</td>
<td>1</td>
<td>*</td>
<td>3/7</td>
<td>√</td>
<td>35.21</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>*</td>
<td>0/7</td>
<td>-</td>
<td>29.66</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>*</td>
<td>-/7</td>
<td>√</td>
<td>37.11</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>*</td>
<td>-/7</td>
<td>-</td>
<td>29.64</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>*</td>
<td>3/7</td>
<td>√</td>
<td>3.77</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>*</td>
<td>0/7</td>
<td>-</td>
<td>24.92</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>*</td>
<td>-/7</td>
<td>√</td>
<td>4.16</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>*</td>
<td>-/7</td>
<td>-</td>
<td>24.68</td>
</tr>
<tr>
<td>sum’ :: [Int] -&gt; Int</td>
<td>1</td>
<td>*</td>
<td>0/1</td>
<td></td>
<td>0.04</td>
</tr>
<tr>
<td>takeWhile’ :: (a -&gt; Bool) -&gt; [a] -&gt; [a]</td>
<td>2</td>
<td>*</td>
<td>3/5</td>
<td>√</td>
<td>0.24</td>
</tr>
<tr>
<td>insert :: [rd a -&gt; a] -&gt; [a] -&gt; [a]</td>
<td>4</td>
<td>gunfold</td>
<td>?/7</td>
<td>?/e</td>
<td>2062.90</td>
</tr>
</tbody>
</table>

*Without base-case example.

*The system generalises Booleans.

*Analysis impractical due to large log.

**Fig. 8.2:** The system at work: identifiers, problem data, and statistics.
that when one template is defined using an if-statement 3 new templates are
introduced.

• \( f \): Mark ‘\( \vee \)’ means that factored equations lead to a solution, mark ‘\( \times \)’ that
factored equations never lead to a solution, mark ‘\(-\)’ that factoring is disabled,
and otherwise there is no mark. The latter case is when factoring is enabled, but
never used.

• \( t \): Time used to synthesise the definition, in seconds. Times are measured using
the Unix \texttt{time} command and includes both user and system time. It measures
the time of the whole synthesis process, including reading problem specification
and writing the synthesised definition. Times are from single runs, but further
runs gave only minor differences.

The example sets and synthesised definitions can be found in the appendix. I have
attempted to make the examples as simple as possible while still having the system
synthesise a correct definition. For some runs the default parameters are modified:

• The maximal size of definitions must be increased for bobble, dist, sort, and
insert.

• The system is informed that sort requires 2 schemas, not the default 1.

The results in the figure allow some conclusions to be drawn.

First, the system uses generalisation when it should, but the lack of patterns means
that generalisation may fail, for example, the then-part of the if-statement in bobble
(strict insert). The lack of factoring for the unfolds makes generalisation fail. In the
synthesis of such definitions the algorithm tries first to use the foldr schema and
factoring, but to no avail; see the \( \times \) in the column for \( f \). There is a bogus factoring
for deleteAll. The factoring heuristic used does not find the right factoring.

Both bobble and sort are synthesised from one example, although two examples
give faster synthesis in the former case. Synthesis of sort includes synthesising bobble
as an auxiliary definition. There are 8 runs for sort in the figure, in two groups of 4:
First with all schemas, then with just foldr. Each group tries all four combinations
of factoring and generalisation. In the first group disabling the algorithms for
genralisation and factoring leads to faster synthesis. This is due to the cost associated
with these algorithms: a number of failures, in combination with enumeration, must
be carried out before arriving at foldr. Once foldr is used a solution is found quickly
with good use of the algorithms, see the second group of 4 runs. This suggests that
a new search strategy is needed. It is somewhat surprising that generalisation has so
little effect here. In general, however, avoiding enumeration gets more important with
a more complex Prelude and datatypes since the number of definitions to enumerate
grows quickly.

Last, the abysmal synthesis time for insert is due to the lack of factoring for
gunfold equations meaning that large expressions must be enumerated in order to
find suitable definitions for the templates.
Chapter 9

Conclusion

When we write programs that “learn”,
it turns out we do and they don’t.
—ALAN J. PERLS

Here I review the research reported in this thesis and give directions for further work.

9.1 Contributions

The main contribution of this thesis is an algorithm for synthesising recursive functional
definitions of a function from examples of that function and its type. The algorithm
requires fewer examples than previous algorithms, for example, to synthesise sort,
and it synthesises definitions for a class of functions that previous algorithms do not
consider: higher-order functions, for example, map and takeWhile, an essential class
of definitions in functional programming.

The main technical results are:

- Extension of the least general generalisation algorithm for expressions (terms) to
definitions, including pattern-matching on arguments.

- A new algorithm for factoring non-example equations, including recursive equa-
tions, into example equations for one or more functions.

- A template expression language with explicit representation of unknown expres-
sions and substitutions on them.

The template language itself is unremarkable, but it leads to simple and general for-
mulation of generalisation, factoring and the use of schemas.

The factoring algorithm is important since it decomposes recursive synthesis prob-
lems into simpler problems, again sort is a good example. Hutchinson [32, p. 415]
calls decomposing synthesis an “open issue”. Factoring provides a partial solution to
this problem for functional program synthesis.
9.2 RELATED WORK

My approach is partly type-driven, for example, generalisation and factoring are only applied to certain polymorphic functions, and the approach represents schemas using higher-order functions and base the schemas on standard higher-order definitions that functional programmers use. This use of polymorphism in inductive synthesis appears to be new.

9.2 Related work

In this section I compare my approach to synthesis from examples with related work. Most of the work I review here is already mentioned in the survey chapter.

Summers's approach uses Lisp programs; typed functional language were not readily available in 1977 [80]. It is instructive, however, to see his approach in light of ML-like type systems. Using my terminology all his examples are of first-order range-polymorphic functions. The requirements that there be no duplicate constants on the left-hand side of an arrow effectively makes all examples “good” and avoids the question of how to match constants from the left-hand side and right-hand side. Summers's algorithm requires example traces, and traces are similar to a series of well-chosen examples showing the structure of recursion. Recurrence relations are found by comparing different traces and using a technique resembling simple generalisation. In my approach the structure of recursion is assumed by pre-defined schemas, and factoring can be seen as a way to produce traces. Summers's heuristic to introduce auxiliary variables is not needed in my approach; fold1 defines list reverse without an auxiliary list append. Summers writes that extension to nested function calls is under investigation. The advantage of Summers's algorithm is its sound theoretical foundation through the synthesis theorem; the disadvantage is its limited applicability.

Ishino and Yamamoto [37] are able to find the type of some polymorphic functions from examples and generate traces for Summers's algorithm, but only as a pre-processing step. For this they use a generalised version of Plotkin's fold algorithm.

Zhu and Jin's approach is the closest to mine. Their schema [90, 32], shown in Fig. 3.6 can be rewritten in Haskell, see Fig. 9.1. The schema, here called zjascm, is a slight generalisation of schema sfold1 from Fig. 4.4. Also shown in Fig. 9.1 are two sets of examples and their synthesised definitions, both using zjascm. The definitions, even1 (called ODD in the paper) and innerProduct are given in Haskell here, but preserves the structure of the original FP code. The definition of even1 fails for lists of even length. The examples given for innerProduct are symbolic; this means that finding out what operators to use becomes simpler. My algorithm would need a schema like gfoldr in Fig. 4.4 to define innerProduct.

In general, Zhu and Jin's algorithm finds the 'schema slots' test and base using the first example, and applies an equation-solving algorithm to find e, d and h. This latter algorithm uses a number of axioms and rules about datatypes to search for definitions. By inspecting the selections of axioms and rules in the paper I believe that this algorithm has equal power to the generalisation algorithm in Fig. 6.2 for polymorphic functions, though the former algorithm is quite involved compared with
CHARTER 9. CONCLUSION

```
-- Zhu and Jin's schema.
zjscm :: (a -> b) -> (a -> Bool) -> (a -> a) -> (a -> c)
    -> (b -> c -> c) -> a -> c
zjscm e test d base h in'
    | test in' = base in'
    | otherwise = h (e in') (zjscm e test d base h (d in'))

-- Examples and synthesised definitions.
even1 [a] = [a]
even1 [a, b, c] = [a, c]
even1 [a, b, c, d, e] = [a, c, e]
even1 :: [a] -> [a]
even1 in'
    = let e xs = [head xs, head (tail xs)]
        test = null tail
        d = tail tail
        base xs = [head xs]
        h xs ys = [head xs] ++ ys -- ... = head xs:ys
        in
        zjscm e test d base h in'
innerProduct [a1] [b1] = a1 * b1
innerProduct [a1, a2] [b1, b2] = a1 * b1 + a2 * b2
innerProduct [a1, a2, a3] [b1, b2, b3] = a1 * b1 + a2 * b2 + a3 * b3
innerProduct :: [[Int]] -> Int
innerProduct in'
    = let e xs = head (xs!!0) * head (xs!!1)
        test xs = null (tail (xs!!0)) && null (tail (xs!!1))
        d xs = [tail (xs!!0), tail (xs!!1)]
        base xs = head (xs!!0) * head (xs!!1)
        h = (+)
        in
        zjscm e test d base h in'
```

Fig. 9.1: Examples and synthesised definitions using Zhu and Jin's schema.
9.2 RELATED WORK

\[
\begin{align*}
\text{incBinRec} & : [\text{Bool}] \rightarrow [\text{Bool}] \\
\text{incBinRec} [\text{[]} & = \text{error} \quad \text{"incBinRec: [] invalid as binary number"} \\
\text{incBinRec} [\text{True}] & = [\text{False, True}] \\
\text{incBinRec} (\text{False}; \text{bs}) & = \text{True}; \text{bs} \\
\text{incBinRec} (\text{True}; \text{bs}) & = \text{False}; (\text{incBinRec bs}) \\
\text{incBin} & : [\text{Bool}] \rightarrow [\text{Bool}] \\
\text{incBin bs} & = \text{let } f_\_ = \text{False} \\
& \quad \text{p xs} = \text{null xs} \mid \text{head xs} = \text{False} \\
& \quad g = \text{tail} \\
& \quad h xs = \text{if null xs then [] else tail xs} \\
& \quad \text{in} \\
& \quad \text{gunfold f p g h bs}
\end{align*}
\]

\textbf{Fig. 9.2:} Hutchinson’s directly recursive definition and a version using \text{gunfold}.

my algorithm. On the other hand Zhu and Jin base their approach on a theoretical foundation, referred to but not explained in detail in their paper [90]. The choice of test and base from the first example corresponding to constructing the base-case, and this makes the algorithm brittle. The authors state that their algorithm can synthesise super-linear definitions, but no examples or details are given. Also it does not seem that the algorithm can introduce conditionals (if-statements), nor synthesise higher-order functions. Simplicity of my generalisation compared to equation solving—the latter is different for each slot—and the ability to factor equations and thereby decompose the synthesis problem, are additional advantages of my approach compared with that of Zhu and Jin.

Hutchinson [32, p. 275] points out that more than one base-case cannot be handled by Zhu and Jin’s algorithm. He gives as example the problem of incrementing a binary number represented as a reversed integer list, see the recursive solution \text{incBinRec} in Fig. 9.2, here rewritten to use Boolean lists. Hutchinson writes that it “would be very interesting” to extend the method to handle such definitions. The second definition in the figure is an equivalent definition that could be synthesised by my algorithm, except that ‘\text{incBin []}’ does not report an error. This indicates that the combination of higher-order functions and if-statements yields a powerful synthesis mechanism.

There is also some interesting work on logic program synthesis. Hamfelt and Nilsson [27] use a predicate-version of \text{foldr} as a schema to synthesise logic programs from examples, see Fig. 9.3, together with a special meta-interpreter given in the paper. Predicate \text{append} is synthesised from example atoms as sketched in the figure: The meta-interpreter transforms given example atoms for \text{append} into simpler example atoms for a predicate \text{p} by using the hypothesis. Then Plotkin’s \text{egg} is applied to generalise the new examples, and the result is recognised as the definition for \text{cons}. The authors also show how their approach can synthesise the \text{reverse} predicate using two schemas and a recursive application of the algorithm, though they have to modify
Predicate foldr, note order of arguments.
foldr([], F, A, A).
foldr([X|XS], F, A, Y) :- foldr(XS, F, A, U), apply(F, X, U, Y).

Meta-predicate defined for relevant predicates p,
apply(p,X₁,...,Xₙ) :- p(X₁,...,Xₙ).
for example for cons:
cons(X, XS,[X|XS]).

Examples:
append([], [], [])
append([a], [], [a])
append([b,c], [d,e], [b,c,d,e])
append([], [f], [f])

Partial definition:
append(XS,YS,ZS) :- foldr(XS,p,YS,ZS).

Synthesis decomposes examples:
append([], [], []) ⇒ foldr([], p, [], []),
append([a], [], [a]) ⇒ foldr([a], p, [], [a]) ⇒ foldr([], p, [], [a]) ∧ p(a, [], [a]) ⇒ U = [] ∧ p(a, [], [a])
append([b,c], [d,e], [b,c,d,e]) ⇒ foldr([b,c], p, [d,e], [b,c,d,e]) ⇒ foldr([b,c], p, [d,e], U) ∧ p(b, U, [b,c,d,e]) ⇒ foldr([], p, [d,e], U) ∧ p(c, U, U) ⇒ U₁ = [d,e] ∧ p(c, [d,e], U)
append([], [f], [f]) ⇒ foldr([], p, [f] [f])

Decomposed examples for unknown predicate p:
p(a, [], [a])
p(b, U, [b,c,d,e])
p(c, [d,e], U)

Generalised clause, equivalent to cons:
p(V, VS, [V|VS]).

Solution:
append(XS,YS,ZS) :- foldr(XS,cons,YS,ZS).

Fig. 9.3: Hansfelt and Nilsson's schema, synthesis of append.
foldr a bit to synthesise the auxiliary append predicate.

This is similar to the steps of my algorithm where the given example equations are transformed into new equations using a definition with a schema and templates. The difference in language of examples and definitions gives Hamfelt and Nilsson some advantages. Their clause-relative transformation of atoms is simpler than my equation transformation. Furthermore, they have no need for factoring: the examples are atoms and decomposition of examples by the meta-interpreter yields new atoms. Last, their algorithm is very compact and elegant. Unlike my approach they have no need for extraneous concepts such as template expressions and explicit substitutions.

Hamfelt and Nilsson do not synthesise programs in a statically typed language, but at least some typing could be added to their setting. This would allow types to be exploited during generalisation like in §6, and in general restrict the number of solutions.

Hamfelt and Nilsson’s approach is simpler than mine because their solution language does not have first-class functions (or predicates). A system synthesising functional programs must be able to synthesise higher-order definitions, like map and filter. It seems unlikely that Hamfelt and Nilsson’s approach applies to the synthesis of functional programs. While the added complexity of first-class functions requires a more complicated approach, this is not relevant to the logic programming case. In a system synthesising programs with higher-order predicates or synthesising functional-logic programs a combination of the two approaches may be fruitful.

ILP algorithms are sometimes classified based on their search strategies. Generally speaking my algorithm is a hybrid: Introducing a schema and then finding the templates corresponds to top-down search while generalising examples into a definition corresponds to bottom-up search.

9.3 Applications

My synthesis algorithm is of interest to researchers working on the problem of synthesising recursive programs from examples, but the algorithm itself is somewhat technical and probably not of interest outside this research area. I therefore consider the class of such synthesis algorithms in general when pointing out possible application areas.

Examples can be seen as partial description of the causality of some process, for example, human user actions, the Dow Jones Industrial Average, or communication between Internet hosts. The ability to compact such process data, or to predict it, can make systems easier to use, save storage, or help in analysis of future behaviour. Some such processes can be modelled with recursive descriptions.

Systems operating in a changing or unknown environment need the ability to ‘learn’ from observations. Synthesis algorithms are part of the reasoning techniques suitable for learning, though problems like uncertainty and inconsistency means that systems need additional reasoning components to function effectively. As a reasoning technique synthesis from examples have an advantage: Synthesis, especially functional program synthesis, works even if little is known about the domain of the examples, though
synthesis may take more time in this case. This is useful for small or autonomous systems without access to a large knowledge base.

One particular application area stands out as the best fit for inductive synthesis algorithms—programming assistants. A programming assistant is a system that helps a programmer write code in a programming-in-the-small development settings. The assistant’s task could be to synthesise a function from examples given to it by the programmer, possibly querying the programmer about the validity of new examples or about other information regarding the function to be defined. The programmer inspects the synthesised program and either accepts it or modifies the examples and re-launches synthesis. For use in such ‘dialectic’ program development synthesis would benefit from being more interactive and less batch-oriented.

A more specialised domain still is end-user programming where the user gives examples but never looks at synthesised definitions due to lack of programming skills or resources. An example is construction of editor macros. It is then important that computations can be supervised by users or that their effects can be reversed.

9.4 Critique

Here I list what I consider to be the main criticisms that can be levelled against this thesis.

The lack of a formal analysis of the algorithm is a main weakness. A proof of correctness and analysis of complexity along the lines indicated in §7.2 would both ensure the quality of the algorithm, and, in all likelihood, help to improve it. The current lack of such results means that I cannot make any formal assessment of the algorithm despite the practical results reported in §8. Furthermore, the algorithm manipulates expressions and equations in various ways, including much use of substitution, and this indicates that a less ad-hoc formulation should be investigated, starting with (term) rewriting theory, and unification and generalisation modulo equations [5]. A related weakness is the lack of an analysis of the expressibility of the synthesis algorithm: What precisely is the class of functions that it can synthesise definitions for, and how does the set of schemas available affect this? Expressibility of higher-order functions has been studied in category theory [12].

As presented the algorithm strives to be independent of search strategies. While this is a good idea during initial design, a search strategy is crucial to efficient implementation, and must be investigated—it may effect the design of the algorithm. The depth-first strategy used by the implementation and described in §8.1 could be characterised as a trivial non-strategy. It should be replaced with a more sophisticated strategy, for example, first refining states where generalisation or factoring applies and leaving those where enumeration is the only possibility for later. Generalisation is preferable to enumeration since it is more likely to lead to a solution, though having a global search preference for generalisation over enumeration is not without problems: It may keep more states in memory since all successors to a state cannot be computed at the same time. Note that pruning the search space can affect completeness; possibly
some compromise definition of completeness must be found to get efficient search, and an analysis of the complexity of the algorithm should be useful for search strategy development. Search strategies has been studied at length in the AI literature.

During the course of synthesis the algorithm may synthesise the same auxiliary definition several times or ‘re-invent’ well-known definitions, and this is a waste of resources. An obvious idea is to make a library of definitions part of the problem specification and allow the specification writer to supply useful definitions. In addition the algorithm could add synthesised sub-definitions to this library during its operation, and even store them persistently between successive runs. Unfortunately this leads immediately to a new problem: how to organise the library for efficient checking of library definitions on examples.

Last a criticisms that can, at least partly, be countered: The algorithm scales very poorly to problems of real-world size. Scalability is a serious problem and it is inherent to synthesis from examples. It is a consequence of the fact that a language of examples is too poor a language for representing knowledge in many problem domains. In my view synthesis from examples is an important fundamental technique in synthesis and therefore legitimate to study in its own right. This criticism is one of relevance and not addressed at the technique itself and therefore somewhat off target.

9.5 Further work

I divide the discussion of further work in two: First concrete ideas for improving my synthesis algorithm and then some more speculative notes on new problem formulations that require quite different approaches to synthesis.

Currently the synthesis algorithm places some restrictions on the use of schemas: The number of schemas must be given, and schemas are only used at top-level in definitions. These restrictions can be lifted—at the cost of more search. Consider this definition using two schemas.

\[
\text{inits :: [a] -> [[a]]}
\]
\[
\text{inits xs}
\]
\[
= \text{let f y ys} = [] : \text{mapr (y:) [] ys} \quad \text{-- Note \text{mapr} not on top-level}
\]
\[
a = [[]]
\]
\[
\text{in}
\]
\[
\text{foldr f a xs}
\]

Failing to find f by generalisation the algorithm could instead try using a schema. By examining the failed generalisation, the algorithm can realise that some definition should be applied to the schema, in this case ([1:]).

The algorithms for generalisation and factoring are used intensively during synthesis. Researching efficient versions of these algorithms is therefore important. Amongst the issues to investigate are incremental versions of the algorithms re-using results from failed attempts. Including tuples in the type-language would make the algorithm more expressive, for example, it would make it possible to define \text{take} without the higher-order trickery of Ex. 2.2.
Generalisation uses inverse substitution of explicit substitution to bind constants to variables from the environment—factoring can benefit from the same technique. Inverting such substitutions can be done at the very start of synthesis, before both generalisation and factoring are attempted. This is illustrated by the following example.

Ex. 9.1 Function surround takes an argument x and a list, and returns a list where x has been inserted at the front, at the end, and between all elements in the list.

\[
\text{surround :: } a \rightarrow [a] \rightarrow [a]
\]

\[
\text{surround 1 [2, 3]} = [1, 2, 1, 3, 1]
\]

\[
\text{surround 4 [5]} = [4, 5, 4]
\]

\[
\text{surround :: } a \rightarrow [a] \rightarrow [a]
\]

\[
\text{surround x xs = foldr } \tilde{f} \tilde{a} \text{ xs}
\]

Proceeding as described in §5 results in the following unsatisfiable example set for \( \tilde{a} \).

\[
\tilde{a}\theta_1 = [1, 1, 1]
\]

\[
\tilde{a}\theta_2 = [4, 4]
\]

Instead, invert the explicit substitution first, before factoring, and let factoring heuristically delete one occurrence of \( x \) when reducing an equation. Synthesis proceeds as follows:

\[
(\tilde{f} \theta \ 2 \ (\tilde{f} \theta \ 3 \ \tilde{a}) = [1, 2, 1, 3, 1])\theta^{-1}
\]

\[
\Rightarrow \tilde{f} \ 2 \ (\tilde{f} \ 3 \ \tilde{a}) = [x, 2, x, 3, x]
\]

\[
\Rightarrow \tilde{f} \ 3 \ \tilde{a} = [x, 3, x]
\]

\[
\Rightarrow \ \tilde{a} = [x]
\]

This leads to a correct factoring.

For \textit{foldl} matching must sometimes be done against the accumulated value not the right-hand side value; this is needed to successfully factor equations for the standard Haskell List module definition \texttt{\texttt{\texttt{\texttt{\texttt{|\}}}}}. Obviously factoring should be extended to also handle equations involving non-simple lists, that is, equations with lists of lists and so on.

Constant occurrences in examples can indicate what schemas to use, or not to use. For example consider this equation:

\[
\text{foldr } f \ [\] \text{ xs } = \text{ ys}
\]

Assuming \( f \) does not use an \textit{if}-statement, the equation is only satisfiable if the following holds for some integer \( n > 0 \),

\[
\text{length(ys)} = n \cdot \text{length(xs)}.
\]

Hardy [29] solves an instance of this problem.

My approach to synthesis uses both syntactic and enumerative techniques. A natural continued line of research is to look at semantic methods, for example, whether a
function is total or partial, or allowing examples to be supplemented by laws, thereby increasing the power of the example language. If reverse and init are unknown, and last is in a library of known definitions, these are possible laws:

\[
\text{reverse (reverse xs)} \equiv \text{xs}
\]
\[
\text{init (x:xs)} \cup \text{last (x:xs)} \equiv \text{x:xs}
\]

Also, synthesis could start from examples without types. Inferring a polymorphic type for an argument by generalising constants of different types is investigated by Ishino and Yamamoto [37]; they use types to pre-process the examples. A harder, but probably more fruitful approach would be to synthesise the definition and its type together, instead of a batch-oriented approach. Work on type-checking algorithms is likely to be relevant here.

For the specification writer a convenient extension to the example language would be the ability to provide definitions of new datatypes relevant for the problem-domain together with examples of functions using the datatypes. If the synthesis algorithm does not have schemas for the new datatypes, folds and unfolds can, in some cases, be constructed automatically from datatype declarations using polytypism [38].
Appendix A

Example sets and synthesised definitions

This appendix lists sets of examples and corresponding definitions as synthesised by my system, and it should be read in conjunction with §8.2 where related statistics appear. The examples and definitions, except the last one, together make up a literate Haskell program: lines starting with ‘> ’ are code lines and other lines are comments. The imported module Schemas contains the definitions of the list schemas from Fig. 4.3. Synthesised definitions have the types given in the example sets, but the types given there lack contexts, for example, for elem and sort. Some names have a ‘’ added to avoid clashes with the Haskell Prelude.

Some definitions are not explained elsewhere, nor defined by the Haskell Prelude and libraries: deleteAll deletes every occurrence of an element from a list; dlist doubles every integer in a list; dup1 appends a list to itself; even1 keeps only elements at even positions in a list (the head position is 0); idList is the identity on lists; parity1 is true for lists of even length; and snoc appends an element at the end of a list.

Some peculiarities of my system, relating to if-statements, should be commented.

- Sometimes if-statements are used when ‘||’ would be simpler. See v3 in the definition of even1. An equivalent simpler definition is the following.

\[
v3 \text{ xs } = \text{null xs } || \text{null (tail xs)}
\]

- Redundant if-statements are sometimes introduced, see the test null v7 in bobble, or v3 in the definition of elem.

This is also the case for v3 in the definition of insert. Here the lack of a post-processing step in the system makes template expressions occur in the solution. The equivalent \( v3 = \text{head} \) yields a perfect definition.
> import Prelude hiding (foldr, foldr1, foldl, foldl1)
> import $cmaiens

and' :: [Bool] -> Bool
and' [True, False] :=: False
and' [] :=: True

> and' v1 =
>  let
>  v2 v5 v6 = v6 && v5
>  v3 = True
>  in
>  foldr v2 v3 v1

append :: [a] -> [a] -> [a]
append [1, 2] [3] :=: [1, 2, 3]

> append v1 v2 =
>  let
>  v3 v6 = v6
>  v4 = v2
>  in
>  map v3 v4 v1

bobble :: a -> [a] -> [a]
bobble 0 [] :=: []
bobble 4 [3, 5, 6, 7] :=: [3, 4, 5, 6, 7]

> bobble v1 v2 =
>  let
>  v3 v6 v7 -
>  let
>  v8 = (null v7) || (v6 <= v1)
>  v9 = v6 : v7
>  v10 = v1 : (v6 : (tail v7))
>  in
>  if v8 then v9 else v10
>  v4 = [v1]
>  in
>  foldr v3 v4 v2
deleteAll :: a -> [a] -> [a]
deleteAll 1 [1, 2, 1] :=: [2]

> deleteAll v1 v2 =
>  let
>  v3 v6 v7 -
let
  v8 = v1 = v6
  v9 = v7
  v10 = v6 : v7
in
if v8 then v9 else v10
  v4 = []
  foldr v3 v4 v2

dist :: [Int] -> [Int]
dist [1, 2, 3] ::= [2, 4, 6]

dist v1 =
  let
    v2 = ((succ (succ 0)) *)
    v3 = []
  in
    map v2 v3 v1

double :: [a] -> [a]
double [1, 2] ::= [1, 1, 2, 2]

double v1 =
  let
    v2 v5 v6 = v5 : (v5 : v6)
    v3 = []
  in
    foldr v2 v3 v1

dup1 :: [a] -> [a]
dup1 [1] ::= [1, 1]

dup1 v1 =
  let
    v2 v5 = v5
    v3 = v1
  in
    map v2 v3 v1

elem' :: a -> [a] -> Bool
elem' 1 [1] ::= True
elem' 2 [3] ::= False
elem' 2 [1, 2] ::= True

elem' v1 v2 =
  let
    v3 v6 v7 =
> let
>   v8 = v7
> v9 = True
> v10 = v1 = v6
> in
> if v8 then v9 else v10
> v4 = False
> in
> foldr v3 v4 v2
>
> embed :: [a] -> [[a]]
> embed [1] ::= [[1]]
>
> embed v1 =
> let
> v2 v5 = [v6]
> v3 = []
> in
> map v2 v3 v1
>
> evenl :: [a] -> [a]
evenl [4, 9, 2] ::= [4, 2]
evenl [1, 2, 3, 4, 5, 6] ::= [1, 3, 5]
>
> evenl v1 =
> let
> v2 v0 = head v0
> v3 v17 =
> let
> v18 = null v17
> v19 = True
> v20 = null (tail v17)
> in
> if v18 then v19 else v20
> v4 = (\v0 -> tail v0) . (\v0 -> tail v0)
> v5 v37 = v37
> in
> gunfold v2 v3 v4 v5 v1
>
> filter' :: (a -> Bool) -> [a] -> [a]
> filter' (\v100 -> not (v100 == 0)) [3, 0, 1, 0, 2] ::= [3, 1, 2]
>
> filter' v1 v2 =
> let
> v3 v6 v7 =
> let
> v8 = v1 v6
> v9 = v6 : v7
v10 = v7
\text{in}
\text{if v8 then v9 else v10
v4 = []
\text{in}
foldr v3 v4 v2

\text{idList :: [a] -> [a]}
\text{idList [0] := [0]}

\text{idList v1 =}
\text{let}
\text{v2 v5 = v5}
\text{v3 = []}
\text{in}
\text{map v2 v3 v1

\text{init' :: [a] -> [a]}
\text{init' [1, 2, 3] := [1, 2]}

\text{init' v1 =}
\text{let}
\text{v2 v0 = head v0}
\text{v3 v16 = null (tail v16)
\text{v4 v0 = tail v0}
\text{in}
\text{unfold v2 v3 v4 v1

\text{last' :: [a] -> a}
\text{last' [1, 2] := 2

\text{last' v1 =}
\text{let}
\text{v2 v4 v5 = v5
\text{in}
\text{foldr1 v2 v1

\text{length' :: [a] -> Int}
\text{length' [1] := 1
\text{length' [] := 0

\text{length' v1 =}
\text{let}
\text{v2 v5 v0 = succ v0}
\text{v3 = 0
\text{in}
\text{foldr v2 v3 v1
map' :: (a -> b) -> [a] -> [b]
map' (\v100 -> succ v100) [0, 1, 2] ::= [1, 2, 3]

> map' v1 v2 =
> let
> v3 = v1
> v4 = []
> in
> map v3 v4 v2

odd :: [a] -> [a]
odd [4, 9, 2] ::= [9]
odd [1, 2, 3, 4, 5, 6] ::= [2, 4, 6]

> odd v1 =
> let
> v2 v6 = head (tail v6)
> v3 v16 =
> let
> v17 = null v16
> v18 = True
> v19 = null (tail v16)
> in
> if v17 then v18 else v19
> v4 = ((\v0 -> tail v0) . (\v0 -> tail v0))
> in
> unfold v2 v3 v4 v1

parity :: [a] -> Bool
parity [3] ::= False
parity [False, False] ::= True

> parity v1 =
> let
> v2 v5 v0 = not v0
> v3 = True
> in
> foldr v2 v3 v1

reverse' :: [a] -> [a]
reverse' [1, 2] ::= [2, 1]

> reverse' v1 =
> let
> v2 v5 = v5
> v3 = []
> in
> map v2 v3 v1
snoc :: a → [a] → [a]
snoc 1 [2] ::= [2, 1]

> snoc v1 v2 =
>   let
>   v3 v6 = v6
>   v4 = [v1]
>   in
>   mapr v3 v4 v2

sort :: [a] → [a]
sort [3, 1, 4, 0, 2] ::= [0, 1, 2, 3, 4]

> sort v1 =
>   let
>   v2 v5 v6 =
>     let
>     v7 v10 v11 =
>       let
>       v12 = (null v11) || (v10 <= v5)
>       in
>       if v12 then v13 else v14
>       v8 = [v8]
>     in
>     foldr v7 v8 v6
>   v3 = []
>   in
>   foldr v2 v3 v1

sum' :: [Int] → Int
sum' [1, 2] ::= 3

> sum' v1 =
>   let
>   v2 v4 v5 = v5 + v4
>   in
>   foldr1 v2 v1

takeWhile' :: (a → Bool) → [a] → [a]
takeWhile' (\v100 -> not (v100 == 0)) [3, 1, 2, 0, 2] ::= [3, 1, 2]
takeWhile' (\v101 -> v101 <= 10) [7, 8, 9] ::= [7, 8, 9]

> takeWhile' v1 v2 =
>   let
>   v3 v6 v7 =
Above are the examples and definition for `insert`. Note the `#`'s indicating template expressions.

```
insert :: a -> [a] -> [a]
insert 0 [] := [0]
ninsert 4 [3, 5, 6] := [3, 4, 5, 6]
ninsert 4 [1, 2] := [1, 2, 4]
insert 2 [1, 3, 4] := [1, 2, 3, 4]
```

```
> insert v1 v2 =
>     let
>         v3 v8 =
>             let
>                 v9 = null v8
>             in
>                 if v9 then #10 else #11
>             v4 v18 = (null v18) || (v1 <- (head v18))
>             v5 v0 = tail v0
>             v6 v38 = v1 : v38
>         in
>         gunfold v3 v4 v5 v6 v2
```
References


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


