Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.),

eingereicht bei der
Fakultät Wirtschaftsinformatik und Angewandte Informatik
der Otto-Friedrich-Universität Bamberg

Schema-Guided Inductive Functional Programming

through

Automatic Detection of Type Morphisms

Martin Hofmann

Oktober 2010

Gutachter

1. Gutachter: Prof. Dr. Ute Schmid (Universität Bamberg)
2. Gutachter: Prof. Dr. Petra Hofstedt (TU Cottbus)

Abstract

Inductive functional programming systems can be characterised by two diametric approaches: Either they apply exhaustive program enumeration which uses input/output examples (IO) as test cases, or they perform an analytical, data-driven structural generalisation of the IO examples.

Enumerative approaches ignore the structural information provided with the IO examples, but use type information to guide and restrict the search. They use higher-order functions which capture recursion schemes during their enumeration, but apply them randomly in a uninformed manner.

Analytical approaches on the other side heavily exploit this structural information, but have ignored the benefits of a strong type system so far and use recursion schemes only either fixed and built in, or selected by an expert user.

In category theory universal constructs, such as natural transformations or type morphisms, describe recursion schemes which can be defined on any inductively defined data type. They can be characterised by individual universal properties. Those type morphisms and related concepts provide a categorical approach to functional programming, which is often called categorical programming.

This work shows, how categorical programming can be applied to Inductive Programming, and how universal constructs, such as catamorphisms, paramorphisms, and type functors, can be used as recursive program schemes for inductive functional programming. The use of program schemes for Inductive Programming is not new. The special appeal and the novelty of this work is that, contrary to previous approaches, the program schemes are neither fixed, nor selected by an expert user: The applicability of those recursion schemes can be automatically detected in the given IO examples of a target function by checking the universal properties of the corresponding type morphisms. Applying this to the analytical system IGOR II, both, the capabilities and the expressiveness can be extended without paying it by efficiency.

An extension of the analytical functional inductive programming system IGOR II is proposed and its algorithms described. An empirical evaluation demonstrates the improvements with respect to efficiency and effectiveness that can be achieved by the use of type morphisms for IGOR II due to reduction of the search space complexity.

Kurzfassung

Systeme zur induktiven Programmsynthese werden bezüglich zweier gegensätzlicher Ansätze beschrieben: Enumerative Systeme zählen Programme vollständig auf und verwenden Eingabe/Ausgabe (E/A) Beispiele lediglich zum Testen; analytische, datengetriebene Systeme hingegegen generieren ein Programm durch strukturelle Generalisierung der E/A Beispiele.

Aufzählende Ansätze ignorieren die in den E/A Beispielen enthaltene strukturelle Information völlig, benutzen aber Typinformation, um den Suchraum zu beschränken und die Suche zu steuern. Sie verwenden Funktionen höherer Ordnung als rekursive Programmschemata während der Aufzählung, wenden diese aber beliebig und nicht zielgerichtet an.

Analytische Ansätze hingegen nutzen extensiv die strukturelle Information der E/A Beispiele, vernachlässigen aber die Vorzüge eines starken Typsystems. Programmschemata verwenden sie lediglich starr und fest codiert oder durch Auswahl eines Experten.

In der Kategorientheorie beschreiben universelle Konstrukte wie zum Beispiel natürliche Transformationen und Typmorphismen Rekursionsschemata auf beliebigen, induktiv definierten Datentypen. Diese Konstrukte zeichnen sich durch spezifische, universelle Eigenschaften aus.

Diese Arbeit zeigt, wie Catamorphismen, Paramorphismen und Typfunktoren als universelle Konstrukte in der induktiven Programmsynthese als rekursive Programmschemata verwendet werden können. Die Verwendung von Schemata in der induktiven Programmierung ist an sich nichts Neues, die Innovation liegt jedoch in der Art und Weise der Einführung der Schemata. Im Gegensatz zu herkömmlichen Ansätzen wird weder ein festes Schema verwendet, noch wählt ein Experte ein Schema aus. Die vorliegende Arbeit zeigt, dass die Anwendbarkeit eines bestimmten Schemas sich aus den E/A Beispielen einer konkreten Zielfunktion ableiten lässt, wenn man die universellen Eigenschaften das dem Programmschema entsprechenden Typmorphismus in den Beispielen erfüllen kann.

Im Folgenden wird eine Erweiterung des funktionalen, induktiven Programmsynthesesystems Igor II vorgestellt und der neue Algorithmus beschrieben. Ein empirischer Vergleich untermauert die Vorzüge der Erweiterung und macht die Steigerung der Effizienz und der Effektivität, die durch die Verwendung von Typmorphismen durch Komplexitätsreduktion des Suchraums erzielt werden kann, deutlich.

Acknowledgements

Without the support of various people, this work would have never been finished in this form.

First of all, I would like to thank my supervisor Prof. Ute Schmid who got me fascinated by Inductive Programming ever since I started working as a student research assistant and wrote my diploma thesis at her group. I was lucky to have a supervisor who gave me plenty of rope and was totally open for my new ideas, far away from usual IP methodology.

Kudos to all the student research assistants who were related to the project: Florian Bader, Nora Fteimi, Thomas Hieber, Olga Yanenko, and Thorsten Spiekers. Through their work, they have contributed a lot to improve the usability of various Igor II versions. Their tools and programs were a big help to evaluate and analyse the systems and essential to finetune my own Igor II^{+} system.

Many thanks to Klaus Stein for his help, hints, hacks, and tricks with LaTeX, Unix, Ruby, and especially for the board game evenings with the colleagues from the Kulturinformatik group at his place.

In addition, I would like to thank Susumu Katayama and Makoto Sakamoto from the University of Miyazaki for their generosity and overwhelming hospitality during my time as visiting researcher in Japan. I also thank Pieter Koopman from the University of Nijmegen for his advice to an inexperienced PhD student.

In particular I yield Prof. Michael "Magic Mike" Mendler thanks for his lectures and reading clubs. Without them, the theoretical approach of this work would have been a completely different one. I also appreciate very much that Prof. Petra Hofstedt made time to act as my external supervisor and gave me the opportunity to present my ideas in her group.

Eventually, I am very much indebted to my parents for all the years of confidence and backing, and especially to my girlfriend Mariko Kato who was a big source of encouragement and comfort in times when things did not went so well.

Contents

Table of Contents 9

List of Listings 15

List of Tables and Figures 17

1. Introduction 19
- 1.1. Motivating Example — Specific a priori Knowledge 20
- 1.2. Motivating Example — No a priori Knowledge 21
- 1.3. Dealing with Combinatorial Explosion in Practice 21
- 1.4. Contribution 23

2. Inductive Programming 25
- 2.1. Basic Concepts 26
- 2.2. Approaches to IP 27
  - 2.2.1. Object Language 29
  - 2.2.2. Search-Space Traversal 31
  - 2.2.3. Schema-based Language Bias and Use of Additional Knowledge . 33

3. Terms and Term Rewriting 43
- 3.1. Terms 43
- 3.2. Substitution, Matching, and Generalisation 46
- 3.3. First-Order Constructor Term Rewriting 47
- 3.4. General Remarks on Notation 50

4. Category Theory and Functional Programming 51
- 4.1. Category Theory 51
  - 4.1.1. Categories 51
  - 4.1.2. Universal Constructions 52
  - 4.1.3. Functors 56
- 4.2. Functional Programming and Category Theory 58
  - 4.2.1. Primitive Data Types and Functions 59
  - 4.2.2. Inductive Data Types 60
  - 4.2.3. Structural Recursion via Catamorphisms 61
  - 4.2.4. Type Functors 69
  - 4.2.5. Paramorphisms 72

5. The Igor II Algorithm **77**
5.1. Definition of a Problem Specification 77
5.2. Main Loop and χ_{init} 80
5.3. Synthesis Operators 81
5.3.1. Split operator χ_{split} 82
5.3.2. Subfunction operator χ_{subfn} 84
5.3.3. Function call introduction with χ_{direct} and χ_{call} 85
5.4. A Synthesis Example 90
5.4.1. Iteration 1 93
5.4.2. Iteration 2 93
5.4.3. Iteration 3 95
5.4.4. Iteration 4 96
5.4.5. Iteration 5 97

6. Guiding Igor II's Search with Type Morphisms **99**
6.1. Data-Driven Detection of Arbitrary Catamorphisms 100
6.1.1. The Categorical Perspective 100
6.1.2. The Term Rewriting Perspective 102
6.2. Formal Definition of the Operator χ_{cata} 103
6.3. Modifications 107
6.3.1. Catamorphism on Lists 107
6.3.2. Using natural transformation with χ_{tyfunc} 111
6.3.3. Primitive recursion via Paramorphisms with χ_{para} 114
6.4. Igor II in Action — Illustrating Examples 114
6.4.1. Filling a list with Zeros 115
6.4.2. The `length` of a List 117
6.4.3. Mirroring Binary Trees 119
6.4.4. A Synthesis Example revisited 120

7. Evaluation **123**
7.1. The Evaluation Design 123
7.1.1. Benchmark Systems and Settings 124
7.1.2. Purpose of Benchmarks 125
7.1.3. Benchmark Problem Specifications 125
7.2. Empirical Results 133
7.2.1. Runtimes and Success 134
7.2.2. Igor II_{H} Algorithm loop cycles 141
7.3. Improvement Remarks 150
7.4. Discussion 151

8. Conclusion **155**

Bibliography **157**

A. Haskell Reference **171**
A.1. Types and Values . . . 171
A.1.1. User-Defined Types and Synonyms . . . 171
A.1.2. Predefined and Built-In Types . . . 172
A.2. Functions . . . 175
A.2.1. Lambda Abstractions . . . 175
A.2.2. Infix, Prefix, and Sections . . . 175
A.2.3. Pointwise and Pointfree . . . 175
A.3. Pattern Matching, Case Expressions and Control Structures . . . 176
A.4. Type Classes and Overloading . . . 177
A.5. Modules . . . 178
A.5.1. Module exports . . . 179
A.5.2. Module imports . . . 179
A.6. Recursion Schemes in HASKELL . . . 180
A.6.1. Reduce-Map-Filter of Lists . . . 180
A.6.2. The Functor class . . . 181
A.6.3. Pointless Schemes . . . 181

B. Using SKI calculus defining `last` **187**

C. MagicHaskeller Specification **189**

D. IgorII$_H$ Specification **211**

E. IgorII$_H$ Manual **239**

F. All Results and Synthesised Programs **245**
F.1. Programs for `ack` . . . 245
F.2. Programs for `add` . . . 246
F.3. Programs for `addN` . . . 247
F.4. Programs for `alleven` . . . 248
F.5. Programs for `allodd` . . . 249
F.6. Programs for `and` . . . 250
F.7. Programs for `andL` . . . 251
F.8. Programs for `append` . . . 252
F.9. Programs for `balloons` . . . 253
F.10. Programs for `concat` . . . 254
F.11. Programs for `drop` . . . 255
F.12. Programs for `enjoySport` . . . 256
F.13. Programs for `eq` . . . 257
F.14. Programs for `even` . . . 258
F.15. Programs for `even, odd` . . . 259
F.16. Programs for `evenLength` . . . 260
F.17. Programs for `evenParity` . . . 261

F.18. Programs for `evenpos` . . . 262
F.19. Programs for `evens` . . . 263
F.20. Programs for `fact` with `mult` . . . 264
F.21. Programs for `fib` with `add` . . . 266
F.22. Programs for `gaussSum` with `add` . . . 267
F.23. Programs for `geq` . . . 268
F.24. Programs for `halves` . . . 269
F.25. Programs for `hanoi` . . . 270
F.26. Programs for `incr` . . . 271
F.27. Programs for `init` . . . 272
F.28. Programs for `init, last` . . . 273
F.29. Programs for `inits` . . . 274
F.30. Programs for `inorder` with `append` . . . 275
F.31. Programs for `intersperse` . . . 276
F.32. Programs for `last` . . . 277
F.33. Programs for `lastM` . . . 278
F.34. Programs for `lasts` . . . 279
F.35. Programs for `length` . . . 280
F.36. Programs for `lengths` . . . 281
F.37. Programs for `lenses` . . . 282
F.38. Programs for `mapCons` . . . 285
F.39. Programs for `mapTail` . . . 286
F.40. Programs for `mirror` . . . 287
F.41. Programs for `mult` . . . 288
F.42. Programs for `mult` with `add` . . . 289
F.43. Programs for `multfst` . . . 290
F.44. Programs for `multlst` . . . 291
F.45. Programs for `nandL` . . . 292
F.46. Programs for `negateAll` . . . 293
F.47. Programs for `norL` . . . 294
F.48. Programs for `nthElem` . . . 295
F.49. Programs for `odd` . . . 296
F.50. Programs for `oddpos` . . . 297
F.51. Programs for `odds` . . . 298
F.52. Programs for `oddslist` . . . 299
F.53. Programs for `or` . . . 300
F.54. Programs for `orL` . . . 301
F.55. Programs for `pack` . . . 302
F.56. Programs for `pepper` . . . 303
F.57. Programs for `pepperF` . . . 304
F.58. Programs for `playTennis` . . . 305
F.59. Programs for `postorder` with `append, snoc` . . . 306
F.60. Programs for `preorder` with `append` . . . 309
F.61. Programs for `replicate` . . . 310

F.62. Programs for `reverse` . . . 311
F.63. Programs for `rocket` . . . 312
F.64. Programs for `sentence` . . . 313
F.65. Programs for `shiftl` . . . 314
F.66. Programs for `shiftl, shiftr` . . . 315
F.67. Programs for `shiftr` . . . 316
F.68. Programs for `snoc` . . . 317
F.69. Programs for `split` . . . 318
F.70. Programs for `splitAt` . . . 319
F.71. Programs for `sum` . . . 320
F.72. Programs for `sub` . . . 321
F.73. Programs for `subseqs` with `append` . . . 322
F.74. Programs for `swap` . . . 323
F.75. Programs for `switch` . . . 324
F.76. Programs for `tail` . . . 325
F.77. Programs for `tails` . . . 326
F.78. Programs for `take` . . . 327
F.79. Programs for `transpose` . . . 328
F.80. Programs for `unzip` . . . 329
F.81. Programs for `weave` . . . 330
F.82. Programs for `weaveL` . . . 331
F.83. Programs for `zeros` . . . 332
F.84. Programs for `zip` . . . 333

G. Search Space Visualisations **335**

Index **343**

Listings

1.1. Primitive recursive schema with head-tail decomposition. 20
1.2. Appropriate instantiation of schema of Listing 1.1 for `last` 20
1.3. input/output examples of the function `lasts` 23
1.4. input/output examples of the function `length` 24

2.1. IO examples for `reverse` 27
2.2. IO examples for `last` 27
2.3. Possible candidate solution 27
2.4. Example library for MagicHaskeller 40

4.1. Catamorphism for Peano Integers 66
4.2. Catamorphism on Lists 68

5.1. n ground examples for `last` on a two-element list. 78
5.2. A single non-ground example for `last` on a two-element list. 78
5.3. Target specification Φ for `lasts` 79
5.4. Background specification B for `lasts` 80
5.5. Data type information Θ for `lasts` 80
5.6. Solution CS P for `lasts` 80
5.7. IO examples of `reverse` 83
5.8. Type information θ 91
5.9. Target specification Φ 92
5.10. Background specification B 92
5.11. Initial Hypothesis $H_0 : \chi_{\text{init}}$ 93
5.12. $H_1 : \chi_{\text{split},\texttt{x}} \overset{1}{\circ} \chi_{\text{init}}$ 93
5.13. $H_2 : \chi_{\text{split},\texttt{x}_1} \overset{2}{\circ} \chi_{\text{split},\texttt{x}} \overset{1}{\circ} \chi_{\text{init}}$ 94
5.14. $H_3 : \chi_{\text{split},\texttt{x}_2} \overset{2}{\circ} \chi_{\text{split},\texttt{x}} \overset{1}{\circ} \chi_{\text{init}}$ 94
5.15. Abduced IOs for $\texttt{fun}_1$ 94
5.16. Abduced IOs for $\texttt{fun}_2$ 95
5.17. $H_4 : \chi_{\text{subfn}} \overset{2}{\circ} \chi_{\text{split},\texttt{x}} \overset{1}{\circ} \chi_{\text{init}}$ 95
5.18. $H_5 : \chi_{\text{split},\texttt{x}_2} \overset{3}{\circ} \chi_{\text{subfn}} \overset{2}{\circ} \chi_{\text{split},\texttt{x}} \overset{1}{\circ} \chi_{\text{init}}$ 96
5.19. $H_6 : \chi_{\text{split},\texttt{x}_3} \overset{3}{\circ} \chi_{\text{subfn}} \overset{2}{\circ} \chi_{\text{split},\texttt{x}} \overset{1}{\circ} \chi_{\text{init}}$ 96
5.20. $H_7 : \chi_{\text{direct},\texttt{last}} \overset{3}{\circ} \chi_{\text{subfn}} \overset{2}{\circ} \chi_{\text{split},\texttt{x}} \overset{1}{\circ} \chi_{\text{init}}$ 96
5.21. $H_{10} : \chi_{\text{direct},\texttt{lasts}} \overset{4}{\circ} \chi_{\text{direct},\texttt{last}} \overset{3}{\circ} \chi_{\text{subfn}} \overset{2}{\circ} \chi_{\text{split},\texttt{x}} \overset{1}{\circ} \chi_{\text{init}}$ 97

6.1. Specification Φ for `reverse` 108
6.2. Type information Θ for Φ of `reverse` as shown in Listing 6.1 108
6.3. Specification Φ for `incr` 109
6.4. Type information Θ for Φ of `incr` as shown in Listing 6.3 109
6.5. The function `nzeros` defined as catamorphism. 115
6.6. The function `length` defined as catamorphism. 117
6.7. $\mathtt{fun}_3$ 120
6.8. $\mathtt{fun}_3^*$ 121
6.9. $H_{1,alt} : \chi_{\text{cata}} \overset{1}{\circ} \chi_{\text{init}}$ 121
6.10. $H_{2,alt} : \chi_{\text{direct},\mathtt{last}} \overset{2}{\circ} \chi_{\text{cata}} \overset{1}{\circ} \chi_{\text{init}}$ 121

7.1. Definition of `pepper` 127
7.2. Data types and examples for the `rocket` problem. 128
7.3. IO examples for the function `sentence` 129
7.4. Data types and examples for the `hanoi` problem. 129
7.5. Abduced IOs for auxiliary function in `postorder` 133
7.6. `reverse` using `snoc` and explicit recursion 152
7.7. `reverse` with auxiliary `foldr` and `snoc` 152

A.1. Type definitions and instance declarations for common inductive data types as used with the `pointless-haskell` library. 182
A.2. Types of `cata` and `para` 184
A.3. Examples for catamorphisms `pointless-haskell` library 184
A.4. Examples for paramorphisms using the `pointless-haskell` library . . 185

C.1. MagicHaskeller specification 189

D.1. Igor II_H specification 211

E.1. Igor II_H manual 239

List of Figures

1.1. From an incomplete specification to a recursive program. 19

2.1. The area of a regular pentagon with side length t, as examples (a), as arithmetic expression (b), and its term tree representation (c) 32
2.2. Divide-and-conquer clause template as used by DIALOGS-II 34
2.3. Grammar of the type `Fun` representing candidate programs 37

3.1. The term $f(g(y, h(a)), x)$ represented as tree with positions. 45

7.1. A phrase-structure grammar for the function `sentence` 129

G.1. Visualisation of the search space for `add` of IGOR II_{H} without (a) and of IGOR II^{+}_{C} with catamorphisms (b). 336
G.2. Visualisation of the search space for `addN` of IGOR II_{H} without (a) and of IGOR II^{+}_{C} with catamorphisms (b). 337
G.3. Visualisation of the search space for `evens` of IGOR II_{H} without (a) and of IGOR II^{+}_{C} with catamorphisms (b). 338
G.4. Visualisation of the search space for `inits` of IGOR II_{H} without (a) and of IGOR II^{+}_{C} with catamorphisms (b). 339
G.5. Visualisation of the search space for `sentence` of IGOR II_{H} without (a) and of IGOR II^{+}_{C} with catamorphisms (b). 340
G.6. Visualisation of the search space for `split` of IGOR II_{H} without (a) and of IGOR II^{+}_{C} with catamorphisms (b). 340
G.7. Visualisation of the search space for `unzip` of IGOR II_{H} without (a) and of IGOR II^{+}_{C} with catamorphisms (b). 341

1. Introduction

Inductive Programming can be considered as a subfield of **artificial intelligence** (AI) and especially **machine learning** (ML). It aims to generate programs from an incomplete specification, i.e. usually from a set of input/output examples only. Figure 1.1 depicts a set of input/output examples for a function `last` to retrieve the last element of a list and a recursive definition of this function[1].

```
last :: [α] → α
last      (a:[]) = a                 last (a:[]) = a
last    (a:b:[]) = b        ⇒        last (x:xs) = last xs
last  (a:b:c:[]) = c
```

Figure 1.1.: From an incomplete specification to a recursive program.

The task itself is typical for machine learning: Given an extensional definition, i.e. a set of examples, find an intensional definition, i.e. a (recursive) program. This program, however, should not only be able to simply reproduce them, but also *explain* them and contain explicit knowledge that was only implicit in the examples.

In the previous example of learning the function `last` as shown in Figure 1.1, apparently the learned function was expected not only to correctly compute the last element of lists up to three elements, but also for lists with arbitrary length. It is obvious, that in practise it is impossible to define a general algorithm that generates *any* program from examples.

However, Inductive Programming differs in two main points from standard machine learning. Firstly, the resulting program is required to be 100% correct on the provided examples. Standard machine learning algorithms try to minimise a classification error on a test set. Apparently in Inductive Programming, there is no correctness in the sense that the generated output is indeed exactly the program the user had in mind. A generated program failing to compute the provided examples correctly has already proven to be incorrect.

Furthermore, contrary to other machine learning algorithms, the object language of an IP system can usually be arbitrarily extended. Consider for example a decision tree learner or a linear regression learner. The object language, i.e. the language bias, of the former can be defined as a tree with nested attribute-value-tests on the inner nodes and class value assignments on the leaves. The language bias of linear regression is simply the set of all linear functions. The task of the learner is to adjust the coefficients in such a way that the overall error is minimised.

[1]The syntax used is HASKELL. A short reference is given in Appendix A.

In Inductive Programming there does not exist such a clear language bias, because the object language can be more or less chosen at will, depending how much knowledge is put a priori into the IP algorithm. It can be either quite restrictive, assuming specific knowledge of the domain to reduce search to a minimum, or it can assume no knowledge at all to be able to theoretically generate any program. The former is of limited capabilities, the latter will sooner or later lead to a combinatorial explosion of the space of candidate programs.

1.1. Motivating Example — Specific a priori Knowledge

One common solution to the combinatorial explosion is to consider oneself satisfied to generate only specific programs, i.e. only those following a certain schema. Or put differently, if it is *known* that the desired program follows a specific schema, this schema will be used. For the case of the introductory `last`-example, a primitive recursive schema as shown in Listing 1.1 can be considered. It used appropriate definitions of the functions `isAtomic` to check whether an input cannot be decomposed anymore, `solve` to compute the output given an atomic input, `decompose` to decompose an input in an atomic and a non-atomic part, where the latter is passed to a recursive call, and `compose` to compose the result of the recursive call with the atomic part of the decomposition.

Listing 1.1: Primitive recursive schema with head-tail decomposition.

```
f l
  | isAtomic l = solve l
  | otherwise  = let (hl,tl) = decompose l
                     tl'     = f tl
                 in compose hl tl'
```

Although it is still not trivial to find an appropriate instantiation of this schema to learn primitive recursive programs from example, it is quite feasible. The relevant functions can be defined as follows, where `isAtomic` checks whether a list is a singleton, `solve` simply returns its input, i.e. the single element of a singleton list, `decompose` is simple head-tail decomposition of lists, and `compose` returns always its second input, i.e. the result of the recursive call. Their definitions are shown in Listing 1.2.

Listing 1.2: Appropriate instantiation of schema of Listing 1.1 for `last`

```
isAtomic l  = case l of
                [a]     → True
                 _owise → False
solve [e]   = e
decompose l = (head l, tail l)
compose _ r = r
```

Of course, generating programs by instantiating a fixed schema is quite limited, because any program not following this schema cannot be generated by such an algorithm.

1.2. Motivating Example — No a priori Knowledge

One can take on another extreme and naïve approach and assume no knowledge at all. Simply all possible programs could be enumerated and checked whether by chance a program that is consistent with our input/output examples has been created.

From combinatory logic the primitive combinators of the untyped **SKI calculus** by Curry [23], which is known to be equivalent to λ calculus, and thus Turing complete, can be used. Although this may seem rather artificial, it demonstrates the extent of those approaches.

The **SKI** calculus consists of three combinators, i.e. functions with no free variables. Applying arguments to a combinator is expressed by juxtaposition. All combinators are left-associative; parenthesis may be used to explicitly state associativity. The enumeration of correct **SKI** terms can easily be done by induction over the following grammar:

$$\langle expr \rangle ::= \langle var \rangle \mid \mathbf{S} \mid \mathbf{K} \mid \mathbf{I} \mid \langle expr \rangle \langle expr \rangle$$

Variables $\langle var \rangle$ denote some arbitrary **SKI** expression:

$$\langle var \rangle ::= x \mid y \mid z \mid \ldots$$

For each combinator a reduction rule is defined to specify its operational semantics by stating expression replacements:

$$\begin{aligned} \mathbf{I}x &\rightarrow x \\ \mathbf{K}xy &\rightarrow x \\ \mathbf{S}xyz &\rightarrow (xz)(yz) \end{aligned}$$

Although programs in the **SKI** calculus are very easy to enumerate, its programs are huge. The equivalent of the two-lines-Haskell-definition of `last` in **SKI**[2] consists of about 3300 combinators, including representations for lists, Booleans, conditionals, head-tail decomposition, and a fixed point combinator. This would require to check about 5×10^{1574} possible candidate programs[3]. Assuming one could generate and test 10^9 candidates a second, it still would take more than 1500 times longer than the expected lifetime of the earth[4].

1.3. Dealing with Combinatorial Explosion in Practice

Both examples show the extreme: Using a fixed schema is far too limited, mindless enumeration far beyond efficiency. So how to cope with the combinatorial explosion

[2] The curious reader may be referred to Appendix B

[3] Ignoring the general undecidability of this equational theory.

[4] Generous estimates say the earth exists since 5×10^9 years and it will take again as long for her to be swallowed by an expanding sun.

of the search space? As usual, a solution lies in the happy medium. However, for any method this implies sacrificing some of its strengths by improving some of its weaknesses. Seldom there is such thing as a free lunch.

Using type information when enumerating programs ensures the generation of type-correct programs only, for example. This reduces the search space, but does not restrict the solution space (cf. MagicHaskeller or PolyGP in Section 2.2.3.5). So there is no risk of accidentally removing the solution from the search space when reducing it. Another benefit of type information is that it is easily available in strongly typed languages like Haskell. MagicHaskeller and PolyGP for example exploit Haskell's strong type system. Once the specification has been successfully type checked, it is not too difficult to make this information also available for the synthesis process. Type constraints then simply prohibit the generation of programs that are not well-typed.

In Artificial Intelligence, usually heuristics are applied during search. This also keeps the search space unchanged, but uses additional knowledge. This knowledge was a priori researched by an expert programmer and compiled into a rating function to provide guidance during the search space traversal. It can be seen as a compass roughly pointing into the correct direction.

Other approaches delegate more responsibility to the user. For example could he be required to provide negative examples. This has shown to be quite useful in the inductive logic programming (cf. Section 2.2.1.1), when learning concepts or relations, to prune irrelevant branches of the search space. When learning programs, i.e. functions, negative examples convey no additional information, because for a given input there can be only a single output.

A second possibility is to give an expert user the possibility to restrict the search space problem-specifically. In analogy to our motivating example in the previous Section 1.2, the user would need to give a new grammar each time depending on the problem. This is done by the G∀st system (cf. G∀st in Section 2.2.3.3). Or the user can choose from a set of predefined program schemes or templates according to his knowledge about his program in mind (cf. Dialogs in Section 2.2.3.1). Problems arise however, if the user is not such an expert as required and provides an inappropriate schema or grammar. In the best case this only deteriorates the efficiency, in the worst the synthesis fails.

In the domain of **automated theorem proving** (ATP) similar problems arise. Here the aim is to split a goal, which is to prove, into subgoals which then may be proven automatically. For this purpose high level **tactics** are used, i.e. programmed strategies to split a goal into subgoals. Milner firstly used tactics in the theorem prover LCF [88] which were later adopted for HOL [38], Isabelle [108] and their combination Isabelle/HOL[102], or NuPRL[19]. Although some tactics are applied automatically an automated theorem prover still relies on an expert user to apply appropriate tactics interactively.

Proof Planning [16, 15] tries to apply planning methods to guide the search of a proof in ATP. A proof plan can be considered as a plan or an outline of a proof. To prove a conjecture, proof planning first constructs the proof plan for a proof and then uses it to guide the construction of the proof itself. It has been implemented in Clam, the proof planner of the Oyster System [17]. Although several approaches have been made to construct a proof plan automatically [124, 27], e.g. from examples [18, 56, 91, 90], it still requires an expert to construct an appropriate proof plan.

1.4. Contribution

After all it does not seem possible to gain efficiency without giving up expressiveness. Or at least, it always involves the help of an informed user. It would be desirable, similar to tactics in theorem proofing, to use high-level schemas, or program patterns, but applying them without the help of a user. The question is, though, how to obtain this information? One can show that the necessary information is just available in a strong type system as it is used by e.g. HASKELL.

Listing 1.3: input/output examples of the function `lasts`

```
lasts :: [[a]] → [a]
lasts []                  = []
lasts [[a]]               = [a]
lasts [[a,b]]             = [b]
lasts [[a,b,c]]           = [c]
lasts [[b],[a]]           = [b,a]
lasts [[c],[a,b]]         = [c,b]
lasts [[c,d],[b]]         = [d,b]
lasts [[a,b],[c,d]]       = [b,d]
lasts [[c],[d,e],[f]]     = [c,e,f]
lasts [[c,d],[e,f],[g]]   = [d,f,g]
```

Consider the examples for `lasts` in Listing 1.3, which show a modification of our first function `last`. Now `last` is applied to each list inside a list. A functional programmer would immediately see a well known pattern, the so called `map`-pattern, which applies a function to each element in a list. He derives this knowledge from the type information of the target function, knowing that `map` is a polymorphic higher-order function, defined on arbitrary lists:

```
map :: (α → β) → [α] → [β]
map _ []     = []
map f (x:xs) = (f x) : (map f xs)
```

A simple solution would use the higher-order function `map` in HASKELL:

```
lasts = map last
```

A similar problem would be the function `length`, which examples are shown in Listing 1.4. A common way to define `length` is to use the higher-order function `foldr` (cf. Appendix A.6.1) which for each element in the given list constantly increments the default value zero by one. The function `const` ignores its second input and always returns its first.

```
length = foldr (const (+1)) 0
```

Listing 1.4: input/output examples of the function `length`

```
length :: [a] → Int
length []          = 0
length [a]         = 1
length [a,b]       = 2
length [a,b,c]     = 3
length [a,b,c,d]   = 4
length [a,b,c,d,e] = 5
```

Both functions used higher-order functions to incorporate a schema for structural recursion on lists. Once it is clear that a function follows a certain higher-order scheme, only its arguments have to be determined and the synthesis effort decreases. This is the usual benefit one gets from the use of program schemes. The novelty presented in this work is that for the selection of an appropriate schema, neither an expert user nor additional knowledge is required. This work makes furthermore the following contributions:

- It recalls the well known fact that for any inductively defined data type so called type morphisms exist and that they are distinguished by universal properties. Those type morphisms do not only capture program schemes for structural recursion, but also for primitive recursion and many other.
- Given the input/output examples of some function together with its type, it shows that it is possible to automatically detect whether this function can be expressed by a morphism by checking its universal properties in the given set of examples.
- This gives rise to a shortcut operator in the IP system Igor II. If the universal properties of type morphism hold for a set of examples at some state in the search space, Igor II can skip several subsequent search steps. At this point Igor II needs only to synthesise the argument function of the higher-order scheme implementing this specific type morphisms.
- It describes the algorithms for operators for three of those morphisms (catamorphism, type functor, paramorphism).
- It underpins in an empirical evaluation that with this approach improvements in efficiency are not at the cost of expressiveness. It shows that due to the use of type morphisms the Igor II algorithm is now both, faster and more powerful w.r.t. the programs synthesisable.

The reminder of this thesis is organised as follows. Chapter 2 gives the reader a short introduction into the basic concepts of IP (Section 2.1) and the main approaches to IP (Section 2.2). Then the theoretical foundations for terms and term rewriting (Chapter 3) and category theory (Chapter 4) are laid down, before the main Igor II-algorithm can be recapitulated in Chapter 5 and the extensions can be described in Chapter 6. Chapter 7 evaluates the new algorithms. Finally, Chapter 8 concludes.

2. Inductive Programming

Traditionally, the first appearance of **Inductive Programming** (IP) or **Inductive Program Synthesis** (IPS) is dated back into the late 1970s, when Summers put IP on a strong theoretical foundation introducing his Thesys system [128]. Although this field can now look back on more than four decades of more or less continuous research, there is no precise, comprehensive, and widely agreed definition or understanding of the term *Inductive Programming*. Considering everything the term Inductive Programming is attached to, one looks on a conglomerate of techniques and methods from various disciplines for solving problems of different domains. This is not a satisfying way to approach its meaning, because as soon as one topic is examined more closely, immediately adjacent fields that might be considered as IP, but are not coined IP, may be identified.

A first profound study about inductive inference was published by Angluin and Smith [3] motivated by the problem of finding patterns common to strings [2]. However, one can say that the term *Inductive Programming* first occurred in a paper by Partridge [107] and was taken up again by Flener and Partridge in their introductory article of a special issue on Inductive Programming in *Automated Software Engineering* [32].

Starting in the 80s, often motivated by theories and methods of inductive inference introduced by Angluin and Smith, Inductive Programming also got a connotation, which was also considered as *Programming by Example*, cf. Myers et al. in [101].

The understanding of researchers dealing with this topic was that of *programming in the user interface* [40] and was tackled quite problem-specific. The idea was to capture the users intention by analysing his actions in a *demonstrational interface*. The terms *Programming by Demonstration* used by Myers [99], *Programming by Direct Manipulation* [123], or Finzer and Gould's methodology of *Programming by Rehearsal* [29] describes this quite well.

The focus was mostly on detecting repetitive user actions [25, 24], providing the basis for end-user programming [40, 39], learning macros in text editors [80, 103], or guessing and predicting user actions and intentions [26, 85, 99, 100, 101].

However, with focus on an end-user, this ignored the strong relation to software engineering and the synthesis, i.e. the (semi-)automatic generation of code from a specification, as emphasised by Partridge [107]. He stressed induction as a scientific principle in contrast to deduction. Inductive inference—as a reasoning technique from specific to general—is, similar to abductive inference and analogical inference, per se unsound. Thus, instead of deducing a program from an *assumed-to-be-complete* specification, the focus is to infer a program from a *known-to-be-incomplete* specification. In this case *incomplete* means that the specification does not *fully and unambiguously* define the program behaviour.

In Deductive Program Synthesis, given a complete specification, all programs derived from the specification must be equivalent. In Inductive Program Synthesis this is not necessarily the case. Usually, a program is specified incompletely in terms of input/output (IO) examples as an incomplete specification, which normally does not cover the compete domain of the target program. This can be compared with extrapolation or regression in mathematics.

2.1. Basic Concepts

Before it is possible to talk about specific IP systems and discuss their different approaches, some basic IP concepts need to be fixed. As mentioned before, IP infers programs by generalising over an incomplete specification. The problem at hand, i.e. the program the specifier has in mind, is usually called the **target (program)**.

An incomplete specification usually comprises only examples of the program behaviour on a part of the program's domain. In most cases simple input/output examples describe the desired behaviour of the target program. Sometimes also exemplary computation traces are used. Usually type information and data type definitions are also considered as part of the specification.

Given such a set of IO examples, unless the examples do not cover a restricted domain completely, the problem is normally under-specified, as there are infinitely many, semantically different programs satisfying such a specification. In fact the IO examples themselves are also a program trivially satisfying the given specification. However, there are no objective criteria to determine which program the specifier really had in mind. Strictly speaking, there is usually nothing like a *unique and correct solution* to the induction problem, i.e. a unique program compliant with the specification. Therefore, a program satisfying the specification at hand is called an **hypothesis**. The language in which those hypotheses are described is called **hypothesis language** or **target language**.

However, when doing IPS a probably infinite set of hypothesis is quite unsatisfying as output of an IP system, or a machine learning algorithm in general. Thus, a learner generalising over such a specification needs to make some assumptions. The concept of **inductive bias** [89] captures exactly the assumptions made by a learner a priori.

Two kinds of inductive bias can be distinguished. The first is the so called **language bias** (or **restriction bias**). As the name suggests, it poses a restriction on the language used to represent the hypothesis considered. It may for example only allow first-order Horn clauses as hypothesis, require hypotheses to follow some syntactical restrictions, or being expressed in a subset of some other language, etc. This kind of bias is quite easy to grasp, because it is impossible to learn something which cannot be expressed in the hypothesis language.

The second bias, the so called **preference bias** (or **search bias**), is more crucial, as it decides on success and failure of an IP system. Briefly, it determines which hypotheses are preferred over others, and additionally guides the search through the **hypotheses space**, i.e. the space of all hypotheses, by imposing an order on them. In the simplest case, the preference bias applies *Occam's razor* which states that *"entia praeter* *necessitatem non sunt multiplicanda"*, i.e. it prefers the simplest hypothesis. However, preferring the least general hypotheses, or those with minimal costs, w.r.t. a fitness- or cost-function, are conceivable, too.

Apart from examples for the target program, often additional, problem specific knowledge, so called **background knowledge**, can be provided to an IP system. Listing 2.1 shows some IO examples for the function reverse. Examples for `last` as shown on Listing 2.2 are given as background knowledge. A possible candidate solution could look like the program shown in Listing 2.3. Haskell syntax is used here.

Listing 2.1: IO examples for `reverse`

```
reverse :: [a] → [a]
reverse []        = []
reverse [a]       = [a]
reverse [a,b]     = [b,a]
reverse [a,b,c]   = [c,b,a]
reverse [a,b,c,d] = [d,c,b,a]
```

Listing 2.2: IO examples for `last`

```
last :: [a] → a
last [a]       = a
last [a,b]     = b
last [a,b,c]   = c
last [a,b,c,d] = d
```

Listing 2.3: Possible candidate solution

```
1  reverse []      = []
2  reverse (a0:a1) =
3          last (a0:a1) : reverse (sub (a0:a1))
4  sub [_]         = []
5  sub (a0:(a1:a2)) = a0 : sub (a1:a2)
```

Listing 2.3 shows another feature of some IP systems: The invention of so called **subfunctions**. These are auxiliary functions which are neither target function nor defined in the background knowledge. Lines 4 and following in Listing 2.3 show the definition of such a subfunction which was introduced on the fly by the IP system.

2.2. Approaches to IP

When regarding different approaches to IP, they can traditionally be characterised by two orthogonal dimensions. The first is the kind of the target or object language, shown in the vertical dimension of Table 2.1, which are usually declarative languages. Therefore, one can distinguish between functional, logic, and functional-logic IP systems.

	analytical	*generate & test*	
		systematically	*evolutionary*
functional	IGOR IGOR II THESYS	G∀ST MAGICHASKELLER	ADATE POLYGP
logic	DIALOGS DIALOGS-II GOLEM	FFOIL PROGOL ATRE	
functional-logic		FLIP	

Table 2.1.: Classification of IP systems

The second describes the way an IP algorithm traverses its search space. This is shown in the horizontal dimension of Table 2.1. Either this is done more analytical, and thus data-driven, i.e. the structure of the data given in the specification guides the search. Contrarily, the system may follow a generate and test approach by simply enumerating and testing all correct programs exhaustively. If this is done systematically, usually by some size complexity measure, in most cases additional restrictions are used, because the search space is tremendous. If a system operates on a rather unrestricted search-space, genetic algorithms are used to more or less randomly traverse the program space.

In the context of this work it is useful to bring in and examine a third dimension: Namely the language bias, or the use of additional knowledge. Saying this, not background knowledge is meant, but knowledge that is not specific to a certain synthesis problem.

Usually, the language bias was mostly considered together with the hypothesis or object language, as it tendentially comprises some restriction of the class of synthesisable programs. Traditionally, this was done by a fixed program scheme for a well-defined class of programs. However, other approaches were used, such as the use of type information, or user selected program schemes, which are not as restrictive as a hard-coded language bias.

The next sections discuss the different object languages (Subsection 2.2.1), the different strategies to traverse the search space (Subsection 2.2.2), and finally Subsection 2.2.3 will examine the use of additional knowledge for the synthesis process.

Hereby in each subsection only those systems are considered that are not of interest in subsequent ones, such that at the end Subsection 2.2.3 describes only those systems in more detail that are of interest from the perspective of the use of program schemes and type information.

2.2.1. Object Language

Traditionally, most IP systems use declarative languages as target or object languages, i.e. as language of their generated programs, because only in declarative languages the expression of a data type value, e.g. `(a:(b:(c:[])))`, represents not only an object of the given type, but also conveys all information about the construction of this value. Since all data type values are composed of data type constructors, there is only a single way to construct such a value by successive constructor applications. Especially analytic approaches make excessive use of this paradigm.

However, let us first concentrate on the different object languages, because every type of declarative language, be it logic, functional, or functional-logic, has its own characteristics which fosters language-specific program representations and inference mechanisms.

2.2.1.1. Inductive Logic Programming

One line of research is the field of **Inductive Logic Programming** (ILP), a term which was first coined by Muggleton [94]. Though, ILP has a focus on non-recursive concept learning problems, there has also been research in inducing recursive logic programs on inductive data types in the field of ILP, see Flener and Yilmaz [34] for an overview.

In ILP, all examples, background knowledge, and hypothesis are represented as definite Horn clauses in a subset of first order logic. Definite Horn clauses have the form $H \vee \neg B_1 \vee \ldots \vee \neg B_n$, where the positive literal H is called the head and represents the predicate, or relation to be learnt.

Given a set of clauses B representing the background knowledge, a set of positive examples E^+, and a set of negative examples E^-, the task in the typical ILP setting is about finding the simplest consistent hypothesis H such that

$$B \wedge H \models E^+ \text{ and } B \wedge H \not\models E^-.$$

Or in other words, the hypothesis H is complete and consistent with respect to the training data given B. Finding this simplest consistent hypothesis H is done by search. Therefore, the hypothesis space, i. e. all possible Horn clauses that can be learnt, are partially ordered in a lattice based on θ-subsumption [114]. Also based on θ-subsumption, additionally a syntactic notion of generality is introduced which makes it possible to systematically search this lattice, from general-to-specific or vice versa, for an appropriate hypothesis.

Well known ILP systems are Quinlan's Foil/Ffoil [115, 116], Progol developed by Muggleton [95, 96], Muggleton and Firth [93], and Golem developed by Muggleton and Feng [97]. A relatively young analytical system is the schema-guided, interactive, inductive, and abductive recursion synthesiser Dialogs-II (**D**ialogue-based **I**nductive and Abductive **LOG**ic program **S**ynthesiser)[30, 135]. Dialogs-II will be discussed as an schema-guided, interactive analytical IP system later on (see 2.2.3.1). A system specialised on recursive rules is Atre by Malerba [83].

Most of the ILP systems are geared towards learning relational predicates and not programs in a functional sense. This has two consequences which both hamper many ILP systems. First, many ILP systems, excluding Progol, require both, positive and negative examples. The latter are used to prune overly general rules. This is easy and intuitive when learning relational predicates. In a setting where the goal is to learn programs, given one input there is exactly one correct output, but infinitely many incorrect outputs, i.e. negative examples. Thus, giving just random negative examples does not help the system in generalising correctly. Giving appropriate negative examples is extremely tedious and requires expert knowledge about the IP algorithm at hand.

Second, most ILP systems have adopted a strategy called **sequential covering**. In each iteration of their algorithms, one rule is learned which covers some positive and no negative examples. Then all examples covered by this rule are removed and the algorithm starts again. Considering rules independently might be appropriate for learning relations. In the case of (mutual) recursive programs, where rules have a high interdependency, this is in most cases not successful, because interdependencies between IO examples are ignored.

Various empirical studies showed that the outdated, though prominent ILP systems are superseded by modern functional approaches [51, 52, 54].

2.2.1.2. Functional Programming

IP systems with a functional language as object language make up the largest part. Apart from the ILP hype in the 90s, functional languages are the language of choice for many IP systems.

This ranges from the early Lisp-programs [128, 58, 8], to evolutionary systems in ML [104], λ-abstractions in Haskell [138], or the polymorphic-typed λ calculus *System F* [10], to systems with a strong focus on term-rewriting in Maude [69], or systems that make heavily use of type information in Haskell [63] or Clean [74].

An early survey about inductive synthesis of Lisp programs was written by Smith [125]. A more up-to-date survey of program synthesis techniques, with most of them in a functional setting, can be found in [66, 67]. Section 2.2.2 discusses those systems in more detail.

2.2.1.3. Functional Logic Programming

The aim of functional logic programming is to combine the most important concepts of functional languages with those of logic languages, to support features like function inversion, existential variables, and non-deterministic search from logic languages and efficient operational behaviour and evaluation strategies from functional languages.

As functional logic programming is not a run-of-the-mill programming paradigm, so are there only a few **Inductive Functional Logic Programming** (IFLP) systems.

Notably Flip by Hernández-Orallo and Ramírez-Quintana [41, 42] is an implemented representative of the approach. In functional logic languages, *narrowing* combines resolution from logic programming and term reduction from functional programming. Flip uses its inverse, analog to inverse resolution, called inverse narrowing to solve the induction problem.

A paper by Bowers et al. [14] describes a framework for higher-order inductive programming, in the functional logic programming language Escher, which allows to augment the usual functional programming syntax with predicates. Due to the fact that Escher claims to have high meta-programming facilities this would be a promising system. Unfortunately, its algorithm lacks crucial parts and it has never been implemented.

2.2.2. Search-Space Traversal

2.2.2.1. Generate and Test Approaches

With the trailblazing work by Koza [75, 76, 77, 78], founding the field of **Genetic Programming** (GP), evolutionary methods entered the field of IP. Inspired by evolution in biology, evolutionary methods build populations of possible candidate programs or individuals. Programs are usually represented as syntax trees, with functions on the inner nodes and constants and variables on the leafs. Instead of systematically traversing the search space, a general Monte Carlo search is applied. Individuals are randomly modified by biological inspired operators such as reproduction, selection, crossover, and mutation. In each iteration, always the "fittest" individuals of one population are evolved, hoping that finally a desirable program will be created.

Although, GP is applied to various problems in different domains, having standard libraries in every major programming language, it is preferably used to evolve arithmetic expressions. Consider the arithmetic expression of a function *pentA* to compute the area of a regular pentagon with side length t in Figure 2.1b. Representing arithmetic expressions as term trees as shown in Figure 2.1c, and using the examples (Figure 2.1a) for testing, a genetic algorithm would soon come to a solution similar to the tree in Figure 2.1c. For this purpose it composes and recombines elements from a fixed collection of symbols, the so called **terminal set**, and a set of function symbols, the **function set**, containing at least $\{t, 1, 2, \ldots, 4, 5, \ldots, 25, \ldots\}$ and $\{*, +, \mathit{div}, \mathit{pow}, \mathit{sqrt}\}$, respectively.

Through this focus on numerical problems, recursion is a less important issue. However, it is occasionally discussed for numerical data [60] or for restricted embedding of functions into object-oriented languages [1], but it is easier to model iterations or repetitions by loops with carefully crafted termination criteria [77] compared to recursion. Koza [76] for example, evolves a recursive program to check the parity of a binary digit of size n. For digits of size $n + 1$ a new program would be necessary.

So it is commonly agreed that the recursive program learning problem is very difficult for GP, because this can lead to nonterminating programs, which are impossible to test, and thus it is difficult to assign a fitness value to non-terminating functions. However, recursion is crucial when dealing with programs on structural data.

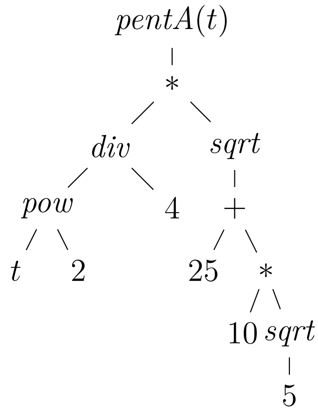
One possibility in GP is to allow non-terminating recursion and use a time limit for executing individual programs. This done in Adate by Olsson [104, 105]. Wong and Mun [134] proposed a method to detect structural similarities of non-terminating programs, and to modify the GP-algorithm on-the-fly to prevent the generation of future

t	$pentA(t)$
1	1.7205
2	6.8820
3	15.4843
4	27.5276
5	43.0119

(a) examples

$$pentA(t) = \frac{t^2}{4} * \sqrt{25 + 10 * \sqrt{5}}$$

(b) arithmetic expression



(c) term tree

Figure 2.1.: The area of a regular pentagon with side length t, as examples (a), as arithmetic expression (b), and its term tree representation (c)

programs with similar structure.

Another approach avoids non-termination by including special recursion operators in the terminal set. This leads to an extension called **Strongly Typed Genetic Programming** (STGP) [92]. Traditionally, GP systems face a limitation called *closure*, meaning that all variables, constants, arguments of functions and return values must have the same type. STGP lifts this restriction, such that they can be of any a priori fixed type.

Work by Yu and Clack paved the way for PolyGP by Yu [138, 137] which uses well-known higher-order functions such as `map` or `fold`, capturing termination in recursion schemes. Similarly, the system by Binard and Felty [10, 9] provides higher-order programming capabilities within the polymorphic λ-calculus F.

Nevertheless are GP algorithms quite time-intensive, because they only use a minimum of the information available in the IO examples. Attempts to marry GP approaches, represented by the Adate system and analytical approaches, represented by Igor II, have been conducted by Crossley et al. [22, 21].

2.2.2.2. Analytical Approaches

As already mentioned, the foundation of IP, and in particular functional IP, was laid by Summers [128] with its seminal Thesys system. He developed a theoretical framework to automatically synthesise S-expressions in Lisp from exemplary computation traces only. Summers' system worked in two steps. In the first, for each IO example a so called *program fragment*, consisting of Lisp primitives, predicates and McCarthy conditionals, was constructed, computing the exact output for the specific input. In a second step, these fragments were analysed for recurrences in their expression, and detected recurrences were folded into a recursive definition.

The underlying program scheme of Summers' approach was quite restricted, so Kodratoff et alter proposed various generalisations of Summers' approach based on the so called BMWk[1]-algorithm [58, 59, 70, 71]. BMWk was taken up again by Le Blanc [81] and generalised further.

Summers' approach avoided search completely by sticking to a restrictive program scheme. Work by Biermann [8] was also based on exemplary computation traces, similar to Summers' first step, but used them to speed up search by pruning an exhaustive program space of a well-defined program class.

Another system heavily inspired by Summers' approach is IGOR I [119, 98, 69]. Which also follows a two-step approach. Given IO examples are transformed into a finite approximating term of a *recursive program scheme*, a special kind of term-rewriting system. In a second step this finite approximating term was analysed for recurrences which finally were folded in a recursive definition. IGOR I is more powerful then THESYS, because its program scheme is less restricted and the IO examples need not to be ordered linearly. However, it suffers from the first step being its bottleneck, because it turned out that generating this finite approximating term is anything but trivial.

Its successor IGOR II [65, 121, 69] finally overcomes this two-step approach by combining analytical methods and an integrated search in the space of rules or unfinished programs. It is described in detail in Chapter 5.

2.2.3. Schema-based Language Bias and Use of Additional Knowledge

In this subsection we talk about IP systems that have a schema-based language bias or use, apart from background knowledge, additional information.

2.2.3.1. Dialogs-II

DIALOGS-II [135, 30] is an ILP system interactively querying the user for required examples. In terms of Flener and Yilmaz it is *schema-guided*, i.e. a schema is chosen by an informed user. This is contrary to systems with hard-wired schemas which are called *schema-based* (cf. the early analytic approaches). Flener et al. [33, 36, 35, 31] did various research to capture general knowledge about structured program design principles as well as domain specific knowledge in programming schemes, such as divide-and-conquer, together with formal semantics to reason about its correctness.

The basic idea is that a scheme fixes the control flow of a program. If, furthermore, the induction argument and its decomposition function is given by the user, examples for the target function can be used to abduce examples fur subfunctions.

Consider the *clause template* shown in Figure 2.2, notated in PROLOG-syntax, which constitutes a *divide-and-conquer scheme* assuming the user to have chosen a decomposition predicate which decomposes an input type into a single atomic element and two sub-parts of the same type. X is always the input variable and the output is bound to Y. If X is primitive, i.e. it is minimal or atomic in such a sense that it cannot be

[1] Boyer-Moore-Wegbreit-Kodratoff

$$
\begin{array}{rcl}
r(X,Y) & \leftarrow & prim(X), solve(X,Y) \\
r(X,Y) & \leftarrow & \neg prim(X), dec(X,H,X_1,X_2), r(X_1,Y_1), \\
 & & r(X_2,Y_2), comp(H,Y_1,Y_2,Y)
\end{array}
$$

Figure 2.2.: Divide-and-conquer clause template as used by Dialogs-II

decomposed anymore, the output can directly be computed from it. The predicate *solve* binds the output to Y. If it is not primitive, X is decomposed into H, X_1, and X_2. Recursive calls to r bind the sub-solutions for X_1 and X_2 to Y_1 and Y_2, respectively. The predicate *comp* binds the composition of H, Y_1, and Y_2 to the output variable Y.

Given such a template and the decomposition predicate, the only open or undefined predicates are *solve* for the base case and *comp* for composing the partial results.

However, the system is constructed to allow an expert user to make additional, domain specific knowledge available for the synthesis process. Despite that, the system cannot determine if a program scheme is appropriate or not.

2.2.3.2. PolyGP

PolyGP [139, 136] is a GP system with a polymorphic type system á la Girard-Reynolds [117, 37], able to evolve programs containing higher-order functions. It is of special interest for us, because it can use well known Haskell higher-order functions such as `foldr`, `map`, or `scanl` as recursion schemes [138]. Equipped with a user-defined terminal set T, and a function set F, its generated programs have the following syntax:

$$
\begin{array}{rll}
exp :: & c & \text{constant} \in T \\
| & x & \text{identifier} \in T \\
| & f & \text{function symbol} \in F \\
| & exp1\; exp2 & \text{function application of one expression to another} \\
| & \lambda x.exp & \lambda \text{ abstraction}
\end{array}
$$

Each program expression is associated with a type, which abstract syntax is defined as follows:

$$
\begin{array}{rll}
\sigma :: & \tau & \text{primitive built-in type} \\
| & \alpha & \text{type variable} \\
| & \alpha \rightarrow \beta & \text{function type} \\
| & [\alpha] & \text{list type with elements of type } \alpha \\
| & (\alpha \rightarrow \beta) & \text{bracketed function type}
\end{array}
$$

All user-provided constants, variables, and function symbols have to be attached with a type: constants and variables with a primitive type, function symbols with a function type. Together they comprise the context Γ. The following well-known typing rules apply:

- For any constant, or variable $t \in T$:

$$\frac{}{\Gamma \vdash t :: \tau}\ (t :: \tau) \in T$$

The type of a constant or variable in the set of terminal set is the type attached to it.

- For any function symbol $f \in F$:

$$\frac{}{\Gamma \vdash f :: \tau}\ (f :: \tau) \in F$$

Similarly, the type of a function is the function-type attached to it in the set of function symbols.

- For any application $\mathit{exp1}\,\mathit{exp2}$:

$$\frac{\Gamma \vdash \mathit{exp1} :: \sigma \to \tau \qquad \Gamma \vdash \mathit{exp2} :: \sigma}{\Gamma \vdash \mathit{exp1}\,\mathit{exp2} :: \tau}$$

For any application of two expressions, if the type of the first expression is a function type : $\sigma \to \tau$, and the type of the second expression is σ, the type of their application is τ.

- For any abstraction $\lambda x.\mathit{expr}$:

$$\frac{\Gamma, x :: \sigma \vdash \mathit{expr} :: \tau}{\Gamma \vdash \lambda x.\mathit{expr} :: \sigma \to \tau}\ x \notin \Gamma$$

Given a variable of type σ that is not already used in Γ and an expression *expr* of type τ, the type of the abstraction $\lambda x.\mathit{expr}$ is the function type $\sigma \to \tau$.

Its general algorithm is typical. First an initial population of n randomly generated, type-correct programs up to a predefined maximal length are created. For type unification Robinson's algorithm [118] is used. Then several cross-over and mutation operators are applied to the fittest individuals, w.r.t. a user-provided fitness-function, until an individual with maximal fitness occurs.

Despite there is no limit to PolyGP's expressiveness as a GP system in general, it suffers from the usual drawbacks of GP approaches. Success or failure crucially depends on the terminal set and the function set provided by the user. Are they too restrictive, the desired program is not contained in the induced search space. Are they too general, the search gets lost in space. Therefore, it requires the user to carefully craft these sets. Furthermore, there is still some amount of randomness involved, as the initial population is randomly generated.

2.2.3.3. G∀st

G∀ST [73] is an automatic tool for software testing implemented in the functional language CLEAN[2] [111]. The user expresses properties about functions and data types in first order logic. G∀ST automatically and systematically [72] generates appropriate test data, evaluates the properties for these values, and analyses the test results.

In general, given a logic expression such as for example $\forall t : T.P(t)$, G∀ST evaluates the predicate $P(t)$ for a large number of values t of type T, where G∀ST represents the predicate P as a function $T \to Bool$. The system uses the potentially infinite list of all possible values of type T as test suite and conducts n tests for some large fixed number n. The test may result in three possible outcomes: *Proof* if for a type, which number of values is less or equal than n, the test succeeded for all values in the test suite. *Pass* indicates that no counterexamples are found in the first n tests. *Fail* indicates that at least one counterexample was found during the first n tests.

The idea behind G∀ST is to state a property about the desired target function and let the system proof this property by providing such a function. For example properties for the factorial function f can be stated, such that $P(f) = f(2) = 2 \wedge f(4) = 24 \wedge f(6) = 729$, which becomes $\exists f.P(f)$. However, test systems are normally geared towards finding counterexamples and proving by contradiction, so it is more convenient to try to proof $\neg\exists f.P(f)$ or even more suitable $\forall f.\neg P(f)$, which is equivalent with the following proposition for G∀ST:

```
prop :: (Int → Int) → Bool
prop f = not (f 2 ≡ 2 ∧ f 4 ≡ 24 ∧ f 4 ≡ 24)
```

However, functional test systems in general have difficulties to generate and print functions. Therefore, instead of a property `prop` over function of type `Int → Int`, a property over a inductively defined data type `Func`, representing functions of type `Int → Int`, is used. The type `Func` represents the grammar of the target language (cf. Figure 2.3), and the function `apply` turns an instance of this data type into an actual function.

```
prop' :: Fun → Bool
prop' d = not (f 2 ≡ 2 ∧ f 4 ≡ 24 ∧ f 4 ≡ 24)
     where f = apply d
```

A hand-crafted grammar as shown in Figure 2.3 defines the target language of the candidate functions. It must be assured that it only describes terminating functions. In the case of the here described integer domain it will only construct terminating (primitive recursive) functions. Either they are non-recursive or they have an explicit stop criterion by carrying the number of further allowed recursive calls around. The conditional part is true if $x \leq c$ for x being the function argument, and c being some integer constant. The **then**-part is a normal non-recursive expression, where the **else**-part contains only one recursive call of the form $f(x-d)$, for some small positive number d. The expressions are either a variable, an integer constant, or a binary operator

[2] The syntax of CLEAN is very similar to that of HASKELL. Thus, HASKELL's syntax is used to describe problems in CLEAN, too.

$$
\begin{array}{rcl}
\mathit{Fun} & :: & \mathbf{f}(\mathbf{x}) = (\mathit{Expr} \mid \mathit{RFun}) \\
\mathit{RFun} & :: & \mathbf{if}(x \leq \mathit{IConst})\ \mathbf{then}\ \mathit{Expr}\ \mathbf{else}\ \mathit{Expr2} \\
\mathit{IConst} & :: & \text{Positive\_Integer} \\
\mathit{Expr} & :: & \text{Variable} \mid \text{Integer} \mid \mathit{BinOP}\ \mathit{Expr} \\
\mathit{Expr2} & :: & \text{Variable} \mid \text{Integer} \mid \\
 & & \mathit{BinOP}\ (\text{Variable} \mid \text{Integer} \mid \mathbf{f}(\mathbf{x} - \text{Integer})) \\
\mathit{BinOP}\ e & :: & e + e \mid e - e \mid e * e
\end{array}
$$

Figure 2.3.: Grammar of the type `Fun` representing candidate programs

applied to an expression. In this example binary operators for addition, subtraction, and multiplication are supported.

This grammar can directly be mapped into an inductive data type. The type `Fun` can now be recursively enumerated, starting with the terminals. Integers are enumerated up to a fixed integer n.

Although this grammar can easily be extended, from an IP perspective this is a bit unsatisfying though. An expert user has to put a lot problem specific knowledge into the problem specification in advance. However, this approach allows more control over the syntax of the generated function. So in general they are more readable for many users and the class of synthesisable functions is better describable. Furthermore, G∀ST is really tuned for enumeration and generates solutions much faster on the same domain. However, it is nearly sure that the enumeration time of G∀ST would deteriorate extremely if the grammar is sufficiently expressive and complex.

2.2.3.4. Djinn

A very interesting system, though not an IP system in the strict sense but rather a deductive system, is DJINN by Augustsson [5]. DJINN is a theorem prover in HASKELL, generating HASKELL expressions for a given type exploiting the Curry-Howard isomorphism.

The Curry-Howard isomorphism states an astonishing correspondence between type theory and proof theory. For instance, minimal propositional logic corresponds to simply typed λ-calculus, first-order logic corresponds to dependent types, second-order logic corresponds to polymorphic types, etc. This also extends to the level of syntax, where types correspond to formulas, expressions to proofs, and term reduction to proof normalisation.

In the same way as a proof for B can be derived from A and a proof for $A \rightarrow B$, one can obtain a value `f x` of type `B` from a value `x` of type `A` and a function `f` of type `A → B`. Thus, in the same way as deriving a proof for $A \rightarrow B$ from a proof of B assuming A, it is possible to derive a function `(λx → e)` of type `A → B` by constructing a value `e` of type `B` using a variable `x` of type `A`.

In DJINN the user gives a types at the prompt and the systems returns a term of that type if one exists. Djinn interprets a Haskell type as a logic formula and then uses a decision procedure for Intuitionistic Propositional Calculus. This decision procedure is based on LJT, a modification of Gentzen's LJ sequent calculus by Dyckhoff [28], which ensures termination. Theoretically, Djinn will always find a function if one exists, and if none exists, Djinn will tell so. The decision procedure has been extended to generate a proof object (i.e., a lambda term) as solution. It is this lambda term (in normal form) that constitutes the Haskell code. Given the type of a *polymorphic* function `f :: a → a`, Djinn generates the identity function as the sole solution.

```
Djinn> f ? a->a
f :: a -> a
f x1 = x1
```

Or it can uncurry a function

```
Djinn> uncurry ? (a -> b -> c) -> ((a,b) -> c)
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry x1 (v3, v4) = x1 v3 v4
```

and induce a case distinction:

```
Djinn> either ?  (a -> b) -> (c -> b) -> Either a c -> b
either :: (a -> b) -> (c -> b) -> Either a c -> b
either x1 x2 x3 = case x3 of
          Left l4 -> x1 l4
          Right r5 -> x2 r5
```

Djinn will always find a (total) function if one exists. If multiple functions exist, the system will only return one of them according to its search strategy.

```
Djinn> cons ? a -> [a] -> [a]
cons :: a -> [a] -> [a]
cons _ x2 = x2
```

Sure, this is not the desired function `cons` to insert an element at the front of a list, but it is a very simple function with this type. Similarly, the only Church numerals Djinn can find are 0 and 1.

```
Djinn> :set +multi
Djinn> num ? (a -> a) -> (a -> a)
num :: (a -> a) -> a -> a
num x1 x2 = x1 x2
-- or
num _ x2 = x2
```

Djinn also allows to add new function types to construct the results.

```
Djinn> foo :: Int -> Char
Djinn> bar :: Char -> Bool
Djinn> f ? Int -> Bool
f :: Int -> Bool
f x3 = bar (foo x3)
```

However, this is less powerful as it might seem at the first place, because Djinn does not instantiate polymorphic functions, but only uses those functions with exactly the type stated. Obviously, there is no polymorphic function to convert arbitrary types.

```
Djinn> cast ? a -> b
-- cast cannot be realized.
```

A more complex example is the generation of `returnS`, `bindS` in the state monad, which allows to encapsulate in Haskell state-full computations in a monadic type `type S s a = (s → (a, s))`

```
Djinn> type S s a = (s -> (a, s))
Djinn> returnS ? a -> S s a
returnS :: a -> S s a
returnS x1 x2 = (x1, x2)
Djinn> bindS ? S s a -> (a -> S s b) -> S s b
bindS :: S s a -> (a -> S s b) -> S s b
bindS x1 x2 x3 =
        case x1 x3 of
        (v4, v5) -> x2 v4 v5
```

Although Djinn is a neat demonstration of the power of types and the Howard-Curry isomorphisms, its limitations are quite obvious. Recursive functions are far beyond its scope, but it might be a good starting point in combination with other IP approaches.

2.2.3.5. MagicHaskeller

MagicHaskeller [63, 62, 61] is an IP system based on systematic and exhaustive enumeration and test of candidate programs. Simply put, the basic idea of functional programming done by MagicHaskeller is to exploit a strong type system just as solving a jigsaw puzzle: By repetitive combination of unifying functions and their arguments, expressions are constructed until eventually the intended program is obtained.

Based on a user-defined library containing primitive expression, MagicHaskeller systematically enumerates all those de Bruijn lambda-expressions that can be built from primitives from the given library. This is an infinite stream of functions which can be constructed by function composition and function application of expressions in the library and which unify with a given target type, i.e. the type of the target function. Finally, all those functions are filtered which pass a user-defined test, usually correctly compute the given IO examples.

Katayama [64] reformulated its search, similar to DJINN, under Curry-Howard isomorphism such that systematic and exhaustive search corresponds to the generation of infinite streams of proofs under second-order intuitionistic propositional logic.

MAGICHASKELLER's elegance lies in its efficient implementation of this enumeration and ingeniously interleaving these infinite streams. A naïve approach would soon run into spacial problems. Its breadth-first search is based on Spivey's algebraic approach for search monads [127, 126] combined with a type-checking monad transformer [82]. To avoid re-computation of subexpressions, Katayama uses a special memoization [63] technique based on generalised tries [45].

Similar to IGOR II, a minimal specification library consists of *complete* definitions of relevant type constructors. However, also a recursion scheme is required. Usually this is the type specific paramorphism. Furthermore, arbitrary function definitions may be added. Listing 2.4 shows a small library including the constructors and the paramorphism for natural numbers and lists. Given the following test function[3]

```
λf  →   f [ ]            ≡  [ ]
     ∧  (f [1 :: Int]    ≡  [3 :: Int])
     ∧  (f [1,2]         ≡  [3,4])
     ∧  (f [1,2,3]       ≡  [3,4,5]),
```

MAGICHASKELLER quickly finds a lambda expression satisfying this test:

```
λa → list_para a [] (λb c d → succ (succ b) : d)
```

In contrast to POLYGP (2.2.3.2), MAGICHASKELLER is less prone to be overwhelmed with too much background knowledge. Although the size of its provided library strongly influences its synthesis time, always only those library functions are used which have the appropriate type, while all others are ignored. POLYGP would simply mingle all elements in the terminal and function sets into its initial population.

Listing 2.4: Example library for MAGICHASKELLER

```
module Library where

zero       :: Int
zero       = 0

succ       :: Int → Int
succ       = succ

nat_para   :: Int → α → (Int → α → α) → α
nat_para   = λi x f →
                 if i ≡ 0
                    then x
                    else f (i-1) (nat_para (i-1) x f)

nil        :: [α]
```

[3] The explicit typing resolves ad-hoc polymorphism, because digits in HASKELL are overloaded

```
nil        = []

cons       :: α → [α] → [α]
cons       = (:)

list_para :: [β] → α → (β → [β] → α → α) → α
list_para = λl x f →
                case l of
                  [] → x
                 a:m → f a m (list_para m x f)
```

3. Terms and Term Rewriting

This chapter formally introduces the concepts of a term and term rewriting and its basic notions. Since IGOR II is a functional IP system, it strongly builds upon terms which describe the syntax of an arbitrary functional programming language. For our purpose, term rewriting gives them their operational semantics and helps us to define a functional program as a term rewriting system. In general, this chapter fixes the syntactic part of problems and algorithms described in other chapters, as well as the operational semantics w.r.t. term rewriting. Approaching problems from a term rewriting perspective can also be seen as adopting a micro perspective, i.e. looking at *how a problem is solved*. This we need later in the Chapters 5, and 6.1.2. The nomenclature and definitions follow the standard text books on this topic by Baader and Nipkow [6] or Terese [129].

3.1. Terms

Before we can talk about term rewriting, we need to have a clear understanding of terms which make the syntax of any functional programming language. Intuitively, terms are built from function symbols and variables. Given a binary function symbol f and two variables x and y, then $f(x, y)$ is a term. To prevent ambiguities, we need to be sure which function symbols are available in a certain context and what arity they have. This is fixed by a *signature*.

Definition 3.1.1. Let a **signature** Σ be a set of **function symbols**, where each symbol $f \in \Sigma$ is associated with a fixed natural number, the **arity**, indicating the number of arguments it is supposed to have. Function symbols with arity 0, i.e. nullary (0-ary) symbols, are called **constant symbols** or just **constants**.

However, functions are usually not defined for the whole domain, but only for a subset of it, so functions taking arbitrary terms as input are too general. So the domain is in fact an indexed family or collection of sets, a so called sorted set. Let us recall some standard terminology.

Definition 3.1.2. For a set S, let an **S-sorted set** be an S-indexed family of sets $(X_s)_{s\in S}$. For S-sorted sets S and Y, an **S-sorted mapping** $\phi\colon X \to Y$ is defined by the family of mappings $(\phi_s\colon X_s \to Y_s)_{s\in S}$.

Definition 3.1.3. A **many-sorted signature** Σ is a pair $\Sigma := (S, \Omega)$, where

- S is a set (of sort names); and
- Ω is an $S^* \times S$-sorted set (of function symbols).

S^* is the finite, possibly empty sequence of elements of S and $\times$ the product of two sets.

NOTATION: Saying that $f\colon s_1 \times \ldots \times s_n \to s$ is in $\Sigma = (S, \Omega)$ means that it holds that $s_1 \ldots s_n \in S^*, s \in S$, and $f \in \Omega_{s_1 \ldots s_n, s}$. The f is said to have *arity* n and *(result) sort* s. The abbreviation $f\colon s$ will be used for $f\colon \epsilon \to s$ where ϵ denotes the empty sequence. However, to avoid confusions with a colon being the operator to name a rule (cf. Definition 3.3.2), we will also write $t :: s$ to denote that the term t is of sort (or type) s. Usually the semantics should be clear from the context though.

From now on, all signatures are considered to be sorted. After this formal preparatory work, we are now well-equipped to continue with a definition of terms. In general, *terms*, or sorted terms, are strings of symbols from an *alphabet*. The symbols are drawn from a signature $\Sigma = (S, \Omega)$ and a countably infinite s-sorted set of **variables** $\mathcal{X}$, assumed to be disjoint from the function symbols in the signature Σ.

Definition 3.1.4. The set of **terms** over a signature $\Sigma = (S, \Omega)$ and an S-sorted set of variables $\mathcal{X}$ is indicated as $\mathcal{T}_\Sigma(\mathcal{X})$ and defined inductively:

(i) $x \in \mathcal{T}_\Sigma(\mathcal{X})$ for every $x \in \mathcal{X}$ (i.e. every variable is a term).

(ii) Given an n-ary function symbol $f\colon s_1 \times \ldots \times s_n \to s, f \in \Omega$, and terms $t_i\colon s_i, i \in \{1_1, \ldots, 1_n\}, t_i \in \mathcal{T}_\Sigma(\mathcal{X})$, then $f(t_1, \ldots, t_n) \in \Sigma$ is a term of sort s (i.e. the application of function symbols to terms yields terms).

(iii) These are all terms.

Terms containing no variables are called **closed** or **grounded**. Terms in which no variable occurs more than once are called **linear**.

NOTATION: In the context of terms, variables will be usually denoted by $x, y, z, x', y'', \ldots$, etc., or indexed $x_0, x_1, x_2, \ldots$ if needed. The symbols $f, g, h, \ldots$ stand for function symbols, $a, b, c, \ldots$ are constants. If appropriate we will use more meaningful names, e.g. 0 for a constant or $+$ for a binary function. We will also use infix or postfix when in the context the meaning is clear.

We also need to be able to address parts of terms or *subterms*. For this purpose, terms suggest for a straight forward representation as trees. Function symbols are parent nodes, and arguments are child nodes. Figure 3.1 depicts the structure of the term $f(g(y, h(a)), x)$ represented as a tree. Each subterm can now be addressed by a unique position. The term itself has the position ϵ. The first argument of a function symbol at position p has the position $p1$, the second the position $p2$, and so on. By induction over the structure of a term, we can give a more formal definition of subterms and positions.

Definition 3.1.5. Let Σ be a signature, and $\mathcal{X}$ a set of variables, and furthermore are $s, t \in \mathcal{T}_\Sigma(\mathcal{X})$.

1. For a term s, let $\mathcal{P}os\,(s)$ be the set of **positions** as sequences over an alphabet of natural numbers which is defined by induction as follows:

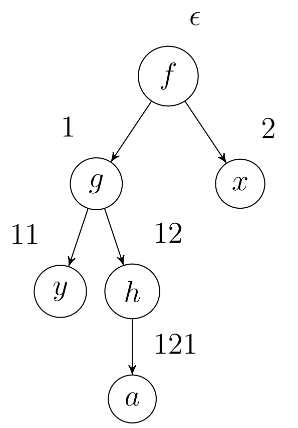


Figure 3.1.: The term $f(g(y,h(a)),x)$ represented as tree with positions.

(i) If $s = x \in \mathcal{X}$, then $\mathcal{P}os\,(s) := \{\epsilon\}$, where ϵ be the empty sequence.

(ii) If $s = f(s_1, \dots, s_n)$, then

$$\mathcal{P}os\,(s) := \{\epsilon\} \cup \bigcup_{i=1}^{n} \{ip \mid p \in \mathcal{P}os\,(s_i)\}.$$

The terms s_i are called *arguments*, and the symbol f is the **head symbol**, **root**, or simply the *head*. The position ϵ is the **root position**.

2. The **size** $|s|$ of a term s is the cardinality of $\mathcal{P}os\,(s)$.

3. For a position $p \in \mathcal{P}os\,(s)$, the **subterm of** s **at position** p, denoted by $s|_p$, is defined by induction on the length of p:

$$\begin{array}{rcl} s|_\epsilon & := & t, \\ f(s_1, \dots, s_n)|_{iq} & := & s_i|_q. \end{array}$$

4. For $p \in \mathcal{P}os\,(s)$, a term obtained by **replacing the subterm at position** p **by** t is denoted by $s[t]_p$, i.e.

$$\begin{array}{rcl} s[t]_\epsilon & := & s, \\ f(s_1, \dots, s_n)[t]_{iq} & := & f(s_1, \dots, s_i[t]_q, \dots, s_n). \end{array}$$

5. The set of all **variables occurring in** s is denoted by $\mathcal{V}ar\,(s)$, s.t.

$$\mathcal{V}ar\,(s) := \{x \in \mathcal{X} \mid \text{ there exists } p \in \mathcal{P}os\,(s) \text{ s.t. } s|_p = x\}.$$

6. When we talk about identity of two terms s and t, we mean **syntactic identity** and denote it by $s \equiv t$.

Notation: We may call a sequence of terms or arguments $t_1, \ldots, t_n$ a **(term) vector** and may abbreviate it by $\boldsymbol{t}$. The element at position i is denoted by $\boldsymbol{t}|_i$. The position ϵ is undefined.

Specific subterms on arbitrary positions are best described together with their context they are occurring in. Let for our purpose a **context** be a term with zero or more occurrences of a special symbol $\Box$ (called **hole**), i.e. a term over an extended signature $\Sigma \cup \{\Box\}$. If C is a context containing exactly n holes, then $C[t_1, \ldots, t_n]$ denotes the result of replacing each hole from left to right by $t_1, \ldots, t_n$. If there is exactly one occurrence of $\Box$ in C, we call C a **one-hole context**. If $t \in \mathcal{T}_\Sigma(\mathcal{X})$ can be written as $t \equiv C[s]$, s is a **subterm** of t written $s \subseteq t$. For the trivial context $C = \Box$, for any term t it holds that $C[t] \equiv t$, so $t \subseteq t$. Another subterm s of t different from t is called a **proper subterm**, written $s \subset t$.

3.2. Substitution, Matching, and Generalisation

Replacing variables in a term by other terms is called *substitution*. In particular, a substitution only affects variables so it may be defined using the set of variables $\mathcal{X}$ as domain i.e. $\sigma\colon \mathcal{X} \to \mathcal{T}_\Sigma(\mathcal{X})$. Usually it is defined as a finite set of variable assignments $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. It acts as identity on all variables not mentioned. Assume a signature Σ to be given.

Definition 3.2.1. A **substitution** is a function on terms $\sigma\colon \mathcal{T}_\Sigma(\mathcal{X}) \to \mathcal{T}_\Sigma(\mathcal{X})$ defined as a variable assignment $A = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ s.t.

$$\begin{array}{lcl}
\sigma(x) & \equiv & t,\ \text{for } (x \mapsto t) \in A, \\
\sigma(x) & \equiv & x,\ \text{for } (x \mapsto t) \notin A \text{ for some t}, \\
\sigma(c) & \equiv & c,\ \text{for some constant term} c, \\
\sigma(f(t_1, \ldots, t_n)) & \equiv & f(\sigma(t_1), \ldots, \sigma(t_n)).
\end{array}$$

Notation: Usually we write a substitution in postfix ($t\sigma$ or t^σ) instead of prefix (σt) and drop the surrounding brackets.

Due to the fact that a substitution is interpreted as a set of variable assignments, it is common sense that all variables in a term are replaced simultaneously. A substitution that replaces variables by variables is called a **(variable) renaming**.

Definition 3.2.2. The finite set of those variables a substitution does not map to themselves is the **domain of** σ, written $\mathcal{D}om\,(\sigma)$.

We will tacitly mix both notations, i.e. assume a substitution σ to operate on terms, but define it as a set of variable assignments with identity on all other terms and variables. This is more convenient for our later work and facilitates the definition of other concepts, as e.g. the composition of two substitutions.

Definition 3.2.3. The **composition** $\sigma\tau$ of two substitutions σ and τ is defined as $\sigma\tau(x) := \sigma(\tau(x))$.

More set theoretic, the composition $\sigma\tau$ of two substitutions σ and $\tau = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ with disjoint domains yields the finite set of variable assignments s.t. $\sigma\tau := \{x_1 \mapsto t_1\sigma, \ldots, x_n \mapsto t_n\sigma\} \cup \sigma$.

The concept of substitution gives rise to a partial ordering relation of terms, the so called **subsumption** order.

Definition 3.2.4. Given two terms s and t. Let $s \equiv t\sigma$ for some substitution σ. We say the term t **subsumes** s and write $s \preceq t$. The term t is also said to **match** its **(substitution) instance** s. The term t is **more general** than the term s.

The subsumption order of terms extends naturally to substitutions.

Definition 3.2.5. For two substitutions we define $\sigma \preceq \tau \Leftrightarrow (\exists\rho).(\sigma = \tau\rho)$ for some substitution ρ. Similarly, if $\sigma \preceq \tau$ we say τ is **more general** than σ.

Given two terms it may be interesting if there is a substitution which makes both terms equal when applied.

Definition 3.2.6. Let s and t be terms, then a substitution σ is called a **unifier** if $s\sigma \equiv t\sigma$. If σ is minimal w.r.t. the ordering on substitutions $\preceq$, then σ is called a **most general unifier** (MGU).

Given two terms, *unification* computes the *most general unifier*, i.e. a substitution (if any) that equalises both terms. Conversely, the so called *anti-unification* computes a term matching both of them. Such a term is a generalisation, but a simple variable is also a valid generalisation, but matching every other term though. Therefore, it is required to be minimal, or the *least general*, w.r.t. the ordering on terms $\preceq$.

Definition 3.2.7. Given a set of terms $S = \{s_1, \ldots, s_n\}$, then there exists a term t subsuming all terms in S which itself is subsumed by every other term subsuming all terms in S. The term t is called **least general generalisation** (LGG) of S [112, 113].

3.3. First-Order Constructor Term Rewriting

In the last sections we explained the basic syntax of terms and possible relations between individual terms. In this section we will look in more detail on reduction rules, rewrite relations and *constructor (term rewriting) systems* (CS). Since constructor systems are in our main focus of interest, all other concepts like rules or reductions are introduced w.r.t. them, although, they are in fact much more general and apply for any rewrite system. Therefore, the reader should be aware that, compared to other textbooks, especially the mentioned ones [6, 129], our definitions may be overly specific.

Before we come to constructor systems, we again have to define some basic building blocks first.

Definition 3.3.1. Let $\Sigma = (S, \Omega)$ be a signature s.t. the function symbols Ω may be partitioned into two disjoint set of **defined function symbols** $\mathcal{D}$ and **constructor symbols** or just **constructors** $\mathcal{C}$. Terms over a set of variables $\mathcal{X}$ and symbols from $\mathcal{C}$ are called **constructor terms** and denoted by $\mathcal{T}_{\mathcal{C}}(\mathcal{X})$.

Notation: For convenience we may ambiguously identify both, a signature and the set of function symbols by Σ. However, the semantics should always be clear from the context. From now on, in the context of constructor term rewriting, assume an arbitrary, but fixed, signature $\Sigma = (S, \Omega)$ to be given.

In general, a *reduction rule* is a pair of terms (l, r) over some signature where $\mathcal{V}ar\,(r) \subseteq \mathcal{V}ar\,(l)$. In our case we are more specific and allow only constructor terms below the root position of the left term l.

Definition 3.3.2. A **reduction rule** (or rewrite rule) for a signature Σ is a pair of terms (l, r), usually written as $l \to r$. Under some circumstances we may want to name a rule and write $\rho\colon l \to r$. The term l is called the **left-hand side** (LHS), r the **right-hand side** (RHS) of the rule (In plural we use LHSS and RHSS.). If the LHS l in $\rho\colon l \to r$ is linear, ρ is called **left-linear**.

Definition 3.3.3. For a rule $f(\boldsymbol{p}) \to r$ we may call $\boldsymbol{p}$ the **pattern** and f the **head**.

Notation: In some cases, when we talk about an arbitrary set of rules R, we may nevertheless pose some restrictions on it. For this purpose we denote with $R_{f,\boldsymbol{p}}$ a set of rules with head f and pattern $\boldsymbol{p}$.

Since a left-hand side of a rule usually contains variables, an application of a rewrite rule is quite intuitive. Given a term t that matches the LHS of a rule $\rho\colon l \to r$ with substitution σ, the term t can be rewritten to r^σ. The result is an *atomic* reduction step $l^\sigma \to_\rho r^\sigma$. The LHS l^σ is called a **redex** (from **red**ucible **ex**pression), the right-hand side r^σ is called **contractum**.

Definition 3.3.4. Given a term t, a **reduction step** (or rewrite step) according to a rule $\rho\colon l \to r$, rewrites the redex to the contractum within an arbitrary context:

$$C[l^\sigma] \to_\rho C[r^\sigma]$$

We call $\to_\rho$ the **one-step reduction relation** or just **reduction** generated by ρ.

Several (constructor) rewrite rules make up a constructor (term rewriting) system.

Definition 3.3.5. A **constructor (term rewriting) system** (CS) over a signature $\Sigma = (S, \Omega)$ with $\Omega = \mathcal{D} \cup \mathcal{C}$ and $\mathcal{D} \cap \mathcal{C} = \emptyset$, is a pair $\mathcal{R} = (\Sigma, R)$ consisting of a signature Σ and a set of rewrite rules R. For any rule $\rho\colon l \to r \in R$ following restrictions apply:

(i) $l \equiv f(\boldsymbol{p})$ and $f\colon s_1 \times \ldots \times s_n \to s \in \Sigma$, the term $\boldsymbol{p}|_i \in \mathcal{T}_\mathcal{C}(\mathcal{X})$ with $\boldsymbol{p}|_i\colon s_i$ for $i = 1 \ldots n$,

(ii) $r \in \mathcal{T}_\Sigma(\mathcal{X})$ with $r\colon s$, i.e. r is of type s,

(iii) ρ is left-linear, and

(iv) $l \notin \mathcal{X}$ with $\mathcal{V}ar\,(r) \subseteq \mathcal{V}ar\,(l)$.

The *one-step reduction relation* of $\mathcal{R}$, denoted by $\to$ (or $\to_{\mathcal{R}}$ if we are more specific), is defined as the union $\bigcup\{\to_\rho \mid \rho \in R\}$. So there is a one-step reduction in R, whenever there is a one-step reduction of a rule ρ in R.

Concatenating multiple reductions we can generate a (possible infinite) **reduction sequence** or **reduction** $t_0 \to t_1 \to t_2 \to \dots t_n$. If there is a finite reduction sequence $t_0 \to \dots \to t_n$ we may write $t_0 \twoheadrightarrow t_n$ to emphasise that there are multiple reductions and call t_n a **reduct** of t_0. If t_n is not a redex, i.e. not an instance of a LHS, of any rule in R we call t_n a **normal form** (of t_0). If there is a reduction sequence s.t. $s \twoheadrightarrow t$ and t is a normal form we say s **normalises** to t and write $s \xrightarrow{!} t$.

NOTATION: Even if $s \twoheadrightarrow t$ or $s \xrightarrow{!} t$ we still may write $s \to t$ for the sake of brevity if the meaning is clear from the context.

Definition 3.3.6. A CS is called **confluent** if for all its rules, any two redexes have a common reduct. It is **normalising** if for any terms s there is a term t, such that $s \xrightarrow{!} t$. If it is both normalising and confluent, it is **complete**.

A sufficient criteria for confluence is that no two LHSS of a CS **overlap**, i.e. do not unify. If a CS is confluent, each term has at most one normal form. If such a unique normal form for a term t exists, we denote it by $t\downarrow$.

Definition 3.3.7. We call a CS (Σ, R) **terminating** if and only if, there exists a well-founded order $<$ on $\mathcal{T}_\Sigma(\mathcal{X})$, s.t. $t > u$ for every $t, u \in \mathcal{T}_\Sigma(\mathcal{X})$ for which $t \to_R u$.

A strict order $<$ is called **well-founded** if it does not have an infinite ascending sequence $t_0 > t_1 > t_2 \dots$. For a given CS a *compatible reduction order* is a well-founded order s.t. the CS is terminating. If the CS terminates all terms have normal forms. Thus, each term has a unique normal form, if the CS is complete.

Definition 3.3.8. A **reduction order** on $\mathcal{T}_\Sigma(\mathcal{X})$ is a well founded order $<$ on $\mathcal{T}_\Sigma(\mathcal{X})$ that is

- *closed under substitutions*, i.e. for an arbitrary substitution σ, if $t < u$ then $t^\sigma < s^\sigma$
- *closed under contexts*, i.e. for an arbitrary context C, if $t < s$ then $C[t] < C[s]$.

Definition 3.3.9. A reduction order $<$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is called **compatible** with a CS (Σ, R) if $l > r$ for every rewrite rule $l \to r$ in R.

Functional programming is the main application of CS, which are left-linear and confluent. We call such CSs **orthogonal**[1]. Thus, for our purpose, a functional program is a constructor system, which is left-linear and no two LHSS overlap.

[1]Orthogonality is in fact less restricted than in our definition, but it suffices for us.

3.4. General Remarks on Notation

If it is necessary to make the relation to functional programming more precise and avoid ambiguities with functional types we may also write $l = r$ and call it **equation** or simply **rule** instead of rewrite rule $l \to r$.

We will also call things as they are, and may call a set of $n \geq 1$ rules with the same head (but different pattern) a **function definition** or just **function**, f may then also be called **function head**.

If in some context it is unambiguous, we may also treat the symbol $=$ as a distinguished symbol of some signature, so assume $l = r$ to be a single term. In this case all concepts and operations on terms naturally extend to rules and equations, respectively.

To facilitate speaking about terms and rules, we will use many concepts in a "functional" manner. If for example, r is a rule and f is a concept, applying r to f as argument means "f of r". For example $head(r)$ denotes the *head* of the rule, i.e. the head of its LHS. Similarly, $lhs(r)$ and $rhs(r)$ would denote the LHS and the RHS of r, respectively.

If the semantics is clear from the context, we may extend this to multiple arguments. For example shall $lgg(l_1, l_2, l_3)$ denote the least general generalisation of the terms l_1, l_2 and l_3.

4. Category Theory and Functional Programming

Chapter 3 already introduced the theory needed to describe our problem and later our algorithms from a syntactic point of view. Term rewriting is appropriate to explain *how it is done*. To describe *what is done*, i.e. to look at the problem from a semantic point of view, concepts from **category theory** have shown to be quite suitable. The following sections introduce necessary concepts from category theory, relate them to functional programming, and provide the theoretical background for our algorithms. They follow the standard texts by Barr and Wells [7] and Pierce [110].

4.1. Category Theory

Originally, categories were introduced in a mathematical context as a generalisation of set theory to describe mathematical structures and their relationships in an abstract way. Despite its abstractness and generality—or even because of it—it took a great influence on the design and implementation techniques of (especially functional) programming languages. Instead of reasoning about properties of certain objects, category theory more or less neglects the individual structure of objects and focuses only on the relations among themselves. In this way, it achieves its high level of abstractness and generality.

4.1.1. Categories

In one sentence, a *category* consists of *objects* related to each other by *arrows* (or *morphism*), including *identity arrows* and the *composition of morphisms*.

Definition 4.1.1. A **category** $\mathcal{C}$ consists of

(i) a set of **objects**;

(ii) a set of **arrows**;

(iii) two operations assigning to each arrow f an object $dom\ f$, its **domain**, and an object $cod\ f$, its **codomain** ($f\colon A \to B$ denotes an arrow with $dom\ f = A$ and $cod\ f = B$);

(iv) an *associative* **composition operator** assigning to any pair of arrows f and g, with $cod\ f = dom\ g$, a **composite** arrow $g \circ f\colon dom\ f \to cod\ g$, s.t. for any three arrows $f\colon A \to B$, $g\colon B \to C$, and $h\colon C \to D$

$$h \circ (g \circ f) = (h \circ g) \circ f;$$

(v) for each object A, it consists of an **identity arrow** $\mathrm{id}_A\colon A \to A$, s.t. for any arrow $f\colon A \to B$

$$\mathrm{id}_B \circ f = f \text{ and } f \circ \mathrm{id}_A = f.$$

Example 4.1.1
An intuitive example of a category is the category of sets $\mathcal{S}et$. Objects in $\mathcal{S}et$ are sets, a morphism $f\colon A \to B$ is a total function mapping from A into B. Composition of morphisms is the set-theoretic composition of functions. Identity morphisms are identity functions.

A common way to display categories is in form of **commuting diagrams**. Below the diagrams of categories with one, two and three objects are shown.

$$\begin{tikzcd} A \arrow[loop above, "\mathrm{id}_A"] \end{tikzcd} \qquad \begin{tikzcd} A \arrow[loop above, "\mathrm{id}_A"] \arrow[r, "f"] & B \arrow[loop above, "\mathrm{id}_B"] \end{tikzcd} \qquad \begin{tikzcd} & C \arrow[loop above, "\mathrm{id}_C"] & \\ A \arrow[ur, "h"] \arrow[rr, "f"'] \arrow[loop below, "\mathrm{id}_A"] & & B \arrow[ul, "g"'] \arrow[loop below, "\mathrm{id}_B"] \end{tikzcd}$$

Definition 4.1.2. A **diagram** in a category $\mathcal{C}$ is a collection of vertices and edges labelled with morphisms and objects of $\mathcal{C}$, respectively. If and only if an edge is labelled with a morphisms f and f has the domain A and the codomain B, then the edge is an arrow starting at a vertex labelled with A and pointing to a vertex labelled with B.

Diagrams are widely used to state and prove certain properties of categories or categorial constructions. Saying that a diagram commutes suffices often to prove certain properties.

Definition 4.1.3. A diagram in a category $\mathcal{C}$ is said to **commute**, if for any pair of vertices X and Y, all paths from X to Y are equal, in the sense that any path from X to Y is an arrow and all these arrows are equal in $\mathcal{C}$.

4.1.2. Universal Constructions

As already mentioned, commuting diagrams allow to reason about objects in a category and their relationships to each other, or briefly, about the structure of a category. However, apart from morphisms, no other concept has been introduced so far which may induce a structure on a category. A broad range of so called **universal constructions** are known to characterise the structure of a category. Only those universal constructions are introduced here which are needed later to elaborate this work.

The simplest of those universal constructions is the initial object and its dual, the terminal object.

Definition 4.1.4. An **initial object** of a category is an object $\mathbf{0}$, s.t. for every object A, there is exactly one arrow from $\mathbf{0}$ to A.

Definition 4.1.5. Dually, an object $\mathbf{1}$ is called a **terminal** or **final object** if, for every object A, there exists exactly one arrow from A to $\mathbf{1}$.

Categories can be seen as a generalisation of sets. So it is not surprising that, similar to sets, products and sums can be defined for categories, too.

Definition 4.1.6. Given two objects A and B, their **product** is the object $A \times B$ together with two **projections** $fst_{A\times B}\colon A \times B \to A$ and $snd_{A\times B}\colon A \times B \to B$, s.t. for any object C and arrows $f\colon C \to A$ and $g\colon C \to B$ there exists a *unique* **mediating arrow** $\langle f, g\rangle\colon C \to A \times B$ that makes the following diagram commute.

$$
\begin{tikzcd}
 & C \arrow[dl, "f"'] \arrow[d, dashed, "\langle f, g\rangle"] \arrow[dr, "g"] & \\
A & A \times B \arrow[l, "fst_{A,B}"] \arrow[r, "snd_{A,B}"'] & B
\end{tikzcd}
$$

Thus, satisfying the universal property, s.t. for $h = \langle f, g\rangle$

$$fst_{A,B} \circ h = f \text{ and } snd_{A,B} \circ h = g \qquad \text{(prod-UniProp)}$$

for all $h\colon C \to A \times B$.

The function $\langle f, g\rangle$ is called "the **pairing** of" or "**fork of functions** f and g".

Corollary 4.1.1 *From the diagram above, a couple of useful laws for products can directly be obtained.*

- ***Cancellation**: This follows directly from Equation prod-UniProp.*

$$\begin{aligned} fst_{A,B} \circ \langle f, g\rangle &= f \\ snd_{A,B} \circ \langle f, g\rangle &= g \end{aligned} \qquad \text{(prod-Cancel)}$$

- ***Reflection**: Assume $f = fst_{A,B}$ and $g = snd_{A,B}$ in $\langle fst_{A,B}, snd_{A,B}\rangle$, then*

$$\mathrm{id}_{A\times B} = \langle f, g\rangle. \qquad \text{(prod-Refl)}$$

- ***Fusion**: Assuming $\langle j, k\rangle \circ m = h$ in Equation prod-*UNIPROP *and use it in Equation prod-*CANCEL*, the equations*

$$\begin{aligned} j \circ m &= f \\ k \circ m &= g \end{aligned} \quad \Rightarrow \quad \langle j, k\rangle \circ m = \langle f, g\rangle, \qquad \text{(prod-Fuse)}$$

can be obtained. They are equivalent to

$$\langle j, k\rangle \circ m = \langle j \circ m, k \circ m\rangle.$$

Given two morphisms, their product can be defined as a morphism between two product objects in terms of projections.

Definition 4.1.7. Let $A \times C$ and $B \times D$ be product objects, then for any pair of arrows $f\colon A \to B$ and $g\colon C \to D$ their **product arrow** $f \times g\colon A \times C \to B \times D$ is defined as

$$f \times g = \langle f \circ \mathit{fst}_{A,C}, g \circ \mathit{snd}_{A,C}\rangle, \qquad \text{(prod-Arrow)}$$

making the following diagram commute:

$$\begin{tikzcd} A & A \times C \arrow[l, "\mathit{fst}_{A,C}"'] \arrow[r, "\mathit{snd}_{A,C}"] \arrow[d, dashed, "f \times g"] & C \arrow[d, "g"] \\ B & B \times D \arrow[l, "\mathit{fst}_{B,D}"] \arrow[r, "\mathit{snd}_{B,D}"'] & D \arrow[from=1-1, to=2-1, "f"] \end{tikzcd}$$

Putting the rules from the definitions 4.1.6 and 4.1.7 together, the following **absorption law for products** can be obtained:

$$(f \times g) \circ \langle h, k\rangle = \langle f \circ h, g \circ k\rangle. \qquad \text{(prod-Absorp)}$$

Dual to a product of objects, which relates to the Cartesian product on sets, is the sum of two object, relating to the disjoint union of sets.

Definition 4.1.8. Given two objects A and B, their **sum** or **coproduct** is the object $A + B$ together with two **injections** $\mathit{inl}_{A,B}\colon A \to A + B$ and $\mathit{inr}_{A,B}\colon B \to A + B$, s.t. for any object C and arrow $f\colon A \to C$ and $g\colon B \to C$ there exists a *unique* **mediating arrow** $[f, g]\colon A + B \to C$ that makes the following diagram

$$\begin{tikzcd} A \arrow[r, "inl_{A,B}"] \arrow[dr, "f"'] & A+B \arrow[d, dashed, "{[f,g]}"] & B \arrow[l, "inr_{A,B}"'] \arrow[dl, "g"] \\ & C & \end{tikzcd}$$

commute. Hence, the coproduct is defined by the universal property, s.t. for $h = [f, g]$

$$h \circ inl_{A,B} = f \text{ and } h \circ inr_{A,B} = g \qquad \text{(sum-UniProp)}$$

for all $h \colon A + B \to C$.

The function $[f, g]$ is called the "**case analysis** for" or "the **join of the functions** f and g".

Corollary 4.1.2 *Again, in analogy to products, a couple of useful laws for coproducts can directly be obtained from the diagram above.*

- ***Cancellation****: This follows directly from Equation sum-UniProp.*

$$\begin{aligned} [f, g] \circ inl_{A,B} &= f \\ [f, g] \circ inr_{A,B} &= g \end{aligned} \qquad \text{(sum-Cancel)}$$

- ***Reflection****: Assume* $f = inl_{A,B}$ *and* $g = inr_{A,B}$*, then*

$$\mathrm{id} = [inl_{A,B}, inr_{A,B}]. \qquad \text{(sum-Refl)}$$

- ***Fusion****: Taking* $h = m \circ [j, k]$ *in Equation sum-UniProp, the equations*

$$\begin{aligned} m \circ j &= f \\ m \circ k &= g \end{aligned} \quad \Rightarrow \quad m \circ [j, k] = [f, g], \qquad \text{(sum-Fuse)}$$

  can be obtained. They are equivalent to

$$m \circ [j, k] = [m \circ j, m \circ k].$$

Similar to products, the sum of objects extends naturally to morphisms.

Definition 4.1.9. Let $A + C$ and $B + D$ be sum objects, then for any pair of arrows $f \colon A \to B$ and $g \colon C \to D$ their **sum arrow** or **coproduct arrow** $f+g \colon A+C \to B+D$ is defined as

$$f + g = [inl_{B,D} \circ f, inr_{B,D} \circ g], \qquad \text{(sum-Arrow)}$$

making the following diagram commute:

$$\begin{tikzcd}
A \arrow[r, "inl_{A,C}"] \arrow[d, "f"] & A + C \arrow[d, dashed, "f + g"] & C \arrow[l, "inr_{A,C}"'] \arrow[d, "g"] \\
B \arrow[r, "inl_{B,D}"'] & B + D & D \arrow[l, "inr_{B,D}"]
\end{tikzcd}$$

Hence, for any morphisms $h \colon B \to E$ and $k \colon D \to F$ it holds:

$$(f + g) \circ (h + k) = f \circ h + g \circ k \qquad \text{(sum-Comp)}$$

Similar to products, the rules from the definitions 4.1.8 and 4.1.9 can be used to obtain an **absorption law for coproducts** :

$$[f, g] \circ (h + k) = [f \circ h, g \circ k]. \qquad \text{(sum-Absorp)}$$

4.1.3. Functors

In general, category theory does not impose any restrictions on the objects in a category, only on the morphisms: For any two morphisms, there must be a composite in the category, and any object must have an identity arrow. Thus, nothing speaks against a **category of categories** $\mathcal{C}at$, whilst a sensible structure-preserving mapping between categories can be defined to represent morphisms in $\mathcal{C}at$. Such mappings between categories are called functors, mapping objects to objects and morphisms to morphisms.

Definition 4.1.10. Let $\mathcal{C}$ and $\mathcal{D}$ be two categories. A **functor** $\mathsf{F} \colon \mathcal{C} \to \mathcal{D}$ maps every $\mathcal{C}$-object A to a $\mathcal{D}$-object $\mathsf{F}(A)$ and every $\mathcal{C}$-morphism $f \colon A \to B$ to a $\mathcal{D}$-morphism $\mathsf{F}(f) \colon \mathsf{F}(A) \to \mathsf{F}(B)$, s.t. for all $\mathcal{C}$-objects A and $\mathcal{C}$-morphisms f and g identities

$$\mathsf{F}(\mathrm{id}_A) = \mathrm{id}_{\mathsf{F}(A)} \qquad \text{(func-Id)}$$

and composition

$$\mathsf{F}(g \circ f) = \mathsf{F}(g) \circ \mathsf{F}(f) \qquad \text{(func-Comp)}$$

are preserved.

Notation: The parenthesis in functor applications may be dropped, writing $\mathsf{F}A$ instead of $\mathsf{F}(A)$. The composition of two functors F and G (“G after F”) is written using juxtaposition GF just as in normal functor application of objects and morphisms. $\mathsf{GF}A$ may be parsed as $\mathsf{G}(\mathsf{F}A)$ or $(\mathsf{GF})A$, both meaning the same.

Some special functors of interest will be introduced now. They are of importance later. Endofunctors map from a category in the same category. Furthermore, there are identity and constant functors, as well as binary functors.

Definition 4.1.11. Let $\mathcal{C}$ be a category, a functor $\mathsf{F}\colon \mathcal{C} \to \mathcal{C}$ is called an **endofunctor**.

Definition 4.1.12. The endofunctor $\mathsf{Id}\colon \mathcal{C} \to \mathcal{C}$ is the **identity functor**, s.t. for any $\mathcal{C}$-object A and $\mathcal{C}$-morphisms f:

(i) $\mathsf{Id}A = A$, and

(ii) $\mathsf{Id}f = f$.

Definition 4.1.13. For a $\mathcal{C}$-object A the endofunctor $\mathsf{K}_A\colon \mathcal{C} \to \mathcal{C}$ is called the **constant functor**, s.t. for any $\mathcal{C}$-object B and $\mathcal{C}$-morphisms f:

(i) $\mathsf{K}_A B = A$, and

(ii) $\mathsf{K}_A f = \mathrm{id}_A$.

Definition 4.1.14. A **bifunctor** $\dagger\colon \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is a binary functor, s.t. if for any $\mathcal{C}$-morphisms $f\colon A \to C$ and $g\colon B \to D$, $f \dagger g\colon A \dagger B \to C \dagger D$ is preserving identities

$$\mathrm{id} \dagger \mathrm{id} = \mathrm{id}, \qquad \text{(bifunc-Id)}$$

and composition

$$(f \dagger g) \circ (h \dagger j) = (f \circ h) \dagger (g \circ j). \qquad \text{(bifunc-Comp)}$$

Looking at a bifunctor more closely, it becomes apparent that it has been used already, without explicitly naming it. Both, sum and product can bee seen as a bifunctor. They map two objects to their product, respectively sum, and likewise morphisms.

Definition 4.1.15. If each pair of objects in $\mathcal{C}$ has products, one says the $\mathcal{C}$ *has products*. Using the product arrow defined in Equation prod-Arrow $\times$ extends to a bifunctor $\mathcal{C} \times \mathcal{C} \to \mathcal{C}$. It is defined pointwise as:

$$(\mathsf{F} \times \mathsf{G})h = \mathsf{F}h \times \mathsf{G}h.$$

Of course, this also applies to sums.

Definition 4.1.16. We say that $\mathcal{C}$ *has sums* or *has coproducts* if each pair of objects in $\mathcal{C}$ has sum. By Equation sum-Arrow of a coproduct arrow, $+$ extends to a bifunctor $\mathcal{C} \times \mathcal{C} \to \mathcal{C}$, being pointwise defined as:

$$(\mathsf{F} + \mathsf{G})h = \mathsf{F}h + \mathsf{G}h.$$

Notation: In cases where the infix notation of a bifunctor is inappropriate we will write $\mathsf{F}(A, X)$ for a bifunctor $\mathsf{F}\colon \mathcal{C} \times \mathcal{C} \to \mathcal{C}$. When the first argument of a bifunctor is arbitrary but fixed, $\mathsf{F}_A X$ is written instead and treated as it were a parameterised endofunctor. Consequently, $\mathsf{F}_A f = \mathsf{F}(\mathrm{id}_A, f)$.

A special class of functors is worth mentioning. Functors exclusively built from identities, constants, products, and coproducts are called *polynomial functors*.

Definition 4.1.17. A **polynomial functor** is defined inductively:

- Given an arbitrary object A, the identity functor Id and the constant functor K_A are polynomial;
- if F and G are polynomial, then their composition FG, their product $\mathsf{F} \times \mathsf{G}$, and their coproduct $\mathsf{F} + \mathsf{G}$ are polynomial.

Example 4.1.2
Consider a functor defined by $\mathsf{F}X = \mathbf{1} + (A \times X)$ and $\mathsf{F}f = \mathrm{id}_{\mathbf{1}} + (\mathrm{id}_A \times f)$. It is a polynomial functor, because $\mathsf{F} = \mathsf{K}_{\mathbf{1}} + (\mathsf{K}_A \times \mathsf{Id})$, where $+$ and $\times$ are pointwise.

The last piece of category theoretic basics we need are *natural transformations*. Natural transformations are structure-preserving mappings between functors. They map each object of one category to an object of another category, s.t. its structure is preserved.

Definition 4.1.18. Given two functors $\mathsf{F}, \mathsf{G} \colon \mathcal{A} \to \mathcal{B}$ between two categories $\mathcal{A}$ and $\mathcal{B}$, a **transformation** from F to G is a collection of arrows $\phi_A \colon \mathsf{F}A \to \mathsf{G}A$ one for each object A in $\mathcal{A}$. A transformation is called **natural** if

$$\mathsf{G}f \circ \phi_A = \phi_B \circ \mathsf{F}f$$

for all arrows $f \colon A \to B$ in $\mathcal{A}$, s.t. the following diagram commutes:

$$\begin{tikzcd}
\mathsf{F}A \arrow[r, "\phi_A"] \arrow[d, "\mathsf{F}f"'] & \mathsf{G}A \arrow[d, "\mathsf{G}f"] \\
\mathsf{F}B \arrow[r, "\phi_B"'] & \mathsf{G}B
\end{tikzcd}$$

Natural transformations can be seen as functions which are independent on the structure of their argument elements. For example in functional programming languages, all polymorphic functions are natural transformations [133].

4.2. Functional Programming and Category Theory

Functional Programming and Category Theory are closely related and their interdependencies are especially researched in *Constructive Algorithmics*. The *Bird-Meertens formalism* [13, 11] makes use of such an algebraic approach to *calculate* programs from a specification and prove their correctness. For a comprehensive study of type morphisms see Vene [131]. By now, category theory has found it's way into functional programming through Meijer et al. [87].

The following section will borrow those concepts from the previously mentioned authors and will introduce them in the context of this work. At some points it is suitable to be more specific and refer to, or give examples in a specific functional programming language. The language of choice is HASKELL [57]. For the standard introduction see Bird [12], a more recent introduction with focus on real applications was written by O'Sullivan et al. [106]. In Appendix A a short reference can be found.

4.2.1. Primitive Data Types and Functions

In general, the essence of functional programming is captured within a fixed category $\mathcal{C}$, where types are objects and total functions are morphisms of $\mathcal{C}$. The values constituting a type A are represented by morphisms in this category from the terminal object to A. Thus, a value a of type A is a morphism $a\colon \mathbf{1} \to A$. Applying a function $f\colon A \to B$ to a value a of type A is identified by composing the two morphisms a and f in $\mathcal{C}$ to $f \circ a$. Lifting values (points) to functions leads to a so called point-free style of programming, where functions can exclusively be described by functional composition.

When considering only total functions, this easily can be motivated by identifying $\mathcal{C}$ with the category of sets $\mathcal{S}et$, which has sets as objects and total functions between those sets as morphisms. Primitive data types like sums and products are represented by (categorial) products and coproducts, function types by exponentials and so forth. However, it is permissible to be more general here. For example, $\mathcal{C}$ is not necessarily required to have exponentials, i.e. be Cartesian closed, well-pointed, and be locally small. $\mathcal{C}$ only needs to be *distributive*, i.e. $\mathcal{C}$ has finite products and finite coproducts, and consequently terminal and initial objects, and the distribution of products over coproducts.

NOTATION: From now on, if not specifically stated otherwise, our categories are *distributive*!

Definition 4.2.1. A category $\mathcal{C}$ with products and coproducts is **distributive** if there are two unique isomorphisms

$$distr\colon A \times (B + C) \to (A \times B) + (A \times C)$$

defined as $distr = [\mathrm{id} \times inl_{B,C}, \mathrm{id} \times inr_{B,C}]$ and with inverse

$$distr^{-1}\colon (A \times B) + (A \times C) \to A \times (B + C),$$

and

$$null\colon A \times \mathbf{0} \to \mathbf{0}$$

with inverse

$$null^{-1}\colon \mathbf{0} \to A \times \mathbf{0}.$$

Thus, in $\mathcal{C}$ there are natural isomorphisms resulting into the following equivalences:

$$\begin{aligned} A \times (B + C) &\cong (A \times B) + (A \times C) \\ A \times \mathbf{0} &\cong \mathbf{0}, \end{aligned}$$

as well as

$$\begin{aligned} A \times (B \times C) &\cong (A \times B) \times C & \qquad A + (B + C) &\cong (A + B) + C \\ A \times B &\cong B \times A & A + B &\cong B + A \\ A \times \mathbf{1} &\cong A & A + \mathbf{0} &\cong A, \end{aligned}$$

which follow from the fact that $\mathcal{C}$ has products and coproducts with terminal and initial objects.

4.2.2. Inductive Data Types

The last ingredient of our categorical model of a functional programming language are inductively defined data types. Inductive types are generated by constructors and come equipped with a scheme for structural recursion induced by these constructors. From a categorical point of view they correspond to initial objects in a category of functor-algebras.

Definition 4.2.2. Given an endofunctor $\mathsf{F}\colon \mathcal{C} \to \mathcal{C}$, an **$\mathsf{F}$-algebra $\mathbf{A}$** $= (A, \varphi)$ is a tuple consisting of an object A, the **carrier** of the algebra, and a morphism $\varphi\colon \mathsf{F}A \to A$ which represents the **algebraic structure** of $\mathbf{A}$.

Given two algebras, a structure preserving mapping, i.e. a homomorphism, can be defined between them.

Definition 4.2.3. Let $\mathbf{A} = (A, \varphi)$ and $\mathbf{B} = (B, \psi)$ be two F-algebras, a homomorphism, called **F-morphism**, between them is a morphism $f\colon A \to B$, s.t.

$$f \circ \varphi = \psi \circ \mathsf{F}f,$$

which makes the following diagram commute:

$$\begin{tikzcd} \mathsf{F}A \arrow[r, "\varphi"] \arrow[d, "\mathsf{F}f"'] & A \arrow[d, "f"] \\ \mathsf{F}B \arrow[r, "\psi"] & B \end{tikzcd}$$

The collection of all algebras induced by a functor F itself gives rise to a category, where objects are F-algebras and arrows are homomorphisms between them, because identity arrows are homomorphisms and composed homomorphisms are homomorphisms, too.

Definition 4.2.4. $\mathcal{A}lg_{\mathsf{F}}$ is the category of functor algebras with F-algebras as objects and F-morphisms as arrows.

The category $\mathcal{A}lg_{\mathsf{F}}$ may or may not contain initial objects, i.e. *initial* F-*algebras*, but if they exist, they are uniquely defined up to isomorphism. Such an initial algebra is called *the* initial algebra. The existence of initial objects in $\mathcal{A}lg_{\mathsf{F}}$ depends on the functor F and especially for polynomial functors of a distributive category they exist [84][1].

Definition 4.2.5. An **initial F-algebra** is an initial object in the category $\mathcal{A}lg_{\mathsf{F}}$ and denoted by $\mu\mathbf{F} = (\mu\mathsf{F}, \mathrm{in}_{\mathsf{F}})$ with *carrier* $\mu\mathsf{F}$ and *algebraic structure* in_{F}.

Notation: If the morphism φ that gives an F-algebra $\mathbf{A} = (A, \varphi)$ its structure is fully defined with domain and codomain, φ may denote both, the morphism and the algebra.

4.2.3. Structural Recursion via Catamorphisms

By definition, initial objects have a unique arrow to every other object in the category, and so do initial F-algebras.

Definition 4.2.6. Given an endofunctor $\mathsf{F}\colon \mathcal{C} \to \mathcal{C}$ and the initial algebra $\mu\mathbf{F} = (\mu\mathsf{F}, \mathrm{in}_{\mathsf{F}})$, for any F-algebra $\mathbf{A} = (A, \varphi)$ there exists a *unique* morphism $f\colon \mu\mathsf{F} \to A$, s.t.

$$f \circ \mathrm{in}_{\mathsf{F}} = \varphi \circ \mathsf{F}f \quad \iff \quad f = \llparenthesis \varphi \rrparenthesis_{\mathsf{F}},$$

making the diagram commute:

$$\begin{tikzcd}
\mathsf{F}\mu\mathsf{F} \arrow[r, "\mathrm{in}_{\mathsf{F}}"] \arrow[d, "\mathsf{F}f"'] & \mu\mathsf{F} \arrow[d, "f"] \\
\mathsf{F}A \arrow[r, "\varphi"] & A
\end{tikzcd}$$

The distinguished morphism $f\colon \mu\mathsf{F} \to A$ which is witness of initiality is called **F-catamorphism**[2] or **F-fold** of φ. The unique catamorphism f solely depends on the structure of F and the mediating morphism φ. Therefore, it is denoted by putting this mediating arrow into *banana-brackets* $\llparenthesis \varphi \rrparenthesis_{\mathsf{F}}$. Catamorphisms, like other constructions by universal properties, satisfy special fusion and reflection laws.

Corollary 4.2.1 *Let* $(\mu\mathsf{F}, \mathrm{in}_{\mathsf{F}})$ *be an initial* F-*algebra.*

- ***Cancellation****: For any* F-*algebra* $\varphi\colon \mathsf{F}A \to B$

$$\llparenthesis \varphi \rrparenthesis_{\mathsf{F}} \circ \mathrm{in}_{\mathsf{F}} = \varphi \circ \mathsf{F}\llparenthesis \varphi \rrparenthesis_{\mathsf{F}} \qquad \text{(cata-Self)}$$

[1] In fact, if F is ω-cocontinuous, i.e. the base category is a partial ordering and F is monotonic (preserving colimits of ω-chains), the initial algebras are guaranteed to exist and polynomial functors are ω-cocontinuous [131], i.e. preserve the monotonicity.

[2] The word *catamorphism* comes from the Greek work $\kappa\alpha\tau\alpha$ meaning 'downwards'.

- ***Reflection***:

$$\mathrm{id} = \llparenthesis \mathrm{id} \rrparenthesis_{\mathsf{F}} \tag{cata-Refl}$$

- ***Fusion***: *For any* F*-algebras* $\varphi\colon \mathsf{F}A \to A$, $\psi\colon \mathsf{F}B \to B$ *and an arrow* $f\colon A \to B$

$$f \circ \varphi = \psi \circ \mathsf{F}f \Rightarrow f \circ \llparenthesis \varphi \rrparenthesis_{\mathsf{F}} = \llparenthesis \psi \rrparenthesis_{\mathsf{F}} \tag{cata-Fuse}$$

Let us summarise the rules from Corollary 4.2.1 in the following diagram.

$$\begin{tikzcd}
F\mu F \arrow[r, "\mathrm{in}_{\mathsf{F}}"] \arrow[d, "F\llparenthesis \varphi \rrparenthesis_{\mathsf{F}}"'] & \mu F \arrow[d, "\llparenthesis \varphi \rrparenthesis_{\mathsf{F}}"'] \arrow[dd, bend left, "\llparenthesis \psi \rrparenthesis_{\mathsf{F}}"] \\
FA \arrow[r, "\varphi"] \arrow[d, "F\,f"'] & A \arrow[d, "f"] \\
FB \arrow[r, "\psi"] & B
\end{tikzcd}$$

The cata-Self rule follows directly from the catamorphism definition and its uniqueness. However, when read from left to right, it can also be seen as a reduction rule for terms where a catamorphism is applied to a data constructor. The reduction recurses into the term, replacing all constructors with an algebra with the same signature.

The cata-Refl equation states that when constructors are replaced by themselves, nothing is changed.

The cata-Fuse law simply says that if between two arbitrary algebras a homomorphism $f\colon A \to B$ exists, f composed with a catamorphism from the initial algebra to A must yield a direct catamorphism to B, because of uniqueness.

Intuitively, the initial algebra $\mathrm{in}_{\mathsf{F}} : \mathsf{F}\mu\mathsf{F} \to \mu\mathsf{F}$ represents the collection of constructors for an inductively defined type $\mu\mathsf{F}$. The catamorphism is the witness of initiality and corresponds to a simple, structurally defined recursive function. The result type A and the step function φ are modelled by an algebra (A, φ) which together with the initial algebra uniquely define the catamorphism.

Formally this can be justified by the fact that in_{F} is an isomorphism which was first stated by Lambek [79].

Theorem 4.2.1. The initial algebra $\mathrm{in}_{\mathsf{F}}\colon \mathsf{F}\,\mu\mathsf{F} \to \mathsf{F}\mu$ is an isomorphism with inverse

$$\mathrm{in}_{\mathsf{F}}^{-1} = \llparenthesis \mathsf{F}\,\mathrm{in}_{\mathsf{F}} \rrparenthesis_{\mathsf{F}} \tag{in-inv-Def}$$

Proof. It is necessary to show that $\mathrm{in}_{\mathsf{F}}^{-1}\colon \mu\mathsf{F} \to \mathsf{F}\mu\mathsf{F}$ is the pre- and post-inverse of $\mathrm{in}_{\mathsf{F}}\colon \mathsf{F}\mu\mathsf{F} \to \mu\mathsf{F}$:

$\Longrightarrow$

$$
\begin{array}{ll}
 & \mathrm{in}_{\mathsf{F}} \circ \mathrm{in}_{\mathsf{F}}^{-1} \\
= & \qquad [\text{in-inv-}\textsc{Def}] \\
 & \mathrm{in}_{\mathsf{F}} \circ (\!|\mathsf{F}\,\mathrm{in}_{\mathsf{F}}|\!)_{\mathsf{F}} \\
= & \qquad [\text{cata-}\textsc{Fuse}] \quad
\left|\begin{array}{ll}
 & f \circ \varphi = \psi \circ \mathsf{F} f \\
\equiv & \qquad [f = \psi = \mathrm{in}_{\mathsf{F}}] \\
 & \mathrm{in}_{\mathsf{F}} \circ \varphi = \mathrm{in}_{\mathsf{F}} \circ \mathsf{F}\,\mathrm{in}_{\mathsf{F}} \\
\equiv & \qquad [\varphi = \mathsf{F}\,\mathrm{in}_{\mathsf{F}}] \\
 & \mathrm{in}_{\mathsf{F}} \circ \mathsf{F}\,\mathrm{in}_{\mathsf{F}} = \mathrm{in}_{\mathsf{F}} \circ \mathsf{F}\,\mathrm{in}_{\mathsf{F}} \\
\Rightarrow & \qquad [\text{cata-}\textsc{Fuse}] \\
 & \mathrm{in}_{\mathsf{F}} \circ (\!|\mathsf{F}\,\mathrm{in}_{\mathsf{F}}|\!)_{\mathsf{F}} = (\!|\mathrm{in}_{\mathsf{F}}|\!)_{\mathsf{F}}
\end{array}\right. \\
 & (\!|\mathrm{in}_{\mathsf{F}}|\!)_{\mathsf{F}} \\
= & \qquad [\text{cata-}\textsc{Refl}] \\
 & \mathrm{id}
\end{array}
$$

For proving the post-inverse, the previous step can be used.

$\Longleftarrow$ For proving the post-inverse, the previous step can be used.

$$
\begin{array}{ll}
 & \mathrm{in}_{\mathsf{F}}^{-1} \circ \mathrm{in}_{\mathsf{F}} \\
= & \qquad [\text{in-inv-}\textsc{Def}] \\
 & (\!|\mathsf{F}\,\mathrm{in}_{\mathsf{F}}|\!)_{\mathsf{F}} \circ \mathrm{in}_{\mathsf{F}} \\
= & \qquad [\text{cata-}\textsc{Self}] \\
 & \mathsf{F}\,\mathrm{in}_{\mathsf{F}} \circ \mathsf{F}\,(\!|\mathsf{F}\,\mathrm{in}_{\mathsf{F}}|\!)_{\mathsf{F}} \\
= & \qquad [\text{func-}\textsc{Comp}] \\
 & \mathsf{F}(\mathrm{in}_{\mathsf{F}} \circ (\!|\mathsf{F}\,\mathrm{in}_{\mathsf{F}}|\!)_{\mathsf{F}}) \\
= & \qquad [\Longrightarrow\text{, see above}] \\
 & \mathsf{F}\,\mathrm{id} \\
= & \qquad [\text{func-}\textsc{Id}] \\
 & \mathrm{id}
\end{array}
$$

□

Thus, the carrier $\mu\mathbf{F}$ of the algebra in_{F} is a fixed point of the functor F, i.e. the initial algebra $\mathrm{in}_{\mathsf{F}} : \mathsf{F}\mu\mathsf{F} \to \mu\mathsf{F}$ is an isomorphisms with inverse $\mathrm{in}_{\mathsf{F}}^{-1} = (\!|\mathsf{F}\,\mathrm{in}_{\mathsf{F}}|\!)_{\mathsf{F}}$. Roughly speaking, $\mu\mathbf{F}$ is the least fixed point, because it is initial in $\mathcal{A}lg_{\mathsf{F}}$. See [79] for a full proof.

Theorem 4.2.1 generalises the notion of the least fixed point from lattice theory in such a sense that if the base category is a preorder and consequently an endofunctor is a monotonic functor, i.e. it preserves the preorder, then the carrier of the initial algebra is the least fixed point of the given functor.

Before deepening these findings with some examples, it may be convenient to summarise the last theoretical part. Any arbitrary functor F induces F-algebras which form the category $\mathcal{A}lg_{\mathsf{F}}$ with F-algebras as objects and F-homomorphisms as arrows. Depending on the functor F, the category $\mathcal{A}lg_{\mathsf{F}}$ may have initial objects. If an initial object exists, there is also a witnessing morphism to any other object in $\mathcal{A}lg_{\mathsf{F}}$ which is called F-catamorphism. Inductive data types can be interpreted as F-algebras induced by a polynomial functor. Also, they are the least fixed point of this functor. Categories of polynomial functor algebras have initial objects, namely the least fixed point of this functor, which are the inductive data types.

Example 4.2.1

The initial algebras in a distributive category are named by type declarations common in functional programming. For the sake of simplicity, just assume $\mathcal{S}et$ as our base category. In Haskell for example, the type of natural number `Nat` may be defined as Peano's integers by the following syntax:

```
Nat = Zero | Succ Nat
```

We won't let confuse ourselves by Haskell's convention to write constructors in upper case, because all it defines can be interpreted as a homomorphism to the set of natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$ with a constant function `zero` and the successor function `succ`:

```
zero   = 0
succ n = n + 1.
```

The data type declaration simply defines the initial algebra $\mathrm{in}_{\mathsf{N}} = [\mathit{zero}, \mathit{succ}] \colon \mathsf{N}\mathit{Nat} \to \mathit{Nat}$ of the functor N defined by

$$\begin{aligned} \mathsf{N} &= \mathsf{K}_{\mathbf{1}} + \mathsf{Id} \text{ , s.t.} \\ \mathsf{N}A &= \mathbf{1} + A \text{ , and} \\ \mathsf{N}f &= \mathrm{id}_{\mathbf{1}} + f. \end{aligned}$$

The constructors of the corresponding data type are:

$$\begin{aligned} \mathit{zero} &: \mathit{Nat} \\ \mathit{succ} &: \mathit{Nat} \to \mathit{Nat}. \end{aligned}$$

The functor N is polynomial, so the category $\mathcal{A}lg_{\mathsf{N}}$ has an initial object. It is the fixed point $\mu\mathsf{N}$ of the functor N and is now simply called *Nat*. Assume now an arbitrary algebra φ such that the following diagram commutes:

$$\begin{tikzcd}
\mathsf{N}Nat \arrow[r, "{[zero, succ]}"] \arrow[d, "\mathsf{N}f"] & Nat \arrow[d, "f"] \\
\mathsf{N}C \arrow[r, "\varphi"] & C
\end{tikzcd}$$

Note that for any algebra (C, φ) of the functor $\mathsf{N}\colon \mathcal{S}et \to \mathcal{S}et$ the object $\mathsf{N}C$ is a sum and any morphism out of $\mathsf{N}C$, i.e. any algebra φ, is a join of two functions $\varphi = [c, h]$, where $c\colon \mathbf{1} \to C$ and $h\colon C \to C$, s.t. the diagram commutes:

$$\begin{tikzcd}
\mathbf{1} \arrow[r, "zero"] \arrow[d, equal, "\mathrm{id}_{\mathbf{1}}"] & Nat \arrow[d, "f"] & Nat \arrow[l, "succ"'] \arrow[d, "f"] \\
\mathbf{1} \arrow[r, "c"] & C & C \arrow[l, "h"']
\end{tikzcd}$$

The relation of the last two diagrams can exactly be formalised by spelling out the function f defined by $f = (\!|\varphi|\!) = (\!|[c, h]|\!)$. Starting with the definition of the catamorphism, simplification reveals:

$$
\begin{array}{ll}
 & f \circ \mathrm{in}_{\mathsf{N}} = \varphi \circ \mathsf{N} f \\
\equiv & \qquad [\text{definition of } \mathsf{N}] \\
 & f \circ \mathrm{in}_{\mathsf{N}} = \varphi \circ (\mathrm{id}_{\mathbf{1}} + f) \\
\equiv & \qquad [\text{definition of } \varphi] \\
 & f \circ \mathrm{in}_{\mathsf{N}} = [c, h] \circ (\mathrm{id}_{\mathbf{1}} + f) \\
\equiv & \qquad [\text{sum-}\textsc{Absorp}] \\
 & f \circ \mathrm{in}_{\mathsf{N}} = [c \circ \mathrm{id}_{\mathbf{1}}, h \circ f] \\
\equiv & \qquad [\mathrm{in}_{\mathsf{N}} = [\mathit{zero}, \mathit{succ}]] \\
 & f \circ [\mathit{zero}, \mathit{succ}] = [c \circ \mathrm{id}_{\mathbf{1}}, h \circ f] \\
\equiv & \qquad [\text{sum-}\textsc{Fuse}] \\
 & [f \circ \mathit{zero}, f \circ \mathit{succ}] = [c \circ \mathrm{id}_{\mathbf{1}}, h \circ f] \\
\equiv & \qquad [\text{sum-}\textsc{Cancel}] \\
 & f \circ \mathit{zero} = c \circ \mathrm{id}_{\mathbf{1}} \text{ and } f \circ \mathit{succ} = h \circ f
\end{array}
$$

So for any algebra $\varphi = [c, h]$ in $\mathcal{A}lg_{\mathsf{N}}$ the function f is exactly defined by the universal property

$$
\begin{array}{lcl}
f \circ \mathit{zero} & = & c \\
f \circ \mathit{succ} & = & h \circ f.
\end{array}
$$

The function f can be turned into a higher-order Haskell function `foldn` parametrised with two arguments, i.e. a function and a constant as shown in Listing 4.1.

Listing 4.1: Catamorphism for Peano Integers

```
foldn :: (α →α) → α → Nat → α
foldn h c Zero     = c
foldn h c (Succ p) = h (foldn h c p)
```

Thus, `foldn f c i` yields the result of applying i-times the function f to the default value c. Let now $f = (\!|\varphi|\!) = (\!|[c, h]|\!)$ be defined as the partial function `foldn h c` of type `Nat` $\to$ α. For example, addition and multiplication of two natural numbers can be defined as catamorphisms:

```
add,mult :: Nat → Nat → Nat
add a b  = foldn Succ a b
mult a b = foldn (add a) Zero b .
```

Example 4.2.2

A further example is the data type of cons-lists over arbitrary but fixed elements A. Let $\mathsf{L}_A \colon \mathcal{S}et \to \mathcal{S}et$ be a bifunctor parameterised in A and defined as

$$\begin{aligned}
\mathsf{L}_A &= \mathsf{K}_{\mathbf{1}} + (\mathsf{K}_A \times \mathsf{Id}) \text{ , s.t.}\\
\mathsf{L}_A X &= \mathbf{1} + (A \times X) \text{ , and}\\
\mathsf{L}_A f &= \mathrm{id}_{\mathbf{1}} + (\mathrm{id}_A \times f).
\end{aligned}$$

As before, the initial L_A-algebra $(\mu\mathsf{L}_A, \mathrm{in}_{\mathsf{L}_A})$ will be renamed for convenience s.t. the data type of lists with elements of type A is called $List_A$ with constructors

$$\begin{aligned}
nil &: List_A\\
cons &: A \times List_A \to List_A
\end{aligned}$$

For an arbitrary algebra φ the following diagram commutes:

$$\begin{tikzcd}
\mathsf{L}_A List_A \arrow[r, "{[nil, cons]}"] \arrow[d, "\mathsf{L}_A f"] & List_A \arrow[d, "f"] \\
\mathsf{L}_A C \arrow[r, "\varphi"] & C
\end{tikzcd}$$

To get a better understanding let us this time make the application of the functor L_A explicit and spell everything out in all details:

$$\begin{tikzcd}
\mathbf{1} + (A \times List_A) \arrow[r, "{[nil, cons]}"] \arrow[d, "\mathrm{id}_{\mathbf{1}} + (\mathrm{id}_A \times (\!|\varphi|\!)_{\mathsf{L_A}})"] & List_A \arrow[d, "(\!|\varphi|\!)_{\mathsf{L_A}}"] \\
\mathbf{1} + (A \times C) \arrow[r, "\varphi"] & C
\end{tikzcd}$$

Because the algebra φ is an arrow out of a sum, it must be a join of two functions $c\colon \mathbf{1} \to C$ and $h\colon A \times C \to C$ s.t. $\varphi = [c, h]$. The catamorphism $f = (\![c, h]\!)_{\mathsf{L_A}}$ is the unique solution of the equation system which can be obtained by spelling out the commuting condition of the diagram above as in Example 4.2.2:

$$\begin{aligned}
f \circ nil &= c\\
f \circ cons &= h \circ (\mathrm{id}_A \times f).
\end{aligned}$$

In functional programming f is known as `foldr` and in HASKELL defined as a higher order function, s.t. `foldr f c` takes a function `f` $::\ \alpha \to \beta \to \beta$ and a default value `c` $::\ \alpha$ as input and returns a function of type `c` $::$ `[`α`]` $\to \beta$.

Listing 4.2: Catamorphism on Lists

```
foldr :: (α → β → β) → α → [α] → β
foldr f c []     = c
foldr f c (x:xs) = x `f` (foldr f c xs)
```

Intuitively, `foldr` replaces all *nil* constructors by the provided default value and all *cons* constructors by a call to its argument function.

Let us now be even more concrete and assume $length = (\!|\varphi|\!)_{\mathsf{L_A}} : List_A \to Nat$ to be a homomorphism between lists and natural numbers. The issue is described in the next diagram. To improve readability and get a more intuitive understanding the sums on the left side "swing out" to the right side making the functor application explicit:

$$\begin{tikzcd}
\mathbf{1} \arrow[r, "nil"] \arrow[d, equal, "\mathrm{id}_{\mathbf{1}}"] & List_A \arrow[d, "length"] & A \times List_A \arrow[l, "cons"'] \arrow[d, "\mathrm{id}_A \times length"] \\
\mathbf{1} \arrow[r, "c"] & Nat & A \times Nat \arrow[l, "h"']
\end{tikzcd}$$

The question which now arises is "*How does the algebra* $[c, h]$ *look like?*". Well, this completely depends on the semantics of *length*. Since the empty list has length zero, it must hold that $c = zero$. The length of any non-empty list is one plus the length of the list with the first element removed. Putting it into a function yields $h = succ \circ snd_{A,Nat}$. So $length = (\![zero, succ \circ snd_{A,Nat}]\!)_{\mathsf{L_A}}$ or without catamorphism:

$$\begin{aligned}
length \circ nil &= zero \\
length \circ cons &= succ \circ snd \circ (\mathrm{id}_A \times length),
\end{aligned}$$

which is the same as `foldr (Succ ∘ snd) Zero` in Haskell.

Example 4.2.3
Another typical example of an inductive data type induced by a bifunctor are binary node trees. Given the parameterised bifunctor

$$\mathsf{B}_A = \mathsf{K}_{\mathbf{1}} + (\mathsf{K}_A \times \mathsf{Id} \times \mathsf{Id}) \text{ , s.t.}$$

$$\begin{aligned}
\mathsf{B}_A X &= \mathbf{1} + (A \times X \times X) \text{ , and} \\
\mathsf{B}_A f &= \mathrm{id}_{\mathbf{1}} + (\mathrm{id}_A \times f \times f).
\end{aligned}$$

Then the initial B_A-algebra $[empty, node]$ with constructors

$$\begin{aligned}
empty &: BTree_A \\
node_A &: A \times BTree_A \times BTree_A \to BTree_A,
\end{aligned}$$

defines the data type of binary node trees with A-elements $BTree_A = \mu \mathsf{B}_A$.

For an arbitrary B_A-algebra (C, φ) with $\varphi = [c, h]$ the catamorphism $f = (\![c, h]\!)_{\mathsf{B_A}}$ is the unique solution for the equation system stating its universal properties:

$$\begin{array}{lcl} f \circ \mathit{empty} & = & c \\ f \circ \mathit{node} & = & h \circ (\mathrm{id}_A \times f \times f). \end{array}$$

Take for example the function $\mathit{mirror} \colon \mathit{BTree}_A \to \mathit{BTree}_A$ which swaps left and right branches in the whole tree. Using the B_A-catamorphism it can be defined as

$$\mathit{mirror} = (\![\mathit{empty}, \mathit{node} \circ \mathit{swap}]\!)_{\mathsf{B_A}},$$

where *swap* simply exchanges the third and the second argument of a triple.

4.2.4. Type Functors

So far, only initial algebras of bifunctors which where parameterised in their first argument have been considered. Keeping the arguments of the bifunctor unfixed, it is possible to be more general and reason not only about a single initial functor algebra, but about a whole collection of them. Such a collection of initial algebras can be interpreted as a polymorphic data type, e.g. lists in general.

So let F be a binary functor, then $\mathrm{in}_A \colon \mathsf{F}(A, \mathsf{T}A) \to \mathsf{T}A$ is the collection of initial algebras induced by F, i.e. all types with the same structure induced by F, but parameterised in A.

Section 4.1.3 already pointed out earlier that there is a close relationship between polymorphic types and natural transformation. It should intuitively be clear that it must be possible to somehow *naturally* transform a list of characters into a list of say, natural numbers.

Assume a bifunctor F and two arbitrary F-algebras $\mathrm{in}_A \colon \mathsf{F}(A, \mathsf{T}A) \to \mathsf{T}A$ and $\mathrm{in}_B \colon \mathsf{F}(B, \mathsf{T}B) \to \mathsf{T}B$. For a function $f \colon A \to B$ and given an F_A-algebra φ, the functor $\mathsf{T}f \colon \mathsf{T}A \to \mathsf{T}B$ can be defined as an F_A-catamorphisms with $\mathsf{T}f = (\!|\varphi|\!)_{\mathsf{F_A}}$ s.t. the following diagram commutes:

$$\begin{tikzcd}
\mathsf{F}_A\mathsf{T}A \arrow[r, "\mathrm{in}_A"] \arrow[d, "\mathsf{F}_A\mathsf{T}f"] & \mathsf{T}A \arrow[d, "\mathsf{T}f = (\!|\varphi|\!)_{\mathsf{F_A}}"] \\
\mathsf{F}_A\mathsf{T}B \arrow[r, "\varphi"] & \mathsf{T}B
\end{tikzcd}$$

Making use of the fact that $\mathsf{T}B$ is the carrier of an initial algebra in_B, the mediating function φ can be defined in terms of in_B and F. In the following diagram the left part commutes because of equation bifunc-Comp, the right part commutes because:

$$\mathsf{T}f \circ \mathrm{in}_A = \mathrm{in}_B \circ \mathsf{F}(f, \mathrm{id}_{\mathsf{T}B}) \circ \mathsf{F}(\mathrm{id}_A, \mathsf{T}f) = \mathrm{in}_B \circ \mathsf{F}(f, \mathsf{T}f).$$

$$
\begin{tikzcd}
\mathsf{F}(A, \mathsf{T}A) \arrow[r, equal, "\mathrm{id}"] \arrow[d, "\mathsf{F}(\mathrm{id}_A, \mathsf{T}f)"] & \mathsf{F}(A, \mathsf{T}A) \arrow[r, "\mathrm{in}_A"] \arrow[d, "\mathsf{F}(f, \mathsf{T}f)"] & \mathsf{T}A \arrow[d, "\mathsf{T}f"] \\
\mathsf{F}(A, \mathsf{T}B) \arrow[r, "\mathsf{F}(f, \mathrm{id}_{\mathsf{T}B})"] & \mathsf{F}(B, \mathsf{T}B) \arrow[r, "\mathrm{in}_B"] & \mathsf{T}B
\end{tikzcd}
$$

Thus, given a bifunctor F and a function $f\colon A \to B$, there exists a transformation between any two F-algebras $\mathsf{T}f = (\!|\mathrm{in}_B \circ \mathsf{F}(f, \mathrm{id}_{\mathsf{T}B})|\!)_{\mathsf{F_A}}$ with $\mathsf{T}f\colon \mathsf{T}A \to \mathsf{T}B$. Further more, the construct T is indeed a functor, if it can bee proven that it preserves identities and composition.

Theorem 4.2.2. Given a bifunctor F which induces a collection of initial algebras $\mathrm{in}\colon \mathsf{F}(A, \mathsf{T}A) \to \mathsf{T}A$, the mapping T can be extended from objects to initial algebras s.t.

$$\mathsf{T}A = \mu\mathsf{F},$$

is an endofunctor

$$\mathsf{T}f = (\!|\mathrm{in}_B \circ \mathsf{F}(f, \mathrm{id})|\!)_{\mathsf{F}}. \tag{tyfunc-Def}$$

The functor T is called the **type functor** of F.

Proof. To show that the functor T preserves composition, it is necessary to show that there is a homomorphism from $\mathsf{T}f \circ (\!|\mathrm{in} \circ \mathsf{F}(g, \mathrm{id})|\!)_{\mathsf{F}}$ to $(\!|\mathrm{in} \circ \mathsf{F}(f \circ g, \mathrm{id})|\!)_{\mathsf{F}}$ and use cata-

FUSE.

$$
\begin{array}{ll}
 & \mathsf{T}f \circ \mathsf{T}g \\
= & \quad [\text{tyfunc-DEF}] \\
 & \mathsf{T}f \circ (\!|\text{in} \circ \mathsf{F}(g, \text{id})|\!)_{\mathsf{F}} \\
= & \quad [\text{cata-FUSE}] \left[
\begin{array}{ll}
 & \mathsf{T}f \circ \text{in} \circ \mathsf{F}(g, \text{id}) \\
= & \quad [\text{tyfunc-DEF}] \\
 & (\!|\text{in} \circ \mathsf{F}(f, \text{id})|\!)_{\mathsf{F}} \circ \text{in} \circ \mathsf{F}(g, \text{id}) \\
= & \quad [\text{cata-SELF}] \\
 & \text{in} \circ \mathsf{F}(f, \text{id}) \circ \mathsf{F}(\text{id}, (\!|\text{in} \circ \mathsf{F}(f, \text{id})|\!)_{\mathsf{F}}) \circ \mathsf{F}(g, \text{id}) \\
= & \quad [\text{tyfunc-DEF}] \\
 & \text{in} \circ \mathsf{F}(f, \text{id}) \circ \mathsf{F}(\text{id}, \mathsf{T}f) \circ \mathsf{F}(g, \text{id}) \\
= & \quad [\text{bifunc-COMP}] \\
 & \text{in} \circ \mathsf{F}(f \circ g, \text{id}) \circ \mathsf{F}(\text{id}, \mathsf{T}f)
\end{array}
\right. \\
 & (\!|\text{in} \circ \mathsf{F}(f \circ g, \text{id})|\!)_{\mathsf{F}} \\
= & \quad [\text{tyfunc-DEF}] \\
 & \mathsf{T}(f \circ g)
\end{array}
$$

□

Proof. The functor T preserves identities:

$$
\begin{array}{ll}
 & \mathsf{T}\text{id} \\
= & \quad [\text{tyfunc-DEF}] \\
 & (\!|\text{in} \circ \mathsf{F}(\text{id}, \text{id})|\!)_{\mathsf{F}} \\
= & \quad [\text{bifunc-ID}] \\
 & (\!|\text{in}|\!)_{\mathsf{F}} \\
= & \quad [\text{cata-SELF}] \\
 & \text{id}
\end{array}
$$

□

Having shown that T is indeed a functor, the initial algebra $\text{in}\colon \mathsf{F}(A, \mathsf{T}A) \to \mathsf{T}A$ is, according to Definition 4.1.18, a natural transformation.

$$\mathsf{T}f \circ \text{in} = \text{in} \circ \mathsf{F}(f, \text{id}) \circ \mathsf{F}(\text{id}, \mathsf{T}f) = \text{in} \circ \mathsf{F}(f, \mathsf{T}f).$$

Example 4.2.4
Behind this theoretical construct lies a technique every functional programmer uses in his daily work. Consider our previous introduced functor for lists $\mathsf{L}_A \colon \mathcal{S}et \to \mathcal{S}et$ which is parameterised in A and defined as $\mathsf{L}_A X = \mathbf{1} + (A \times X)$ and $\mathsf{L}_A f \colon \mathrm{id}_{\mathbf{1}} + (\mathrm{id}_A \times f)$. For the functor L_A our functor T is defined as

$$\mathsf{T}f = (\!|[\mathit{nil}, \mathit{cons}] \circ (\mathrm{id}_{\mathbf{1}} + (f \times \mathrm{id}_{\mathsf{L}_A}))|\!)_{\mathsf{L_A}}$$

After some simplification and renaming, things become self-evident:

$$\mathit{map}\, f = (\!|[\mathit{nil}, \mathit{cons} \circ (f \times \mathrm{id}_{\mathsf{L}_A})]|\!)_{\mathsf{L_A}},$$

This would implement the well-known `map`-function, which iterates over a list and applies the provided function on each element, in HASKELL as:

```
map f []     = []
map f (x:xs) = (f x) : (map f xs)
```

4.2.5. Paramorphisms

Catamorphisms, however, capture only structural recursive functions over an inductive data type. There are recursive functions, though, with an inductive type as source which are not a catamorphism. Consider a function `fac :: Int → Int` computing the factorial of a given natural number:

```
fac 0     = 1
fac (n+1) = (n+1) * fac(n)
```

It is obvious that the factorial of a natural number n does not only depend on the factorial of its predecessor $n - 1$, but also on n itself. So the recursive scheme does not follow a catamorphism, where the result depends on a constant part and the result of a recursive call only. The factorial function is the standard example of a primitive recursive function, which is more general than mere structural recursion.

However, it is possible to derive a catamorphic solution for the factorial by pairing the factorial and the number, allowing the catamorphisms to compute both in parallel:

$$\mathit{fac} = \mathit{fst} \circ (\!|[\lambda x.(1, 0), \lambda(f, n).((n + 1) * f, n + 1)]|\!)$$

Meertens [86] showed that this trick of pairing the intermediate result of a primitive recursive function and its input can be done for any inductive type.

Theorem 4.2.3. Given two arbitrary arrows $f \colon \mu\mathsf{F} \to A$ and $\varphi \colon \mathsf{F}(A \times \mu\mathsf{F}) \to A$, it holds that:

$$f \circ \mathrm{in}_{\mathsf{F}} = \varphi \circ \mathsf{F}\langle f, \mathrm{id}\rangle \quad \Longleftrightarrow \quad f = \mathit{fst} \circ (\!|\langle \varphi, \mathrm{in}_{\mathsf{F}} \circ \mathsf{F}(\mathit{snd})\rangle|\!)_{\mathsf{F}}.$$

The left side states, that f can be expressed by a catamorphism of the functor F. The fork of f and id simply builds the pair of applying a value to f and the value itself. The right side says that f can be expressed by composing the projection to the first element and a catamorphism.

Proof. This equivalence is proved showing the left and the right identity.

$\Longrightarrow$, assuming: $f \circ \mathrm{in}_{\mathsf{F}} = \varphi \circ \mathsf{F}\langle f, \mathrm{id}\rangle$

$$
\begin{array}{ll}
 & f \\
= & \qquad [\text{prod-}\textsc{Cancel}] \\
 & \mathit{fst} \circ \langle f, \mathrm{id}\rangle \\
= & \qquad [\text{Definition 4.2.6}] \left[\begin{array}{ll}
 & \langle f, \mathrm{id}\rangle \circ \mathrm{in} \\
= & \qquad [\text{prod-}\textsc{Fuse}] \\
 & \langle f \circ \mathrm{in}, \mathrm{in}\rangle \\
= & \qquad [\text{func-}\textsc{Id}] \\
 & \langle f \circ \mathrm{in}, \mathrm{in} \circ \mathsf{F}\mathrm{id}\rangle \\
= & \qquad [\text{prod-}\textsc{Cancel}] \\
 & \langle f \circ \mathrm{in}, \mathrm{in} \circ \mathsf{F}(\mathit{snd} \circ \langle f, \mathrm{id}\rangle)\rangle \\
= & \qquad [\text{assumption}] \\
 & \langle \varphi \circ \mathsf{F}\langle f, \mathrm{id}\rangle, \mathrm{in} \circ \mathsf{F}(\mathit{snd} \circ \langle f, \mathrm{id}\rangle)\rangle \\
= & \qquad [\text{func-}\textsc{Comp}] \\
 & \langle \varphi \circ \mathsf{F}\langle f, \mathrm{id}\rangle, \mathrm{in} \circ \mathsf{F}\mathit{snd} \circ \mathsf{F}\langle f, \mathrm{id}\rangle\rangle \\
= & \qquad [\text{prod-}\textsc{Fuse}] \\
 & \langle \varphi, \mathrm{in} \circ \mathsf{F}\mathit{snd}\rangle \circ \mathsf{F}\langle f, \mathrm{id}\rangle
\end{array}\right. \\
 & \mathit{fst} \circ (\!|\langle \varphi, \mathrm{in} \circ \mathsf{F}(\mathit{snd})\rangle|\!)
\end{array}
$$

$\Longleftarrow$, assuming: $f = \mathit{fst} \circ (\!|\langle \varphi, \mathrm{in} \circ \mathsf{F}(\mathit{snd})\rangle|\!)$

$$
\begin{array}{ll}
 & f \circ \mathrm{in} \\
= & \quad \text{[assumption]} \\
 & \mathit{fst} \circ (\!|\langle \varphi, \mathrm{in} \circ \mathsf{F}\mathit{snd}\rangle|\!) \circ \mathrm{in} \\
= & \quad \text{[Definition 4.2.6]} \\
 & \mathit{fst} \circ \langle \varphi, \mathrm{in} \circ \mathsf{F}\mathit{snd}\rangle \circ \mathsf{F}(\!|\langle \varphi, \mathrm{in} \circ \mathsf{F}\mathit{snd}\rangle|\!) \\
= & \quad \text{[prod-}\textsc{Cancel}\text{]} \\
 & \varphi \circ \mathsf{F}(\!|\langle \varphi, \mathrm{in} \circ \mathsf{F}\mathit{snd}\rangle|\!) \\
= & \quad \text{[prod-}\textsc{Fuse}\text{]} \\
 & \varphi \circ \mathsf{F}\langle \mathit{fst} \circ (\!|\langle \varphi, \mathrm{in} \circ \mathsf{F}\mathit{snd}\rangle|\!), \mathit{snd} \circ (\!|\langle \varphi, \mathrm{in} \circ \mathsf{F}\mathit{snd}\rangle|\!)\rangle \\
= & \quad \text{[assumption]} \\
 & \varphi \circ \mathsf{F}\langle f, \mathit{snd} \circ (\!|\langle \varphi, \mathrm{in} \circ \mathsf{F}\mathit{snd}\rangle|\!)\rangle \\
= & \quad \text{[cata-}\textsc{Fuse}\text{]} \left[\begin{array}{ll} & \mathit{snd} \circ \langle \varphi, \mathrm{in} \circ \mathsf{F}\mathit{snd}\rangle \\ = & \quad \text{[prod-}\textsc{Cancel}\text{]} \\ & \mathrm{in} \circ \mathsf{F}\mathit{snd} \end{array}\right. \\
 & \varphi \circ \mathsf{F}\langle f, (\!|\mathrm{id}|\!)\rangle \\
= & \quad \text{[cata-}\textsc{Refl}\text{]} \\
 & \varphi \circ \mathsf{F}\langle f, \mathrm{id}\rangle
\end{array}
$$

□

Let this function f which provides a primitive recursive scheme for arbitrary inductive data types formally be defined as follows.

Definition 4.2.7. Given an endofunctor $\mathsf{F}\colon \mathcal{C} \to \mathcal{C}$ and the initial algebra $\mu\mathbf{F} = (\mu\mathsf{F}, \mathrm{in}_{\mathsf{F}})$, for any F-algebra $\mathbf{A} = (A, \varphi)$, s.t. $\varphi\colon \mathsf{F}(A \times \mu\mathsf{F}) \to A$, the arrow $\langle\!|\, \varphi \,|\!\rangle\colon \mu\mathsf{F} \to A$ is defined as:

$$\langle\!|\, \varphi \,|\!\rangle = \mathit{fst} \circ (\!|\langle \varphi, \mathrm{in}_{\mathsf{F}} \circ \mathsf{F}(\mathit{snd})\rangle|\!)_{\mathsf{F}}$$

An arrow of the form $\langle\!|\, \varphi \,|\!\rangle$ is called **paramorphism**. This is due to Meertens [86], who derived it from the Greek preposition $\pi\alpha\rho\alpha$ meaning “near to”, “at the side of”, or “towards”.

Looking at the definition of the paramorphism in particular, it is apparent that Theorem 4.2.3 already states its universal property.

Corollary 4.2.2 *Given an endofunctor* $\mathsf{F}\colon \mathcal{C} \to \mathcal{C}$ *and the initial algebra* $\mu\mathbf{F} = (\mu\mathsf{F}, \mathrm{in}_\mathsf{F})$, *for any morphism* $\varphi\colon (A \times \mu\mathsf{F}) \to A$ *the paramorphism* $f = \langle\!| \varphi |\!\rangle_\mathsf{F}\colon \mu\mathsf{F} \to A$ *is the unique morphism s.t. the following diagram commutes:*

$$\begin{tikzcd}
\mathsf{F}\mu\mathsf{F} \arrow[r, "\mathrm{in}_\mathsf{F}"] \arrow[d, "\mathsf{F}\langle f, \mathrm{id}\rangle"'] & \mu\mathsf{F} \arrow[d, "f = \langle\!| \varphi |\!\rangle_\mathsf{F}"] \\
\mathsf{F}(A \times \mu\mathsf{F}) \arrow[r, "\varphi"] & A
\end{tikzcd}$$

From the diagram in Corollary 4.2.2 it is immediately apparent that the sole difference between a catamorphism and a paramorphism is the amount of information available to the mediating function φ. While a catamorphism simply gets a value of type $\mathsf{F}A$, a paramorphism has additionally a value of type $\mathsf{F}\mu\mathsf{F}$ available to combine the intermediate results.

As in previous sections, some examples of paramorphisms for different inductive data types will be given.

Example 4.2.5
Consider the type of natural numbers *Nat*, as e.g. defined in Example 4.2.1. Given an arbitrary algebra $\varphi = [c, h]$ with $c\colon \mathbf{1} \to A$ and $h\colon A \times \mathit{Nat} \to A$, the paramorphism $\langle\!| [c, h] |\!\rangle\colon \mathit{Nat} \to A$ is the unique morphism solving the following equation system:

$$\begin{array}{rcl} f \circ \mathit{zero} & = & c \\ f \circ \mathit{succ} & = & h \circ \langle f, \mathrm{id}\rangle. \end{array}$$

This primitive recursive scheme corresponds to the factorial function which can be defined as a paramorphism as follows.

$$\mathit{fac} = \langle\!| [\mathit{succ} \circ \mathit{zero}, \lambda(f, n).\mathit{mult}(\mathit{succ} n, f)] |\!\rangle_\mathsf{N},$$

where $\mathit{mult}\colon \mathit{Nat} \times \mathit{Nat} \to \mathit{Nat}$ is the multiplication of two natural numbers as defined in Example 4.2.1.

Example 4.2.6
Similarly, referring to the functor L_E for lists over elements of type E from Example 4.2.2, given an arbitrary algebra $\varphi = [c, h]$ with $c\colon \mathbf{1} \to A$ and $h\colon E \times A \times \mathit{List}_E \to A$, the paramorphism $\langle\!| [c, h] |\!\rangle\colon \mathit{List}_E \to A$ is the unique morphism solving the following equation system:

$$\begin{array}{rcl} f \circ \mathit{nil} & = & c \\ f \circ \mathit{cons}(x, \mathit{xs}) & = & h(x, f(\mathit{xs}), \mathit{xs}). \end{array}$$

The typical example of a paramorphism on lists is $\mathit{tails}\colon \mathit{List}_E \to \mathit{List}_{\mathit{List}_E}$, returning the tails of all sublists of a list. The paramorphic definition is:

$$\mathit{tails} = \langle\!| [\mathit{cons}(\mathit{nil}, \mathit{nil}), \lambda(x, \mathit{ys}, \mathit{xs}).\mathit{cons}(\mathit{cons}(x, \mathit{xs}), \mathit{ys})] |\!\rangle_{\mathsf{L}_\mathsf{E}}$$

Example 4.2.7
As described in Example 4.2.3, the initial algebra of the functor B_E is the data type of binary trees $BTree_E$, containing elements of type E in its nodes. Given an arbitrary algebra $\varphi = [c, h]$ with $c\colon \mathbf{1} \to A$ and $h\colon E \times A \times BTree_E \times A \times BTree_E \to A$, the paramorphism $\langle\!|\, [c, h] \,|\!\rangle\colon Tree_E \to A$ is the unique morphisms solving the following equation system:

$$\begin{array}{lcl} f \circ empty & = & c \\ f \circ node(e, l, r) & = & h(e, f(l), l, f(r), r). \end{array}$$

The function $subtrees\colon BTree_E \to List_{BTree_E}$ returning a list of all subtrees for a given input tree is defined as the paramorphism:

$$\begin{array}{lcl} subtrees & = & \langle\!|\, [c, h] \,|\!\rangle_{\mathsf{B_E}} \\ c & = & cons(empty, nil) \\ h & = & \lambda(e, fl, f, fr, r).cons(node(e, l, r), fl +\!\!+ fr) \end{array}$$

The function $+\!\!+$ denotes list concatenation.

5. The Igor II Algorithm

IGOR II [65, 69, 121] is an analytical, functional inductive programming system. It combines several methodologies. On the one hand, it is based on the pure analytical procedure of *recurrence detection*. This technique was also used by its predecessor IGOR I [68, 69, 98, 119], which itself was heavily inspired by Summers's THESYS system [128]. On the other hand, it integrates a *search in the space of rules or unfinished programs*. This allows using background knowledge during the synthesis process and overcomes its predecessors restrictions to be fixed to a specific program scheme. From the beginning on, all IGOR versions aimed for extending the expressiveness of analytical systems without being hampered by the restrictions of fixed program schemes and without falling back to generate-and-test.

This chapter recapitulates the basic IGOR II algorithm which serves as bedrock for extensions and improvements presented later on. Although this chapter is based on the thesis by Kitzelmann [66] and will borrow many terms and concepts, not all features described there have been re-implemented in the HASKELL version IGOR II_H, or they may differ from the implementation described in [66]. It is meant to be the starting point of the formalisation of extensions as described in Chapter 6.

Section 5.1 defines the problem specification used by IGOR II, in Section 5.2 IGOR II's main algorithm is described, whereas Section 5.3 formally defines the refinement operators used by the system. Section 5.4 shall give a very intuitive understanding of IGOR II by demonstrating an exemplary synthesis process as a hand simulation.

5.1. Definition of a Problem Specification

IGOR II synthesises functions from incomplete specifications given as input/output (IO) examples which describe the behaviour of these functions on a part of their domain only. These IO examples are given as a CS which incompletely specifies the problem.

Definition 5.1.1. A **specification** is an orthogonal CS over a signature Σ. As defined in Definition 3.3.1, $\mathcal{D}$ and $\mathcal{C}$ are the sets of defined function symbols and constructor symbols, respectively. Each rule r in CS is called **(IO) example** or **(IO) equation**. If the LHS of r is ground, we call r ground. Otherwise it is a non-ground example or an **IO pattern**. The LHS of any $r \in$ CS is called **input**, the RHS is the **output**.

In general IGOR II allows both, ground and non-ground IO examples. The intuitive semantics of non-ground examples are that one non-ground example describes all matching ground examples. Listing 5.1 shows examples of n ground examples for computing the last element of a two-element list. Listing 5.2 shows the equivalent non-ground example. Note that terms starting with small caps are variables in Haskell.

Listing 5.1: n ground examples for `last` on a two-element list.

```
last [1,1] = 1
last [1,2] = 2
last [1,3] = 3
...
last [2,1] = 1
last [2,2] = 2
...
last [3,1] = 1
...
```

Listing 5.2: A single non-ground example for `last` on a two-element list.

```
last [a,b] = b
```

Igor II gets two specifications as input: the **target specification** Φ and the **background specification** or **background knowledge** B. The defined function symbols $\mathcal{D}_\Phi$ of Φ are called **target functions** or **targets**.

In the Haskell re-implementation Igor $\mathrm{II_H}$, as well as in its extension Igor II^+, additionally type information Θ is required. The type information Θ represents information about the types of all terms, used data types and their constructors, type classes, and type class instances (cf. Appendix A).

The result of a synthesis is a CS P, which redefines all target functions in Φ, maybe using functions from B. "Redefines" means here that P may consist of different rules not in Φ and B, but given an input it computes the same output as Φ, if Φ is defined on this input. If this holds, we say P *is correct w.r.t.* Φ.

Definition 5.1.2. A CS P is **correct w.r.t.** a CS Φ, if for any term s the following holds:

$$s \to_\Phi t \Rightarrow s \twoheadrightarrow_P t$$

This definition, however, is only applicable if P is indeed a terminating and especially a *closed* CS. If the CS P contains open, i.e. unfinished rules, there is no properly defined rewrite relation $\to_P$.

Definition 5.1.3. A rule r is an **open rule** or an **unfinished**, if $\mathcal{V}ar\,(rhs(r)) \setminus \mathcal{V}ar\,(lhs(r)) \neq \emptyset$. A CS which contains at least one open rule is called an **open CS** or an **unfinished CS**.

Intuitively we say a CS P, that possibly contains unfinished rules, is *extensional correct* if it is possible, after one rewrite step in P, to rewrite an input correctly to its specified output only using the specification Φ now.

Definition 5.1.4. Given a specification Φ, we say a (possibly unfinished) **rule** $f(\boldsymbol{p}) = t$ is **extensionally correct** w.r.t. Φ, if and only if for any rule $(f(\boldsymbol{i}) = o) \in \Phi$ s.t. $f(\boldsymbol{i}) \equiv f(\boldsymbol{p})^\sigma$ for a substitution σ with $\mathcal{D}om\,(\sigma) = \mathcal{V}ar\,(f(\boldsymbol{p}))$, there exists a substitution θ with $\mathcal{D}om\,(\theta) = \mathcal{V}ar\,(t) \setminus \mathcal{V}ar\,(f(\boldsymbol{p}))$, s.t. $t^{\sigma\theta} \rightarrow_\Phi o$.

A (candidate) CS is said to be *extensional correct* w.r.t. Φ, if all its rules are extensional correct and each input of Φ matches some LHS of the candidate CS.

Definition 5.1.5. Given a specification Φ, a **CS** P is **extensional correct** w.r.t. Φ, if and only if:

- Each rule in P is extensional correct w.r.t. Φ, and
- each LHS of Φ matches a LHS of P.

The task of constructing an orthogonal and terminating CS P given a specification of the target Φ and some background knowledge is defined as the *induction problem.*

Definition 5.1.6. Given a target specification Φ and a background specification B with disjoint sets of defined function symbols $\mathcal{D}_\Phi \cap \mathcal{D}_B = \emptyset$, the **induction problem** is to find a CS P with defined functions $\mathcal{D}_P$, s.t. :

(i) P is orthogonal,

(ii) $\mathcal{D}_P \cap \mathcal{D}_B = \emptyset$, and

(iii) $P \cup B$ is correct w.r.t. Φ.

Such a CS P complying to the restrictions defined above is called a **solution** (of the induction problem).

Example 5.1.1
Consider a function `lasts` which returns all last elements of a list of lists. Listings 5.3, 5.4, and 5.5 are its target specification Φ, its background specification B, and its type information Θ, respectively. Listing 5.6 shows a solution of this induction problem. Note that $\mathcal{D}_P \supset \mathcal{D}_\Phi$, i.e. additional functions have been introduced.

Listing 5.3: Target specification Φ for `lasts`

```
lasts []                  = []
lasts [[a]]               = [a]
lasts [[a,b]]             = [b]
lasts [[a,b,c]]           = [c]
lasts [[b],[a]]           = [b,a]
lasts [[c],[a,b]]         = [c,b]
lasts [[c,d],[b]]         = [d,b]
lasts [[a,b],[c,d]]       = [b,d]
lasts [[c],[d,e],[f]]     = [c,e,f]
lasts [[c,d],[e,f],[g]]   = [d,f,g]
```

Listing 5.4: Background specification B for `lasts`

```
last       (a:[]) = a
last     (a:b:[]) = b
last   (a:b:c:[]) = c
```

Listing 5.5: Data type information Θ for `lasts`

```
data [α] = [] | α : [α]

lasts :: [[α]] → [α]
last  ::   [α] → [α]
```

Listing 5.6: Solution CS P for `lasts`

```
lasts               []  = []
lasts i@((x0:x1):x2)  = fun1 i : fun2 i
fun1     ((x0:x1):x2) =  last (x0:x1)
fun2     ((x0:x1):x2) =  lasts x2
```

5.2. Main Loop and χ_{init}

The search of a solution CS itself is organised as a uniform-cost search. Starting from a root node, the search tree is traversed by expanding at each step the node with minimal costs. Each node in the tree represents an hypothesis as a probably unfinished CS, where unfinished means that at least one rule in CS has a variable in its RHS which does not occur on its LHS. All hypotheses are extensional correct w.r.t. the given specification Φ.

The search starts with the initial CS P, which contains the least general generalisation of all rules of a given target function name in the target specification. It is computed by the function `initialCandidate` (Algorithm 1). Let $\Phi(f)$ denote the subset of Φ containing all rules which head is f, i.e.

$$\Phi(f) := \{\rho \mid \mathit{head}(\rho) = f, \rho \leftarrow \Phi\}.$$

Definition 5.2.1. Given a target specification Φ with defined functions $\mathcal{D}_\Phi$, the **initial rule operator** χ_{init} is defined as

$$\chi_{\text{init}}(\Phi(f)) := \{\mathit{lgg}(\Phi(f)) \mid f \in \mathcal{D}_\Phi\}.$$

The application of χ_{init} to a specification Φ without specifying a function name is defined as

$$\chi_{\text{init}}(\Phi) := \{\mathit{lgg}(\Phi(f)) \mid f \leftarrow \mathcal{D}_\Phi\}.$$

Given a set of specifications $\{\Phi_1, \dots, \Phi_n\}$ we generalise χ_{init} to χ_{INIT}, defined as

$$\chi_{\text{INIT}}(\Phi_1, \dots, \Phi_n) := \{\chi_{\text{init}}(\Phi_1), \dots, \chi_{\text{init}}(\Phi_n)\}.$$

Remark: Technically speaking is the result of χ_{init} not a left-linear CS, because computing the least general generalisation may introduce one variable multiple times if the respective terms subsumed by this variable are identical. However, it won't hurt if we silently ignore this and agree to rename all variables at the end to restore left-linearity.

Algorithm 1: `initialCandidate`(Φ)

---

input : a target specification Φ
output : an initial CS P containing one rule for each defined function

$P \leftarrow \emptyset$
foreach $f \in \mathcal{D}_\Phi$ **do insert** $lgg(\Phi(f))$ **into** P
return P

In each iteration of the algorithm, all CS with minimal costs are selected. If one of them is closed, it is returned as solution, which is correct w.r.t. Φ. Otherwise, one of them is chosen and one of its open rules is selected for development. We call this CS and this rule **candidate constructor system** and **candidate rule**, respectively. All other hypotheses are left unchanged.

The **costs of a candidate CS** are the number of maximal general patterns. The motivation behind this is that a maximal general pattern, i.e. a pattern that does not match any other pattern in the CS, can be considered as a case distinction. Thus, according to Occam's razor, it is desirable to have as few cases as possible. Otherwise, the algorithm would tend to prefer the most specific patterns, and thus reproduce the IO examples. As a machine learning algorithm, preferring maximal general patterns is Igor II's *inductive bias.*

To break ties the minimal number of open rules, the minimal number of free variables, and the minimal number of total rules are preferred successively.

Several operators are applied to the candidate rule, replacing it by one or more **successor rules**. Due to the fact that the operators are not applied exclusively, but quasi in parallel, i.e. each operator is applied to the same candidate CS, multiple **successor CSs** are generated from one candidate CS. All successor CS are required to remain extensional correct w.r.t. the given specification Φ.

Algorithm 2 describes this outer main loop. Since some operators introduce new defined functions, we will attach a specification Φ to each candidate program P. This facilitates writing the algorithm.

5.3. Synthesis Operators

After selecting a candidate CS, four operators are applied to an open candidate rule. Each of them generating one or more successor rule sets, eventually with new corresponding specifications.

Given a candidate CS P with specification Φ and selected candidate rule $\rho\colon f(\boldsymbol{p}) \to t$, the **specification subset covered by** ρ, or the IO examples covered by ρ, is the set

---

Algorithm 2: Igor II main loop.

---

input : a target specification Φ
input : a background specification B
input : type information Θ
require : $\mathcal{D}_\Phi \cap \mathcal{D}_B = \emptyset$
output : (maximal general) CS P
ensure : P satisfies Definition 5.1.6

$\langle P, \Phi \rangle \leftarrow$ `initialCandidate` (Φ)
$\mathcal{P} \leftarrow \{\langle P, \Phi \rangle\}$
while $\langle P, \Phi \rangle$ *open* **do**
  $r \leftarrow$ open rule from P
  $\mathcal{S} \leftarrow$ `successorRuleSets` (r, Φ, B); `// cf. Alg. 3`
  remove $\langle P, \Phi \rangle$ **from** $\mathcal{P}$
  foreach $\langle S, \phi_S \rangle \in \mathcal{S}$ **do**
    $P' \leftarrow (P \backslash \{r\}) \cup S$
    insert $\langle P', \Phi \cup \phi_S \rangle$ **into** $\mathcal{P}$
  end
  $\langle P, \Phi \rangle \leftarrow$ a maximal general CS (with corresponding specification) in $\mathcal{P}$
end
return P

$\Phi(\rho)$, defined by

$$\Phi(\rho) := \{\varphi \mid \varphi \leftarrow \Phi, lhs(\varphi) \preceq f(\boldsymbol{p})\}.$$

It contains all IO examples whose LHS match the LHS of the open rule ρ. The rule ρ is called **covering rule**, the set $\Phi(\rho)$ **covered rules** or covered rule set. The Algorithm 5.3 describes how all operators (χ_{split},χ_{subfn},χ_{direct}, and χ_{call}) are applied to an open rule "in parallel". This means that given one candidate rule, successors w.r.t. all operators are computed. Note that χ_{direct} is a special case of χ_{call}, i.e. χ_{call} is only applied if χ_{direct} yields no result.

5.3.1. Split operator χ_{split}

The operator χ_{split} splits a rule by pattern refinement, introducing a case distinction. Since in functional languages cases are usually modelled by rules with the same head, but different patterns, χ_{split} induces a partitioning on the set of rules covered by the candidate rule. Since a covering rule ρ is the LGG of all its covered rules, a variable at position p on the LHS of ρ means that there are at least two rules in $\Phi(\rho)$ which differ at this position p. We call p a *pivot position*.

Definition 5.3.1. Given a rule ρ and an arbitrary covered rule $\varphi \in \Phi(\rho)$, a position $p \in \mathcal{P}os\,(lhs(\rho)))$ is a **pivot position** if $\rho|_p \in \mathcal{V}ar\,(\rho)$, i.e. the subterm of ρ at position

Algorithm 3: `successorRuleSets`(r, Φ, B)

input : an open rule r
input : a target specification Φ
input : a background specification B

output : a set of successor rules with corresponding specification

$\mathcal{S}_1 \leftarrow \chi_{\text{split}}(r, \Phi)$
$\mathcal{S}_2 \leftarrow \chi_{\text{subfn}}(r, \Phi)$
$\mathcal{S}_3 \leftarrow \chi_{\text{direct}}(r, \Phi, B)$
if $\mathcal{S}_3 = \emptyset$ **then** $\mathcal{S}_3 \leftarrow \chi_{\text{call}}(r, \Phi, B)$
return $\mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S}_3$

p is a *variable*, and *root*$(\varphi|_p) \in \mathcal{C}_\Phi$, i.e. the subterm of ρ at position p has a *constructor* at the root position.

An equivalence relation $\sim_p$ between any two terms φ and φ' in $\Phi(\rho)$ is defined upon a pivot position p by

$$\varphi \sim_p \varphi' \quad \Longleftrightarrow \quad \varphi|_p \equiv \varphi'|_p.$$

The expression $\Phi(\rho)/\sim_p$ denotes the quotient set of $\Phi(\rho)$ w.r.t. $\sim_p$. The operator χ_{split} computes the quotient set of $\Phi(\rho)$ for all pivot positions of ρ and applies χ_{INIT} to each of them. This is described in Algorithm 4.

Definition 5.3.2. Given a candidate CS $\langle P, \Phi\rangle$, and a candidate rule ρ, the **splitting operator** $\chi_{\text{split}}(\rho, \Phi)$ is defined as:

$$\chi_{\text{split}}(\rho, \Phi) := \{\chi_{\text{INIT}}(\langle \Phi(\rho)/\sim_p \rangle) \mid p \text{ is a pivot position of } \Phi(\rho)\}.$$

Example 5.3.1
Listing 5.7 shows some IO examples of a function `reverse` on lists.

Listing 5.7: IO examples of `reverse`

```
reverse      []        =          []
reverse    (d:     []) =       (d:[])
reverse    (c:   d:[]) =     (d:c:[])
reverse    (b:c:d:[])  =   (d:c:b:[])
```

These examples are covered by the rule:

```
reverse x = y
```

The term `x` at the root position of the argument is on a pivot position, because `x` is a variable, and in each covered examples there is a constructor, either `(:)` or `[]`, at the pivot position[1]. Thus, the first IO example would be in one equivalent class of the quotient set, all other examples in another, yielding the following new initial rules computed by χ_{init}:

[1]Note that the cons-constructor `(:)` is, as usually, written in infix!

```
reverse      [] = []
reverse  (x:xs) = (y:ys)
```

Algorithm 4: The splitting operator χ_{split}

input : an open rule ρ
input : a specification Φ
output : A finite set $\mathcal{S} = \{\langle S_1, \emptyset\rangle, \ldots, \langle S_n, \emptyset\rangle\}$ of successor rule sets and empty specifications

$\mathcal{S} \leftarrow \emptyset$
foreach $p \in \mathcal{P}os\,(lhs(\rho))$ **do**
  if $\forall \varphi \in \Phi(\rho).\quad \rho|_p \in \mathcal{V}ar\,(\rho) \wedge root(\varphi|_p) \in \mathcal{C}_\Phi$ **then**
    $S \leftarrow \{\chi_{\text{INIT}}(\phi) \mid \phi \leftarrow (\Phi(\rho)/\sim_p)\}$
    insert $\langle S, \emptyset\rangle$ **into** $\mathcal{S}$
  end
end
return $\mathcal{S}$

5.3.2. Subfunction operator χ_{subfn}

Recall that in each iteration of the IGOR II algorithm, the aim is to close an open rule, i.e. remove unbound variables. This may be done by replacing each subterm on the RHS of a covering rule which contains an open variable, by a call to some function. This function f is unknown yet, though. However, we can abduce IO examples for it by analysing the terms of the covering rule's covered IO examples. Solving the new induction problem for the function f is done in succeeding iterations using the given examples.

Example 5.3.2
Look again at the `reverse`-examples in Listing 5.7 of the previous Example 5.3.1 except the first one. They are all in the second subset of the quotient set induced by the described pivot position, and thus covered by `reverse (x:xs) = y:ys`. This rule is unfinished due to two unbound variables on the RHS. Now, we close them by replacing the variables with two auxiliary functions.

```
reverse (x:xs) = fun1 (x:xs) : fun2 (x:xs)
```

Solving $\texttt{fun}_1$ and $\texttt{fun}_2$ are treated as new induction problems, but hereto we need IO examples for these functions. Given the same input as `reverse`, $\texttt{fun}_1$ needs to compute the subterm at the position of `y` in the accordant outputs of `reverse`. This leads to following example equations:

```
fun1        (d:[]) = d
fun1      (c:d:[]) = d
fun1    (b:c:d:[]) = d
fun1  (a:b:c:d:[]) = d
```

Similar, we can obtain the relevant examples for $\texttt{fun}_2$:

```
fun₂         (d:[]) =         []
fun₂       (c:d:[]) =     (c:[])
fun₂     (b:c:d:[]) =   (c:b:[])
fun₂ (a:b:c:d:[]) = (c:b:a:[])
```

Finally, we have to add the initial rules for $\texttt{fun}_1$ and $\texttt{fun}_2$ to our currently processed hypotheses:

```
fun₁ (x:xs) = d
fun₂ (x:xs) = ys
```

We formally define the subfunction operator as follows in Definition 5.3.3, Algorithm 5 computes χ_{subfn}.

Definition 5.3.3. Given a candidate CS $\langle P, \Phi\rangle$, a background specification B s.t. $\mathcal{D}_\Phi \cap \mathcal{D}_B = \emptyset$, and a candidate rule $\rho\colon f(\boldsymbol{p}) = c(t_1, \ldots, t_n)$, where $c \in \mathcal{C}_\Phi$. Let $I = \{i \in [1..n] \mid \mathcal{V}ar\,(t_i) \nsubseteq \mathcal{V}ar\,(\boldsymbol{p})\}$ be the set of all positions on the RHS of the candidate rule ρ that contain unbound variables.

Let further be $\{g_i\}_{i\in I}$ a set of new function symbols neither occurring in P, nor in B, i.e. $g_i \notin \mathcal{D}_{P\cup B} \cup \mathcal{C}_{P\cup B} \cup \mathcal{X} = \emptyset$. Then the **subfunction operator** χ_{subfn} is defined as

$$\chi_{\text{subfn}}(\rho, \Phi, B) := \{\langle\{f(\boldsymbol{p}) = c(t'_1, \ldots, t'_n)\} \cup P_S, \phi_S\rangle\},$$

where

- for all $j \in [1..n], t'_j = \begin{cases} g_j(\boldsymbol{p}) & \text{if } j \in I \\ t_j & \text{otherwise,} \end{cases}$
- $\phi_S = \{g_j(\boldsymbol{i}) = o_j \mid j \leftarrow I, (f(\boldsymbol{i}) = c(o_1, \ldots, o_n)) \leftarrow \Phi(\rho)\}$, and
- P_S is an initial candidate CS of ϕ_S.

If there is no constructor symbol at the root position of the RHS of the candidate rule ρ, χ_{subfn} returns the empty set $\emptyset$.

5.3.3. Function call introduction with χ_{direct} and χ_{call}

So far, none of the operators introduced used background knowledge. The operator χ_{split} does not introduce new functions at all, and χ_{subfn} virtually reuses existing IO examples. Instead of replacing only subterms with unbound variables, the function call operator completely discards a RHS of an open rule and tries to replace it by a call to a defined function, i.e. a previously introduced subfunction, a function from the background knowledge, or a recursive call to the target function itself.

We distinguish two possibilities: A direct call, and a call via subfunction. Given a rule $\rho\colon f(\boldsymbol{p}) = t$, the direct call produces a rule of the form $\rho\colon f(\boldsymbol{p}) = f'(\boldsymbol{p'})$. It is direct, because the arguments $\boldsymbol{p'}$ of the call to f' are directly constructed from the pattern $\boldsymbol{p}$.

Algorithm 5: The subfunction operator χ_{subfn}

input : an open rule $\rho\colon f(\boldsymbol{p}) = t$
input : a target specification Φ
input : a background specification B
require : $\mathcal{D}_\Phi \cup \mathcal{D}_B = \emptyset$
output : Either the empty set $\emptyset$, or the set $\{\langle S, \phi_S\rangle\}$ containing a pair of a successor rule set with according new specification subset

switch t **do**
  case $c(t_1, \ldots, t_n)$
    $\phi_S \leftarrow \emptyset$
    foreach $j \in [1..n]$ **do**
      if $\mathcal{V}ar\,(t_j) \not\subseteq \mathcal{V}ar\,(\boldsymbol{p})$ **then**
        $g_j \leftarrow$ a new defined symbol, s.t. $g_j \notin (\mathcal{D}_{P\cup B} \cup \mathcal{C}_{P\cup B} \cup \mathcal{X})$
        $\phi_S \leftarrow \phi_S \cup \{g_j(\boldsymbol{i}) = o_j \mid$
          $j \leftarrow I,$
          $(f(\boldsymbol{i}) = c(o_1, \ldots, o_n)) \leftarrow \Phi(\rho)\}$
        $t'_j \leftarrow g_j(\boldsymbol{p})$
      else
        $t'_j \leftarrow t_j$
      end
      $P_S \leftarrow$ `initialCandidate(`ϕ_S`)` **return** $\{\langle\{f(\boldsymbol{p}) = c(t'_1, \ldots, t'_n)\} \cup P_S, \phi_S\rangle\}$
    end
  otherwise
    return $\emptyset$
  end
end

If the call is via a subfunction a rule $\rho\colon f(\boldsymbol{p}) = f'(g_1(\boldsymbol{p}), \ldots, g_n(\boldsymbol{p}))$ is produced, where for each argument i of f' a new subfunction $g_i(\boldsymbol{p})$ is constructed which takes the original input $\boldsymbol{p}$ and produces the specific input for the i^{th} argument.

In both cases, it is required that each RHS in $\Phi(\rho)$ matches a RHS of f' in such a way that the LHSS of $\Phi(\rho)$ are mapped appropriately instantiated to the according LHSS of f'.

When we allow calls to any previously defined functions, we have to take care not to destroy the properties of our solution CS postulated in Definition 5.1.6. It demands completeness w.r.t. a given target specification, which includes that the CS is terminating. Calls to previously defined functions exactly jeopardise termination, because they allow recursion, directly or via subfunctions, or even mutual recursion.

In Definition 3.3.7 we have already stated the properties for a terminating CS, i.e. that there exists a well-founded reduction-order (Def. 3.3.8) that is compatible (Def. 3.3.9) with the CS.

In practise, this means that given a candidate rule $f(\boldsymbol{p}) = t$, the operator χ_{direct} replaces its RHS t only by those calls $f'(\boldsymbol{p'})$ which assure that if $\boldsymbol{p}^\sigma$ is an input for f, then $\boldsymbol{p'}^\sigma$ is an input for f' for some substitution σ and $f(\boldsymbol{p})^\sigma > f'(\boldsymbol{p'})^\sigma$ w.r.t. a reduction order $>$.

Similarly, χ_{call} replaces the open RHS t only by those function calls $f'(g_1(\boldsymbol{p}), \ldots, g_n(\boldsymbol{p}))$, s.t. if $\boldsymbol{p}^\sigma$ is an input for f, then $\boldsymbol{p'}^\sigma$ is the corresponding output of f, if and only if for some substitution σ $\boldsymbol{p}^\sigma$ is the input for g_i with corresponding output o_i and $f(\boldsymbol{p})^\sigma > f'(o_1, \ldots, o_n)^\sigma$ w.r.t. a reduction order $>$.

However, recursion can occur indirectly in form of mutual recursion, too. Therefore, the conditions above apply to any call to a function $f' \in \mathcal{D}_\Phi$ of specification Φ of the candidate CS. The only exception is a call to a function $f \in \mathcal{D}_B$, because we assume only terminating functions in the background knowledge.

Given a fixed reduction order for all rules in a CS, the condition above together with extensional correctness (Def. 5.1.5) assures termination of the inputs specified. The **default reduction order** of IGOR II is the order $(s_1, \ldots, s_n) > (t_1, \ldots, t_n)$, if and only if $|\mathcal{P}os\,(s_1)| > |\mathcal{P}os\,(t_1)|$, or if $|\mathcal{P}os\,(s_1)| = |\mathcal{P}os\,(t_1)|$ then $(s_2, \ldots, s_n) > (t_2, \ldots, t_n)$. Similarly, $(s_1, \ldots, s_n) < (t_1, \ldots, t_n)$, if and only if $|\mathcal{P}os\,(s_1)| < |\mathcal{P}os\,(t_1)|$, or if $|\mathcal{P}os\,(s_1)| = |\mathcal{P}os\,(t_1)|$ then $(s_2, \ldots, s_n) < (t_2, \ldots, t_n)$. This is called the **argument-wise** order.

An alternative, the so called **linear** reduction order is implemented for $s = (s_1, \ldots, s_n)$ and $t = (t_1, \ldots, t_n)$ $s > t$, if and only if $\sum_i^n |\mathcal{P}os\,(s)| > \sum_i^n |\mathcal{P}os\,(t)|$.

Definition 5.3.4. Let $\langle P, \Phi \rangle$ be a candidate CS with corresponding specification, let further be B a background specification, s.t. $\mathcal{D}_\Phi \cap \mathcal{D}_B = \emptyset$, $\mathcal{C} := \mathcal{C}_P \cup \mathcal{C}_B$, $\rho\colon f(\boldsymbol{p}) = t$ is a candidate rule in P, and $\Phi(\rho)$ the examples covered by ρ.

The **direct-call operator** $\chi_{\text{direct}}(\rho, \Phi, B)$ yields a (possibly empty) set of singleton rule sets and empty specifications. Each successor rule in a singleton rule set has the form $f(\boldsymbol{p}) = f'(\boldsymbol{p'})$, where $f' \in \mathcal{D}_{\Phi \cup B}$ and $\boldsymbol{p} \in \mathcal{T}_\mathcal{C}(\mathcal{V}ar\,(\boldsymbol{p}))$. This set is uniquely defined as:

$$\langle \{f(\boldsymbol{p}) = f'(\boldsymbol{p'})\}, \emptyset \rangle \in \chi_{\text{direct}}(\rho, \Phi, B),$$

if and only if for each candidate rule $(f(\boldsymbol{i}) = o) \in \Phi(\rho)$ there is a rule $(f'(\boldsymbol{i'}) = o') \in \Phi \cup B$, s.t. the following conditions are satisfied. Let σ be a substitution which matches a candidate rule $\rho\colon f(\boldsymbol{p}) = t$ with the according specification rule $(f(\boldsymbol{p}) = o) \in \Phi(\rho)$, i.e. $f(\boldsymbol{i}) = o \equiv f(\boldsymbol{p})^\sigma = t^\sigma$:

(i) $t^\sigma \equiv o'^\tau$ for a substitution τ, with $\mathcal{D}om\,(\tau) = \mathcal{V}ar\,(o')$.

(ii) $f'(\boldsymbol{p'}) \preceq f'(\boldsymbol{i'})^\tau$ and $f'(\boldsymbol{p'})^\sigma \equiv f'(\boldsymbol{i'})^{\tau\theta}$ for any substitution θ with $\mathcal{D}om\,(\theta) = \mathcal{V}ar\,(f'(\boldsymbol{i'})) \setminus \mathcal{V}ar\,(o')$.

(iii) If $f' \in \mathcal{D}_\Phi$, then $f(\boldsymbol{p})^\sigma > f'(\boldsymbol{p'})^\sigma$ w.r.t. some reduction order $>$.

Condition (i) states that, given the new rule $\rho'\colon f(\boldsymbol{p}) = f'(\boldsymbol{p'})$, for any example $(f(\boldsymbol{p})^\sigma = t^\sigma) \in \Phi(\rho)$ covered by the old rule ρ with substitution σ, $f'(\boldsymbol{p'})$ must reduce to t^σ. This can be checked extensionally on $\Phi(f')$: There must be a rule $f'(\boldsymbol{i'}) = o'$, s.t. its output subsumes t^σ with substitution τ.

Condition (ii) states that, if condition (i) holds, the argument $\boldsymbol{p'}$ of the call matches indeed the desired inputs for f', i.e. those inputs of f' which compute the required outputs o'^τ, i.e. those indeed covered by ρ. The additional substitution θ instantiates those variables occurring in o' which are not affected by τ. This occurs if $f'(\boldsymbol{i'}) = o'$ is not ground. Actually, these variables may be instantiated arbitrarily since the output is independent of them; at least as long as $f' \notin \mathcal{D}_B$. Examples of a direct call can be found in a hand simulation in the next section (5.4.3 and 5.4.4).

Algorithm 6: The direct call operator χ_{direct}

input : candidate rule ρ: $f(\boldsymbol{p}) = t$
input : target specification Φ
input : background specification B
output : a (possibly empty) set $\mathcal{S} = \{\langle\{\rho'_1\}, \emptyset\rangle, \ldots, \langle\{\rho'_n\}, \emptyset\rangle\}$ of pairs of singleton successor rule sets and empty specifications.

1 $\mathcal{S} \leftarrow \emptyset$
2 **foreach** $f' \in \mathcal{D}_{\Phi\cup B}$ **do**
3 $\quad r \leftarrow \min_{r\in\Phi(\rho)} |\mathcal{P}os\,(lhs(r))|$, i.e. covered rule with the smallest LHS
4 $\quad \sigma \leftarrow$ substitution s.t. $f(\boldsymbol{p})^\sigma \equiv lhs(r)$
5 $\quad$ **foreach** $(f'(\boldsymbol{i'}) = o') \in \Phi(f')$ *with* $f'(\boldsymbol{i'}) < lhs(r)$ **do**
6 $\quad\quad \theta \leftarrow$ substitution s.t. $o'^\theta \equiv rhs(r)$
7 $\quad\quad \mathcal{P} \leftarrow$ `makePatterns(`$\boldsymbol{i'^\theta}, \sigma$`)`
8 $\quad$ **end**
9 $\quad$ **foreach** $\boldsymbol{p'} \in \mathcal{P}$ **do**
10 $\quad\quad \tau \leftarrow$ a substitution s.t. $f'(\boldsymbol{p'})^\sigma \equiv f'(\boldsymbol{i'})^{\tau\theta}$
11 $\quad\quad$ **if** $t^\sigma \equiv o'^{\tau\theta}$ **then**
12 $\quad\quad\quad$ **insert** $\langle\{f(\boldsymbol{p}) = f'(\boldsymbol{p'})\}, \emptyset\rangle$ **into** $\mathcal{S}$
13 $\quad\quad$ **end**
14 $\quad$ **end**
15 **end**
16 **return** $\mathcal{S}$

Some explanations for Algorithm 6 may be necessary. The algorithm iterates over all defined function symbols of the current specification and the background knowledge (line 2). The pattern $\boldsymbol{p'}$ for the call to f' is first of all constructed only w.r.t. one covered example of ρ, namely that with the smallest LHS (line 3), because the pattern is meant to be as general as possible. Later (line 11) only the compatible patterns, i.e. those which satisfy condition (i) of Definition 5.3.4, are kept. Possible patterns are generated (line 7) for any covered example that satisfies the reduction order as stated in condition (iii) of Definition 5.3.4. Another remark is necessary for θ as computed in line 6. The substitution is chosen in such a way that we achieve a renaming of variables in terms of

ρ, i.e. we normalise all variables that won't be captured by τ.

The auxiliary function `makePatterns` defined in Algorithm 7 is straight forward. It can be seen as inverting a substitution application. Given the result term t and the substitution σ, the task is to find a term (probably containing variables) which yields the result term t after applying the substitution σ. If we can find our target term in one of the variable assignments, we return the variable (line 3). Otherwise, we invert the substitution recursively over the structure of t, apply `makePatterns` to all its subterms (line 10), and combine the results under the root symbol (line 13), because it was not affected by the substitution. Similarly, if t is a constant (line 6), we return it, as it was not affected by σ, too.

Algorithm 7: `makePatterns`(t, σ)

input : a linear term t
input : a substitution σ
output : a (possibly empty) set T of terms p, s.t. $p \preceq t$, s.t. $s^\sigma \equiv t$

1 $T \leftarrow \emptyset$
2 **foreach** *variable assignment* $(x \mapsto t') \in \sigma$ **do**
3   **if** $t' = t$ **then insert** x **into** T
4 **end**
5 **switch** t **do**
6   **case** $t = c()$, *i.e. is a constant*
7   **insert** t **into** T
8   **case** $t = c(t_1, \ldots, t_n)$
9   **foreach** $i = [1..n]$ **do**
10             $T_i \leftarrow$ `makePatterns`(t_i, σ)
11   **end**
12   **foreach** $(s_1, \ldots, s_n) \in T_1 \times \ldots \times T_n$ **do**
13   **insert** $c(s_1, \ldots, s_n)$ **into** T
14   **end**
15   **end**
16 **end**
17 **return** T

As the operator for a direct call χ_{direct}, the operator χ_{call} introduces a call to a previously defined function. However, the arguments for the call are not constructed using bindings to the pattern variables of the current candidate rule, but via subfunctions which take the same input as the calling function.

Definition 5.3.5. Let $\langle P, \Phi \rangle$ be a candidate CS with corresponding specification, B a background specification, s.t. $\mathcal{D}_\Phi \cap \mathcal{D}_B = \emptyset$, ρ: $f(\boldsymbol{p}) = t$ a candidate rule in P, and $\Phi(\rho)$ the examples covered by ρ. Let further be $\{g_i\}_{i \in \mathbb{N}}$ a set of new function symbols neither occurring in P nor in B, i.e. $g_i \notin \mathcal{D}_{P \cup B} \cup \mathcal{C}_{P \cup B} \cup \mathcal{X}$). Then the result of **function call**

operator χ_{call} is the (possibly empty) set containing all elements of the form

$$\langle\{f(\boldsymbol{p}) = f'(g_1(\boldsymbol{p}), \dots, g_n(\boldsymbol{p}))\} \cup P_S, \phi_S\rangle \in \chi_{\text{call}}(\rho, \Phi, B),$$

where $f' \in \mathcal{D}_{\Phi \cup B}$ with arity n, if and only if there is a total mapping $\mu\colon \Phi(r) \mapsto (\Phi \cup B)(f')$ s.t. for each $(f(\boldsymbol{i}) = o) \in \Phi(\rho))$ and $\mu(f(\boldsymbol{i})) := (f'(\boldsymbol{i}) = o')$ the following conditions are satisfied:

(i) $\mathcal{V}ar\,(f'(\boldsymbol{i})) \equiv \mathcal{V}ar\,(o')$,

(ii) $o \equiv o'^{\tau}$ for some substitution τ, and

(iii) if $f' \in \mathcal{D}_\Phi$, then $f(\boldsymbol{i}) > f'(\boldsymbol{i'})^{\tau}$ w.r.t. a reduction order $>$.

If there exists such a mapping μ, then

- $\phi_S := \{g_j(\boldsymbol{i}) = i'_j\tau \mid j \leftarrow [1..n],$
  $(\varphi\colon f(\boldsymbol{i}) = o) \leftarrow \Phi(\rho),$
  $(f'(i'_1, \dots, i'_n) = o') \leftarrow \mu(\varphi)\}$, and
- P_S is the initial candidate CS of ϕ_S.

The operator χ_{call} is based on the idea, that given a covering rule ρ and for each covered rule in $\Phi(\rho)$ there is a rule of some defined function f', s.t. the RHS of this f'-rule subsumes the RHS of one covered rule, we can use f' to compute the same outputs as with ρ. Thus, we introduce a new rule $f(\boldsymbol{p}) = f'(g_1(\boldsymbol{p}), \dots, g_n(\boldsymbol{p}))$ where each $g_i(\boldsymbol{p})$ reduces to the appropriate input argument for f'.

Algorithm 8 computes the function call operator χ_{call}. It iterates over all defined function symbols, i.e. all function calls possible. An auxiliary function `possibleMapping`, defined in Algorithm 9, computes for the current defined function symbol a mapping $\hat{\mu}\colon \phi(\rho) \mapsto \mathfrak{P}(\Phi \cup B)$[2] (line 3), which maps each IO example $f(\boldsymbol{i}) = o \in \Phi(\rho)$ to all examples of f'. It assures that conditions (i) and (ii) of Definition 5.3.5 are satisfied. In the second loop (line 4) only those examples are selected that satisfy condition (iii) of Definition 5.3.5 (line 5). In the innermost loop (line 8) the corresponding function call and the specification for all new introduced subfunctions are constructed and added to the result set.

5.4. A Synthesis Example

This section demonstrates one run of the IGOR II$_\text{H}$ algorithm on a simple example. It is based in a perviously published tool demo [47]. For now, only the operators for partitioning χ_{split}, for introducing auxiliary functions χ_{subfn}, and calls to previously defined functions χ_{direct} and χ_{call} will be used. The introduction of a higher order schema (χ_{cata}), as presented in Chapter 6, will be considered later in Section 6.4.4.

[2] $\mathfrak{P}(S)$ denotes the power set of S.

Algorithm 8: Function call operator χ_{call}

input : an open rule ρ: $f(\boldsymbol{p}) = t$
input : a target specification Φ
input : a background specification B
output : a set $\mathcal{S} = \{\langle S_i, \phi_i\rangle, \ldots, \langle S_n, \phi_n\rangle\}$ containing pairs of a successor rule set with according new specification subset

$\mathcal{S} \leftarrow \emptyset$
foreach $f' \in \mathcal{D}_{\phi \cup B}$ **do**
  $\hat{\mu} \leftarrow$ `possibleMappings`$(\rho, \Phi \cup B, f')$
  foreach *mapping* $(\mu\colon \Phi(\rho) \mapsto \Phi \cup B)$ *with* $\mu(\varphi) \in \hat{\mu}(\varphi)$ *for all* $\varphi \in \Phi(\rho)$ **do**
    if *all* $\{(f' \in \mathcal{D}_B \ \vee \ f(\boldsymbol{i}) > f'(\boldsymbol{i'})) \ \wedge \ o \equiv o'^{\tau} \mid$
        $(f'(\boldsymbol{i'}) = o') \leftarrow \mu(f(\boldsymbol{i}) = o)\}$ **then**
      $\phi_S \leftarrow \emptyset$
      foreach $j \in [1..n]$ **do**
        $g_j \leftarrow$ a new defined symbol, s.t. $g_j \notin (\mathcal{D}_{P \cup B} \cap \mathcal{C}_{P \cup B} \cap \mathcal{X})$
        $\phi_S \leftarrow \phi_S \cup \Big\{ g_j(\boldsymbol{i}) = \boldsymbol{i}'^{\tau}_j \mid$
            $(\varphi\colon f(\boldsymbol{i}) = o) \leftarrow \Phi(\rho),$
            $(f'(i'_1, \ldots, i'_n) = o') \leftarrow \mu(\varphi),$
            $o \equiv o'^{\tau}\Big\}$
      end
      $P_S \leftarrow$ `initialCandidate`(ϕ_S)
      insert $\langle \{f(\boldsymbol{p}) = f'(g_1(\boldsymbol{p}), \ldots, g_n(\boldsymbol{p}))\} \cup P_S, \phi_S\rangle$ **into** $\mathcal{S}$
    end
  end
end
return $\mathcal{S}$

The example problem is the function `lasts` of type `[[a]]` $\to$ `[a]`, getting a list of lists and returning a list of all last elements. The type information Θ defines the types of all defined functions and the data type definition and is shown in Listing 5.8[3].

Listing 5.8: Type information θ

```
data [α]   =   α |α : [α] -- built in
lasts      :: [[a]] → [a]
last       :: [a] → a
```

The target specification is depicted in Listing 5.9. Note that we use some syntactic sugar for lists to keep the examples readable. As before, list `(a:b:c:d:[])` is abbreviated by `[a,b,c,d]`, and `(:)` denotes an infix *cons* operator. We switch between the

[3]The list type is special in HASKELL and built in. Therefore, it is built into IGOR II$_{\text{H}}$, too, and it is not necessary to explicitly define it. We just mention it to be self contained. In fact, it is not possible to redefine lists in HASKELL like this.

Algorithm 9: `possibleMapping`(r, ϕ, f')

input : an open rule $\rho\colon f\,(\boldsymbol{p}) = t$
input : a set of specification rules ϕ
input : a defined function symbol $f' \in \mathcal{D}_\phi$
output : a total mapping $\hat{\mu}\colon \phi(\rho) \mapsto \mathfrak{P}(\phi)$

$\hat{\mu} \leftarrow \emptyset$
foreach $(f(\boldsymbol{i}) = o) \in \phi(\rho)$ **do**
    $P_{f(\boldsymbol{i})} \leftarrow \emptyset$
    foreach $(f'(\boldsymbol{i'} = o') \in \phi$ *with* $\mathcal{V}ar\,(f'(\boldsymbol{i'})) \equiv \mathcal{V}ar\,(o')$ **do**
        if $o \equiv o'^{\tau}$ *for any substitution* τ **then**
            insert $f'(\boldsymbol{i'}) = o'$ **into** $P_{f(\boldsymbol{i})}$
        end
        insert $(f(\boldsymbol{i}) = o) \mapsto P_{f(\boldsymbol{i})}$ **into** $\hat{\mu}$
    end
end
return $\hat{\mu}$

two representations where appropriate. Variables are written in small caps in HASKELL.

Listing 5.9: Target specification Φ

```
lasts []                    = []
lasts [[a]]                 = [a]
lasts [[a,b]]               = [b]
lasts [[a,b,c]]             = [c]
lasts [[b],[a]]             = [b,a]
lasts [[c],[a,b]]           = [c,b]
lasts [[c,d],[b]]           = [d,b]
lasts [[a,b],[c,d]]         = [b,d]
lasts [[c],[d,e],[f]]       = [c,e,f]
lasts [[c,d],[e,f],[g]]     = [d,f,g]
```

Listing 5.10 defines the background specification B containing the examples for the background knowledge function `last`. Sure, IGOR II could solve the problem without additional help and with this function as additional knowledge there is not much left for IGOR II, but we want to keep the simulation as clear as possible.

Listing 5.10: Background specification B

```
last [a]         = a
last [a,b]       = b
last [a,b,c]     = c
last [a,b,c,d]   = d
```

The initial hypothesis is a single rule covering all examples of the target function, but with an unbound variable on the RHS (cf. Listing 5.11).

Listing 5.11: Initial Hypothesis $H_0 : \chi_{\text{init}}$

```
lasts x = y
```

To keep track of the operator application we will label the resulting hypothesis H with the sequence of operator applications $H : \chi \circ \chi \circ \ldots$. If we want to make clear to which rule an operator was applied, we will write the rule number as superscript to the function composition operator $\circ$, i.e. $\chi \overset{n}{\circ} \chi$. When applying χ_{split}, χ_{direct}, or χ_{call} we add the variable w.r.t. which the partitioning was performed, and the corresponding called function as subscript. Now we start to stepwise develop our initial hypothesis. In each iteration of the algorithm all available operators are applied to the currently best hypothesis.

5.4.1. Iteration 1

The initial hypothesis H_0 is the only one in our search space at the moment, covering all example equation of `lasts`.

Partitioning We start with the partition operator. There is only the variable `x` on the LHS of the rule in H_0. This rule is the LGG of rules $\{1 \ldots 10\}$ of the example equations of `lasts`, which explains this variable, because rule 1 has the constructor `[]` on the position where rules 2 to 10 have the symbol `(:)`. This induces a partition of all examples into the subsets $\{1\}$ and $\{2 \ldots 10\}$. Generalising both subsets, we get a new hypothesis with specialised patterns as shown in Listing 5.12.

Listing 5.12: $H_1 : \chi_{\text{split,x}} \overset{1}{\circ} \chi_{\text{init}}$

```
lasts             []  = []
lasts ((x0:x1):x2)    = (y:ys)
```

Auxiliary Introduction The rule 1 in hypothesis H_0 has a variable at the root position of the RHS, so the operator A is not applicable.

Function Call In rule 1 of hypothesis H_0 the RHS can not be replaced by a call to the background `last`, because of different output types. A recursive call to `lasts` would be conceivable, but since the argument must decrease according to the reduction order in size to prevent non-termination, this is not allowed here.

5.4.2. Iteration 2

Still only one hypothesis, namely H_1, is in our hypotheses space and rule 2 is the sole open one.

Partitioning The LHS of rule 2 contains three variables, so we can generate successor hypotheses partitioning w.r.t. to each of them.

The first variable `x0` does not induce a partition, because at this position all example equations contain a variable.

Partitioning w.r.t. the second variable `x1` separates all rules which have a one-element list as first element as input from those with more. The induced partitioning subsets are $\{2, 5, 6, 9\}$ and $\{3, 4, 7, 8, 10\}$ of the original IO examples, leading to the new hypothesis shown in Listing 5.13.

Listing 5.13: $H_2 : \chi_{\mathrm{split},\mathtt{x}_1} \overset{2}{\circ} \chi_{\mathrm{split},\mathtt{x}} \overset{1}{\circ} \chi_{\mathrm{init}}$

```
lasts                 []   =  []
lasts           ([x]:xs)   =  (x:ys)
lasts  ((x0:(x1:x2)):x3)   =  (x4:x5)
```

Partitioning w.r.t. the third variable i.e. variable `x2`, separates all rules with a singleton input list from those with more. The induced partitioning subsets are $\{2, 3, 4\}$ and $\{5, \dots, 10\}$, leading to a new hypothesis as shown in Listing 5.14:

Listing 5.14: $H_3 : \chi_{\mathrm{split},\mathtt{x}_2} \overset{2}{\circ} \chi_{\mathrm{split},\mathtt{x}} \overset{1}{\circ} \chi_{\mathrm{init}}$

```
lasts                      []   =  []
lasts                 [x0:x1]   =  [x2]
lasts  ((x0:x1):((x2:x3):x4))   =  (x5:(x6:x7))
```

One can see that the partitions get more and more fragmented, and finally will lead to an overfitting. However, Igor II's bias is to prefer those hypotheses, which have the least number of partitions.

Auxiliary Introduction The rule number 2 of our current hypothesis H_1 has the infix constructor symbol `(:)` at root position. So we can replace both subterms, subsumed by `y` and `ys`, respectively, by calls to the auxiliary functions $\mathtt{fun}_1$ and $\mathtt{fun}_2$.

We do not have both of them, yet. To treat them as new induction problems, we need example equations for them. Consider $\mathtt{fun}_1$ first. Using the input of our target function, $\mathtt{fun}_1$ has to compute the first element in the output list. Listing 5.15 shows the resulting IO examples.

Listing 5.15: Abduced IOs for $\mathtt{fun}_1$

```
fun1  [[a]]                =  a
fun1  [[a,b]]              =  b
fun1  [[a,b],[c,d]]        =  b
fun1  [[b],[a]]            =  b
fun1  [[a,b,c]]            =  c
fun1  [[c],[a,b]]          =  c
fun1  [[c],[d,e],[f]]      =  c
fun1  [[c,d],[b]]          =  d
fun1  [[c,d],[e,f],[g]]    =  d
```

Listing 5.16 shows the IO examples of `fun`$_2$. The function `fun`$_2$ removes the first element of the input list and returns the rest.

Listing 5.16: Abduced IOs for `fun`$_2$

```
fun2 [[a]]                 =  []
fun2 [[a,b]]               =  []
fun2 [[a,b,c]]             =  []
fun2 [[b],[a]]             = [a]
fun2 [[c],[a,b]]           = [b]
fun2 [[c,d],[b]]           = [b]
fun2 [[a,b],[c,d]]         = [d]
fun2 [[c],[d,e],[f]]       = [e,f]
fun2 [[c,d],[e,f],[g]]     = [f,g]
```

With these additional example equations for the new auxiliary functions we can develop our new hypothesis. The variables in the RHS of H_1's second rule have been replaced by calls to auxiliary functions. The initial rules for `fun`$_1$ and `fun`$_2$ have been included in the new hypothesis of Listing 5.17, too.

Listing 5.17: $H_4 : \chi_{\text{subfn}} \overset{2}{\circ} \chi_{\text{split,x}} \overset{1}{\circ} \chi_{\text{init}}$

```
1 lasts              []    = []
2 lasts  i@((x0:x1):x2)    = fun1 i : fun2 i
3 fun1     ((x0:x1):x2)    =  x3
4 fun2     ((x0:x1):x2)    =  x3
```

The `@` is not supported by IGOR II, but is a common syntax in HASKELL to bind a complex pattern to a simple variable. This keeps our code a little less messy.

Function Call Because of type mismatch, a call to `last` is not allowed. Also not to `lasts`, because we cannot find both, a matching RHS and a smaller LHS for all equations covered by our rule 2.

5.4.3. Iteration 3

Now there are three hypotheses in the search space, but only H_4 (Listing 5.17) has the least number of partitions. However both, the third and the forth rule, are open and one is chosen arbitrarily. So we continue by developing rule 3 of `fun`$_1$.

Partitioning Rule 3 of H_4 (Listing 5.17) contains three variables. The first variable `x`$_0$ does not induce a partition, because at this position all example equations contain a variable.

Partitioning w.r.t. to `x`$_1$ also induces two subsets of the example equation of `fun`$_1$. Namely one where the first element is a singleton list $\{1, 4, 6, 7\}$ and the other where the first element is a list with at least two elements $\{2, 3, 5, 8, 9\}$. We replace rule 3 of H_4 in Listing 5.17 by two successor rules in H_5 in Listing 5.18.

Listing 5.18: $H_5 : \chi_{\text{split},\texttt{x}_2} \overset{3}{\circ} \chi_{\text{subfn}} \overset{2}{\circ} \chi_{\text{split},\texttt{x}} \overset{1}{\circ} \chi_{\text{init}}$

```
1 lasts               [] = []
2 lasts i@((x0:x1):x2) = fun1 i : fun2 i
3 fun1        ([x0]:x1) = x0
4 fun1 ((x0:x1:x2):x3) = x4
5 fun2      ((x0:x1):x2) = x3
```

Partitioning w.r.t. to $\texttt{x}_2$ induces two subsets of the examples of $\texttt{fun}_1$. One where all inputs are a singleton list $\{1, 2, 5\}$ and another where the input list has at least two elements $\{3, 4, 6, 7, 8, 9\}$. Refer to H_6 in Listing 5.19.

Listing 5.19: $H_6 : \chi_{\text{split},\texttt{x}_3} \overset{3}{\circ} \chi_{\text{subfn}} \overset{2}{\circ} \chi_{\text{split},\texttt{x}} \overset{1}{\circ} \chi_{\text{init}}$

```
1 lasts                        [] = []
2 lasts          i@((x0:x1):x2) = fun1 i : fun2 i
3 fun1                   [x0:x1] = x2
4 fun1  ((x0:x1):((x2:x3):x4)) = x5
5 fun2              ((x0:x1):x2) = x3
```

Auxiliary Introduction This operator is again not applicable, because the RHS is a variable and does not have a constructor at root position.

Function Call Considering the examples of function $\texttt{fun}_1$: Calls to `lasts` and $\texttt{fun}_2$ are not possible due to type constraints, but to `last`. Matching the RHSS of $\texttt{fun}_1$ against the RHSS of `last`, IGOR II detects that it is possible to compute the output of $\texttt{fun}_1$ by a call to `last`. It also detects that the argument for the call can be directly constructed from variables and constructors from the LHS of the covering rule 3. Thus, no auxiliary function is needed and the RHS can be replaced, which leads to a new hypothesis H_7 presented in Listing 5.20.

Listing 5.20: $H_7 : \chi_{\text{direct},\texttt{last}} \overset{3}{\circ} \chi_{\text{subfn}} \overset{2}{\circ} \chi_{\text{split},\texttt{x}} \overset{1}{\circ} \chi_{\text{init}}$

```
1 lasts               [] = []
2 lasts i@((x0:x1):x2) = fun1 i : fun2 i
3 fun1      ((x0:x1):x2) = last (x0:x1)
4 fun2      ((x0:x1):x2) = x3
```

5.4.4. Iteration 4

The search space now contains five hypotheses H_2, H_3, H_5, H_6 and H_7, where all but H_7 have three partitions, which has only two. So the only rule in this hypothesis (rule 4) is developed.

Partitioning The patterns of the induced partitions in rule 4 of H_7 and Rule 3 of H_4 are the same and only the RHSS of the example equations covered by them are different, of course. Therefore, we have done this partitioning before and the construction of resulting hypotheses H_8 and H_9 is straight forward and reveals nothing new. As partitioning increases the costs and other hypotheses have less patterns, we can omit them here.

Auxiliary Introduction This operator is not applicable here.

Function Call Now it is only the function `lasts` to which a call is allowed and applicable. Again, IGOR II can directly construct the argument for the call from variables and constructors from the LHS of the covering rule 4. So no auxiliary function is needed and we can close our current rule and make a new hypothesis. The resulting hypothesis H_{10} is shown in Listing 5.21.

Listing 5.21: $H_{10} : \chi_{\text{direct},\texttt{lasts}} \overset{4}{\circ} \chi_{\text{direct},\texttt{last}} \overset{3}{\circ} \chi_{\text{subfn}} \overset{2}{\circ} \chi_{\text{split},\texttt{x}} \overset{1}{\circ} \chi_{\text{init}}$

```
lasts                 [] = []
lasts i@((x0:x1):x2) = fun1 i : fun2 i
fun1     ((x0:x1):x2) = last (x0:x1)
fun2     ((x0:x1):x2) = lasts x2
```

5.4.5. Iteration 5

At the beginning of this iteration IGOR II finds the best hypothesis H_{10} closed. It still has only two partitions compared to the others with three. So H_{10} is returned as the final solution.

6. Guiding Igor II's Search with Type Morphisms

Chapter 5 introduced the basic Igor II-algorithm as it has been re-implemented in Haskell for this work. It serves as a reference foundation on which the extensions introduced in this chapter are built on. The main point where the original Igor II algorithm, as presented in [66], was modified is the way how new successor rule sets are computed. Instead of calling `successorRuleSets` directly (cf. Algorithm 3), a new operator χ_{cata} which uses type morphisms, especially catamorphisms, as recursive program schemes to guide the search is used. Only if the operator for type morphisms introduction is not applicable, the successor rule sets are computed as usual. The modification of Algorithm 3 is shown in Algorithm 10.

Algorithm 10: `successorRuleSets'`(r, Φ, B, Θ)

---

input : an open rule r
input : a target specification Φ
input : a background specification B
input : type information Θ

output : a set of successor rules with corresponding specification

$\mathcal{S} \leftarrow \chi_{\text{cata}}(r, \Phi, \Theta)$
if $\mathcal{S} = \emptyset$ **then**
  $\mathcal{S} \leftarrow$ `successorRuleSets`(r, Φ, B)
end
return $\mathcal{S}$

Section 6.1 describes how catamorphisms on arbitrary inductively defined data types can be detected in a given set of IO examples. Section 6.2 formally defines the new operator, where Section 6.3 proposes some modifications of this operator. It describes how catamorphisms on lists and the detection of natural transformation represented by type functors can be seen as special cases of this operator. It also explains how χ_{cata} can be generalised to detect paramorphisms, which describe some form of primitive recursion on inductive types. Section 6.4 gives some illustrating examples on well known inductive types and continues the hand simulation of the Igor II algorithm from Section 5.4, now using the new operator χ_{cata}.

6.1. Data-Driven Detection of Arbitrary Catamorphisms

Chapter 4 formalised inductive data types as fixed points of a polynomial functor F and introduced catamorphisms as unique morphisms from a inductive data type to any other type which is an F-algebra.

This chapter develops how this can be used for inductive program synthesis, especially as an extension of the IGOR II algorithm. It is based on prior published work [48, 49, 50]. It first takes on a macro view and analyses the semantics of this task from a categorical perspective (Subsection 6.1.1), then it changes to a micro view to concentrate on the syntactic level in a term rewriting perspective (Subsection 6.1.2). The reader is asked to refer to the examples in Section 6.4 at any time.

6.1.1. The Categorical Perspective

The starting point for IGOR II is always the specification Φ containing IO examples $\Phi(f)$ which partially define a target function $f\colon A \to B$. Furthermore, let A be the fixed point of a polymorphic functor F, s.t. $A = \mu\mathsf{F}$. This can be again described in the following, now well known, commuting diagram.

$$
\begin{tikzcd}
\mathsf{F}\mu\mathsf{F} \arrow[r, "\mathrm{in}_{\mathsf{F}}"] \arrow[d, "\mathsf{F}f"'] & \mu\mathsf{F} \arrow[d, "f"] \\
\mathsf{F}B \arrow[r, "\varphi"] & B
\end{tikzcd}
$$

It is known that, given φ, f can be expressed as a catamorphism s.t. $f = (\!|\varphi|\!)_{\mathsf{F}}$. The function φ is unknown, though. However, its IO behaviour can be abduced from the examples in $\Phi(f)$ using the following equality:

$$\varphi \circ \mathsf{F}f \circ \mathrm{in}_{\mathsf{F}}^{-1} = f, \tag{6.1}$$

where $\mathrm{in}_{\mathsf{F}}^{-1}$ is the inverse of the isomorphism in_{F} (cf. Definition in-inv-DEF), and thus

$$
\begin{aligned}
\mathit{cod}\ f &= \mathit{cod}\ \varphi \\
\mathit{dom}\ (\mathsf{F}f \circ \mathrm{in}_{\mathsf{F}}^{-1}) &= \mathit{dom}\ f \\
\mathit{cod}\ (\mathsf{F}f \circ \mathrm{in}_{\mathsf{F}}^{-1}) &= \mathit{dom}\ \varphi,
\end{aligned}
$$

where *dom* and *cod* in this case not only refer to the object, i.e. type, in our category, but in this special case really extend to the term level of the target function's IOs.

The set of IO examples for φ is completely determined by $\Phi(f)$ and the function $\mathsf{F}f \circ \mathrm{in}_{\mathsf{F}}^{-1}$. For each rule $\rho \in \Phi(f)$ there is an IO example for φ sharing its RHS, whereas its LHS is the result of applying the LHS of ρ to $\mathsf{F}f \circ \mathrm{in}_{\mathsf{F}}^{-1}$. To define f as a catamorphism in terms of φ three steps are necessary:

1. Abduce IO examples for φ, s.t. for each rule $\rho \in \Phi(f)$
   a) its RHS is kept, and
   b) its new LHS is constructed by applying the old LHS of ρ to $\mathsf{F}f \circ \mathrm{in}_{\mathsf{F}}^{-1}$.
2. Re-express f as $(\!|\varphi|\!)_{\mathsf{F}}$.
3. Use IGOR II to synthesise φ.

Hence, to obtain the appropriate IO examples for φ, figuratively speaking, one just needs to follow the arrows from the diagram above counterclockwise starting from $\mu\mathsf{F}$.

Recall that our category of choice is distributive. Therefore, there are polynomial inductive data types, i.e. types, only built from primitive types by products and coproducts. The functors inducing those types are polynomial too, i.e. only built from products, coproducts, the identity functor, and the constant functor. Referring to the diagram above, one can say that for the functor F of the F-algebra it holds that $\mathsf{F} = \mathsf{F}_1 + \cdots + \mathsf{F}_n$, i.e. F is the coproduct of n functors F_i for $i = 1 \ldots n$. Similar, $\mathrm{in}_{\mathsf{F}} = [c_1, \ldots, c_n]$ is the sum of n constructors. According to Definition in-inv-DEF, it holds that $\mathrm{in}_{\mathsf{F}}^{-1} = (\!|\mathsf{F}[c_1, \ldots, c_n]|\!)_{\mathsf{F}}$. Hence, the mediating function of the catamorphism $\varphi : \mathsf{F}B \to B$ is in fact a case distinction $\varphi = [\varphi_1, \ldots, \varphi_n]$. For each coefficient of the type $\mathsf{F}A$ there exists one function $\varphi_i : \mathsf{F}_i B \to B$. Equation 6.1 can be spelled out to:

$$\varphi \circ \mathsf{F}f \circ \mathrm{in}_{\mathsf{F}}^{-1} = [\varphi_1, \ldots, \varphi_n] \circ [\mathsf{F}_1 f, \ldots, \mathsf{F}_n f] \circ (\!|\mathsf{F}[c_1, \ldots, c_n]|\!)_{\mathsf{F}} = f. \tag{6.2}$$

Therefore, abducing IO examples for φ splits up into the task of abducing IO examples for n functions $\varphi_i \colon \mathsf{F}B \to B$, for $i = 1 \ldots n$, i.e. one for each constructor of $\mu\mathsf{F}$, or summand of F. We will come back to this later in the next section (6.1.2). To spare subscripts, for now it suffices to keep this in mind and consider the coproducts only.

To abduce IO examples for each φ_i from the inputs of f, we need to statically evaluate $\varphi \circ \mathsf{F}f \circ \mathrm{in}_{\mathsf{F}}^{-1}$, using the IOs of f, as much as possible. The original inputs given in $\Phi(f)$ are taken and first $\mathrm{in}_{\mathsf{F}}^{-1}$ is applied to deconstruct the inductive type $\mu\mathsf{F}$ into a sum of product types. The function in_{F} is an isomorphism, and thus defining $\mathrm{in}_{\mathsf{F}}^{-1}$ by itself. Where in_{F} fuses a product from $\mathsf{F}\mu\mathsf{F}$ into a value of type $\mu\mathsf{F}$, via $\mathrm{in}_{\mathsf{F}}^{-1}$ it is possible to break up a value of type $\mu\mathsf{F}$ to retrieve this product.

The mapping from $\mathsf{F}\mu\mathsf{F}$ to $\mathsf{F}B$ is determined by F and our target function f. $\mathsf{F}\mu\mathsf{F}$ is a sum of products, because F is polynomial, i.e. is a sum of functors. Hence $\mathsf{F}f$ is a sum, too. Consequently, the structure of $\mathsf{F}\mu\mathsf{F}$ and $\mathsf{F}B$ are identical. The only difference is that wherever there is a summand of type $\mu\mathsf{F}$ in the sum of $\mathsf{F}\mu\mathsf{F}$, the corresponding summand in $\mathsf{F}B$ is of type B. Hence, $\mathsf{F}f$ maps values of type $\mu\mathsf{F}$ to values of type B, and acts as identity on all others.

We said that φ is a sum, because F is a sum. If for φ_i the corresponding functor F_i is a constant functor K_A, it holds that $\varphi_i : A \to B$ and that φ_i takes the same input as f to compute the same output, because $\mathsf{F}_i f = \mathrm{id}_A$.

If the functor is the identity functor $\mathsf{Id}_{\mu\mathsf{F}}$, it holds that $\mathsf{F}_i f = f$. Again, φ_i computes the same output tas f but with a different input. This input, however, can be computed by evaluating $\mathrm{in}_{\mathsf{F}}^{-1}$ on an input term of f, taking the corresponding summand of $\mathsf{F}_i f$ from this result, and evaluating it on $\Phi(f)$. If we do this for each IO example of f, we get the corresponding IOs for φ_i.

If F_i is a product functor, where each coefficient is, again, either a constant functor or the identity functor. The input for φ_i is a product, i.e. a nested tuple too. By applying the procedure above for each coefficient, the corresponding coefficients for the input of φ_i can be abduced in the same way. Examples for these procedures can be found in Subsections 6.4.1, 6.4.2, and 6.4.3, respectively.

Thus, given a specification Φ, type information Θ, and a candidate rule $\rho\colon f(\boldsymbol{p}) = t$, one can construct a rule $\rho'\colon f(\boldsymbol{p}) = (\![\varphi_1, \ldots, \varphi_n]\!)\ \boldsymbol{p}$, and abduce corresponding specifications $\Phi(\varphi_i)$ with $i \in [1..n]$ and initial rules for each new subfunction φ_i.

6.1.2. The Term Rewriting Perspective

However, with category theory we won't get any further, so lets put on the term rewriting goggles. Let Φ be a target specification, Θ the corresponding type information and $\rho\colon f(\boldsymbol{p}) \to t$ a candidate rule. Let further be any rule in $\Phi(\rho)$ of the form $f(\boldsymbol{i}) = o$. Assume further, to avoid complicating things with even more indices, $\boldsymbol{p}$ to be a vector with only one field, i.e. f has arity 1 for now. Let $\boldsymbol{p}$ be of type α, i.e. $(\boldsymbol{p} :: \alpha)$, and α be an inductive data type. Let $\Theta(\alpha)$ be the set of constructors of α, i.e.

$$\Theta(\alpha) := \{c_1, \ldots, c_m\}.$$

Since all terms subsumed by $\boldsymbol{p}$ are of the same type α, its type constructors induce a natural partitioning into m disjoint subsets. The constructor is always at the root position of the first (and only) argument, so this partitioning is the quotient set of $\Phi(\rho)$ w.r.t. $\sim_{pos}$ (cf. Definition 5.3.1) for $pos = 1$, i.e.

$$\Phi(\rho)/\sim_{pos} = \{\Phi(\rho)_{c_1}, \ldots, \Phi(\rho)_{c_m}\}.$$

So for each constructor symbol c_j, with $j \in [1..m]$, there is one quotient. The LHSS of all rules in one quotient $\Phi(\rho)_{c_j}$ are subsumed by a term, say $c_j(\boldsymbol{p'_j})$.

Each quotient of the example equations $\Phi(\rho)_{c_j}$ gives rise to a new mediating function φ_j. The example equations can be abduced using the examples in $\Phi(\rho)_{c_j}$ which are of the form $f(c_j(\boldsymbol{p'_j})) = o$. We create an new set of examples ϕ_j, such that for each equation $r \in \Phi(\rho)_{c_j}$ we build an equation $\varphi_j(\boldsymbol{q}) = o$, where o is the RHS of r and $\boldsymbol{q} := (q_1, \ldots, q_n)$ is (at the moment) a vector containing a single n-ary nested tuple.

For $i = [1..n]$, if the i^{th} argument of the constructor c_j in r, i.e. $\boldsymbol{p'_j}|_i$, is of type α and there is an equation $f(\boldsymbol{p}) = o$ in $\Phi(\rho)$ s.t. $\boldsymbol{p}^\sigma \equiv \boldsymbol{p'_j}|_i$ for some substitution σ, we assign q_i to o^σ. If $\boldsymbol{p'_j}|_i$ is not of type α then q_i is assigned to $\boldsymbol{p'_j}|_i$.

In plain words, an m-ary constructor term $c_j(\boldsymbol{p'_j})$ which is input to f is transformed to an m-ary nested tuple and given as input to φ_j. That is, we apply $\mathrm{in}_{\mathsf{F}}^{-1}$. Each direct subterm t of c_j of type α is replaced by the result of a recursive call to f, i.e. by the RHS of the equation of f that subsumes t. All other direct subterms are kept unchanged. This is the application of $\mathsf{F}f$.

Thus, for each constructor symbol of the inductive type α we get one function. The coproduct of those functions, i.e. a case distinction on the constructor symbol, is exactly the mediating function needed for the catamorphism.

The function f was said to have only one argument for the sake of simplicity. If f has now n arguments, and the n^{th} is of an inductive type, and assuming the quotient set is computed w.r.t. n, then all other arguments are transferred unchanged to the IO examples of the mediating functions φ_j. So if the candidate rule is $\rho\colon f(p_1, \ldots, p_n) = o$ and the catamorphism is applied to p_n, all mediating functions are of the form $\varphi_j(p_1, \ldots, p_{n-1}, p'_j) = o'$, where the terms $p_1, \ldots, p_{n-1}$ subsume the same subterms as in ρ, and then p'_j is the argument for the catamorphism. When calling the catamorphism, the mediating functions are partially applied, s.t. $\rho'\colon f(p_1, \ldots, p_n) = (\![\varphi_1(p_1, \ldots, p_{n-1}), \ldots, \varphi_{n-1}(p_1, \ldots, p_{n-1})]\!)p_n$. Keep this in mind when we soon relax this and allow the catamorphism applied to any argument.

6.2. Formal Definition of the Operator χ_{cata}

The previous section described now the structure of an inductive data type can be used to introduce type morphisms as program schemes. This section gives a formal definition of an operator χ_{cata} for IGOR II to introduce catamorphisms on those types.

To give a formal definition of the new operator χ_{cata}, one more assumption is needed. The operator χ_{cata}, which introduces primitive structural recursion for a specific inductive data type, assumes the recursion scheme, i.e. the catamorphism, to be defined in the target language, HASKELL in our case. This requires to extend the language bias of IGOR II. In the current IGOR II re-implementation in HASKELL primitive structural recursion is defined via the polymorphic function `cata` from the `Generics.Pointless` library[1]. It also provides an operator for a join on functiosn ($\oplus$), which is required to define the mediating function of a catamorphisms as a sum of functions. For lists, the better known function `foldr`, `map`, and `filter` are used, which are just specialised catamorphisms for lists. For now let $(\!|\ |\!)_\alpha$ denote the catamorphisms of a type α and $[\ ]$ the sum operator for functions.

Definition 6.2.1. Let $\langle P, \Phi\rangle$ be a candidate CS and the corresponding specification, let Θ contain the corresponding type information, let $\rho\colon f(p_1, \ldots, p_n) = t$ be a candidate rule in P with n arguments, and $\Phi(\rho)$ the examples covered by ρ. Let p_i be any argument of type α, an inductive type with catamorphism $(\!|\ |\!)_\alpha$ and type constructor $\Theta(\alpha) = \{c_1, \ldots, c_m\}$.

Assume further $(\!|\ |\!)_\alpha$ to be a polymorphic function defining the catamorphism for the inductive type α and $[\varphi_1, \ldots, \varphi_m]$ to be the sum of the functions $\varphi_1, \ldots, \varphi_m$. The **structural recursion operator**χ_{cata} is defined as the (possibly empty) set

$$\chi_{\text{cata}}(\rho, \Phi, \Theta) := \{\langle\{\rho'\} \cup P_{\mathcal{S}}, \phi_{\mathcal{S}}\rangle\},$$

if and only if

[1] Available from `hackageDB :: [Package]`, the GHC library data base.

(i) $\Phi(\rho)/\sim_i = \{\Phi(\rho)_{c_1}, \dots, \Phi(\rho)_{c_m}\}$ and $\Phi(\rho)_{c_j} \neq \emptyset$ for all $j \in [1..m]$, i.e. for each constructor $c_j \in \Theta(\alpha)$ there exists a non-empty quotient, and

(ii) for all $(f(s_i, \dots, s_{i-1}, c_j(a_1, \dots, a_k), s_{i+1}, \dots, s_n) = o) \in \Phi(\rho)_{c_j}$, with $j \in [1..m]$, there exists a mapping $\mu\colon \mathcal{T}_\Sigma(\mathcal{X}) \mapsto \mathcal{T}_\Sigma(\mathcal{X})$ for all a_l with $l \in [1..k]$ defined as:

$$\mu(a) = \begin{cases} \text{if } a :: \alpha & \text{then } o'^{\sigma} \text{ for any substitution } \sigma \text{ s.t. for all} \\ & (f(\boldsymbol{p}) = o') \in \Phi(\rho) \text{ it holds that} \\ & f(s_1, \dots, s_{i-1}, a, s_{i+1}, \dots, s_n)) \equiv f(\boldsymbol{p})^{\sigma}, \\ \text{if } a \not:: \alpha & \text{then } a. \end{cases}$$

If conditions (i) and (ii) are satisfied, then:

- $\rho'\colon f(p_1, \dots, p_n) = (\![\varphi_1(p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n), \dots, \varphi_m(p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n)]\!)_\alpha\ p_i$,
- $\phi_{S_j} := \{\varphi_j(s_1, \dots, s_{i-1}, s_{i+1}, \dots, s_n, s'_i) = o\}$ for all $(f(s_1, \dots, s_{i-1}, c_j(a_1, \dots, a_k), s_{i+1}, \dots, s_n)) = o) \in \Phi(\rho)_{c_j}$ and $j \in [1..m]$, where $s'_i = (\mu(a_1), \dots, \mu(a_k)))$,
- $\phi_{\mathcal{S}} = \bigcup_{j=1}^{m} \phi_{S_j}$, and
- $P_{\mathcal{S}}$ is the initial candidate CS of $\phi_{\mathcal{S}}$.

To prevent the catamorphism to be partially undefined, condition (i) demands each constructor of α must occur in some rule of $\Phi(\rho)$ at position i at least once. Condition (ii) assures that there is indeed evidence in $\Phi(\rho)$ that justifies the application of a structural recursive program scheme. Given a catamorphisms $(\![\varphi_1, \dots, \varphi_m]\!)_\alpha$, for a rule $f(s_1, \dots, s_{i-1}, c_j(a_1, \dots, a_k), s_{i+1}, s_n) = o$, the mediating function φ_j for the constructor c_j accepts a tuple (a sum) $(\mu(a_1), \dots, \mu(a_k))$ as input, where each a_l, for $l = [1..k]$, which is of type α is replaced by the corresponding term a recursive call to f would yield. If a_l is not of type α, then μ is the identity. The task of each φ_j is just to combine the intermediate results, s.t. it yields the corresponding output o.

Algorithm 12 describes the computation of χ_{cata}. The outermost loop (line 2) iterates over all arguments of the candidate rule ρ and checks for each if a structural recursive scheme is applicable. It checks if a catamorphism is defined for the type of the current argument i and whether for each constructor of the type, there is a non-empty quotient w.r.t. $\sim_i$. The next loop (line 7) iterates over all quotients and abduces IO examples for each function φ_j in $(\![\varphi_1, \dots, \varphi_m]\!)$ if possible, i.e. if condition (ii) as stated in Definition 6.2.1 is satisfied (line 9). The case that the condition is not satisfied coincides with `makeMediatorArg` returning an undefined value ($\perp$). The auxiliary function `makeMediatorArg` constructs the appropriate argument for each mediating function φ_j (line 12) or returns undefined ($\perp$) if condition (ii) in Definition 6.2.1 cannot be satisfied.

In the case that for some quotient $\Phi(\rho)_{c_j}$ no appropriate arguments can be constructed, the local variables are reset and the loop (line 18) is exited. No structural recursion scheme is introduced for this argument.

Algorithm 11 implements the condition (*ii*) of Definition 6.2.1: The decomposition of the argument the catamorphism is applied to and the construction of the appropriate arguments for each φ. Actually, it applies the morphism $\mathsf{F}f \circ \text{in}_{\mathsf{F}}^{-1}$ described in Equation (6.2). Note that if a catamorphism is applied to a constant constructor, the morphism $\mathsf{F}f \circ \text{in}_{\mathsf{F}}^{-1}$ maps it to the unit type (), and so does `makeMediatorArg`.

Algorithm 11: `makeMediatorArg`$(\boldsymbol{t}, i, \phi))$

input : a term vector $\boldsymbol{p} = (s_1, \ldots, s_n)$
input : a position i
input : specification ϕ
output : undefined ($\perp$) or (possibly empty) tuple of terms $\boldsymbol{p'}$

switch $t = \boldsymbol{p}[i]$ **do**
  case $c(a_1, \ldots, a_k)$
    let $\mu(a) = \begin{cases} o'^{\sigma} & \text{if } a \text{ and } t \text{ have the same type and there is a } \sigma \text{ s.t.} \\ & \quad f(s_i, \ldots, s_{i-1}, a, s_{i+1}, \ldots, s_k)) \equiv f(\boldsymbol{p'})^{\sigma} \text{ for some} \\ & \quad (f(\boldsymbol{p'}) = o') \in \phi, \\ a & \text{if } a \text{ and } t \text{ have not the same type} \\ \perp & \text{otherwise} \end{cases}$
    if *for* $i = [1..k]$ *any* $\mu(a_i) = \perp$ **then**
      return $\perp$
    else
      return $(\mu(a_1), \ldots, \mu(a_k))$
    end
  otherwise
    return ()
  end
end

Algorithm 12: Structural recursion introduction operator χ_{cata}

input : an open rule $\rho\colon f(p_1, \ldots, p_n) = t$
input : a target specification Φ
input : corresponding type information Θ

output : a finite (possibly empty) set $\mathcal{S} = \{\langle S_l, \phi_l \rangle\}_{l \in \mathbb{N}}$ containing pairs of a successor rule set with corresponding new specification subsets

$\mathcal{S} \leftarrow \emptyset$
for $i \in [1..n]$ **do**
    $m \leftarrow |\Theta(\alpha)|$, the number of constructors of the type α
    if $p_i :: \alpha \wedge$ *defined* $(\!|\;|\!)_\alpha \wedge \quad |\Phi(\rho)/\sim_i| = m \wedge \forall s \in (\Phi(\rho)/\sim_i).\ s \neq \emptyset$ **then**
        $\phi_i \leftarrow \emptyset$
        $P_i \leftarrow \emptyset$
        foreach $\Phi(\rho)_{c_j} \in \Phi(\rho)/\sim_n$ *with* $j \in [1..m]$ **do**
            $\varphi_j \leftarrow$ a new defined function symbol, s.t. $\varphi_j \notin (\mathcal{D}_{P \cup B} \cup \mathcal{C}_{P \cup B} \cup \mathcal{X})$
            if $\forall (f(\boldsymbol{p}) = o) \in \Phi(\rho)_{c_j}$. `makeMediatorArg(`$\boldsymbol{p}, i, \Phi(\rho)$`)`$\neq \bot$ **then**
                $\phi \leftarrow \Big\{ \varphi_j(p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_{n-1}, p'_j) = o \mid$
                        $(f(\boldsymbol{p}) = o) \leftarrow \Phi(\rho)_{c_j}$,
                        $p'_j \qquad \leftarrow$ `makeMediatorArg(`$\boldsymbol{p}, i, \Phi(\rho)$`)` $\}$
                $P_i \leftarrow P_i \cup$ `initialCandidate(`ϕ`)`
                $\phi_i \leftarrow \phi_i \cup \phi$
            else
                $\phi_i \leftarrow \emptyset$
                $P_i \leftarrow \emptyset$
                break
            end
        end
    end
    if $\phi_i \neq \emptyset \ \wedge \ \mathcal{P}_i \neq \emptyset$ **then**
        insert $\langle \{ f(p_1, \ldots, p_n) =$
                $(\![\varphi_n(p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_{n-1}), \ldots,$
                $\varphi_m(p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_{n-1})]\!)_\alpha \ p_i \} \cup P_i, \phi_i \rangle$
        into $\mathcal{S}$
    end
end
return $\mathcal{S}$

6.3. Modifications

The previous Section 6.2 defined the operator χ_{cata} as general as possible for arbitrary inductively defined data types. In this section we will present some modifications which use specialised catamorphisms for lists. This also allows to specialise catamorphisms to natural transformations on lists.

The implementation of catamorphisms in the target language HASKELL is based on the polymorphic function `cata`. Although it is very general and captures catamorphisms on arbitrary inductive data types, even for experienced programmers it looks a bit unfamiliar.

Especially for the list type the functions `foldr`, `map`, and `filter`, together implementing the *reduce-map-filter pattern*, are much more often in use. Using those functions instead would not only improve the readability of the synthesised program, but again, also would reduce the complexity of the problem.

In the case of `map` for example, the fact that the partial results returned from a recursive call need to be combined into a value of the original type is already captured in the scheme and need not to be generated in subsequent iteration of the synthesis algorithm.

Similarly, the notion of type functors are usually captured in HASKELL by the type class `Functor` and the class function `fmap`. They can be seen as a generalisation of `map` for other types. Furthermore, as described in Section 4.2.4, are type functors special catamorphisms describing a natural transformation on the target type. Both would simplify the synthesis process: Using `fmap` directly for types parameterised in one type which are instance of the `class Functor`, or extending the mediating function for `cata` as described in Equation (tyfunc-DEF).

6.3.1. Catamorphism on Lists

This subsection is mainly based on the first paper published about the use of list catamorphisms for IP [49]. Only the syntax has been adjusted.

Assume that in our target language there is a data type for lists over arbitrary elements of the same type and a type for Boolean values. Thus, one can say that the set of constructor symbols contains at least two designated constructor symbols for constructing lists: $nil, cons \in \mathcal{C}$ denoting the empty list and the insertion of an element into a list, respectively, and two constructors for `True` and `False`, i.e. $true, false \in \mathcal{C}$. The set of defined function symbols contains a ternary higher-order symbol for `foldr` and two binary higher-order symbols for `map` and `filter`, respectively, i.e. $foldr, map, filter \in \mathcal{D}$, where `foldr`, `map`, and `filter` are defined as explained in Section A.6.1.

6.3.1.1. Simplification using `foldr`

Recall from Example 4.2.2 that `foldr`'s universal property can be captured in the following equations:

$$\begin{array}{lcl} \mathit{foldr} \circ \mathit{nil} & = & v \\ \mathit{foldr} \circ \mathit{cons} & = & \mathit{fun} \circ (\mathrm{id}_A \times \mathit{foldr}). \end{array}$$

Given a candidate rule $\rho\colon f(\boldsymbol{p}) = t$, a corresponding specification Φ, and the position i of an argument of ρ of type list, then be $\Phi(\rho)/\sim_i \;:=\; \{\Phi(\rho)_{cons}, \Phi(\rho)_{nil}\}$ the quotient set w.r.t. position i, containing one non-empty quotient for each constructor. For the sake of simplicity assume $\boldsymbol{p} \equiv (l)$, i.e. f has only one input argument, and consequently be $i = 1$. Since f must be a function, the quotient for the empty list constructor nil $\Phi(\rho)_{nil} := f(\mathit{nil}) = v$ contains exactly one rule with v as an output term. This fixes the default value of `foldr`, and thus satisfies the first part of the universal property.

To satisfy the second part, it must hold that and for each $f(\mathit{cons}(x, xs)) = o \in \Phi(\rho)_{cons}$ with $x, xs \in \mathcal{T_C}(\mathcal{X})$, there exists another example equation $(f(xs') = o') \in \Phi(\rho)$ such that $f(xs')^{\sigma} \equiv f(xs)$ for some substitution σ. Then it is possible to abduce an example $\mathit{fun}(x, o'^{\sigma}) = o$ for each rule in $\Phi(\rho)_{cons}$ for a new defined function symbol fun. The original function f can now be rewritten to $f(l) = \mathit{foldr}(\mathit{fun}, v, l)$.

Example 6.3.1
Consider the following target specification Φ of the function `reverse`, and the corresponding type information Θ, shown in the Listings 6.1, and Listing 6.2 respectively. No background specification B is provided.

Listing 6.1: Specification Φ for `reverse`

```
reverse          []  =          []
reverse       (d:[]) =       (d:[])
reverse     (c:d:[]) =     (d:c:[])
reverse   (b:c:d:[]) =   (d:c:b:[])
reverse (a:b:c:d:[]) = (d:c:b:a:[])
```

Listing 6.2: Type information Θ for Φ of `reverse` as shown in Listing 6.1

```
data [α] =  []  | α:[α]
reverse  :: [α] → [α]
last     :: [α] → α
```

We can now try to satisfy the universal property of `foldr` using the IO examples of `reverse`. Obviously, `reverse` is defined on the empty list, so the default value `v` is fixed to `[]`. Now a function `fun` is required which composes the first element of the input list with the result of the recursive call of the rest list. Following the procedure described above, this yields:

```
fun d         [] =     (d:[])
fun c     (d:[]) =   (d:c:[])
fun b   (d:c:[]) = (d:c:b:[])
fun a (d:c:b:[]) = (d:c:b:a[])
```

The function `reverse` in our current hypotheses can now be rewritten. The function `fun` is still undefined, because its initial rule, as IGOR II would compute it, contains open variables:

```
reverse x    = foldr fun [] x
fun x xs     = (d:ys)
```

It is obvious, that `fun = snoc`, i.e. it is a function inserting an element at the end of a list. So it is renamed it for better readability. After some more iterations, IGOR II would terminate with the following solution. Note that `snoc` is solved using `foldr`, too.

```
reverse x  = foldr snoc [] x
snoc x0 x1 = foldr fun' [x0] x1
fun' x0 x1 = snoc x0 x1
```

6.3.1.2. Simplification using `map`

In the previous subsection (6.3.1.1), we have abduced the IO examples for an auxiliary function *fun* according to the universal properties of *foldr*. If each rule $\rho \in \Phi(\mathit{fun})$ is of the form $\rho\colon \mathit{fun}(a_1, a_2) = \mathit{cons}(x, xs)$, for $a_1, a_2, x, xs \in \mathcal{T}_{\mathcal{C}}(\mathcal{X})$, and $a_2 \equiv xs$, one can define example equations $\Phi(\mathit{fun}')$ for a modified function fun' such that for each $\rho \in \Phi(s)$ there is a rule $\rho' \in \Phi(\mathit{fun}')$ of the form $\rho'\colon \mathit{fun}'(a_1) = x$, then f can be modified to $f(x) = \mathit{map}(\mathit{fun}', x)$.

Example 6.3.2
Consider the target specification Φ shown in Listing 6.3 of a simple function `incr` of type `[Nat] → [Nat]` incrementing each Peano integer in a list. The corresponding type information Θ is shown in Listing 6.4.

Listing 6.3: Specification Φ for `incr`

```
incr            []  =                  []
incr        (Z:[])  =            ((S Z):[])
incr    ((S Z):[])  =         ((S(S Z)):[])
incr  (Z:(S Z):[])  =  ((S Z):(S(S Z)):[])
incr  ((S Z):Z:[])  =  ((S(S Z)):(S Z):[])
```

Listing 6.4: Type information Θ for Φ of `incr` as shown in Listing 6.3

```
data [α] =  [] | α:[α]
data Nat =   Z | S Nat
incr :: [Nat] → [Nat]
```

It is easy to check that `incr` satisfies our universal property and `fun` is our new, abduced auxiliary function with the following IO examples:

```
fun     Z            []  =          ((S Z):[])
fun  (S Z)           []  =       ((S(S Z)):[])
fun     Z ((S(S Z)):[])  =  ((S Z):(S(S Z)):[])
fun  (S Z)   ((S Z):[])  =  ((S(S Z)):(S Z):[])
```

However, for all outputs of `fun`, `(:)` is the constructor at root positions and the second argument occurs unchanged at the second position below root. Thus, `fun` can be simplified to `fun'` by dropping the second argument. Its new IOs are shown below.

```
fun'     Z   =   (S Z)
fun' (S Z) =  S(S Z)
fun'     Z   =   (S Z)
fun' (S Z) =  S(S Z)
```

Now the initial rule for `incr` can be modified using `map`. Adding the initial rule for `fun'` yields the following program:

```
incr x = map fun' x
fun' x = S x
```

Note that the initial rule for `fun'` is immediately closed by anti-unification.

6.3.1.3. Simplification using `filter`

Similarly, assume one can part $\Phi(\mathit{fun})$ from Subsection (6.3.1.1) into n non-trivial equivalence classes w.r.t. equality on the second argument ($\equiv_2$), s.t. $(\Phi(\mathit{fun})/\equiv_2) := \{\Phi(\mathit{fun})_1, \ldots, \Phi(\mathit{fun})_n\}$. Assume further, that each equivalence class $\Phi(\mathit{fun})_i$ for $i = 1, \ldots, n$ can be further parted into two non-empty disjoint sets $\Phi(\mathit{fun})_i^\top$ and $\Phi(\mathit{fun})_i^\perp$. Each rule $\rho \in \Phi(\mathit{fun})_i^\top$ must be of the form ρ: $\mathit{fun}(a_1, a_2) = \mathit{cons}(x, xs)$, s.t. $a_1 \equiv x$ and $a_2 \equiv xs$, and each rule $\rho \in \Phi(\mathit{fun})_i^\perp$ must also be of the form ρ: $\mathit{fun}(a_1, a_2) = xs$, where $a_2 \equiv xs$.

One can see that the second argument occurs always unchanged either at root positions, or below the *cons*. Since we are in a functional setting and processing lists, it is possible to deduce that whether the first argument is contained in the output depends on a predicate on it.

So we can again define example equations $\Phi(\mathit{fun}')$ for a modified function fun' such that for each $\rho \in \Phi(\mathit{fun})_i^\top$ there is an $\rho \in \Phi(\mathit{fun}')$ of the form $\mathit{fun}'(a_1) = \mathit{true}$, and for each $\rho \in \Phi(\mathit{fun})_i^\perp$ there is an $\rho' \in \Phi(\mathit{fun}')$ of the form $\mathit{fun}'(a_1) = \mathit{false}$. Now, f can be reformulated to $f(x) = \mathit{filter}(\mathit{fun}', x)$.

Example 6.3.3
Consider a function `zeros` of type `zeros :: [Nat] → [Nat]` filtering out all zeros from a list of Peano integers. It is easy to check that this function obeys the universal property of `foldr` and the abduced auxiliary function `fun` would be as follows:

```
fun        Z      []   =   (Z:[])
fun     (S Z)     []   =      []
fun  (S(S Z))     []   =      []

fun       Z    (Z:[]) = (Z:Z:[])
fun    (S Z)   (Z:[]) =   (Z:[])
fun  (S(S Z)) (Z:[]) =   (Z:[])
```

The example equations of `fun` can be partitioned w.r.t. equality on the second argument, as the layout suggests. Now, whether the first argument occurs in the output below the root position or not, this depends only on a predicate on this term. Thus, `fun` can be simplified to `fun'`, by dropping the second argument s.t.

```
fun        Z   = True
fun     (S Z)  = False
fun   (S(S Z)) = False
```

We add an initial rule for it, modify the current rule, and construct a new hypotheses as before:

```
zeros x = filter fun' x
fun'  x = y
```

Subsequent iterations of the IGOR II-algorithm would then close the unfinished rule of the function `fun'`, yielding the following solution:

```
zeros    x  = filter fun' x
fun'     Z  = True
fun' (S x)  = False
```

6.3.2. Using natural transformation with χ_{tyfunc}

Recall from Section 4.2.4 that given a bifunctor F and two arbitrary F-algebras $\mathrm{in}_A\colon \mathsf{F}(A, \mathsf{T}A) \to \mathsf{T}A$ and $\mathrm{in}_B\colon \mathsf{F}(B, \mathsf{T}B) \to \mathsf{T}B$ with type functor T, that given a function $f\colon A \to B$ and a F_A-algebra φ, one could define $\mathsf{T}f\colon \mathsf{T}A \to \mathsf{T}B$ as an F_A-catamorphisms with $\mathsf{T}f = (\!|\varphi|\!)_{\mathsf{F_A}}$ s.t. the following diagram commutes:

$$\begin{tikzcd}
\mathsf{F}_A\mathsf{T}A \arrow[r, "\mathrm{in}_A"] \arrow[d, "\mathsf{F}_A\mathsf{T}f"] & \mathsf{T}A \arrow[d, "\mathsf{T}f = (\!|\varphi|\!)_{\mathsf{F_A}}"] \\
\mathsf{F}_A\mathsf{T}B \arrow[r, "\varphi"] & \mathsf{T}B
\end{tikzcd}$$

From Equation tyfunc-DEF we also know that

$$\varphi = \mathrm{in}_B \circ \mathsf{F}(f, \mathrm{id})$$

is exactly the algebra, s.t. $\mathsf{T}f\colon \mathsf{T}A \to \mathsf{T}B$ is a natural transformation.

How can we exploit this for IP? Assume we get two types $\mathsf{T}A$ and $\mathsf{T}B$ which are "instances" of the same polymorphic type. Think of `(Tree Int)` and `(Tree Bool)` and the polymorphic type `(Tree` α`)`, for instance. Given the specification Φ of a function $\mathsf{T}f :: \mathsf{T}A \to \mathsf{T}B$, to check whether $\mathsf{T}f$ is a natural transformation we just need to check for all rules $\rho \in \Phi(\mathsf{T}f)$ if for all positions $p \in (\mathcal{P}os\,(lhs(\rho)) \cup \mathcal{P}os\,(rhs(\rho)))$ either $lhs(\rho)|_p :: \mathsf{T}A \iff rhs(\rho)|_p :: \mathsf{T}B$ or $lhs(\rho)|_p :: A \iff rhs(\rho)|_p :: B$. This checks whether $\mathsf{T}f$ is really a structure preserving mapping, which maps any subterm on the left-hand side of type $\mathsf{T}A$ (A) to a subterm of type $\mathsf{T}B$ (B). We can express $\mathsf{T}f$ using the type functor, s.t. $\mathsf{T}f = (\!|\mathrm{in}_B \circ \mathsf{F}(f, \mathrm{id})|\!)_{\mathsf{F_A}}$ after abducing IO examples for f. Let us formally define an operator for IGOR II to introduce natural transformations.

Definition 6.3.1. Let $\langle P, \Phi\rangle$ be a candidate CS with corresponding specification, let Θ contain the corresponding type information, be $\rho\colon \mathsf{T}f(p_1, \ldots, p_n) = o$ a candidate rule in P with n arguments, and be $\Phi(\rho)$ the examples covered by ρ. Assume p_i to be of type $\mathsf{T}A$ with defined catamorphism. Let further be o of type $\mathsf{T}B$, and let $\Theta(\mathsf{T}A)$ and $\Theta(\mathsf{T}B)$ denote the initial F-algebras in_A and in_B, i.e. the sum of the constructors of $\Theta(\mathsf{T}A)$ and $\Theta(\mathsf{T}B)$, respectively. Let the operator for **natural transformation introduction** χ_{tyfunc} be defined as the possibly empty set, s.t.

$$\chi_{\mathrm{tyfunc}}(\rho, \Phi, \Theta) := \{\langle\{\rho'\} \cup P_{\mathcal{S}}, \phi_{\mathcal{S}}\rangle\},$$

if and only if for all rules $(\mathsf{T}f(s_1, \ldots, s_n) = t) \in \Phi(\rho)$, and all position $l \in (\mathcal{P}os\,(s|_i) \cup \mathcal{P}os\,(t))$ it holds that

$$s_i|_l :: \mathsf{T}A \Longleftrightarrow o|_l :: \mathsf{T}B \;\vee\; s_i|_l :: A \iff t|_l :: B,$$

then

- $\rho'\colon \mathsf{T}f(p_1, \ldots, p_n) = (\!|\mathrm{in}_B \circ \mathsf{F}(f(p_i, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n), \mathrm{id})|\!)_{\mathsf{F_A}}\; p_i,$
- $\phi_{\mathcal{S}} := \bigcup\{f(s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n, s_i|_l) = t|_l\}$ for all $(\mathsf{T}f(s_1, \ldots, s_n) = t) \in \Phi(\rho)$ and all position $l \in (\mathcal{P}os\,(s_i) \cup \mathcal{P}os\,(t))$ where $s_i|_l :: A \iff t|_l :: B$, and
- $P_{\mathcal{S}}$ is the initial candidate CS of $\phi_{\mathcal{S}}$.

Algorithm 13: Natural transformation introduction operator χ_{tyfunc}

input : an open rule $\rho\colon f(p_1, \ldots, p_n) = o$
input : a target specification Φ
input : corresponding type information Θ

output : a finite (possibly empty) set $\mathcal{S} = \{\langle P_l, \phi_l \rangle\}_{l \in \mathbb{N}}$ containing pairs of a successor rule set with corresponding new specification subsets

$\mathcal{S} \leftarrow \emptyset$
foreach $i \in [1..n]$ **do**
    $\phi_i \leftarrow \emptyset$
    $f_i \leftarrow$ a new defined function symbol, s.t. $\phi_j \notin (\mathcal{D}_{P \cup B} \cup \mathcal{C}_{P \cup B} \cup \mathcal{X})$
    if $p_i :: \mathsf{T}\alpha \;\wedge\; o :: \mathsf{T}\beta \;\wedge\;$ *defined* $(\!|\;|\!)_{\mathsf{T}\alpha}$ **then**
        foreach $(\mathsf{T}f(s_1, \ldots, s_n) = t) \in \Phi(\rho)$ *and each position* $l \in (\mathcal{P}os\,(s_i) \cup \mathcal{P}os\,(t))$ **do**
            if $s_i|_l :: \alpha \iff t|_l :: \beta$ **then**
                $\phi_i \leftarrow \phi_i \cup \{f(s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n, s_i|_l) = t|_l\}$
            else if $s_i|_l :: \mathsf{T}\alpha \iff t|_l :: \mathsf{T}\beta$ **then**
                `// noop`
            else
                $\phi_i \leftarrow \emptyset$
                break
            end
        end
    end
    $P_i \leftarrow$ `initialCandidate(`ϕ_i`)`
    $\mathrm{in}_\beta \leftarrow \Theta(\mathsf{T}\beta)$
    insert
     $\langle \{\mathsf{T}f(p_1, \ldots, p_n) = (\!|\mathrm{in}_\beta \circ \mathsf{F}(f_i(p_i, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n), \mathrm{id})|\!)_{\mathsf{F_A}}\; p_i\}$
        $\cup P_i, \phi_{\mathcal{S}_i} \rangle$
    into $\mathcal{S}$
end
return $\mathcal{S}$

6.3.3. Primitive recursion via Paramorphisms with χ_{para}

Section 4.2.5 defined paramorphisms and their relation to catamorphisms and already pointed out that their only difference is the amount of information available to their mediating functions. Thus, it is very easy to extend the operator χ_{cata} to a new operator for primitive recursion χ_{para}. Let, $\langle\!|\ \ |\!\rangle_\alpha$ denote the paramorphisms of a type α.

Definition 6.3.2. Let $\langle P, \Phi \rangle$ be a candidate CS and the corresponding specification, let Θ contain the corresponding type information, be $\rho\colon f(p_1, \ldots, p_n) = t$ a candidate rule in P with n arguments, and be $\Phi(\rho)$ the examples covered by ρ. Let p_i be any argument of type α, an inductive type with paramorphism $\langle\!|\ \ |\!\rangle_\alpha$ and type constructor $\Theta(\alpha) = \{c_1, \ldots, c_m\}$.

The polymorphic function $\langle\!|\ \ |\!\rangle_\alpha$ defines the paramorphism for the inductive type α and $[\varphi_1, \ldots, \varphi_m]$ to be the sum of the functions $\varphi_1, \ldots, \varphi_m$. The **primitive recursion operator** χ_{para} is exactly defined as χ_{cata} (cf. Definition 6.2.1), s.t.

$$\chi_{\text{para}}(\rho, \Phi, \Theta) := \{\langle \{\rho'\} \cup P_{\mathcal{S}}, \phi_{\mathcal{S}} \rangle\},$$

with the following differences:

1. $\rho'\colon f(p_1, \ldots, p_n) = \langle\!|\, [\varphi_1(p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n), \ldots, \varphi_m(p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n)]\, |\!\rangle_\alpha\ p_i,$

2. $\phi_{S_j} := \{\varphi_j(s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n, s_i') = o\}$
   for all $(f(s_i, \ldots, s_{i-1}, c_j(a_1, \ldots, a_k), s_{i+1}, \ldots, s_n)) = o) \in \Phi(\rho)_{c_j}$
   and $j \in [1..m]$, where $s_i' = (s_i, \mu(a_1), \ldots, \mu(a_k))$),
   and $s_i = c_j(a_1, \ldots, a_k)$.

So we (1) apply a paramorphism instead of a catamorphism, and (2) when abducing the IO examples from a rule $(f(s_i, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n)$, we simply carry the argument s_i over to the IO examples of the mediating functions.

One practical remark need to be made here. Catamorphisms reduce complexity by splitting up the argument. The operator χ_{para} however, rather adds more information in each step and may tend to clutter the mediating functions and the hypothesis more and more which leads to more algorithm loop cycles.

6.4. Igor II in Action — Illustrating Examples

The previous sections formally described the detection of type morphisms from both, the categorical and the term rewriting perspective, but without taking the inductive programming system Igor II into account. This section shows by three examples for the most common inductive types how, given examples for a specific target function, the catamorphism can be detected.

6.4.1. Filling a list with Zeros

An example of a simple catamorphism over natural numbers the following function which takes a number as input and returns a list containing the respective amount of zeros. Natural numbers, as previously, are represented as Peano's integers. List are defined as usual.

```
data [α] = [] | (α : [α]) --quasi Haskell
data Nat =  Z | (S Nat)
```

The IO examples for this simple function are the following.

```
nzeros :: Nat → [Nat]
nzeros Z               = []
nzeros (S Z)           = [Z]
nzeros (S(S Z))        = [Z,Z]
nzeros (S(S(S Z)))     = [Z,Z,Z]
nzeros (S(S(S(S Z)))) = [Z,Z,Z,Z]
```

As already described, the starting point for IGOR II is the least general generalisation of the given IO examples of `nzeros`:

```
nzeros :: Nat → [Nat]
nzeros x = y
```

The straight forward solution IGOR II would find without using catamorphisms applies the usual case distinction on the constructors together with a recursive call:

```
nzeros (Z)    = []
nzeros (S a) = Z : (nzeros a)
```

This scheme can be captured by our hand-crafted catamorphism on Peano integers from Listing 4.1.

```
nzeros :: Nat → [Nat]
nzeros a = foldn h c
    where
    c = []
    h = (Z:)
```

However, our aim is not to invent a type-specific catamorphism each type, but use a general polymorphic implementation, as e.g. that from the Pointless HASKELL library. The function `cata` is a generic, polymorphic implementation of catamorphisms on inductive data types (see Subsection A.6.3).

Listing 6.5: The function `nzeros` defined as catamorphism.

```
nzeros = cata ⊥ (c a ⊕ h a)
c _ = []
h x = (Z:x)
```

Although the latter is based on more general constructs than the further, both take a coproduct of functions as argument, which contains one function per input type's constructor.

Since we are in an IP context, we just need IO examples for `c` and `h` to induce them. The construction of these examples is guided by the type functor inducing the input type.

Consider the following commuting diagram, which depicts the current problem. All we need to do now is, figuratively speaking, to traverse both diagrams starting from *Nat* to $List_{Nat}$ to. The left diagram counterclockwise, the right clockwise. The rest is syntactic manipulation of terms. If we traverse it counterclockwise, we abduce the IO for the base case, i.e. function `c`, which needs to output the empty list when given `Zero` as input. To abduce the inputs for `h`, we first remove one constructor application of the corresponding input, evaluate the result on the IOs of `nzeros`, and relate this with the corresponding output of the original IO example.

$$\begin{tikzcd}
\mathbf{1} \arrow[r, bend right, "zero"'] \arrow[d, equal, "\mathrm{id}_{\mathbf{1}}"] & Nat \arrow[l, bend right, "zero^{-1}"'] \arrow[r, bend left, "succ^{-1}"] \arrow[d, "nzeros"] & Nat \arrow[l, bend left, "succ"] \arrow[d, "nzeros"] \\
\mathbf{1} \arrow[r, "c"] & List_{Nat} & List_{Nat} \arrow[l, "h"']
\end{tikzcd}$$

To solve `nzeros` using a polymorphic catamorphism, we proceed in three steps.

Applying $zero^{-1}$ **and** $succ^{-1}$ We part the original set of IO examples depending on the constructor at the root position of its input argument. Furthermore, we discard each constructor and keep only the product of its arguments, i.e. either the unit type if the input was already `Zero`, or the predecessor of the input number.

Resolving the functor N Terms of the target type in these products are replaced by the result of a recursive call to the target function, `nzeros` in this case.

If they consist of a constant constructor, they are replaced by the unit type.

Applying c **and** h Here we use IGOR II recursively to synthesise `c` and `h`.

IGOR II closes its initial rule from `nzeros`, by introducing the catamorohism and two, yet unknown, functions.

```
nzeros = cata (⊥ :: Nat) (c ⊕  h)
```

In subsequent iterations it will synthesise the functions `c` and `h` using the following abduced IO examples.

```
c ()    =          []

h []    =       [Z]
h [Z]   =     [Z,Z]
h [Z,Z] =   [Z,Z,Z]
```

The mediating argument functions `c` and `h` are very simply and Igor II can directly solve them by antiunification when constructing their initial rule. The final solution is shown in Listing 6.5.

6.4.2. The `length` of a List

Consider the problem of computing the length of a list. The data type definition of a list with elements of type α is standard. Either the list is empty or an element of type α is inserted at the front of an α-list (`[α]`). Natural numbers are represented as Peano's integers.

```
data [α] = [] | (α : [α]) --quasi Haskell
data Nat =  Z | (S Nat)
```

Four simple IO equations together with the type of the function specify the problem of computing the length of a list.

```
length :: [α] → Nat
length         []   =         Z
length      (a:[]) =      (S Z)
length    (a:b:[]) =    (S(S Z))
length (a:b:c:[]) = (S(S(S Z)))
```

Without catamorphisms, Igor II would find the following straight forward solution applying the usual case distinction on the constructors together with a recursive call:

```
length []     = Z
length (_:xs) = S (length xs)
```

A functional programmer, however, used to think in terms of recursion schemes and higher-order functions, maybe would come to an alternative solution. It is common to define `length` pointfree in terms of the higher-order function `foldr` (see Listing 4.2).

```
length = foldr h c
   where
   c     = Z
   h _ n = S n
```

As mentioned before, we would like to use a general polymorphic implementation, as shown in Listing 6.6 which uses functions from the pointless-haskell library.

Listing 6.6: The function `length` defined as catamorphism.

```
length = cata (⊥:: [α]) (c ⊕ h)
   where
   c   _   = Z
   h (_,n) = S n
```

As before, both take in fact a coproduct of functions as argument, which consists of as many functions as the input type's number of constructors. The last one is the solution we are aiming for, so lets see how Igor II can take advantage of detecting the list catamorphism when synthesising `length`.

Igor II starts with the least general generalisation of the given IO examples of `length`. From the type information we know that the input type of α-lists `[α]` is an inductive type, so catamorphisms are applicable. The list type has two constructors, so the mediating function of its catamorphism consists of a sum of two functions `c` and `h`. Since we are in an IP context, we just need IO examples for `c` and `h` to induce them. The construction of these examples is guided by the type functor inducing the input type.

Again, we argue on the following commuting diagram, which depicts the current problem. Recall that $List_A$ is the fixed point of the bifunctor L_A parameterised in A and defined as $\mathsf{L}_A X = \mathbf{1} + (A \times X)$ and $\mathsf{L}_A f\colon \mathrm{id}_{\mathbf{1}} + (\mathrm{id}_A \times f)$. All we need to do now is, figuratively speaking, to traverse both diagrams starting from $List_A$ to Nat. The left diagram counterclockwise, the right clockwise. The rest is syntactic manipulation of terms.

$$
\begin{tikzcd}[column sep=huge, row sep=huge]
\mathbf{1} \arrow[r, bend right, "nil"'] \arrow[d, equal, "\mathrm{id}_{\mathbf{1}}"] & List_A \arrow[l, bend right, "nil^{-1}"'] \arrow[r, bend left, "cons^{-1}"] \arrow[d, "length"] & A \times List_A \arrow[l, bend left, "cons"] \arrow[d, "\mathrm{id}_A \times length"] \\
\mathbf{1} \arrow[r, "c"] & Nat & A \times Nat \arrow[l, "h"']
\end{tikzcd}
$$

To solve `length` using a polymorphic catamorphism, we proceed in three steps.

Applying nil^{-1} **and** $cons^{-1}$ We part the original set of IO examples depending on the constructor at the root position of its input argument. Furthermore, we discard each constructor and keep only the product of its arguments.

Resolving the functor $\mathsf{L_A}$ Terms of the target type in these products are replaced by the result of a recursive call to the target function, `length` in this case. If they consist of a constant constructor, they are replaced by the unit type.

Applying c **and** h Here we use Igor II recursively to synthesise `c` and `h`.

Igor II closes its initial rule from `length`.

```
length = cata (⊥:: [α]) (c ⊕ h)
```

In subsequent iterations it will synthesise the functions `c` and `h` using the following abduced IO examples.

```
c ()            =         Z

h (a,Z)         =      (S Z)
h (a,(S Z))     =    (S(S Z))
h (a,(S(S Z))) = (S(S(S Z)))
```

However, we can easily see that both functions are already solved when they are antiunified for the initial hypothesis. We have shown the final result already in the beginning of this section in Listing 6.6.

6.4.3. Mirroring Binary Trees

Consider another recursive problem to mirror binary trees, where either the tree is empty or it contains a node holding an element of type α and two subtrees.

```
data Tree α = E | N α (Tree α) (Tree α)
```

Again, four simple equations specify the problem of mirroring a tree.

```
mirror  :: (Tree α) →(Tree α)
mirror  E                            = E
mirror (N a E E)                     = (N a E E)
mirror (N b (N a E E) (N c E E))     =
       (N b (N c E E) (N a E E))
mirror (N d (N b (N a E E) (N c E E))
            (N f (N e E E) (N g E E))) =
       (N d (N f (N g E E) (N e E E))
            (N b (N c E E) (N a E E)))
```

The next diagram depicts the problem. The data type $Tree_A$ is induced by the parameterised bifunctor $\mathsf{B}_A X = \mathbf{1} + (A \times X \times X)$ and $\mathsf{B}_A f = \mathrm{id}_{\mathbf{1}} + (\mathrm{id}_A \times f \times f)$.

The constructors `E` and `N` correspond to the function $empty_E$ and $node_E$ and `mirror` is the function $f = (\![[g, h]]\!)$.

$$\begin{tikzcd}
\mathbf{1} \arrow[r, bend left, "empty_E"] \arrow[d, equal, "id_{\mathbf{1}}"] & Tree_E \arrow[l, bend left, "empty_E^{-1}"] \arrow[d, "f"] \arrow[r, bend right, "node_E^{-1}"'] & E \times Tree_E \times Tree_E \arrow[l, bend right, "node_E"'] \arrow[d, "id_E \times f \times f"] \\
\mathbf{1} \arrow[r, "g"'] & Tree_E & E \times Tree_E \times Tree_E \arrow[l, "h"]
\end{tikzcd}$$

We proceed in the same manner as described above. We resolve the arrows by traversing the diagram.

Applying $empty_E^{-1}$ **and** $node_E^{-1}$ We part the original set of IO examples depending on the constructor at the root position of its input argument and apply the inverse constructors to the inputs.

Resolving the functor B_E Terms of type `Tree α` are replaced by the result of a recursive call to the target function, `mirror` in this case. If they consist of a constant constructor, they are replaced by the unit type.

Applying g **and** h Here we use IGOR II recursively to synthesise `c` and `h`.

The result are the following sets of IO examples for `g` and `h`. The function g is becomes a constant function, solved per definition, and constituting the base case. For h, the outputs remain the same, but the inputs are turned into nested tuples where *all tree-subterms were replaced by there mirror image*, i.e. the result of a recursive call.

```
g ()                              =  E

h (a, (E,E))                      = (N a E E)
h (b, ((N a E E), (N c E E)))     =
  (N b (N c E E) (N a E E))
h (d, ((N b (N c E E) (N a E E))
    , (N f (N g E E) (N e E E)))) =
  (N d (N f (N g E E) (N e E E))
       (N b (N c E E) (N a E E)))
```

Computing the LGG of all examples of h reveals that it is solved too, because all variables are already bound. The function `mirror` can now be written using a generic implementation of catamorphisms on inductive data types.

```
mirror t          = cata (⊥ ::(Tree α)) (g ⊕ h) t
    where
    g     _          = E
    h (x, (t₁, t₂)) = N x t₂ t₁
```

6.4.4. A Synthesis Example revisited

The synthesis example in Section 5.4 used only the four operators of the base algorithm as defined in Chapter 5. Using the new operator χ_{cata} would have changed the synthesis process and the result completely. Continuing the example synthesis, the following shows how the use of the new operator changes the algorithm behaviour.

6.4.4.1. Alternative Iteration 1

We enter again, directly after constructing the initial candidate H_0 in Listing 5.11 from Section 5.4, into the first iteration. Checking the conditions for χ_{cata}, we easily see that `lasts` is defined for the empty list `[]`. Now we need to find a function $\texttt{fun}_3$ such that `lasts (x:xs) = fun x (lasts xs)` for each example equation covered by the pattern on the LHS. We see that this is true for all example equations, namely $\{1\ldots10\}$. Listing 6.7 shows the accordingly abduced example equations for our new auxiliary rule.

Listing 6.7: $\texttt{fun}_3$

```
fun₃ [a]      []  =  a:[]
fun₃ [a,b]    []  =  b:[]
fun₃ [a,b,c]  []  =  c:[]
fun₃ [b]     [a]  =  b:[a]
```

```
fun3 [c]      [b]   =  c:[b]
fun3 [c,d]    [b]   =  d:[b]
fun3 [a,b]    [d]   =  b:[d]
fun3 [c]    [e,f]   =  c:[e,f]
fun3 [c,d][f,g]     =  d:[f,g]
```

Already now, we could rewrite the initial rule to `lasts x = foldr` $\texttt{fun}_3$ `[] x`, but we can simplify it even more. Note that the second argument of $\texttt{fun}_3$ occurs unchanged in the output, so only the first argument is modified. The formatting of $\texttt{fun}_3$ is supposed to illustrate this. The first argument of $\texttt{fun}_3$ came from the first element in the argument list of `lasts`, the second argument is the result of the recursive call with the rest list. Thus, $\texttt{fun}_3$ modifies always the first element and inserts it at the front of the result list of the recursive call. This is exactly how the function `map` is defined. Therefore, it is admissible to always ignore the second argument for the auxiliary function $\texttt{fun}_3$ in an alternative function $\texttt{fun}_3^*$. Its IO examples are shown in Listing 6.8.

Listing 6.8: $\texttt{fun}_3^*$

```
fun3* [a]      = a
fun3* [a,b]    = b
fun3* [a,b,c]  = c
fun3* [b]      = b
fun3* [c]      = c
fun3* [c,d]    = d
fun3* [a,b]    = b
fun3* [c]      = c
fun3* [c,d]    = d
```

Instead of `foldr` we now use `map` to rewrite the initial rule and create a new hypothesis.

Listing 6.9: $H_{1,alt} : \chi_{\text{cata}} \overset{1}{\circ} \chi_{\text{init}}$

```
lasts   x     = map fun3* [] x
fun3* (x:xs) = y
```

6.4.4.2. Alternative Iteration 2

It is apparent from the example equations that $\texttt{fun}_3^*$ is `last`. So IGOR II will detect a call to the background knowledge as similarly already described in the third iteration (5.4.3). We spare the details and finish with the final solution (Listing 6.10) which is output at the beginning of the third iteration.

Listing 6.10: $H_{2,alt} : \chi_{\text{direct},\texttt{last}} \overset{2}{\circ} \chi_{\text{cata}} \overset{1}{\circ} \chi_{\text{init}}$

```
lasts   x     = map fun3* [] x
fun3* (x:xs) = last (x:xs)
```

7. Evaluation

In the last section, we've described an extension of the Igor II algorithm to detect the applicability of recursive schemes in the given set of IO examples. To avoid ambiguities, let Igor II refer to the original Igor II algorithm as described by Kitzelmann [66] and let Igor II_H refer to its re-implementation in Haskell. Igor II^+ denotes the Haskell implementation using the extensions as described in Chapter 6. If it is necessary to be more precise Igor II_C^+ denotes the extension using catamorphisms and Igor II_P^+ denotes the extension with paramorphisms. Now the extended system will be evaluated to provide evidence for the resulting improvements in efficiency and expressiveness.

Section 7.1 explains the design of the empirical evaluation, justifies the choice of the benchmark systems, explains necessary settings and options of those systems, describes the aim of this benchmark study and introduces the benchmark problems and the system specific specifications. Section 7.2 presents the empirical results and comments on the runtimes, efficiency and successes of the different systems. All synthesised programs of the specific systems are shown in Appendix F. Section 7.3 mentions some ideas for further implementations of the algorithm and summarises the improvements of Igor II_C^+ and Igor II_P^+ w.r.t. the old algorithm Igor II_H.

7.1. The Evaluation Design

An impartial empirical evaluation of IP systems is quite difficult, because both, the language and the restriction bias of two IP systems may be completely different and consequently the amount of additional information, like types, background knowledge, or even mode declarations for predicates in ILP which determine the bound and unbound variables, varies from system to system.

In various previous, elaborate studies and empirical evaluations, we've already compared Igor II/Igor II^+ with other standard IP systems in a unifying framework which tries to take each system's individual peculiarities into account [51, 52, 54]. The results were quite clear. In general, Igor II needs less information than other IP systems, i.e. just the examples and the type information. It is faster or about as fast as and even more powerful than the traditional ILP systems and other analytic approaches. Of course, search-based, and especially evolutionary approaches, are much more powerful, but for problems which lie within the scope of Igor II they where substantially slower.

Our focus of interest lies on IP systems which may adopt a schema-based language bias and which make additionally use of supplementary knowledge, especially type information or user hints, to apply these schemes. Section 2.2.3 has already introduced those systems for which this applies. However, Igor II^+ is, to our present knowledge, the only IP system that is able to apply a program scheme automatically, without specific user interaction or explicit inclusion. Dialogs-II requires the user to choose a scheme and MagicHaskeller, PolyGP, and G∀st could only use them if they are provided explicitly in their specification or if they are hard coded into their language bias.

In fact, MagicHaskeller is the only system able to synthesise programs with about as little knowledge as Igor II/Igor II$_\mathrm{H}$ in general. The previously mentioned evaluations have already shown that Igor II outperforms Dialogs-II w.r.t. expressiveness and time efficiency. For PolyGP or G∀st, it is necessary to specifically adjust the language bias to a certain problem to produce satisfying results at all. If the language bias is too general, in fact as general as it is for MagicHaskeller or Igor II/Igor II$_\mathrm{H}$, both systems struggle with the combinatorial explosion of the search space.

7.1.1. Benchmark Systems and Settings

Due to the arguments mentioned above, the performance of the new Igor II extensions, i.e. Igor II$^+$, will be evaluated against two benchmarks.

The first is the Igor II$_\mathrm{H}$ system without the new extensions for type morphisms introduction. The more efficient re-implementation in Haskell will be used here. It is clear that this depends not on the implementation but on the language choice: A compiled Haskell program is faster than interpreted Maude code (cf. Hofmann et al. [53] for a comparison). Its core algorithm, as described in Chapter 5, corresponds to Kitzelmann's [66] with some deviations, which are described in the following.

Maximum depth The operator χ_subfn is cost-neutral and does not increase the costs associated with a hypothesis. Kitzelmann uses a maximum depth to cap the maximum number of subfunction introductions to prevent infinite sequences of subfunction applications. In the Haskell re-implementation such a maximum bound does not exist. However, as described in Section 5.2, the number of rules in a hypothesis is included into the cost function, i.e. χ_subfn is not cost neutral anymore. If two hypotheses have the same number of patterns, the hypothesis with the least number of rules is preferred.

Rapid rule-splitting extension This extension, as proposed by Kitzelmann, has been included in Igor II$_\mathrm{H}$ under the name **greedy rule splitting**. Assume a lhs has more than one pivot position as required by the χ_split operator (cf. Definition 5.3.1). The usual way is to create for each pivot position one hypothesis partitioning the examples w.r.t. this position. Greedy rule splitting creates only one hypothesis, which however induces a partitioning w.r.t. all pivot positions.

Conditional equations This extension is not supported by Igor II$_\mathrm{H}$.

Extensional quantified variables Not supported by Igor II$_\mathrm{H}$.

If not stated differently, no specific settings were made and the default argument-wise reduction order (cf. Section 5.3.3) is used.

The second benchmark system is MagicHaskeller, which exhaustively, but still efficiently, enumerates de-Bruijn-style λ-expression in the strongly typed language Haskell (cf. Section 2.2.3.5). Its specification consists of the constructors of all used data types and a type specific recursion scheme (catamorphism or paramorphism), which may also be used for case distinctions and data type deconstruction.

7.1.2. Purpose of Benchmarks

Comparing Igor II$^+$ against Igor II$_\mathrm{H}$ aims to show the improvements in efficiency and expressiveness that can be achieved with data-driven detection of program schemes. To assess the improvement in efficiency the iterations needed to solve a specific problem will be compared. The runtimes in both cases lie in most cases within the range of milliseconds, and therefore differences measured in fractions of seconds are considered as not conclusive enough.

Comparing Igor II$^+$ against MagicHaskeller aims to underpin the advantage of an analytical, data-driven strategy compared to mere enumeration. It is intuitively clear that systematic search should be more efficient than enumeration. However, one can argue, that with syntactically simple problems there should not be a big difference.

Furthermore, contrary to Igor II$_\mathrm{H}$, MagicHaskeller is unable to invent auxiliary subfunctions, i.e. use functions that were not provided by the user in advance. Comparing the analytical Igor II$_\mathrm{H}$ with function invention (FI) against the enumerative MagicHaskeller without function invention ($\overline{\mathrm{FI}}$) on the one side, and the use of catamorphisms and paramorphisms on the other side splits up in six test scenarios depicted in Table 7.1. One can already be excluded from the beginning, because using MagicHaskeller without any recursion schemes at all does not make much sense.

Concerning Igor II$^+$ using paramorphisms, another remark in necessary. The use of paramorphisms is still experimental. Although, their introduction similar to catamorphisms is implemented and works correctly, their use requires a more sophisticated reduction order. Recall from Section 4.2.5 that the difference between catamorphisms and paramorphisms is the amount of information available to a recursive call. In fact, in a recursive call to a paramorphism all subterms of the original input are available. An invented subfunction may at this point easily recombine them to the original input, and thus causing a non-terminating hypothesis. This would require a more complex reduction order keeping track of the arguments used for morphisms, but this has not been solved yet. A lot solutions with correctly introduced paramorphisms are expected to use subfunctions which lead to non-termination.

7.1.3. Benchmark Problem Specifications

As it has been already pointed out in the beginning, it is quite difficult to compare different IP systems based on identical specifications. The amount of knowledge presented to both systems has been kept as comparable and equal as possible. Both were given the same set of IO examples: Igor II$_\mathrm{H}$ as equations and MagicHaskeller as

	$\times$	$(\!\vert\cdot\vert\!)$	$\langle\!\vert\cdot\vert\!\rangle$
FI	IGOR II_{H}	IGOR II_{C}^{+a}	IGOR II_{P}^{+b}
$\overline{\text{FI}}$	—	MH using $(\!\vert\cdot\vert\!)$	MH using $\langle\!\vert\cdot\vert\!\rangle$

No recursion scheme ($\times$), cata- ($(\!|\cdot|\!)$), and paramorphism ($\langle\!|\cdot|\!\rangle$)
MH MAGICHASKELLER
With (FI) and without $\overline{\text{FI}}$ function invention
[a]Option `:s +enhanced` [b]Option `:s +enhanced, :s +para`

Table 7.1.: Classify systems into test scenarios.

a higher-order test function, accepting a function as input and returning whether this function computed all IOs correctly.

IGOR II_{H} gets the whole specification with all used data types and type class instance declarations as a valid HASKELL module. With this specification IGOR II_{H} naturally had already all necessary information about the type constructors and recursion schemes available. When requested to solve a particular problem, eventually with some background knowledge, no problem specific adoption is necessary.

The same information was provided to MAGICHASKELLER explicitly within its library, which was changed depending on a specific problem, though. To compensate for pattern matching, which MAGICHASKELLER is natively unable to perform, for each type either a recursion scheme (catamorphisms, or paramorphisms) or all relevant deconstructors, as in the case of tuples, have been provided.

After loading the specification of all problems, IGOR II_{H} synthesised them all at once in batch mode. MAGICHASKELLER was given as much problems at once as possible, but it tends to run out of memory quickly. This is aggravated by polymorphic functions as for example the pair function `(,)`, because in fact it can in always be used. Therefore, contrary to IGOR II_{H}, for each problem it was given an individual library set, consisting of only those constructors and schemes of those types actually occurring in the IO examples. IGOR II_{H} always had all type information of the whole specification available, but used only as much as necessary to solve the problem. See Appendix C and Appendix D for MAGICHASKELLER's and IGOR II_{H}'s specification, respectively.

MAGICHASKELLER performs memoization during synthesis and caches subexpressions. On the one side this may speed up the generation of subsequent programs. On the other side the cache is never emptied. Thus, the chance to run out of memory increases over time. Therefore, if one problem ran out of memory in batch mode, it was tested alone again. Only if finally all available memory did not suffice, it is stated.

MAGICHASKELLER version 0.8.5 from `hackageDB :: [Package]`[1] and IGOR II_{H} version 0.8.0 from the homepage of the DFG-project *Effiziente Algorithmen zur induktiven Programmsynthese*[2] (SCHM 1239/6 10/2007 – 9/2011) have been used. The manual of the IGOR II_{H} system is shown in Appendix E.

[1]`http://hackage.haskell.org/package/MagicHaskeller`
[2]`http://www.cogsys.wiai.uni-bamberg.de/effalip/download.html`

The benchmark problems itself are a collection of mostly recursive programs using the most frequent inductive data types such as natural numbers represented as Peano integers, lists, and binary (node) trees. Some user-defined types for specific problems or mixed types are also included, as well as classification problems on tuples. Many of these function are standard functions present in base packages of functional languages. Others are common benchmarks used in IP and ILP context. It is noteworthy that our focus was not on complex problems in the sense of program size, but in the sense of structural or intellectual complexity. For example the function `evenParity` is not a big program w.r.t. lines of code, nevertheless its recursive solution is not immediately apparent in the first place.

Table 7.2 shows the name, the type, and a short description of each problem function. The following paragraphs provide some additional explanations for selected problems.

Functions on mixed inputs The function `pepper` originated from a colloquium held at the group ÜBB of Prof. Peter Pepper at TU Berlin. It annotates all elements with an index and the index of its predecessor. The variant `pepperF` only adds the index.

Listing 7.1: Definition of `pepper`

```
pepper :: Nat → [a] → [(Nat, Maybe (a,Nat))]
pepper i [] = [(i,Nothing)]
pepper i (x:xs) = (i,Just (x,S i)):(pepper (S i) xs)

pepperF = (map onlyIdx ∘ ) ∘ pepper
   where
   onlyIdx =
     λ(l,r) → (l, maybe (Just ∘ fst) Nothing r)
```

Functions on other data types The functions in this group are inspired by *cognitive psychology*, *human problem solving*, *natural language processing* and *AI planning* and were part of a joint publication with Ute Schmid and Emanuel Kitzelmann [120] and have been elaborated in more detail in [66].

Humans' *problem solving*, reasoning and verbal behaviour often show a high degree of systematication and productivity which can best be characterised by a competence level reflected by a set of recursive rules. Speed-up effects in problem solving are explained by a general rule acquisition mechanism, which extracts such rules from only positive examples from the environment. In [120] analytical inductive programming has been suggested as a model of such a rule acquisition device. Similarly, in AI planning macro learning was modelled as composition of primitive operators to more complex ones.

The function `rocket` is a simple planning benchmark to illustrate the so called *Sussman anomaly*, a weakness of noninterleaved planning. The problem is to transport a number of objects from earth to moon where the *rocket* can only fly in one direction. That is, the problem cannot be solved by first solving the goal "*object 1 on the moon*" by loading it, moving it to the moon and then unloading it. Because with this strategy there is no possibility to transport further objects from earth to moon. The correct procedure is first to load all objects, then to fly to the moon and finally, to unload the objects. The state is modelled by an inductive data type representing the action which can be performed by a planner: Loading (`LOD`), Flying (`FLY`), Unloading (`UNL`). The `Cargo` is a list of objects. The function `rocket` takes a cargo list and some state as input and returns a state in which the appropriate actions have been performed. Listing 7.2 shows the data type definitions and the examples.

Listing 7.2: Data types and examples for the `rocket` problem.

```
data State = START | LOD Object State
           | UNL Object State | FLY State
data Object = O1 | O2 | O3 | O4
data Cargo = NOCARGO | IN Object Cargo

rocket :: Cargo → State → State
rocket NOCARGO                      s = FLY s
rocket (IN x NOCARGO)               s =
  UNL x (FLY (LOD x s))
rocket (IN x (IN y NOCARGO))        s =
  UNL x (UNL y (FLY (LOD y (LOD x s))))
rocket (IN x (IN y (IN z NOCARGO))) s =
  UNL x (UNL y (UNL z (
     FLY (LOD z (LOD y (LOD x s))))))
```

The function `sentence` models the problem of learning a phrase-structure grammar. Learning word-category associations has been avoided and the provided examples abstract from concrete words. In particular, the function generates words (or sentences) of the target grammar of particular depths. Figure 7.1 shows the grammar to be learned and the corresponding examples that were provided as specification are shown in Listing 7.3. The abstract example sentence structures correspond to sentences as described by Covington [20]:

1. *The dog chased the cat.*

2. *The girl thought the dog chased the cat.*

3. *The butler said the girl thought the dog chased the cat.*

The *Towers of Hanoi* puzzle is modelled by a function taking a stack of discs and three pegs (source, auxiliary, and target) and returning a sequence of `Move` actions. Concerning the discs a small semantical trick has been used. Whereas `(D(D D0))` represents a stack of discs, with the smallest disc `D0` on top and the largest `(D(D D0))` right at the bottom, when given as input of `hanoi`, it represents just the largest disc, when given as input to the move action `MV`. For example line 9 and following in Listing 7.4 reads as follows. The sequence of corresponding actions is read backwards.

S	$\to$	NP VP
NP	$\to$	D N
VP	$\to$	V NP | V S

non-terminal	:	**S**entence, **N**oun **P**hrase, **V**erb **P**hrase
terminal	:	**D**eterminant, **V**erb, **N**oun

Figure 7.1.: A phrase-structure grammar for the function `sentence`

To solve the problem of moving the stack of two discs from peg `src` to peg `dst` using peg `aux` in some context `s`, after moving the smallest disc `D0` from `src` to `aux` in some context `s`, move the disc next in size `D D0` from `src` to `dst`, and finally move the smallest disc `D0` from `aux` to `dst`.

Listing 7.3: IO examples for the function `sentence`

```
sentence :: Nat → [Char]
sentence Z         = [ 'D', 'N', 'V', 'D', 'N']
sentence (S Z)     = [ 'D', 'N', 'V', 'D', 'N'
                     , 'V', 'D', 'N']
sentence (S( S Z)) = [ 'D', 'N', 'V', 'D', 'N'
                     , 'V', 'D', 'N', 'V', 'D', 'N']
```

Listing 7.4: Data types and examples for the hanoi problem.

```
data Disc = D0 | D Disc
data Action = NOOP | MV Disc Peg Peg Action
data Peg = PegA | PegB | PegC

hanoi :: Disc → Peg → Peg → Peg
      → Action → Action
hanoi D0 src aux dst s       = MV D0 src dst s
hanoi (D D0) src aux dst s   =
   MV D0 aux dst
    (MV (D D0) src dst
     (MV D0 src aux s))
hanoi (D(D D0)) src aux dst s =
   MV D0 src dst
    (MV (D D0) aux dst
     (MV D0 aux src
      (MV (D(D D0)) src dst
       (MV D0 dst aux
        (MV (D D0) src aux
         (MV D0 src dst s))))))
```

UCI repository The problems in this category are data sets from the UCI machine learning repository[3]. In these standard noise-free classification problems the task is to learn a multi-valued class attribute based on a nominal attribute vector. The concepts `enjoySports` and `playTennis` are from Mitchell [89]. Of course, those concepts are originally non-recursive, but IGOR II will demonstrate that those problems can be learned based, on a small set of examples, as a special case.

Table 7.2.: Description of functions

	functions on natural numbers	
`ack`	`Nat → Nat → Nat`	The Ackermann function.
`add`	`Nat → Nat → Nat`	Addition on natural numbers.
`even`	`Nat → Bool`	Is the number even?
`eq`	`Nat → Nat → Bool`	Equality on natural numbers.
`gaussSum`	`Nat → Nat`	Sum of all naturals from 0 to n.
`fact`	`Nat → Nat`	The factorial function.
`fib`	`Nat → Nat`	The n^{th} number in the Fibonacci sequence.
`geq`	`Nat → Nat → Bool`	Greater-or-equal.
`mult`	`Nat → Nat → Nat`	Multiplication on naturals.
`odd`	`Nat → Bool`	Is the number odd?
`sub`	`Nat → Nat → Nat`	Subtraction on natural numbers.
	predicates, functions on Booleans	
`andL`	`[Bool] → Bool`	Conjunction of a lists of Booleans.
`and`	`Bool → Bool → Bool`	Conjunction of two Booleans.
`evenParity`	`[Bool] → Bool`	Check whether the number of `True` elements is even.
`negateAll`	`[Bool] → [Bool]`	The complement of all Booleans in a list.
`nandL`	`[Bool] → Bool`	Negated conjunction of a lists of Booleans.
`norL`	`[Bool] → Bool`	Negated disjunction of a lists of Booleans.
`or`	`Bool → Bool → Bool`	Disjunction of two Booleans.
`orL`	`[Bool] → Bool`	Disjunction of a list of Booleans.
	functions on lists	
`append`	`[a] → [a] → [a]`	Appending two lists.
`evenLength`	`[a] → Bool`	Is the length of the list even?

Continued on next page

[3] http://archive.ics.uci.edu/ml/

Table 7.2 – continued from previous page

Name	Type	Description
`evenpos`	`[a] → [a]`	Select all elements at even positions.
`halves`	`[a] → ([a],[a])`	Splits a list in two halves.
`init`	`[a] → [a]`	Removes the last element from a list.
`inits`	`[a] → [[a]]`	All initial segments of the input list, shortest first.
`intersperse`	`a → [a] → [a]`	Intersperses the given element between all two consecutive elements in the list
`last`	`[a] → a`	The last element of a list.
`lastM`	`[a] → Maybe a`	`last`, defined as total function.
`multfst`	`[a] → [a]`	Replaces all elements by the first one.
`multlst`	`[a] → [a]`	Replaces all elements by the last one.
`oddpos`	`[a] → [a]`	Selects all elements at odd positions.
`pack`	`[a] → [[a]]`	Wraps all elements into singletons.
`subseqs`	`[a] → [[a]]`	All subsequences of a list, aka power set on lists.
`reverse`	`[a] → [a]`	Reverses a list.
`shiftl`	`[a] → [a]`	Shifts all elements to the left, by inserting the first at the end.
`shiftr`	`[a] → [a]`	Shifts all elements to the right, by inserting the last at the front.
`snoc`	`a → [a] → [a]`	Inserts an element at the end.
`swap`	`[a] → [a]`	Swaps every two subsequent elements.
`switch`	`[a] → [a]`	Switches the first with the last element.
`split`	`[a] → ([a],[a])`	Computes the lists of elements at odd and even positions.
`tail`	`[a] → [a]`	Removes the first element.
`tails`	`[a] → [[a]]`	`map tail`
`unzip`	`[(a,a)] → ([a],[a])`	Computes the list of first and second projections.
`weave`	`[a] → [a] → [a]`	Combines two lists by interleaving their elements.
`zip`	`[a] → [a] → [(a,a)]`	Computes the list of corresponding pairs.
functions on lists of lists		
`concat`	`[[a]] → [a]`	Concatenates all lists.
`lasts`	`[[a]] → [a]`	`map last`
`mapCons`	`a → [[a]] → [[a]]`	Inserts the element at front of each list.
`mapTail`	`[[a]] → [[a]]`	`map tail`

Continued on next page

Table 7.2 – continued from previous page

Name	Type	Description
`transpose`	`[[a]] → [[a]]`	Transposes a matrix.
`weaveL`	`[[a]] → [a]`	Turns a matrix into a list by appending its columns.
	functions on naturals and lists	
`addN`	`Nat → [Nat] → [Nat]`	Increments all elements by a given number.
`alleven`	`[Nat] → Bool`	Are all numbers even?
`allodd`	`[Nat] → Bool`	Are all numbers odd?
`evens`	`[Nat] → [Nat]`	Selects all even numbers.
`incr`	`[Nat] → [Nat]`	Increments all numbers in a list by one.
`length`	`[a] → Nat`	The length of a list.
`lengths`	`[[a]] → [Nat]`	`map length`
`nthElem`	`[a] → Nat → a`	Returns the n^{th} element.
`oddslist`	`[Nat] → Bool`	Are all elements odd?
`odds`	`[Nat] → [Nat]`	Selects all odd elements.
`drop`	`Nat → [a] → [a]`	Drops the first `n` elements of a list
`splitAt`	`Nat → [a] → ([a],[a])`	Splits a list before a given position.
`sum`	`[Nat] → Nat`	The sum of a list of integers.
`replicate`	`a → Nat → [a]`	A list of length `n` containing only the given element.
`take`	`Nat → [a] → [a]`	Takes the first n elements.
`zeros`	`[Nat] → [Nat]`	Removes all non-zero integers from a list.
	functions on trees	
`preorder`	`(NTree a) → [a]`	Preorder traversal of a binary tree.
`inorder`	`(NTree a) → [a]`	Inorder traversal of a binary tree.
`postorder`	`(NTree a) → [a]`	Postorder traversal of a binary tree.
`mirror`	`(NTree a) → (NTree a)`	Mirrors a binary tree by swapping all its subtrees.
	functions on mixed inputs	
`pepper`	`Nat → [a] → [(Nat,Maybe (a,Nat))]`	Annotates each element with an index and the index of its predecessor.
`pepperF`	`Nat → [a] → [(Nat,Maybe a)]`	Indexes all elements starting by the given number.

Continued on next page

Table 7.2 – continued from previous page

Name	Type	Description
	functions on other data types	
`rocket`	`Cargo → State → State`	The planning problem of loading a rocket and flying it to the moon.
`hanoi`	`Disc → Peg → Peg → Peg → Action → Action`	Recursive definition of The Tower of Hanoi problem.
`sentence`	`Nat → [Char]`	Enumerating all sentences of a grammar (cf. Figure 7.1).
	functions for UCI classification problems	
`balloons`	`(Color, Size, Act, Age) → Inflate`	UCI classification problem
`playTennis`	`(Weather, Weather, Weather, Weather) → Bool`	Classification problem [89]
`enjoySport`	`(Weather, Weather, Weather, Weather, Weather, Weather)`	Classification problem [89]
`lenses`	`(LAge, LPrescription, LAstigmatic, LTears) → LCLType`	UCI classification problem

7.2. Empirical Results

This section presents the results of the empirical analysis. All tests have been conducted under Ubuntu 7.10 on an Intel Dual Core 2.33 GHz with 4GB memory. No test has been stopped abortively, but run until it either produced a result or run out of memory.

Kitzelmann already showed that IGOR II can use background knowledge. Therefore, it was only given if necessary, i.e. if IGOR II could not abduce sufficient IO examples to learn it by itself. Consider for example the function `postorder` for postorder tree traversal. Listing 7.5 shows the abduced IOs of a required auxiliary function IGOR II needs to invent. In fact, `fun` is the function `append` to concatenate lists. However, these IOs are insufficient for IGOR II to learn `append`. IGOR II cannot detect recursive regularities, because these are not the first k examples w.r.t. the input data type list. It cannot find evidence how to solve `fun [a] []` for example, which is crucial to detect a recursive regularity. Therefore, `append` has been provided as background knowledge.

Name	Sections	Options
IGOR II_H	5	`:s simplify`
IGOR II_C^+	5, 6.2, 6.3.1	`:s simplify; :s +enhanced`
IGOR II_P^+	5, 6.3.3	`:s simplify; :s +enhanced; :s +para`

Table 7.3.: Different flavours of IGOR II_H and their settings.

Listing 7.5: Abduced IOs for auxiliary function in `postorder`

```
fun :: [a] → [a] → [a]
fun [] []           = []
fun [a] [b]         = [a,b]
fun [a,b] [c,d]     = [a,b,c,d]
fun [a,b,c][d,e,f]  = [a,b,c,d,e,f]
```

Some functions have been synthesised simultaneously by IGOR II_H. This is indicated by multiple target function names in the tables.

As already mentioned, IGOR II_H was run three times with different settings: Once without type morphism, once with catamorphisms and type functors on lists, and finally only with paramorphisms. Note, that the use of type morphisms is still greedy. If the algorithm detects applicability of a morphism, it tries to use it without backtracking. Table 7.3 gives an overview of the different IGOR II_H implementations, their options, and which section describes their algorithm. IGOR II_H represents the algorithm as described in Section 5. IGOR II_C^+ is equivalent to the original algorithm from Section 5 with extensions for catamorphisms as described in Section 6.2, special treatment of lists catamorphisms and list type functors as described in Section 6.3.1. IGOR II_P^+ extends IGOR II_H only with the algorithm described in Section 6.3.3.

Subsection 7.2.1 compares the different flavours (plain vanilla, catamorphisms, paramorphisms) of IGOR II_H with MAGICHASKELLER using paramorphisms and MAGICHASKELLER using catamorphisms.

With runtimes below one second IGOR II_H is already quite fast. Improvements due to the new extension lie in the range of milliseconds which is not a reliable indicator to measure the improvement. Therefore, Section 7.2.2 compares the number of algorithm loop cycles of the different IGOR II_H version.

7.2.1. Runtimes and Success

This section comments the results shown in Table 7.5. For each example function, the systems with the fastest result are highlighted with a green background. If additional options were required to successfully synthesise the function, it is indicated by indices and explained at the bottom of the table.

Table 7.4 summarises the outcome with simple statistical measures over all correctly synthesised problems. It states the total time, maximum and minimum individual runtime, mean, median, standard deviation, the number of correctly synthesised programs,

Name	Σ	*max*	*min*	$\varnothing$	$\tilde{x}$	σ	$\oplus$	$^{\oplus}/_{N}$
I_H	40.5067	17.3651	0.0001	0.5957	0.0080	2.8406	68	80.95%
I^+_C	25.5419	16.1490	0.0001	0.3361	0.0001	1.9472	76	90.48%
I^+_P	310.1139	161.7141	0.0001	5.6384	0.0040	28.7866	55	65.48%
H_P	754.4210	272.6490	0.0010	13.9708	0.2460	49.0317	54	64.29%
H_C	572.5930	167.0460	0.0001	12.7243	0.1880	34.8618	45	53.57%

I_H Igor II$_\mathrm{H}$ · I^+_C Igor II^+_C · I^+_P Igor II^+_P
H_P MagicHaskeller with $\langle\!|\cdot|\!\rangle$ · H_C MagicHaskeller with $(\!|\cdot|\!)$

Σ total runtime *max* maximum
min minimum $\varnothing$ mean $\tilde{x}$ median
σ standard deviation $\oplus$ absolute successes $^{\oplus}/_{N}$ success rate

Table 7.4.: Runtime statistics for the different systems in seconds based on the number of successfully synthesised programs of totally $N = 84$ problems.

and the success rate. It is quite clear that Igor II outperforms MagicHaskeller, both in time efficiency and success rate. Even the experimental paramorphism feature is better than MagicHaskeller. Compared to Igor II$_\mathrm{H}$, Igor II$^+$ shows significant improvements on nearly all measures. The overall runtime was nearly halved from 40.5 seconds to 25.5 seconds. The mean runtime was reduced by about a third from 2.8 seconds to 1.9 seconds. A reduction of the dispersion of runtimes could decrease the median runtime to 0.0001s. Ultimately, the success rate could be raised by ten percentage points. The relatively poor performances of Igor II^+_P arises from the share of non-terminating programs due to an inapprorpiate reduction order as already indicated in Subsection 7.1.2.

The following paragraphs compare the systems individually and comment on individual phenomenons and deviations from the overall impression of Table 7.4.

MagicHaskeller vs. Igor II In general one can say that MagicHaskeller is slower and tends to run out of memory more quickly than Igor II. Both, the runtimes and the memory leaks strongly depend on the problem specific data types and libraries. The more polymorphic they are, the harder it is for MagicHaskeller to quickly find a solution. Furthermore, the search space tends to explode faster, simply because the number of alternative programs is much greater.

Consider for example the group of predicates and functions on Booleans. The library contained only the morphisms for lists, the Boolean values and a conditional expression. Thus, it provided very little polymorphism but much guidance by these types. Arbitrary lists or general tuples as used in the UCI problems, however show much more polymorphism and more possibilities for type alternatives due to weak guidance by the types. This leads to longer runtimes, and obviously to more stack overflows. Similarly, for function on mixed inputs and other user defined data types the libraries were more extensive than e.g. for the predicates.

Another observation is that MagicHaskeller is faster but less successful when using catamorphisms than with paramorphisms. Compared to catamorphisms, paramorphisms are not more polymorphic w.r.t. the amount of different type variables, but they have more arguments. Admittedly, once one argument is fixed, the type of other arguments with the same type variable is determined. However, to compute these arguments, further functions are needed which may contain type variables which are undetermined at this point. This, of course, increases the amount of search involved, but this is made up by a more expressive recursion scheme. If a catamorphism is rather unsuitable, it is hard for MagicHaskeller to compensate this by search.

Only on some instances MagicHaskeller is faster or as fast as Igor II. Most of them were problems where additional background knowledge was provided, e.g. `gaussSum`, `fact`, or `mult`. So given the appropriate primitives, MagicHaskeller is unsurprisingly fast. On most problems (`length`, `concat`, `gaussSum`) where MagicHaskeller outperformed Igor II, the difference is virtually negligible.

Finally, there is only one problem, `mult`, where MagicHaskeller was successful, but all Igor II-systems were not. Natural numbers are inherently Igor II's weak point. Obviously, when represented as integers there is no structure at all, because from a term perspective, there is no difference between, say 1 and 2. Such a representation suits better for MagicHaskeller's enumerative approach, because basic primitives such as 0, 1, and + suffice to quickly build more complex functions such as `mult`. Representing those primitives as Peano integers makes no big difference for MagicHaskeller. For the Igor II systems, however, such a representation still provides only a little structural guidance, and thus leading to vast search with many alternatives which makes it hard for Igor II to detect structural similarities in the output terms of a set of IOs.

Igor II$^+$ vs. Igor II$_{\mathrm{H}}$ The original Igor II$_{\mathrm{H}}$ algorithm performs only on a few problems better than the new, extended one. Only the numerical problems, such as `ack`, `eq`, and `geq` and `hanoi` on user defined data types, Igor II$^+$ was unable to solve with catamorphisms but succeeded without. This is simply because those problems do not follow a catamorphic recursive scheme. Equality on natural numbers requires simultaneous reduction of the input arguments. Such a scheme is not covered by a catamorphism, nevertheless it is possible to satisfy its universal properties on those examples. Catamorphisms are applied greedily, so once a universal property could be satisfied no backtracking is done. Igor II$^+$ then fails to close all hypotheses due to other restrictions (reduction order, etc.), and step by step the whole search space is discarded and Igor II$^+$ terminates with an empty search space for `eq` and `geq`.

Programs for `ack` and `hanoi` require functions as base cases, i.e. the default value of the catamorphisms is not constant. For example, the base case of `ack` is defined as

```
ack Z n = S n
```

and for `hanoi` it is

```
hanoi D0 a0 _ a2 a3 = MV D0 a0 a2 a3
```

So both have a function call, in this case a constructor application, instead of a constant value as base case. Again, this is not covered by catamorphisms.

Although it is possible to solve `rocket` with catamorphisms, IGOR $\mathrm{II_H}$ was much faster without. This, however, could be explained by the fact that the domain was especially engineered to perfectly fit IGOR $\mathrm{II_H}$ requirements. Forcing IGOR II^+ to use catamorphisms seems to hamper the system on this particular example.

On most other examples, IGOR II^+ was faster, sometimes as fast as, and only in single cases unsignificantly slower than IGOR $\mathrm{II_H}$.

Igor II^+_C vs. Igor II^+_P As already mentioned, the use of paramorphisms is at the moment just an experimental feature. Although a paramorphism will be introduced correctly according to its universal properties, in some cases non-terminating programs are synthesised. The problem is that paramorphisms, contrary to catamorphisms, pass both, the original input and its decomposition to its mediating functions. At the moment, it happens quite often that IGOR II^+_P in consecutive auxiliary functions just recomposes the original input and than trivially detects the applicability of a recursive call to the target function. Consider for example the following solution for `mult` with paramorphisms.

```
mult a0 a1 = para ⊥ (fun1 a0 ⊕ fun2 a0) a1
fun1 _ _ = Z
fun2 a0 (Z, _) = a0
fun2 (S a0) (S _, S a2) = mult (S a0) (S (S a2))
```

The second argument of `fun2` is a pair of the result of the recursive call of the decomposed input and the original input. The recursive call, however, recomposes the original input when calling `mult` recursively. To prevent this, the reduction order has to be extended. A simple approach would prohibit any argument of a recursive call to be syntactically greater then any input argument of the calling function, but this postpones the non-terminating recursive call just one level in the calling hierarchy. To fix this, a reduction order has to keep track of individual terms or input arguments. Once used in a paramorphism, it must assure that it will always decrease in size in any subsequent call.

Nevertheless, the results promise paramorphisms to be a good support for the structural recursive scheme of catamorphisms. Some functions that were unsolvable (`eq`, `geq`) with catamorphisms were solvable with paramorphisms. For others, their partial solutions look quite promising and may be successful, once the reduction-order problem is solved. Contrary to IGOR $\mathrm{II_H}$ and IGOR II^+_C, it was possible to synthesise `weave` with the default reduction order and no additional settings.

For the successfully solved problems, the runtimes were as fast as or insignificantly slower than with IGOR II^+_C. Only synthesising `drop`, `lengths`, and `sub` took more time than with IGOR II^+_C. One reason for longer runtimes, especially for functions on lists, is the use of type functors. IGOR II^+_C privileges lists in the sense that it uses the HASKELL functions `foldr`, `filter`, and `map`. The latter, implementing the type functor on lists, allows to more efficiently synthesise natural transformations of lists. Type functors are not supported by IGOR II^+_P and need to be synthesised “afoot”.

Table 7.5.: Overview of runtimes in seconds.

Name	Back	I_H	I_C^+	I_P^+	H_P	H_C
functions on natural numbers						
ack	—	0.9081[d]	∅	22.0054[ea]	272.6490	81.3010
add	—	0.7480	0.0001	0.0001	0.0680	0.0160
even	—	0.0040	0.0001	0.0001	0.0120	∅
even, odd	—	0.0040	0.0001	0.0001	—	—
eq	—	0.4120	2.3081[eb]	0.0280	∅	∅
gaussSum	add	0.0240	0.1000	0.0040	0.0040	0.0040
fact	mult	16.2290	16.1490	0.0040[ea]	0.0280	28.9100
fib	add	0.6360	0.6200	0.8561[ea]	214.9530	77.7570
geq	—	0.0320	0.1400[eb]	0.0440	∅	∅
mult	—	∅	∅	0.0120[ea]	6.3480	2.8520
mult	add	∅	∅	0.0160[ea]	0.2960	0.2040
odd	—	0.0001	0.0001	0.0001	0.0120	∅
sub	—	∅	0.0001	145.5011	∅	∅
predicates, functions on booleans						
andL	—	0.0040	0.0001	0.0040	0.1240	0.0080
and	—	0.0001	0.0001	0.0001	0.0080	0.0040
evenParity	—	∅	0.0040	2.2921[ea]	8.3210	0.2680
negateAll	—	0.0080	0.0001	0.0040	0.4560	0.1080
nandL	—	0.0040	0.0001	0.0040	0.1320	0.0080
norL	—	0.0040	0.0001	0.0040	0.1240	0.0040
or	—	0.0001	0.0001	0.0001	0.0080	0.0040
orL	—	0.0040	0.0001	0.0040	0.1280	0.0080
functions on lists						
append	—	2.3481	2.3321	0.0001	0.1080	0.0080
evenLength	—	0.0040	0.0001	0.0001	0.0120	∅
evenpos	—	0.0040	0.0001	0.0080[ea]	5.1560	3.8680
halves	—	224.3940[e]	∅	0.0120[ea]	∅	∅
init	—	0.0040	0.0001	0.0001	2.6400	3.8960
init, last	—	0.0040	0.0040	0.0040	—	—
inits	—	0.0920[e]	0.0001	0.0080	∅	∅
intersperse	—	0.0080[e]	0.0001	0.0001[ea]	2.6280	1.7600
last	—	0.0001	0.0001	0.0001	∅	∅
lastM	—	0.0040	0.0001	0.0040	0.0440	∅

Continued on next page

Table 7.5 – continued from previous page

Name	Back	I_H	I_C^+	I_P^+	H_P	H_C
multfst	—	0.0120	0.0001	0.0200	1.8600	0.1560
multlst	—	0.0080	0.0001	0.0001	0.1240	0.0680
oddpos	—	0.0160	0.0120	0.0520	4.9680	3.8640
pack	—	0.0080	0.0001	0.0040	4.1760	2.1840
subseqs	append	0.1560[e]	0.0040	0.0320[ea]	∅	∅
reverse	—	0.0240	0.0001	0.0040[ea]	0.0400	0.0200
shiftl	—	0.0040	0.0040	0.0040[ea]	1.8520	167.0460
shiftl, shiftr	—	0.0240	0.0040	0.0080	—	—
shiftr	—	0.0120	0.0040	0.0001	25.2500	37.6300
snoc	—	0.0040	0.0001	0.0001	0.0440	0.0280
swap	—	0.0080	0.0280	0.0040[ea]	∅	∅
switch	—	0.0280	0.0160	0.0040[ea]	∅	∅
split	—	0.0200	0.0001	0.0001	∅	∅
tail	—	0.0040	0.0001	0.0001	0.0010	3.2680
tails	—	0.0040	0.0001	0.0001	0.0520	7.9040
unzip	—	0.0240	0.0040	0.0240	∅	∅
weave	—	0.0480[c]	0.0440[c]	0.0240	∅	∅
zip	—	0.0640	0.0560	0.0600	∅	∅
functions on lists of lists						
concat	—	0.4640	0.0840	0.1000[ea]	2.5880	0.0720
lasts	—	0.0080	0.0001	0.0080	25.7460	2.5520
mapCons	—	0.0040	0.0001	0.0001	0.0560	0.0320
mapTail	—	0.0040	0.0001	0.0001	0.0640	∅
transpose	—	∅	0.0160	0.1840[ed]	∅	∅
weaveL	—	∅	∅	∅	∅	∅
functions on naturals and lists						
addN	—	0.3560	0.0001	0.3600	4.1880	1.3600
alleven	—	0.0120	0.0001	0.0120[ed]	4.1000	∅
allodd	—	∅	0.0001	0.0080[ed]	3.8680	∅
evens	—	0.0400	0.0001	2.4642[ed]	∅	∅
incr	—	0.0040	0.0001	0.0001	0.0160	0.0160
length	—	0.0040	0.0001	0.0001	0.0040	0.0001
lengths	—	17.3651	0.0001	1.7801	1.7040	0.7080
nthElem	—	0.0040	0.0040	0.0001	∅	∅
oddslist	—	∅	0.0040	0.0080[ed]	4.3200	∅
odds	—	0.0440	0.0001	4.9083[e]	∅	∅
drop	—	∅	0.0001	161.7141	0.0240	∅

Continued on next page

Table 7.5 – continued from previous page

Name	Back	I_H	I_C^+	I_P^+	H_P	H_C
splitAt	—	∅	0.2440	∅	∅	∅
sum	—	0.0080	0.0001	0.0960[e]	0.9600	0.0480
replicate	—	0.0040	0.0001	0.0001	0.0080	0.0040
take	—	0.0080	0.0040	0.1680	19.4130	6.0920
zeros	—	0.0040	0.0001	0.0001	0.2200	0.1960
functions on trees						
preorder	append	0.0120	0.0040	0.0080	0.2720	0.1880
inorder	append	0.1240[e]	0.0080	0.0280[ea]	0.2760	0.1880
postorder	append, snoc	13.9929[e]	0.0400	0.0120[ea]	0.2760	0.1840
mirror	—	0.0080	0.0001	0.0001	0.0360	0.0160
functions on mixed inputs						
pepper	—	0.0520	0.0240	0.0040[ea]	∅	∅
pepperF	—	0.0520	0.0040	0.0040[ea]	∅	∅
functions on other data types						
rocket	—	0.0040	5.4763	0.0280	133.6560	137.7810
hanoi	—	0.0640	∅	∅	∅	∅
sentence	—	0.0120	0.0001	0.0001	∅	∅
functions forUCI classification problems						
balloons	—	0.0080	0.0040	0.0040	∅	∅
playTennis	—	0.0160	0.0160	0.0160	∅	∅
enjoySport	—	0.0040	0.0001	0.0001	∅	∅
lenses	—	0.2400	0.2200	0.2200	∅	∅

∅ stack overflow — no value × not applicable
[a]Inapropriate reduction order [b]exhausted search space
[c]Linear reduction order [d]greedy-rule-splitting [e]wrong
fastest wrong

I_H IGOR II_H · I_C^+ IGOR II_C^+ · I_P^+ IGOR II_P^+
H_P MAGICHASKELLER with ⦃·⦄ · H_C MAGICHASKELLER with ⦇·⦈

Name	Σ	*max*	*min*	$\varnothing$	$\tilde{x}$	σ	$\oplus$	$^{\oplus}/_{N}$
IGOR II	4036	2049	0	59.3529	5	269.0267	68	80.95%
IGOR II$^{+}_{C}$	2141	908	0	28.1711	3	119.6794	76	90.48%
IGOR II$^{+}_{P}$	402	206	0	7.3091	2	27.3542	55	65.48%

Σ total loops *max* maximum *min* minimum
$\varnothing$ mean $\tilde{x}$ median
σ standard deviation $\oplus$ absolute successes $^{\oplus}/_{N}$ success rate

Table 7.6.: IGOR II loop cycle statistics for $N = 84$ example problems

7.2.2. IGOR II$_{\mathrm{H}}$ Algorithm loop cycles

The last subsection compared the runtimes of the different systems on various example problems. Since IGOR II$_{\mathrm{H}}$ is already quite fast, a closer look on the algorithm loop cycles needed to synthesise a particular problem may be desirable, making the improvements more sensible.

Table 7.6 summarises the benchmark w.r.t. the loop cycles needed for all successfully synthesised problems. It lists total number of loops, the maximum and the minimum[6] loops needed for an individual problem, the mean and the median loop number, and the standard deviation. It also recapitulates the absolute successes and the success rate. Except for the success rates and the minimum number of loops, all measures could be decreased by about 50% when using IGOR II^{+} instead of IGOR II$_{\mathrm{H}}$. The maximum dropped from 2049 to 908 cycles which dramatically affects the total number of loops needed as well as the mean loops. 50% of all problems are now solved within 3 loops or less. Again, using catamorphisms decreases the dispersion by sparing loops on complex, long running problems.

Table 7.7 shows the algorithm loop cycles needed by the different IGOR II$_{\mathrm{H}}$ versions for each individual problem. Wrong results are also marked with ($\perp$) and highlighted with a dark background. The symbol for the empty set ($\emptyset$) indicates whether the system runs out of memory during synthesis. MAGICHASKELLER is not able to simultaneously synthesise multiple targets. Those problems have been skipped. This is indicated by the symbol ($\times$). The last two columns show the speedup of IGOR II$^{+}_{C}$ and IGOR II$^{+}_{P}$ w.r.t. the original algorithm IGOR II$_{\mathrm{H}}$. The speedups of IGOR II$^{+}_{C}$ and IGOR II$^{+}_{P}$ w.r.t. IGOR II$_{\mathrm{H}}$ and in particular the symbols are defined as follows:

[6]The pathological minimum loop count of 0 arises during the synthesis of `init`, which is solved immediately via antiunification.

$$
{}^{I_H}/_{I^+} = \begin{cases} \frac{I_H}{I^+} & \text{if } I_H > I^+ \\ -\frac{I^+}{I_H} & \text{if } I_H < I^+ \\ 1 & \text{if } I_H = I^+ \\ \gg & \text{if } I^+ \text{ failed} \\ \ll & \text{if } I_H \text{ failed} \\ \times & \text{if both failed} \end{cases},
$$

where / is integer division, I_H abbreviates the loop number of IGOR II$_\mathrm{H}$, I^+ for IGOR II$^+$, I_C^+ for IGOR II$_C^+$, and I_P^+ for IGOR II$_P^+$. Thus, a positive speedup is denoted by a positive integer, a negative by a negative integer. A failure of IGOR II$_\mathrm{H}$ and success of the new IGOR II$^+$ is denoted by $\ll$, and vice verse by $\gg$. A failure of both is denoted by $\times$.

Table 7.7.: Overview of IGOR II algorithm's loop cycles and speedups.

Name	Back	I_H	I_C^+	I_P^+	${}^{I_H}/_{I_C^+}$	${}^{I_H}/_{I_P^+}$
functions on natural numbers						
ack	—	9	$\emptyset$	$2196^{\perp}$	$\gg$	$\gg$
add	—	6	1	1	6	6
even	—	3	2	2	2	2
even, odd	—	2, 2	2, 2	2, 2	1	1
eq	—	245	$703^{\perp}$	4	$\gg$	61
gaussSum	add	17	57	2	-3	9
fact	mult	908	908	$2^{\perp}$	1	$\gg$
fib	add	173	173	$161^{\perp}$	1	$\gg$
geq	—	4	$114^{\perp}$	3	$\gg$	1
mult	—	$\emptyset$	$\emptyset$	$4^{\perp}$	$\times$	$\times$
mult	add	$\emptyset$	$\emptyset$	$4^{\perp}$	$\times$	$\times$
odd	—	3	2	2	2	2
sub	—	$\emptyset$	2	3	$\ll$	$\ll$
predicates, functions on booleans						
andL	—	3	2	2	2	2
and	—	1	1	1	1	1
evenParity	—	$\emptyset$	4	$3^{\perp}$	$\ll$	$\times$
negateAll	—	4	2	3	2	1
nandL	—	3	2	2	2	2
norL	—	3	2	2	2	2
or	—	1	1	1	1	1

Continued on next page

Table 7.7 – continued from previous page

Name	Back	I_H	I_C^+	I_P^+	${}^{I_H}/_{I_C^+}$	${}^{I_H}/_{I_P^+}$
orL	—	3	2	2	2	2
functions on lists						
append	—	4	4	1	1	4
evenLength	—	3	2	2	2	2
evenpos	—	4	4	$3^{\perp}$	1	$\gg$
halves	—	$12742^{\perp}$	$\emptyset^{\perp}$	$5^{\perp}$	$\times$	$\times$
init	—	3	3	3	1	1
inits	—	$63^{\perp}$	3	5	$\ll$	13
init, last	—	3, 2	3, 2	3, 2	1	1
intersperse	—	$7^{\perp}$	3	$3^{\perp}$	$\ll$	$\times$
last	—	2	2	2	1	1
lastM	—	4	3	3	1	1
multfst	—	8	2	8	4	1
multlst	—	6	2	3	3	2
oddpos	—	11	11	23	1	-2
pack	—	8	3	3	3	3
subseqs	append	$76^{\perp}$	5	$8^{\perp}$	$\ll$	$\times$
reverse	—	13	2	$4^{\perp}$	7	$\gg$
shiftl	—	5	4	$4^{\perp}$	1	$\gg$
shiftl, shiftr	—	5, 11	4, 5	4, 3	2	2
shiftr	—	10	5	3	2	3
snoc	—	4	1	1	4	4
swap	—	6	17	$4^{\perp}$	-3	$\gg$
switch	—	13	14	$4^{\perp}$	-1	$\gg$
split	—	17	1	1	17	17
tail	—	0	0	0	1	1
tails	—	4	1	1	4	4
unzip	—	13	4	13	3	1
weave	—	4	4	3	1	1
zip	—	6	6	6	1	1
functions on lists of lists						
lasts	—	6	3	4	2	2
mapCons	—	5	1	1	5	5
mapTail	—	3	1	1	3	3
transpose	—	$\emptyset$	11	$25^{\perp}$	$\ll$	$\times$
weaveL	—	$\emptyset$	$\emptyset$	$\emptyset$	$\times$	$\times$

Continued on next page

Table 7.7 – continued from previous page

Name	Back	I_H	I_C^+	I_P^+	I_H/I_C^+	I_H/I_P^+
functions on naturals and lists						
addN	—	18	2	10	9	2
alleven	—	6	3	$4^{\perp}$	2	$\gg$
allodd	—	$\emptyset$	3	$4^{\perp}$	$\ll$	$\times$
evens	—	24	3	$868^{\perp}$	8	$\gg$
incr	—	3	1	1	3	3
lengths	—	2049	2	7	1025	293
nthElem	—	2	4	2	-2	1
oddslist	—	$\emptyset$	3	$4^{\perp}$	$\ll$	$\times$
odds	—	19	3	$876^{\perp}$	6	$\gg$
splitAt	—	$\emptyset$	43	$\emptyset^{\perp}$	$\ll$	$\times$
replicate	—	3	1	1	3	3
zeros	—	6	2	2	3	3
functions on trees						
preorder	append	5	3	3	2	2
inorder	append	$31^{\perp}$	6	$7^{\perp}$	$\ll$	$\times$
postorder	append, snoc	$545^{\perp}$	14	$4^{\perp}$	$\ll$	$\times$
mirror	—	4	1	1	4	4
functions on mixed inputs						
pepper	—	28	12	$2^{\perp}$	2	$\gg$
pepperF	—	28	3	$2^{\perp}$	9	$\gg$
functions on other data types						
rocket	—	3	495	10	-165	-3
hanoi	—	13	$\emptyset$	$\emptyset$	$\gg$	$\gg$
sentence	—	13	1	1	13	13
functions for UCI classification problems						
balloons	—	5	5	5	1	1
playTennis	—	18	18	18	1	1
enjoySport	—	1	1	1	1	1
lenses	—	206	206	206	1	1

Continued on next page

Table 7.7 – continued from previous page

Name	Back	I_H	I_C^+	I_P^+	${}^{\mathrm{I}_H}/_{I_C^+}$	${}^{\mathrm{I}_H}/_{I_P^+}$

$\emptyset$ stack overflow $\perp$ failure
$+/-N$ positive/negative speedup
$\ll$ Igor II$_\mathrm{H}$ failed $\gg$ Igor II$^+$ failed $\times$ both failed
fastest failure

I_H Igor II$_\mathrm{H}$ · I_C^+ Igor II$_C^+$ · I_P^+ Igor II$_P^+$
${}^{\mathrm{I}_H}/_{I_C^+}$ speedup of I_C^+ w.r.t. I_H · ${}^{\mathrm{I}_H}/_{I_P^+}$ speedup of I_p^+ w.r.t. I_H

Figure 7.8 and Figure 7.9 show the speedups of Igor II$_C^+$ and Igor II$_P^+$, respectively, as a histogram. Bars to the right represent positive, bars to the left negative speedups. Bars to both sides with a speedup of 1 show that there is no difference between the systems. Bars to the left or right labelled with -1 and 1, respectively, represent a small change levelled out by integer division. There is no bar if both systems failed on a particular problem.

As mentioned in Table 7.6, Igor II$_C^+$ could solve more than 90% of the benchmark problems. Among these, only for `gaussSum`, `swap`, `switch`, `nthElem`, and `rocket` Igor II$_C^+$ needed more loops than Igor II$_\mathrm{H}$. Igor II$_P^+$, however, was faster than the original algorithm on `gaussSum`, and failed only due to an inappropriate reduction order on the others. This indicates that catamorphisms seem inappropriate on those problems.

For about $^2/_3$ of the remaining, successfully synthesised programs, the number of loop cycles needed was at least halved when using Igor II$_C^+$. For about 20% of the problems the speedups of factor 10 or more could be achieved. For highly structured inputs, as for example `lengths`, `splitAt`, `subseqs`, `transpose`, or `inits`, the best improvements could be achieved.

The histogram of Igor II$_P^+$ speedups in Figure 7.9 looks worse than it actually is. Of course, all the non-terminating solutions occur as failures resulting in negative speedups, but with an appropriate reduction order they may turn into successes.

Table 7.8.: Histogram of speedups for IGOR II_C^+ w.r.t. IGOR II_H

ack	$\gg$	
eq	$\gg$	
geq	$\gg$	
hanoi	$\gg$	
rocket	-165	
swap	-3	
gaussSum with add	-3	
nthElem	-2	
switch	-1	
halves	$\times$	halves
mult with add	$\times$	mult with add
mult	$\times$	mult
weaveL	$\times$	weaveL
and	1	and
append	1	append
balloons	1	balloons
enjoySport	1	enjoySport
even, odd	1	even, odd
evenpos	1	evenpos
fact with mult	1	fact with mult
fib with add	1	fib with add
init, last	1	init, last
init	1	init
last	1	last
lenses	1	lenses
oddpos	1	oddpos
or	1	or
playTennis	1	playTennis
tail	1	tail
weave	1	weave
zip	1	zip
	1	lastM
	1	shiftl
	2	alleven
	2	andL
	2	concat
	2	evenLength
	2	even
	2	intersperse
	2	lasts
	2	nandL
	2	negateAll
	2	norL
	2	odd
	2	orL
	2	pepper
	2	preorder with append
	2	shiftl, shiftr
	2	shiftr

Table 7.8 – continued from previous page

2	take
3	incr
3	length
3	mapTail
3	multlst
3	pack
3	replicate
3	sum
3	unzip
3	zeros
4	multfst
4	mirror
4	snoc
4	tails
5	inorder with append
5	mapCons
6	add
6	odds
7	reverse
8	evens
9	addN
9	pepperF
13	sentence
15	subseqs with append
17	split
21	inits
39	postorder with append, snoc
1025	lengths
$\ll$	allodd
$\ll$	drop
$\ll$	evenParity
$\ll$	oddslist
$\ll$	splitAt
$\ll$	sub
$\ll$	transpose

n	positive speedup (rounded to next integer)
$-n$	negative speedup (rounded to next integer)
1	no speedup
$\ll$	IGOR II_H failed
$\gg$	IGOR II_C^+ failed
$\times$	both failed

Table 7.9.: Histogram of speedups for IGOR II$_P^+$ w.r.t. IGOR II$_H$

ack	$\gg$	
alleven	$\gg$	
concat	$\gg$	
evenpos	$\gg$	
evens	$\gg$	
fact with mult	$\gg$	
fib with add	$\gg$	
hanoi	$\gg$	
odds	$\gg$	
pepperF	$\gg$	
pepper	$\gg$	
reverse	$\gg$	
shiftl	$\gg$	
sum	$\gg$	
swap	$\gg$	
switch	$\gg$	
rocket	-3	
oddpos	-2	
allodd	$\times$	allodd
evenParity	$\times$	evenParity
halves	$\times$	halves
inorder with append	$\times$	inorder with append
intersperse	$\times$	intersperse
mult with add	$\times$	mult with add
mult	$\times$	mult
oddslist	$\times$	oddslist
postord. w. app. snoc	$\times$	postord. w. app. snoc
splitAt	$\times$	splitAt
subseqs with append	$\times$	subseqs with append
transpose	$\times$	transpose
weaveL	$\times$	weaveL
and	1	and
balloons	1	balloons
enjoySport	1	enjoySport
even, odd	1	even, odd
init, last	1	init, last
init	1	init
last	1	last
lenses	1	lenses
multfst	1	multfst
nthElem	1	nthElem
or	1	or
playTennis	1	playTennis
tail	1	tail
unzip	1	unzip
weave	1	weave
zip	1	zip
	1	geq
	1	lastM
	1	negateAll

Table 7.9 – continued from previous page

2	addN
2	andL
2	evenLength
2	even
2	lasts
2	multlst
2	nandL
2	norL
2	odd
2	orL
2	preorder with append
2	shiftl, shiftr
2	take
3	incr
3	length
3	mapTail
3	pack
3	replicate
3	shiftr
3	zeros
4	append
4	mirror
4	snoc
4	tails
5	mapCons
6	add
9	gaussSum with add
13	sentence
13	inits
17	split
61	eq
293	lengths
$\ll$	drop
$\ll$	sub

n	positive speedup (rounded to next integer)
-n	negative speedup (rounded to next integer)
1	no speedup
$\ll$	IGOR II$_H$ failed
$\gg$	IGOR II$_P^+$ failed
$\times$	both failed

7.3. Improvement Remarks

This section drops some ideas to improve the IGOR II$^+$ algorithm and makes some remarks for further improvements which have not been implemented yet.

Conditional Rules The original IGOR II-algorithm described by Kitzelmann [66] supported to some extent the synthesis of conditional equations, where partitioning is done w.r.t. a predicate. This allows IGOR II to learn functions like `member` to check whether a list contains a certain element, or insertion into a ordered binary tree. However, the induction of the partitioning predicate is not done data-driven, but more or less in a generate-and-test manner, i.e. all partitions were generated and then tried to find an appropriate predicate for each of them. Obviously, this quickly hits the brick wall. It would be desirable to establish some concise criterion upon which the partitioning and the predicate invention can be established and assessed.

Automatic instance generation IGOR II$^+$ requires a data type to be instance of the type classes `Mu` and `PF` if used for type morphisms. At the moment the user has to write these instance declaration and describe a data type as a polynomial functor and an initial algebra.

It is worth mentioning, that these instance declarations are totally canonic and it is possible to automatically generate them. The author of the pointless-haskell[7] library implemented the algorithm described by Hu et al. [55] in a tool called DrHylo[8]. Since it is based on an outdated library, a re-implementation would be required to be used with IGOR II$^+$.

Additional reduction order As already mentioned several times earlier, the failure of IGOR II^+_P is due to an inappropriate reduction order. To prevent the synthesis of non-terminating programs, a more sophisticated reduction order is required, which keeps track of which input argument is used in which type morphism, to ensure that the corresponding terms really strictly decrease in size.

Automatic option inference Some problems were only synthesisable after setting additional options such as "greedy rule splitting" (cf. 7.1.1) or setting an explicit reduction order. It should be possible to infer both, the recursion argument of a function and an appropriate reduction order by analysis of the IO examples.

Compute default value Catamorphisms require a default value for constant constructors. Consider for example catamorphisms on lists and the following code example. If the target function has, apart from the first argument of type list, a second, additional argument, it may be the case that the default value of `foldr` depends on this second input. Therefore, not only the synthesis of a mediating `f` is required but also of another auxiliary `d` which computes the default value.

[7] http://hackage.haskell.org/package/pointless-haskell
[8] http://wiki.di.uminho.pt/twiki/bin/view/Personal/Alcino/DrHylo

```
g :: [α] → β → γ
g l e   = foldr (f e) (d e) l
f ::  β → α → γ → γ
f e a b = ...
d :: β → γ
d a     = ...
```

Inventing an auxiliary function to compute the default value may enable IGOR II_C^+ to solve `ack` and `hanoi`, because this was exactly the reason why it failed (cf. Paragraph 7.2.1).

Backtracking At the moment, type morphisms are applied greedily. As the empirical evaluation revealed, this may sometimes mislead the search and result in complete failure of the synthesis. This is the case when universal properties of a morphism apply, but the hypotheses cannot be finished in later steps due to other restrictions, as for example requirements of the reduction order. It is desirable to have some criteria to judge whether the application of a morphism fails or at least provide the possibility of backtracking and continue the synthesis without type morphisms.

Ordering morphism When using multiple type morphisms it is desirable to check their universal properties in a particular order. Type functors (6.3.1), for example, are a special case of catamorphisms, which themselves generalise to paramorphisms. Thus, it is suggestive to first check the applicability of type functors, then catamorphisms, and finally paramorphisms to apply the least general recursion scheme.

7.4. Discussion

The results of MAGICHASKELLER show the typical weakness of enumerative approaches: They are fast and reliable, provided with appropriate primitives, but tend to get lost in the search space quickly when given too general and unspecific information, especially if there are too many polymorphic functions in its library.

Compared to MAGICHASKELLER, the analytical, data-driven approach of the IGOR II_H systems, are much more faster, more reliable and more successful in synthesising functions from a selected set of IO examples. In general, the structure of IO examples contains much information which can be successfully exploited to analytically reduce and guide the search.

Catamorphisms are just one further step in this direction. Instead of only using structural information which is explicitly encoded in terms representing IOs, they provide means to use implicit structural information, as e.g. structural recursion scheme of a data type. Their universal properties are an exclusive criteria of their applicability which can be easily checked in the IO examples at hand.

The previous empirical tests showed that using type morphisms significantly improves the efficiency and the effectiveness of the IGOR II algorithm. It drastically reduces runtimes and algorithm loop cycles needed, but also allows to synthesise programs which where beyond the scope before. The success can be explained by the following key benefits of the use of type morphisms:

Complexity reduction Type morphisms are a suitable way to reduce the search space complexity by providing guidance when the algorithm is overwhelmed by equivalent alternatives. They help to quickly traverse such plateaus in the search space. Where the original Igor II algorithm falls back to sequentially process all equivalent alternatives, the universal properties and the introduction of type morphisms directly lead the search to the rim of the plateau. They act like a signpost which can be applied under specific circumstances and provide save conduct for the next few steps. Although not matured, the experiments showed that paramorphisms can complement catamorphisms conveniently. Appendix G shows some search tree visualisations of different Igor II versions.

Gain in expressiveness Using type morphisms extends the expressiveness and the capabilities of the algorithm through the use of recursive program schemes. This makes it possible to solve programs which where beyond its scope before. Apart from adding explicitly recursive schemes to Igor II's language bias, they extend it in another, less apparent way.

Igor II$_{\mathrm{H}}$ cannot invent a new auxiliary function at the root position of a rule's right-hand side. Auxiliary functions require the abduction of appropriate IO examples, which is only possible if the new auxiliary functions occur below a constructor symbol. For example, Igor II$_{\mathrm{H}}$ could never synthesise the solution of `reverse` as shown in Listing 7.6, because solely from the IOs of `reverse`, there is now a possibility to abduce appropriate IOs for `snoc`, if not provided as background knowledge.

If you compare this with the equivalent program synthesised by Igor II$_C^+$ which is shown in Listing 7.7, exactly this happened. The auxiliary `snoc` occurs at the root position. Here it is possible to abduce appropriate IOs, because the recursive scheme of `foldr`, induced by its universal properties, provides Igor II$_C^+$ exactly with the information how to abduce IOs for `snoc` given IOs for `reverse`.

Listing 7.6: `reverse` using `snoc` and explicit recursion

```
reverse :: [α] → [α]
reverse []      = []
reverse (x:xs) = snoc x (reverse xs)
snoc a []      = [a]
snoc a (x:xs)  = x : (snoc a xs)
```

Listing 7.7: `reverse` with auxiliary `foldr` and `snoc`

```
reverse :: [α] → [α]
```

```
reverse l     = foldr snoc [] l
snoc a l      = foldr fun [a] l
fun e (x:xs) = e:x:xs
```

Judging Igor II's capability Another benefit of type morphisms is, that they allow to describe classes of programs synthesisable by IGOR II^{+}_{C}. It is hard to exactly describe which programs IGOR II_{H} is able to synthesise and in general this is still not trivial to describe the class of *all* programs IGOR II^{+} can solve. However, now it is possible to tell that programs which follow a catamorphism can now be solved by IGOR II^{+}_{C}. This for the first time states a positive criteria whether a problem can be solved.

Although it is said that there is no such thing as free lunch, this seems not to be true for the use of type morphisms. In IP it is tacitly agreed that an improvement of expressiveness has to be paid by a deterioration of efficiency and vice versa restricting the language bias to keep the search space as small and search efficient goes at the expense of expressiveness. No so with type morphisms. By skillfully exploiting all available knowledge, both the explicit as well as the implicit, an improvement of the expressiveness and the language bias of an IP system simultaneously leads to improvement in efficiency.

8. Conclusion

Inductive functional programming systems can be characterised by two diametric approaches: Either they apply exhaustive program enumeration which uses Input/Output examples (IO) as test cases, or they perform an analytical, data-driven structural generalisation of the IO examples.

Enumerative approaches ignore the structural information provided with the IO examples, but use type information to guide and restrict the search. They use higher-order functions which capture recursion schemes during their enumeration, but apply them randomly in a uninformed manner.

Analytical approaches, on the other side, heavily exploit this structural information but have ignored the benefits of a strong type system so far and use recursion schemes only either fixed and built in, or selected by an expert user.

This work shows how universal constructs from category theory, such as catamorphisms, paramorphisms, and type functors, can be used as recursive program schemes for inductive functional programming. The use of program schemes for Inductive Programming is not new. The special appeal and the novelty of this work is that, contrary to previous approaches, the program schemes are neither fixed, nor selected by an expert user: The applicability of those recursion schemes can be automatically detected in the given IO examples of a target function by checking the universal properties of the corresponding type morphisms.

An extension of the analytical functional inductive programming system Igor II was proposed and explained how the applicability of those recursion schemes can be detected in the given IO examples of the target function by checking universal properties of the corresponding type morphisms. It shows that the capability and the expressiveness of Igor II can be extended without deteriorating its efficiency.

A comprehensive empirical evaluation underpinned the benefits of extending the original Igor II-algorithm. Type morphisms are a very suitable tool to increase the efficiency by reducing the complexity of the search. They provide guidance in the search space based on universal properties of the applied type morphisms. Once such a morphism has been detected, only its mediating (argument) function has to be synthesised, which usually is structurally less complex than the original one.

Furthermore, they extend the expressiveness of Igor II by extending its language bias and quasi allowing to invent new auxiliary functions on root position of the right-hand side of a rule. The recursion scheme provides sufficient information to abduce appropriate IO examples for this new function. This was impossible before.

All in all, by skillfully exploiting all available knowledge, both the explicit as well as the implicit, an improvement of the expressiveness and the language bias of an IP system simultaneously leads to improvement in efficiency.

Additionally, those recursion schemes allow for the first time to characterise some program classes synthesisable by Igor II, namely those programs following a particular type morphisms.

However, this categorical view on IP is not exhausted yet. More type morphisms may be incorporated into the algorithm in the same way. Section 4.2 described that type functors, which describe natural transformation in a functional language, are a special case of catamorphisms and structural recursion, which themselves generalise to paramorphisms and primitive recursion via tupling. Both have their duals for coinductive types: anamorphisms and apomorphisms [132], which describe the construction of data types instead of their destruction. Composing anamorphisms and catamorphism into hylomorphisms, which require a conditional construct as e.g. **`if then else`**, allows to describe primitive recursion without tupling but with an intermediate data structure [87]. Allegories, as the categories of relations, may give the theoretical foundations of learning classifiers, similar to ILP, within this categorical framework [13]. Including exponentials in the underlying base category may allow learning higher-order functions (cf. Vene [131] describing the Ackermann function as higher-order catamorphism). Augusteijn [4] describes sorting morphisms.

This should not be an end in itself, but push the boundary further towards the automated generation of functions from examples for practical applications. An empirical study showed that IP systems cannot only support professional programmers, but they also give more power to programming novices who are enabled to produce correct code by providing input/output examples [44]. Other possible applications are the domain of test-driven-development, the automatic generation of XSL templates from exemplary user-interaction [46], structural generalisation for incident classification [122], and even its utilisation in the domain of object-oriented programming seems further afar than it actually is. Igor II is able to successfully synthesise functions in such a contest, applying structural generalisation to object-oriented concepts such as message passing, method calls, attributes, etc. [43].

Bibliography

[1] Ros Agapitos and Simon M. Lucas. Learning recursive functions with object oriented genetic programming. In *Proceedings of the 9th European Conference on Genetic Programming, ser. Lecture Notes in Computer Science*, pages 166–177. Springer, 2006. (back to page 31)

[2] Dana Angluin. Finding patterns common to a set of strings (extended abstract). In *STOC '79: Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 130–141, New York, NY, USA, 1979. ACM. (back to page 25)

[3] Dana Angluin and Carl H. Smith. Inductive inference: Theory and methods. *ACM Comput. Surv.*, 15(3):237–269, 1983. (back to page 25)

[4] Lex Augusteijn. Sorting morphisms. In *3rd International Summer School on Advanced Functional Programming, volume 1608 of LNCS*, pages 1–27. Springer-Verlag, 1998. (back to page 156)

[5] Lennart Augustsson. Announcing Djinn, version 2004-12-11, a coding wizard. Available from `http://permalink.gmane.org/gmane.comp.lang.haskell.general/12747`, 2005. (back to page 37)

[6] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, United Kingdom, 1998. (back to page 43 and 47)

[7] Michael Barr and Charles Wells. *Category theory for computing science*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990. (back to page 51)

[8] A. W. Biermann. The inference of regular LISP programs from examples. *IEEE Trans. on Systems, Man, and Cybernetics*, SMC-8:585–600, 1978. (back to page 30 and 33)

[9] Franck Binard and Amy Felty. An abstraction-based genetic programming system. In *GECCO '07: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, pages 2415–2422, New York, NY, USA, 2007. ACM. (back to page 32)

[10] Franck Binard and Amy Felty. Genetic programming with polymorphic types and higher-order functions. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1187–1194, New York, NY, USA, 2008. ACM. (back to page 30 and 32)

[11] R. S. Bird. A calculus of functions for program derivation. In David A. Turner, editor, *Research topics in functional programming*, pages 287–307. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. (back to page 58)

[12] Richard S. Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall, 1998. (back to page 59 and 171)

[13] Richard S. Bird and Oege De Moor. *Algebra of Programming*, volume 100 of *International Series in Computing Science*. Prentice Hall, 1997. (back to page 58 and 156)

[14] A. F. Bowers, C. Giraud-Carrier, C. Kennedy, J. W. Lloyd, and R. MacKinney-Romero. A framework for higher-order inductive machine learning. In *In Proceedings of the COMPULOGNet Area Meeting on Representation Issues in Reasoning and Learning*, 1997. (back to page 31)

[15] Alan Bundy. The use of explicit plans to guide inductive proofs. In *Proceedings of the 9th International Conference on Automated Deduction*, pages 111–120, London, UK, 1988. Springer-Verlag. (back to page 22)

[16] Alan Bundy. A science of reasoning (extended abstract). In *TABLEAUX*, pages 10–17, 1998. (back to page 22)

[17] Alan Bundy, Frank van Harmelen, Christian Horn, and Alan Smaill. The Oyster-Clam system. In *Proceedings of the 10th International Conference on Automated Deduction*, pages 647–648, London, UK, 1990. Springer-Verlag. (back to page 22)

[18] Alan Bundy, Andrew Stevens, Frank van Harmelen, Andrew Ireland, and Alan Smaill. Rippling: a heuristic for guiding inductive proofs. *Artif. Intell.*, 62(2): 185–253, 1993. (back to page 23)

[19] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986. (back to page 22)

[20] M. A. Covington. *Natural Language Processing for Prolog Programmers*. Prentice Hall, 1994. (back to page 128)

[21] Neil Crossley, Emanuel Kitzelmann, Martin Hofmann, and Ute Schmid. Combining Analytical and Evolutionary Inductive Programming. In B. Goertzel, P. Hitzler, and M. Hutter, editors, 2^{nd} *Conference on Artificial General Intelligence*, pages 55–60. Atlantis Press, 2009. (back to page 32)

[22] Neil Crossley, Emanuel Kitzelmann, Martin Hofmann, and Ute Schmid. Evolutionary Programming Guided by Analytically Generated Seeds. In António Dourado, Agostinho C. Rosa, and Kurosh Madani, editors, *Proceedings of the International*

Joint Conference on Computational Intelligence, pages 198–203. INSTICC Press, 2009. (back to page 32)

[23] Haskell Brooks Curry, and Robert Feys. *Combinatory Logic*. North Holland, Amsterdam, 1958. (back to page 21)

[24] Allen Cypher. Eager: Programming repetitive tasks by example. In *CHI '91: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 33–39, New York, NY, USA, 1991. ACM Press. (back to page 25)

[25] Allen Cypher. Eager: Programming repetitive tasks by demonstration. In Cypher et al. [26], pages 205–217. (back to page 25)

[26] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch what I do: programming by demonstration*. MIT Press, Cambridge, MA, USA, 1993. (back to page 25, 159, and 160)

[27] R. V. Desimone. Explanation-based learning of proof plans. In Yves Kodratoff and Alan Hutchinson, editors, *Machine and Human Learning*. Kogan Page, 1989. (back to page 23)

[28] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3):795–807, 1992. (back to page 38)

[29] William F. Finzer and Laura Gould. Rehearsal world: programming by rehearsal. In *Watch what I do: programming by demonstration*, pages 79–100. MIT Press, Cambridge, MA, USA, 1993. (back to page 25)

[30] Pierre Flener. Inductive logic program synthesis with dialogs. In *ILP '96: Selected Papers from the 6th International Workshop on Inductive Logic Programming*, pages 175–198, London, UK, 1997. Springer-Verlag. (back to page 29 and 33)

[31] Pierre Flener and Yves Deville. Logic program transformation through generalization schemata. In *Logic Program Synthesis and Transformation*, pages 171–173, 1995. (back to page 33)

[32] Pierre Flener and Derek Partridge. Inductive programming. *Automated Software Engg.*, 8(2):131–137, 2001. (back to page 25)

[33] Pierre Flener and Ute Schmid. An introduction to inductive programming. *Artif. Intell. Rev.*, 29(1):45–62, 2008. (back to page 33)

[34] Pierre Flener and Serap Yilmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. *Journal of Logic Programming*, 41(2-3):141–195, 1999. (back to page 29 and 33)

[35] Pierre Flener, Kung-Kiu Lau, and Mario Ornaghi. Correct-schema-guided synthesis of steadfast programs. In *ASE*, pages 153–, 1997. (back to page 33)

[36] Pierre Flener, Kung-Kiu Lau, Mario Ornaghi, and Julian Richardson. An abstract formalization of correct schemas for program synthesis. *J. Symb. Comput.*, 30(1): 93–127, 2000. (back to page 33)

[37] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, New York, NY, USA, 1989. (back to page 34)

[38] Michael J. C. Gordon. Introduction to the HOL system. In *TPHOLs*, pages 2–3, 1991. (back to page 22)

[39] Daniel C. Halbert. SmallStar: Programming by demonstration in the desktop metaphor. In Cypher et al. [26], pages 103–123. (back to page 25)

[40] Daniel Conrad Halbert. *Programming by example*. PhD thesis, Department of Electrical Engineering and Computer Sciences, Computer Science Division, University of California, Berkeley, 1984. (back to page 25)

[41] José Hernández-Orallo and M. José Ramírez-Quintana. Inverse narrowing for the induction of functional logic programs. In José Luis Freire-Nistal, Moreno Falaschi, and Manuel Vilares Ferro, editors, *Joint Conference on Declarative Programming*, pages 379–392, 1998. (back to page 30)

[42] José Hernández-Orallo and M. José Ramírez-Quintana. A Strong Complete Schema for Inductive Functional Logic Programming. In S. Dzeroski and P.A. Flach, editors, *Proc. of 9th International Workshop on Inductive Logic Programming*, volume 1634 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, 1999. (back to page 30)

[43] Thomas Hieber and Martin Hofmann. Automated Method Induction: Functional Goes Object Oriented. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *Proceedings of the* 3^{rd} *ACM SIGPLAN Workshop on Approaches and Applications of Inductive Programming, Revised Papers*, volume 5812 of *Lecture Notes in Computer Science*, pages 159–173. Springer Berlin / Heidelberg, 2010. (back to page 156)

[44] Thomas Hieber, Martin Hofmann, Emanuel Kitzelmann, and Ute Schmid. Programming recursive functions by examples. In B. Velichkovsky L. Urbas, T. Goschke, editor, *Proceedings der 9. Jahrestagung der Gesellschaft für Kognitionswissenschaft*, 2008. (back to page 156)

[45] Ralf Hinze. Generalizing generalized tries, 1999. (back to page 40)

[46] Martin Hofmann. *Automatic Construction of XSL Templates – An Inductive Programming Approach*. VDM Verlag, Saarbrücken, 2007. (back to page 156)

[47] Martin Hofmann. IgorII - An Analytical Inductive Functional Programming System: Tool Demo. In John Gallagher and Janis Voigtländer, editors, *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'2010)*, pages 29–32, New York, NY, USA, 2010. ACM. (back to page 90)

[48] Martin Hofmann. Data-driven detection of catamorphisms — towards prolem specific use of program schemes for inductive program synthesis. In *Proceedings of the* 11^{th} *Symposium on Trends in Functional Programming, Oklahoma City*, 2010. (back to page 100)

[49] Martin Hofmann and Emanuel Kitzelmann. I/O Guided Detection of List Catamorphisms: Towards Problem Specific Use of Program Templates in IP. In John Gallagher and Janis Voigtläder, editors, *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'2010)*, pages 93–100, New York, NY, USA, 2010. ACM. (back to page 100 and 107)

[50] Martin Hofmann and Ute Schmid. Data-driven detection of recursive program schemes. In Helder Coelho, Rudi Studer, and Micheal Wooldridge, editors, *Proceedings of the 19th European Conference on Artificial Intelligence*. IOS Press, 2010. (back to page 100)

[51] Martin Hofmann, Andreas Hirschberger, Emanuel Kitzelmannn, and Ute Schmid. Inductive Synthesis of Recursive Functional Programs – A Comparison of Three Systems. In *KI 2007: Advances in Artificial Intelligence*, volume 4667 of *LNCS*, pages 468–472. Springer-Verlag, 2007. (back to page 30 and 123)

[52] Martin Hofmann, Emanuel Kitzelmann, and Ute Schmid. Analysis and Evaluation of Inductive Programming Systems in a Higher-Order Framework. In A. Dengel, K. Berns, T. M. Breuel, F. Bomarius, and T. R. Roth-Berghofer, editors, *KI 2008: Advances in Artificial Intelligence, Proceedings of th* 31^{st} *Annual German Conference on Artificial Intelligence*, volume 5243 of *LNAI*, pages 78–86. Springer-Verlag, 2008. (back to page 30 and 123)

[53] Martin Hofmann, Emanuel Kitzelmann, and Ute Schmid. Porting IgorII from Maude to Haskell—Introducing a System's Design. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *Proceedings of the* 3^{rd} *ACM SIGPLAN Workshop on Approaches and Applications of Inductive Programming, Revised Papers*, volume 5812 of *Lecture Notes in Computer Science*, pages 140–158. Springer Berlin / Heidelberg, 2009. (back to page 124)

[54] Martin Hofmann, Emanuel Kitzelmann, and Ute Schmid. A Unifying Framework for Analysis and Evaluation of Inductive Programming Systems. In B. Goertzel, P. Hitzler, and M. Hutter, editors, 2^{nd} *Conference on Artificial General Intelligence*, pages 55–60. Atlantis Press, 2009. (back to page 30 and 123)

[55] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In *In ACM SIGPLAN International Conference on Functional Programming*, pages 73–82. ACM Press, 1996. (back to page 150)

[56] Mateja Jamnik, Manfred Kerber, and Christoph Benzmüller. Towards learning new methods in proof planning. In Michael Kohlhase Manfred Kerber, editor, *Symbolic computation and automated reasoning*, pages 142–158. A. K. Peters, Ltd., Natick, MA, USA, 2001. (back to page 23)

[57] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming*, pages 97–136, 1995. (back to page 59)

[58] J. P. Jouannaud and Y. Kodratoff. Characterization of a class of functions synthesized from examples by a summers like method using a "b.m.w." matching technique. In *IJCAI'79: Proceedings of the 6th international joint conference on Artificial intelligence*, pages 440–447, San Francisco, CA, USA, 1979. Morgan Kaufmann Publishers Inc. (back to page 30 and 33)

[59] Jean-Pierre Jouannaud and Yves Kodratoff. Program synthesis from examples of behavior. In Alan W. Biermann and Gérard Guiho, editors, *Computer Program Synthesis Methodologies*, pages 213–250. D. Reidel Publ. Co., 1983. (back to page 33)

[60] Stefan Kahrs. Genetic programming with primitive recursion. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 941–942, New York, NY, USA, 2006. ACM. (back to page 31)

[61] Susumu Katayama. Power of brute-force search in strongly-typed inductive functional programming automation. In *PRICAI'04: Trends in Artificial Intelligence*, LNCS, pages 75–84. Springer-Verlag, 2004. (back to page 39)

[62] Susumu Katayama. Library for systematic search for expressions. In *AIC'06: Proceedings of the 6th WSEAS International Conference on Applied Informatics and Communications*, pages 381–387, Stevens Point, Wisconsin, USA, 2006. World Scientific and Engineering Academy and Society (WSEAS). (back to page 39)

[63] Susumu Katayama. Systematic search for lambda expressions. In Marko C. J. D. van Eekelen, editor, *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005*, volume 6, pages 111–126. Intellect, 2007. (back to page 30, 39, and 40)

[64] Susumu Katayama. Efficient exhaustive generation of functional programs using monte-carlo search with iterative deepening. In *PRICAI*, pages 199–210, 2008. (back to page 40)

[65] Emanuel Kitzelmann. Analytical inductive functional programming. In M. Hanus, editor, *Proceedings of th 18th International Symposium on Logic-Based Program*

Synthesis and Transformation (LOPSTR 2008, Valencia, Spain), volume 5438 of *LNCS*, pages 87–102. Springer, 2008. (back to page 33 and 77)

[66] Emanuel Kitzelmann. *A Combined Analytical and Search-Based Approach to the Inductive Synthesis of Functional Programs*. PhD thesis, Otto-Friedrich-Universität Bamberg, 2010. (back to page 30, 77, 99, 123, 124, 127, 133, and 150)

[67] Emanuel Kitzelmann. Inductive programming: A survey of program synthesis techniques. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *Approaches and Applications of Inductive Programming, 3rd Workshop AAIP, Revised Papers*, volume 5812 of *LNCS*, pages 50–73. Springer-Verlag, 2010. (back to page 30)

[68] Emanuel Kitzelmann and Martin Hofmann. IgorII: An Inductive Functional Programming Prototype. In Mailik Ghallab, Constantine D. Spyropoulos, Nikos Fakotakis, and Nikos Avouris, editors, *Proceedings of the System Demonstrations of the* 18^{th} *European Conference on Artificial Intelligence*, pages 29–30, 2008. (back to page 77)

[69] Emanuel Kitzelmann and Ute Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research*, 7:429–454, 2006. (back to page 30, 33, and 77)

[70] Y. Kodratoff. A class of functions synthesized from a finite number of examples and a LISP program scheme. *International Journal of Computer and Information Science*, 8:489–521, 1979. (back to page 33)

[71] Y. Kodratoff and J. Fargues. A sane algorithm for the synthesis of LISP functions from example problems: The Boyer and Moore algorithm. In *Proceedings of the AISB/GI Conference on Artificial Intelligence*, pages 169–175, Hamburg, 1978. AISB and GI. (back to page 33)

[72] Pieter Koopman and Rinus Plasmeijer. Generic generation of elements of types. In *In Sixth Symposium on Trends in Functional Programming (TFP 2005)*, pages 23–24, 2005. (back to page 36)

[73] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: Generic automated software testing. In *The 14th International Workshop on the Implementation of Functional Languages, IFL 2002, Selected Papers*, volume 2670 of *Lecture Notes in Computer Science*, pages 84–100. Springer, 2002. (back to page 36)

[74] Pieter W. M. Koopman and Rinus Plasmeijer. Systematic synthesis of functions. In Henrik Nilsson, editor, *Revised Selected Papers from the Seventh Symposium on Trends in Functional Programming, TFP 2006*, volume 7, pages 35–54. Intellect, 2007. (back to page 30)

[75] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992. (back to page 31)

[76] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge, MA, USA, 1994. (back to page 31)

[77] John R. Koza. *Genetic Programming III: Darwinian Invention and Problem Solving.* Morgan Kaufmann Publishers, 1998. (back to page 31)

[78] John R. Koza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence.* Kluwer Academic Publishers, Norwell, MA, USA, 2003. (back to page 31)

[79] Joachim Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103(2):151–161, April 1968. Springer Berlin / Heidelberg. (back to page 62 and 64)

[80] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. Learning repetitive text-editing procedures with smartedit. *Your Wish Is My Command: Programming by Example*, pages 209–226, 2001. (back to page 25)

[81] Guillaume Le Blanc. BMWk revisited: generalization and formalization of an algorithm for detecting recursive relations in term sequences. In *ECML-94: Proceedings of the European conference on Machine Learning*, pages 183–197, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc. (back to page 33)

[82] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *In Proceedings of the 22nd ACM Symposium on Principles of Programming Languages. ACMPress*, 1995. (back to page 40)

[83] Donato Malerba. Learning recursive theories in the normal ILP setting. *Fundamenta Informaticae*, 57:39–77, 2003. IOS Press. (back to page 29)

[84] Ernest G. Manes and Michael A. Arbib. *Algebraic approaches to program semantics.* Springer-Verlag New York, Inc., New York, NY, USA, 1986. (back to page 61)

[85] Richard G. McDaniel and Brad A. Myers. Getting more out of programming-by-demonstration. In *CHI '99: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 442–449, New York, NY, USA, 1999. ACM Press. (back to page 25)

[86] Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, September 1992. (back to page 72 and 74)

[87] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of th 5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, MA, USA*, pages 124–144. Springer-Verlag, 1991. (back to page 58 and 156)

[88] Robin Milner. Logic for computable functions: description of a machine implementation. Technical report, Stanford University, Stanford, CA, USA, 1972. (back to page 22)

[89] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill Higher Education, 1997. (back to page 26, 130, and 133)

[90] Alan Bundy, Moa Johansson, Lucas Dixon. Case-analysis for rippling and inductive proof. Technical report, Mathematical Reasoning Group, University of Edinburgh, 2010. (back to page 23)

[91] Alan Bundy, Moa Johansson, Lucas Dixon. Conjecture synthesis for inductive theories. In *Proceedings of the Interactive Theorem Proving Conference*, number 6172 in LNCS. Springer, 2010. (back to page 23)

[92] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3:199–230, 1994. (back to page 32)

[93] S.H. Muggleton and J. Firth. CProgol4.4: a tutorial introduction. In S. Dzeroski and N. Lavrac, editors, *Relational Data Mining*, pages 160–188. Springer-Verlag, 2001. (back to page 29)

[94] Stephen Muggleton. Inductive logic programming. *New Generation Computing*, 8 (4):295–318, 1991. (back to page 29)

[95] Stephen Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995. (back to page 29)

[96] Stephen Muggleton. Learning from positive data. In *ILP '96: Selected Papers from the 6th International Workshop on Inductive Logic Programming*, pages 358–376, London, UK, 1997. Springer-Verlag. (back to page 29)

[97] Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, pages 368–381. Ohmsma, Tokyo, Japan, 1990. (back to page 29)

[98] Martin Mühlpfordt. Syntaktische Inferenz rekursiver Programmschemata. Master's thesis, Technische Universität Berlin, 2000. (back to page 33 and 77)

[99] Brad A. Myers. Demonstrational interfaces: A step beyond direct manipulation. *Computer*, 25(8):61–73, 1992. (back to page 25)

[100] Brad A. Myers, Allen Cypher, David Maulsby, David C. Smith, and Ben Shneiderman. Demonstrational interfaces: Coming soon? In *CHI '91: Proceedings of the SIGCHI Conference on Human factors in Computing Systems*, pages 393–396, New York, NY, USA, 1991. ACM Press. (back to page 25)

[101] Brad A. Myers, Richard McDaniel, and David Wolber. Programming by example: intelligence in demonstrational interfaces. *Commun. ACM*, 43(3):82–89, 2000. (back to page 25)

[102] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. (back to page 22)

[103] Robert P. Nix. Editing by example. *ACM Trans. Program. Lang. Syst.*, 7:600–621, 1985. (back to page 25)

[104] Roland J. Olsson. *Inductive functional programming using incremental program transformation and Execution of logic programs by iterative-deepening A* SLD-tree search*. Dr scient thesis, University of Oslo, Norway, 1994. (back to page 30 and 31)

[105] Roland J. Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55–83, 1995. (back to page 31)

[106] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 2008. (back to page 59 and 171)

[107] Derek Partridge. The case for inductive programming. *Computer*, 30(1):36–41, 1997. (back to page 25)

[108] Lawrence C. Paulson. *Isabelle: a Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer – Berlin, 1994. (back to page 22)

[109] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`. (back to page 171)

[110] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists (Foundations of Computing)*. The MIT Press, August 1991. (back to page 51)

[111] Rinus Plasmeijer and Marko van Eekelen. Concurrent clean lamguage report (version 2.1.1). Technical report, Radboud University Nijmegen, 2005. URL `www.cs.ru.nl/~clean`. (back to page 36)

[112] G. Plotkin. A further note on inductive generalization. In *Machine Intelligence*, volume 6. Edinburgh University Press, 1971. (back to page 47)

[113] G.D. Plotkin. A note on inductive generalisation. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163. Edinburgh University Press, Edinburgh, 1969. (back to page 47)

[114] G.D. Plotkin. *Automatic Methods of Inductive Inference*. PhD thesis, Edinburgh University, August 1971. (back to page 29)

[115] J. Ross Quinlan. Learning first-order definitions of functions. *Journal of Artificial Intelligence Research*, 5:139–161, 1996. (back to page 29)

[116] J. Ross Quinlan and R. Mike Cameron-Jones. FOIL: A midterm report. In *Machine Learning: ECML-93, European Conference on Machine Learning, Proceedings*, volume 667, pages 3–20. Springer-Verlag, 1993. (back to page 29)

[117] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423, London, UK, 1974. Springer-Verlag. (back to page 34)

[118] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965. (back to page 35)

[119] U. Schmid. *Inductive Synthesis of Functional Programs: Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003. (back to page 33 and 77)

[120] Ute Schmid, Martin Hofmann, and Emanuel Kitzelmann. Analytical Inductive Programming as a Cognitive Rule Acquisition Device. In B. Goertzel, P. Hitzler, and M. Hutter, editors, 2^{nd} *Conference on Artificial General Intelligence.* Atlantis Press, 2009. (back to page 127)

[121] Ute Schmid, Martin Hofmann, and Emanuel Kitzelmann. Inductive Programming - Example-driven Construction of Functional Programs. *KI – Künstliche Intelligenz*, 23(2):38–41, 2009. (back to page 33 and 77)

[122] Ute Schmid, Martin Hofmann, Florian Bader, Tilmann Häberle, and Thomas Schneider. Incident Mining using Structural Prototypes. In Nicolás García-Pedrajas, Herrera Francisco, Colin Fyfe, José Manuel Benítez, and Moonis Ali, editors, *Trends in Applied Intelligent Systems: 23rd International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems*, volume 6097 of *LNAI*, pages 327–336. Springer Berlin / Heidelberg, 2010. (back to page 156)

[123] Ben Shneiderman. Direct manipulation: A step beyond programming languages (abstract only). In *CHI '81: Proceedings of the joint conference on Easier and more productive use of computer systems. (Part - II)*, page 143, New York, NY, USA, 1981. ACM. (back to page 25)

[124] Bernard Silver. *Meta-Level Inference: Representing and Learning Control Information in Artificial Intelligence.* Elsevier Science Inc., New York, NY, USA, 1986. (back to page 23)

[125] D. R. Smith. A survey of the synthesis of LISP programs from examples. In A.W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques.* MacMillan Co., 1982. (back to page 30)

[126] J. Michael Spivey. Combinators for breadth-first search. *J. Funct. Program.*, 10 (4):397–408, 2000. (back to page 40)

[127] Mike Spivey and Silvija Seres. The algebra of searching. In *Proceedings of a symposium in celebration of the work of*. MacMillan, 2000. (back to page 40)

[128] Phillip D. Summers. A methodology for LISP program construction from examples. *Journal ACM*, 24:162–175, 1977. (back to page 25, 30, 32, 33, and 77)

[129] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003. (back to page 43 and 47)

[130] Simon Thompson. *Haskell: The Craft of Functional Programming (2nd Edition)*. Addison Wesley, March 1999. (back to page 171)

[131] Varmo Vene. *Categorical Programming with Inductive and Coinductive Types*. PhD thesis, Faculty of Mathematics, University of Tartu, Estonia, 2000. (back to page 58, 61, and 156)

[132] Varmo Vene and Tarmo Uustalu. Functional programming with apomorphisms (corecursion). In *9th Nordic Workshop on Programming Theory*, 1998. (back to page 156)

[133] Philip Wadler. Theorems for free! In *FUNCTIONAL PROGRAMMING LANGUAGES AND COMPUTER ARCHITECTURE*, pages 347–359. ACM Press, 1989. (back to page 58)

[134] Man Leung Wong and Tuen Mun. Evolving recursive programs by using adaptive grammar based genetic programming. In *MACHINES*, pages 421–455, 2005. (back to page 31)

[135] Serap Yilmaz. Inductive synthesis of recursive logic programs. Master's thesis, University of Bilkent, Computer Science Department, 1997. (back to page 29 and 33)

[136] Tina Yu. *An analysis of the impact of functional programming techniques on genetic programming*. PhD thesis, University College London, Department of Computer Science, 1999. (back to page 34)

[137] Tina Yu. Polymorphism and genetic programming. In *In Proceedings of the Fourth European Conference on Genetic Programming*, pages 87–100. Springer-Verlag, 2001. (back to page 32)

[138] Tina Yu. A higher-order function approach to evolve recursive programs. In Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice III*, volume 9 of *Genetic Programming*, chapter 7, pages 93–108. Springer, Ann Arbor, 12-14 May 2005. (back to page 30, 32, and 34)

[139] Tina Yu and Chris Clack. Polygp: a polymorphic genetic programming system in haskell. In *Proc. of the 3rd Annual Conf. Genetic Programming*, pages 416–421. Morgan Kaufmann, 1998. (back to page 34)

[140] Tina Yu and Chris Clack. Recursion, lambda-abstractions and genetic programming. In Riccardo Poli, W. B. Langdon, Marc Schoenauer, Terry Fogarty, and Wolfgang Banzhaf, editors, *Late Breaking Papers at EuroGP'98: The First European Workshop on Genetic Programming*, pages 26–30. CSRP-98-10, The University of Birmingham, UK, 1998. (back to page 32)

A. Haskell Reference

This appendix summarises the basic constructs of the purely functional programming language Haskell. It is not intended to give a comprehensive introduction into this language, but to serve as a reference to help the reader not familiar in particular with Haskell, but also not completely unaware of functional programming in general, to follow the code examples given in this text. For a more comprehensive introduction the reader is referred to standard text books, as e.g. by Bird [12], O'Sullivan et al. [106], or Thompson [130]. For the language and library specification see [109]. In the Haskell Wiki[1] a collection of various Haskell related books can be found at `http://www.haskell.org/haskellwiki/Books`.

A.1. Types and Values

In Haskell, as a *functional programming language*, computation is done by **evaluating expressions** yielding **values**. A value is an expression in a normal form that cannot be evaluated further. An expression is a syntactic term built from functions and values. Since Haskell is a strongly typed language, every expression has a **type** which describe values, in some sense.

Names can be assigned to expression—or expressions bound to names—which are here often called variables, but more precisely **bindings** or **identifiers**. Multiple equation starting with the same identifier are called **function definition**, **function binding**, or **binding group**. Following two simple function definitions:

```
five     = 5
double x = 2*x
```

It is important, that variables in Haskell are different from variables in other languages, because they really name an expression and do not allocate any memory. Thus, a name is unique and cannot be assigned twice in the same scope. A binding can take one or more arguments which are called **variables**. In fact, there is not much difference between variables and bindings, only the way they are bound differs. Therefore, both are written in lower case.

A.1.1. User-Defined Types and Synonyms

Types can be defined in Haskell with a `data` declaration. It consists of a **type constructor** and multiple **data constructors** (or just **constructors**), both capitalised.

[1] `http://www.haskell.org/haskellwiki/`

A type of different colours could be defined as follows:

```
data Colour = Red | Green | Blue | Yellow
```

The type constructor can take multiple types as arguments to create a parameterised type, as e.g.

```
data Pair α = P α α
```

or as in a recursive definition:

```
data Tree α = Leaf α | Branch (Tree α) (Tree α)
```

Such types are called **polymorphic types**, because they are universally quantified over all types.

Notation: Quantified type arguments are written in small Greek characters to facilitate the distinction to arguments of functions, i.e. bindings. As common in Haskell, `( :: )` (read "type of") denotes the type of a term.

Thus, in some sense `(Pair α)` describes a family of types, because for every type α there exists a pair of α:

```
P Red Green                        :: Pair Colour
P (P Red Green) (P Blue Yellow) :: Pair (Pair Colour)
```

Another useful type-related construct is the the keyword `type` which defines type synonyms, e.g. :

```
type ColouredPoint = Point Colour
```

A.1.2. Predefined and Built-In Types

Of course, Haskell comes equipped with some predefined and built-in types which will be introduced now shortly.

The Unit Type is a special case of particular use, but simply emerges from Haskell's theoretical foundations. It only has a single value:

```
data () = ()
```

Booleans are represented by the values `True` and `False` of type `Bool` and defined as:

```
data Bool = True | False
```

Characters are denoted by `Char`, a value of type `Char` by writing the character in single quotes, e.g. :

```
'a' :: Char
'1' :: Char
'!' :: Char
```

Note that characters are really built in, but from a basic understanding `Char` can be thought as an enumerated data type consisting only of nullary constructors:

```
data Char = 'a' | 'b' | 'c' | ... -- not
          | 'A' | 'B' | 'C' | ... -- valid
          | '1' | '2' | '3' | ... -- Haskell
          ...                     -- syntax!
```

Integers are represented by their number. `Integer` is the name of the type, fixed precision integers are of type `Int`. From the basic understanding of data types, they would be consistently defined as:

```
data Int     = -65532 |...| -1 |0 |1 |...| 65532-- invalid
data Integer =      ...-2 |-1 |0 | 1 | 2 ...    -- syntax!
```

Lists are a comma separated sequence of values enclosed in squared brackets, e.g. :

```
[]         :: [a]
[1]        :: [Integer]
['a','b']  :: [Char]
```

The empty list is denoted by `[]` and a typical example of a polymorphic type. Note that the squared brackets are syntactic sugar, and lists can be treated as if they would have been defined as

```
data [α] = [] | α : [α]   -- not valid Haskell syntax
```

using the constructor `[]` and the right associative infix constructor `(:)`. Thus, `[1,2,3]` is equivalent to `1:2:3:[]`.

Strings are just lists of characters and enclosed in double quotes, e.g. :

```
"Haskell is fun" :: String
```

As usual, `String` is a just a synonym for `Char`, as defined by

```
type String = [Char]
```

and `"abc"` is syntactic sugar for `['a','b','c']`.

Tuples are a comma separated sequence of values enclosed in parentheses, e.g. :

```
(1,'1')                    :: (Integer,Char)
("two",2,'2')              :: (String,Integer,Char)
((3,'3'),"foo",42,True)    :: ((Integer,Char),String,Integer
                              ,Bool)
```

As with lists, they are built into the compiler, but the understanding is that they would have been defined as:

```
data (α,β)   = (α,β)    -- not valid
data (α,β,γ) = (α,β,γ)  -- Haskell
...                     -- syntax!
```

Sometimes, special prefix operators for tuples are used:

```
(,) a b    = (a,b)
(,,) a b c = (a,b,c)
...
```

Disjoint sums are implemented in HASKELL quite intuitively, because they can take on a value *either* of the *left* type α or the *right* type β.

```
data Either α β = Left α | Right β
```

`Either` α β is often used to model exception, where the `Left` constructor is used to hold an error value, and the `Right` constructor to hold a correct value.

Maybe is used to encapsulate optional values. The type `Maybe` α either contains a value of type α or it is empty.

```
data Maybe α = Just α | Nothing
```

Function types are the type of function which map from one type to another. The arrow $\rightarrow$ is the accordant type constructor. For example a function pair which constructs a pair from its two arguments has type $\alpha \rightarrow \beta \rightarrow (\alpha, \beta)$ defined as

```
pair :: α → β → (α,β)
pair a b = (a,b)
```

Note that $\rightarrow$ associates to the right, i.e. the type of `pair` is in fact $\alpha \rightarrow (\beta \rightarrow (\alpha, \beta))$. Reading the type of `pair` like this, it is a function taking a value of type α and returning a function of type $\beta \rightarrow (\alpha, \beta)$.

Note that anywhere in the code, type annotations can be included using (::) to explicitly force the type of an expression. For example:

```
pair a b = ( (a::Char),(b::Int) )
```

A.2. Functions

A.2.1. Lambda Abstractions

As indicated earlier, a function definition in HASKELL is nothing but an expression bound to a variable. However, it is possible to define an anonymous function, called **lambda abstraction**. Usually, they are written `(\a b → a * b)`. However, the fancier λ is used here. Keeping this in mind, we can bind an expression to `mult`.

```
mult a b = a * b
```

Now `mult` can be used as a shorthand for:

```
λa b → a * b
```

A.2.2. Infix, Prefix, and Sections

So far, all functions have been defined in **prefix notation**. Very naturally a function can be defined in **infix notation** by putting the name (the identifier) between the first and the second argument. The following definition is the standard definition in HASKELL for **function composition** and also a typical example of a higher-order function, i.e. a function taking functions as arguments:

```
( ∘ ) :: (β →γ) →(α → β) →(α →γ)
f ∘ g = λ x → f (g x)
```

In the type declaration it is required to enclose the function in parenthesis and explicitly make it prefix, which is necessary for the type declaration. This is called **sectioning**, i.e. a **section** is a function with partially applied arguments. For example:

```
(x*)  ≈   λy   → x*y
(*y)  ≈   λx   → x*y
(*)   ≈   λx y → x*y
```

Sectioning is especially useful with functions in infix notation, but practically any function can be turned into a section. For example, the previous function `mult` a function `double` can be defined as:

```
double a = (mult 2) a
```

Vice versa, any function `f` can be made infix by enclosing it in backward quotes `‘f‘`.

```
double a = 2 ‘mult‘ a
```

A.2.3. Pointwise and Pointfree

Usually, functions are written **curried**, i.e. with multiple arguments:

```
mult :: Integer → Integer → Integer
mult a b = a * b
```

Uncurried, the same function would only take a single argument, namely a tuple:

```
mult :: (Integer,Integer) → Integer
mult (a,b) = a * b
```

Combining the use of uncurried functions, sections and function composition leads to an interesting programming style called **pointfree programming**[2]. Consider for example the following definition. It just names the expression which takes an integer and quadruples it:

```
quadruplicate :: Integer → Integer
quadruplicate = (2*) ∘ (2*)
```

A.3. Pattern Matching, Case Expressions and Control Structures

Pattern matching in Haskell is a quite simple and intuitive facility to determine which equation of a function definition to evaluate, given a specific input. The arguments after the identifier in a function binding are called patterns. Patterns may consist of the constructors of any type, including tuples, strings, numbers, characters, etc., as well as variable, which are bound after matching. For example:

```
f :: [α] → [α]
f []     = []
f(x:xs) = x:x:(f xs)
```

To facilitate the use of patterns, a couple of constructs exist.

As-patterns (@) allow to bind a pattern to a name and reuse it on the right-hand side of the equation. So instead of

```
f (x:xs)   = x:(x:xs)
```

on may write

```
f l@(x:xs) = x:l
```

Wildcards (_) match against any value, without binding it to a name. This often makes code more readable, because it can emphasise that parts of the input do not matter:

```
fst (a,_) = a
snd (_,b) = b
```

[2]Sometimes it is also referred to as *pointless*, because a program can get quite intricate if this becomes rampant. Ironically, pointless programming usually leads to more points (∘).

Case-Expressions (`case of`) can do a pattern matching in one equation instead of splitting it into a binding group with multiple equations:

```
f l = case l of
        []     → []
      (x:xs)  → x:x:xs
```

Case expressions are checked top-down, wildcards can be used for a default decision:

```
g b = case b of
       True  → "True"
         _   → "False"
```

Conditionals (`if _ then _ else`) are just a shorthand for the `case`-expression above:

```
g b = if b then "True" else "False"
```

Guards are constructs to avoid nested conditionals. As with patterns, they are evaluated top-down, and the first that evaluates to `True` results in a successful match. Often `otherwise` is used in the last guard for readability's sake, which is simply defined as `otherwise = True`. For example:

```
sign x | x > 0      =  1
       | x ≡ 0      =  0
       | otherwise  = -1
```

Local variables can be defined in HASKELL in two ways. With `let`-expressions it is possible to make definitions local to an expression e.g. :

```
foo = let x = 2
          y = 3
      in x*y
```

Another possibility to introduce local variables are `where` clauses, e.g. :

```
foo x = f (x + y)
   where
   y   = 2
   f x = if x ≡ 2 then "bar" else "baz"
```

It is important to notice, that `where` opens a new scope of variables, where `let` does not.

A.4. Type Classes and Overloading

As previously mentioned is a type a collection of values. Furthermore, types which share certain functionalities, i.e. functions, can be grouped to **type classes**. The following **class declaration** can be read as "type α is instance of class `Eq`, if the two overloaded functions (≡) and (≢) are defined on it". Furthermore, two default implementations are given.

```
class Eq α where
  (≡), (≢) :: α → α → Bool

  x ≢ y     = not (x ≡ y)
  x ≡ y     = not (x ≢ y)
```

Since the default implementation are mutual recursive, at least one must be overwritten in an **instance declaration**:

```
instance Eq Bool where
  x ≡ y = if x then y else not y
```

If type classes can depend on each other, they are called **derived classes**. The simplest example of a derived class is the class of ordered types `Ord` which depends on the equality class `Eq`. The following code reads as "a type α which is in class `Eq` is in class `Ord` if the functions `compare`, <, ≤, ≥, >, `max`, `min` are defined on it":

```
class (Eq α) ⇒ Ord α where
  compare              :: α → α → Ordering
  (<), (≤), (≥), (>)   :: α → α → Bool
  max, min             :: α → α → α
```

Instance declarations for some standard classes, as e.g. `Eq`, `Ord`, `Show` can automatically be derived using `deriving`:

```
data Colour = Red | Green | Blue | Yellow
   deriving (Ord,Show)
```

Then, equality is just syntactic equality, the ordering is given by the constructor definition, and instances of `Show` implement a function `show :: (Show α) ⇒ α → String`. This already exemplary shows the syntax of how polymorphic function types can be constraint to certain type classes. Another common example is the `elem` function, which checks whether an element is contained in a lost or not. It requires the type of elements in the input list to be instance of `Eq`.

```
elem              :: (Eq α) ⇒ α → [α] → Bool
a `elem` []       = False
a `elem` (x:xs) = a ≡ x ∨ a `elem` xs
```

A.5. Modules

Related parts of HASKELL programs are organised in **modules**. A single line **comment** is starts with two dashes (--) and ends with a newline. Multiline comments are enclosed in {- and -}. A module declaration starts with the keyword `module` and the *qualified name* of the module. For example:

```
module MyModule where
...
```

It is possible to hierarchically organise modules. Then, the qualified module name must contain the complete path, where folders are separated by colons. For example, if a Module "MyModule" lies in a folder "Package" below the top-level working directory the following qualified name must be used.

```
module Package.MyModule where
...
```

A.5.1. Module exports

Per default, all declarations (of functions, types, classes) in a module are exported. To limit the exports, an export list follows the module name as an **export declaration**. The following example defines an ADT for points, exporting only the type `Point`, but not its type constructor, and functions to construct, destruct and access points:

```
module Point (Point, topair, frompair, xcor, ycor) where

data Point α = P α α

topair          :: P α → (α,α)
topair (P a b) = (a,b)

frompair        :: (α,α) →P α
frompair (a,b) = P a b

xor, ycor     ::  P α → α
xcor (P a _) = a
ycor (P _ b) = b
```

Note that the data constructor `P` of the type `Point` is not exported, but hidden. To explicitly export the type constructors, each must be included in the export list, e.g. `Point(P)` or a wildcard must be used to export all type constructors of a given type, e.g. `Point(..)`.

A.5.2. Module imports

A **import declaration** starts with the keyword `import` followed by the module name. If nothing is specified everything exported by that module is imported.

```
import Point
```

It is possible to get more control over the imports by explicitly listing or hiding identifiers.

```
-- import only the type Point
import Point (Point)

-- import all but the type Point
import Point hiding (Point)
```

Per default a standard module called `Prelude` is imported. It contains all standard functions defined in Haskell. Sometimes it is desired to avoid name clashes. Then **qualified imports** can be used which additionally allows to rename modules.

```
module MyPair (Pair, pair, fst, snd) where

import qualified Prelude as P
import qualified Point

type Pair α = Point.Point α

pair       :: α →α →Pair α
pair a b    = Point.fromPair (a,b)

fst,snd    :: (Pair α) →α
fst         = Point.xcor
snd         = P.snd ∘ Point.topair
```

The code is a bit artificial, but it defines an ADT `Pair` based on `Point` by simply using a type synonym for `Pair`. Its constructor `pair` is defined in terms of the function `frompair` from the module `Point`, which was imported qualified. Furthermore, two functions `fst` and `snd` are redefined, both in terms of the modules `Point` and `Prelude`. To avoid names clashes with the `Prelude` it was imported qualified, too.

A.6. Recursion Schemes in Haskell

The various morphisms described in section 4.2 can all be implemented in Haskell as recursion schemes.

A.6.1. Reduce-Map-Filter of Lists

One of the most basic recursion scheme is that for structural recursion over lists. With catamorphism over lists the so called **map-reduce-filter** scheme can be implemented. In Haskell reduce is usually known as `foldr` which replaces each cons-constructor `(:)` of a list with a call to its first argument, and each empty list constructor `[]` with its second argument:

```
foldr              :: (α →β →β) →β →[α] →β
foldr _ z []        =  z
foldr f z (x:xs)    =  x ‘f‘ (foldr f z xs)
```

Closely related to `foldr` is `map`, another name for the type functor of lists. It applies its first argument to each element of its input list. In terms of `foldr` it would be defined as:

```
map f = foldr (λ e l → (f e) : l) []
```

An alternative, but more readable definition is the following:

```
map :: (α → β) → [α] → [β]
map _ []     = []
map f (x:xs) = f x : map f xs
```

Finally, if depending on a predicate certain elements are discarded from a list, the higher-order function **filter** is used. Similar to `map`, it is a special catamorphism and can be defined in terms of `foldr`:

```
filter p = foldr (λe l → if p then e:l else l)
```

The common definition is the following without catamorphism:

```
filter :: (α → Bool) → [α] → [α]
filter _p []     = []
filter p (x:xs)
    | p x        = x : filter p xs
    | otherwise  = filter p xs
```

A.6.2. The Functor class

The **Functor** class is used for types that can be mapped over, i.e. provides an interface for the implementation of a type functor.

```
class  Functor φ where
fmap         :: (α → β) → φ α → φ β
```

Per convention, instances of `Functor` are required to satisfy the following laws:

```
fmap id      ≡ id
fmap (f ∘ g) ≡ fmap f ∘ fmap g
```

For each data type which has a type functor, it must be explicitly defined. The standard definition for the type functor for lists is simply the function `map` from above.

```
instance Functor [α] where
  fmap = map
```

Consider for example a data type for binary trees. Its type functor would be defined as follows.

```
data Tree α = N | B α (Tree α) (Tree α)

instance Functor (Tree α) where
   fmap f N         = N
   fmap f (B a l r) = B (f a) (fmap f l) (fmap f r)
```

A.6.3. Pointless Schemes

The pointless-haskell library *pointless-haskell*[3] , available from `hackageDB`[4] is a pointfree combinator library for programming with recursion patterns

[3] http://hackage.haskell.org/package/pointless-haskell
[4] http://hackage.haskell.org

defined as polytypic functions. A **polytypic** function is a function that is defined by induction on the structure of user-defined data types.

Pointless-haskell uses two GHC extensions to define type operators and type families. **Type operators** allow to define operators on data types similar to operators on expressions. **Indexed type families**, or just type families, are a Haskell extension for ad-hoc overloading of data types. Type families are parametric types that can be assigned specialised representations based on the type parameters they are instantiated with. They are the data type analogue of type classes: families are used to define overloaded data in the same way that classes are used to define overloaded functions. These options are set via GHC pragmas in the header of a module declaration:

```
{-# OPTIONS_GHC -XTypeOperators -XTypeFamilies   #-}
```

One of the main building blocks of this library is the type family `PF` of *pattern functors* of data types. Instances of this type family consist of identity functors (`Id`), constant functors (`Const`), sums of functors (⊕), products of functors (⊗), and composed functors (`:@:`).

The second building block is the `class Mu`, providing the value-level translation between data types and their representations as sum of products. It provides two class functions. One for packing a sum of products into one equivalent data type (`inn`), and one to unpack a data type into the equivalent sum of products (`out`). From a categorical point of view, `inn` corresponds to the initial algebra of a data type, `out` is its inverse. Listing A.1 shows some examples of the standard inductive types for natural numbers, lists, and various trees follow, defining their instances of the type family `PF` and the type class `Mu`.

Listing A.1: Type definitions and instance declarations for common inductive data types as used with the `pointless-haskell` library.

```
import Generics.Pointless.Combinators
import Generics.Pointless.Functors
import Generics.Pointless.RecursionPatterns

--
-- data type definitions
--

-- Peano's Natural Numbers
data Nat     = Z       | S Nat deriving (Show)

-- Cons Lists
data List α  = NilL    | Cons α (List α)
   deriving (Show)

-- Binary Node Trees
data NTree α = NilT    | Node α (NTree α)(NTree α)
   deriving (Show)

```

```
-- Binary Leaf Trees
data LTree α = Leaf α |Branch (LTree α) (LTree α)
   deriving (Show)

-- Rose Trees, arbitrary branching trees
data Rose α  = Forest α [Rose α] deriving (Show)

--
-- defining type family instances of PF,
-- and type class instances of Mu
--

type instance PF Nat = Const One ⊕ Id

instance Mu Nat where
    inn (Left  _) = Z
    inn (Right p) = S p
    out Z         = Left ⊥
    out (S p)     = Right p

type instance PF (List α) =
   Const One ⊕ (Const α ⊗ Id)

instance Mu (List α) where
    inn (Left _)      = NilL
    inn (Right (a,l)) = Cons a l
    out NilL          = Left ⊥
    out (Cons a l)    = Right (a,l)

type instance PF (NTree α) =
   Const One ⊕ (Const α ⊗(Id ⊗ Id))

instance Mu (NTree α) where
    inn (Left _)          = NilT
    inn (Right (a,(l,r))) = Node a l r
    out NilT              = Left ⊥
    out (Node a l r)      = Right (a,(l,r))

type instance PF (LTree α) = Const α ⊕ (Id ⊗ Id)

instance Mu (LTree α) where
    inn (Left a)      = Leaf a
    inn (Right (l,r)) = Branch l r
    out (Leaf a)      = Left a
    out (Branch l r)  = Right (l,r)
```

```
type instance PF (Rose α) = Const α ⊗ ( [] :@: Id )

instance Mu (Rose α) where
    inn (a,rs)         = Forest a rs
    out (Forest a rs) = (a,rs)
```

The details of the pointless-haskell library are not of special interest here, but giving some examples of using catamorphisms and paramorphism might be suitable. The types of the polymorphic functions `cata` and `para`, as shown in Listing A.2, are quite intricate, so it might be wished-for to spend some words on them.

The type class `Mu` and the type family `PF` have already been introduced. The further defines the initial algebra and its inverse of a type, the latter describes a type as a functor pattern. The class `Functor` is just a polytypic extension of the class `Prelude.Functor`. Both, `cata` and `para`, have a restricted type signature, s.t. α is required to be an instance of `Mu` and `PF`, which itself must be a functor.

The first argument is a dummy value to force a type on the highly polymorphic function. Usually an explicitly typed undefined value $\perp$ is used. The second argument is the mediating function, i.e. a join of multiple functions, for each type constructor of α one. This is indicated by the types `Generics.Pointless.Functors.F α β` and `Generics.Pointless.Functors.F α (β,α)`, respectively, which are just an internal shorthand to express the structurally equivalent sum of products for some data type. The codomain of the mediating function is also the codomain of the morphism. Finally, the third argument is the input type, the last is, of course, the output type.

Listing A.2: Types of `cata` and `para`

```
cata :: (Generics.Pointless.Functors.Mu α,
          Generics.Pointless.Functors.Functor
            (Generics.Pointless.Functors.PF α)) ⇒
        α → (Generics.Pointless.Functors.F α β → β) → α → β

para :: (Generics.Pointless.Functors.Mu α,
          Generics.Pointless.Functors.Functor
            (Generics.Pointless.Functors.PF α)) ⇒
        α → (Generics.Pointless.Functors.F α (β,α)
            →   β) →α →β
```

The catamorphisms for addition and multiplication of natural numbers (Example 4.2.1), length of a list (Example 4.2.2), and for mirroring binary trees (Example 4.2.3) can be defined as shown in Listing A.3.

Listing A.3: Examples for catamorphisms `pointless-haskell` library

```
add, mult :: Nat → Nat → Nat
add  a = cata (⊥::Nat) (const a ⊕ S)
mult a = cata (⊥::Nat) (const Z ⊕ add a)

length :: [α] → Nat
```

```
length = cata (⊥::[α]) (const Z ⊕ S ∘ snd)

mirror :: (NTree α) → (NTree α)
mirror =
   cata (⊥::NTree α)
        (const NilT ⊕ (λ(a,(l,r)) → Node a r l))
```

The paramorphisms from Examples 4.2.5, Examples 4.2.6, and Examples 4.2.7, respectively, would be defined using the pointless-haskell library as shown in Listing A.4.

Listing A.4: Examples for paramorphisms using the `pointless-haskell` library

```
fact  :: Int → Int
fact  =
  para (⊥::Int)((const 1) ⊕ (λ(a,b) → a ∗ (b+1)))

tail  :: [a] → [a]
tail  = para (⊥::[a]) ((const []) ⊕ snd ∘ snd)

tails = para (⊥::[Char]) (f ⊕ g)
    where
      f _          = [[]]
      g (x,(fxs,xs)) = fxs:ys

subtrees = para (⊥::(NTree Int)) (f ⊕ g)
    where
      f _ = [NilT]
      g (v,((fl,l),(fr,r))) = (Node v l r):(fl++fr)
```

B. Using SKI calculus defining `last`

For the curious reader, the equivalent of the Haskell program

```
last [x]    = x
last (x:xs) = last xs
```

defined in the **SKI** calculus:

```
S(S(S(S(KS) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK)))(KI)))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(KK)
(KI)))) (S(KK) (KI))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (KI))))(S(S(KS) (S(S(KS) (S(KK)
(KS))) (S(KK) (KI)))) (S(KK) (KI))))) (S(S(KS)(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS)))(S(S(KS)
(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK)))
(S(KK) (KS)))))(S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS)))(S(S(KS) (S(KK) (KK))) (S(KK)
(KS))))) (S(S(KS) (S(S(KS) (S(KK) (KS)))(S(S(KS) (S(KK) (KK))) (S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK)
(KS))))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK)
(KS))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS)))))
(S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK)(KK))))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS)))))))
(S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK)))(S(KK) (KS))))) (S(S(KS)
(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK)))(S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK) (KK))))))) (S(S(KS)
(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS))))))))) (S(S(KS)
(S(S(KS) (S(KK) (KS)))(S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS)))))(S(S(KS) (S(S(KS)
(S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS)))(S(S(KS) (S(KK) (KK))) (S(KK) (KS))))) (S(S(KS) (S(S(KS) (S(KK)
(KS)))(S(S(KS) (S(KK) (KK))) (S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS))))))) (S(S(KS) (S(S(KS) (S(KK)
(KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS))))) (S(S(KS) (S(S(KS) (S(KK) (KS)))
(S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS)
(S(KK) (KK))) (S(KK)(KK))))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS))))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS)
(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK)))(S(KK) (KS))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK)
(KK)))(S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK) (KK))))))) (S(S(KS)(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK)
(KK))) (S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS))))))))) (S(S(KS) (S(S(KS) (S(KK) (KS)))(S(S(KS)
(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS)))))(S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS)
(S(KK) (KS)))(S(S(KS) (S(KK) (KK))) (S(KK) (KS))))) (S(S(KS) (S(S(KS) (S(KK) (KS)))(S(S(KS) (S(KK) (KK)))
(S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS))))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK)
(KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK)
(KK))))) (S(S(KS) (S(KK) (KK)))(S(KK) (KK))))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK)(KK))) (S(KK)
(KK))))) (S(S(KS) (S(KK) (KK))) (S(KK) (KK)))))))))(S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK)
(KS)))(S(S(KS) (S(KK) (KK))) (S(KK) (KS))))) (S(S(KS) (S(S(KS) (S(KK) (KS)))(S(S(KS) (S(KK) (KK))) (S(KK)
(KK))))) (S(S(KS) (S(KK) (KK))) (S(KK)(KK))))))) (S(S(KS) (S(KK) (KK))) (S(KK) (KI)))))))))) (S(S(KS)(S(S(KS)
(S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK)(KK))) (S(KK) (KS))))) (S(S(KS) (S(S(KS) (S(KK)
(KS))) (S(S(KS)(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS))))) (S(S(KS)(S(S(KS) (S(KK) (KS)))
(S(S(KS) (S(KK) (KK))) (S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS))))))) (S(S(KS) (S(S(KS) (S(KK)
(KS)))(S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS)))))(S(S(KS) (S(S(KS) (S(KK) (KS)))
(S(S(KS) (S(KK) (KK))) (S(KK) (KK)))))(S(S(KS) (S(KK) (KK))) (S(KK) (KK))))))) (S(S(KS) (S(S(KS) (S(KK) (KS)))
(S(S(KS) (S(KK) (KK))) (S(KK) (KK))))) (S(S(KS) (S(KK) (KK)))(S(KK) (KK))))))))) (S(S(KS) (S(S(KS) (S(KK)
(KS))) (S(S(KS) (S(KK)(KK))) (S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK) (KI))))))))))) (S(S(KS) (S(S(KS)
(S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS)))(S(S(KS) (S(KK) (KK))) (S(KK) (KS))))) (S(S(KS) (S(S(KS) (S(KK)
(KS)))(S(S(KS) (S(KK) (KK))) (S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK)(KK))))))) (S(S(KS) (S(S(KS) (S(KK)
(KS))) (S(S(KS) (S(KK) (KK)))(S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK) (KI)))))))) (S(S(KS)(S(S(KS) (S(KK)
(KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS)(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS)))))
(S(S(KS) (S(KK) (KK))) (S(KK) (KI)))))) (S(S(KS) (S(S(KS) (S(KK) (KS)))(S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS)
(S(KK) (KK))) (S(KK) (KS))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS)))(S(S(KS) (S(KK)
(KK))) (S(KK) (KS))))) (S(S(KS) (S(S(KS) (S(KK) (KS)))(S(S(KS) (S(KK) (KK))) (S(KK) (KK))))) (S(S(KS) (S(KK)
(KK))) (S(KK) (KS))))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK)
(KK))) (S(KK) (KS))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK)))
(S(KK) (KS))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK)(KK))))) (S(S(KS) (S(KK) (KK)))
(S(KK) (KS))))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK)))(S(KK)
(KS))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK)))(S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK)
(KK))))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK)
(KS))))))))) (S(S(KS) (S(S(KS) (S(KK) (KS)))(S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK)
(KS)))))(S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS)))(S(S(KS) (S(KK) (KK))) (S(KK) (KS)))))
(S(S(KS) (S(S(KS) (S(KK) (KS)))(S(S(KS) (S(KK) (KK))) (S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS)))))))
(S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS))))) (S(S(KS)
(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK) (KK))))))) (S(S(KS)
(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK)(KK))) (S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK) (KK)))))))))(S(S(KS)
(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS)))(S(S(KS) (S(KK) (KK))) (S(KK) (KS))))) (S(S(KS) (S(S(KS)
(S(KK) (KS)))(S(S(KS) (S(KK) (KK))) (S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK)(KK))))))) (S(S(KS) (S(S(KS)
(S(KK) (KS))) (S(S(KS) (S(KK) (KK)))(S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK) (KK))))))))))) (S(S(KS)(S(S(KS)
(S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK)(KK))) (S(KK) (KS))))) (S(S(KS) (S(S(KS) (S(KK)
(KS))) (S(S(KS)(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS))))) (S(S(KS)(S(S(KS) (S(KK) (KS)))
(S(S(KS) (S(KK) (KK))) (S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS))))))) (S(S(KS) (S(S(KS) (S(KK) (KS)))
(S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS)))))(S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS)
(S(KK) (KK))) (S(KK) (KK)))))(S(S(KS) (S(KK) (KK))) (S(KK) (KK))))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS)
(S(KK) (KK))) (S(KK) (KK))))) (S(S(KS) (S(KK) (KK)))(S(KK) (KK))))))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS)
(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK)
```

```
(KK))) (S(KK) (KK))))) (S(S(KS) (S(KK)(KK))) (S(KK) (KK))))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK)
(KK))) (S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK)(KI)))))))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS)
(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS))))) (S(S(KS) (S(KK)
(KK))) (S(KK) (KI)))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK)))
(S(KK) (KS))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK)))(S(KK) (KK))))) (S(S(KS) (S(KK) (KK)))
(S(KK) (KK))))))) (S(S(KS)(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KK))))) (S(S(KS) (S(KK) (KK)))
(S(KK) (KI)))))))) (S(KK) (KI))))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS)
(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS))))) (S(S(KS) (S(KK) (KK)))(S(KK) (KI)))))) (S(S(KS)
(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK)(KK))) (S(KK) (KK))))) (S(S(KS) (S(KK) (KK))) (S(KK) (KK))))))) (S(KK)
(KI)))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK)))(KI)))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS)
(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KS))))) (S(S(KS) (S(KK)
(KK))) (S(KK) (KI)))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK)))
(S(KK) (KS))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK)))(S(KK) (KK))))) (S(S(KS) (S(KK) (KK)))
(S(KK) (KK))))))) (S(S(KS)(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KK))) (S(KK) (KK))))) (S(S(KS) (S(KK) (KK)))
(S(KK) (KI)))))))) (S(KK) (KI)))))) I
```

Similarly, the equivalent of an exemplary input list `[1,2,3,4]`:

```
S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(KK) (KI)))) (S(S(KS) (S(KK) (KK))) (S(KK)I)))))
(S(S(KS) (S(KK) (KK))) (KI)) 1 (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(KK) (KI))))
(S(S(KS) (S(KK) (KK))) (S(KK)I))))) (S(S(KS) (S(KK) (KK))) (KI)) 2 (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS)
(S(S(KS) (S(KK) (KS))) (S(KK) (KI)))) (S(S(KS) (S(KK) (KK))) (S(KK)I))))) (S(S(KS) (S(KK) (KK))) (KI)) 3 (S(S(KS)
(S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(KK) (KI)))) (S(S(KS) (S(KK) (KK))) (S(KK)I)))))
(S(S(KS) (S(KK) (KK))) (KI)) 4 (KK))))
```

C. MagicHaskeller Specification

Listing C.1: MagicHaskeller specification

```
{--
After installation of MagicHaskeller run with:
ghci -fth -package MagicHaskeller Master_HaskellerP.hs
--}
{-# OPTIONS -XDeriveDataTypeable -XTemplateHaskell #-}
module Master_HaskellerP where

import MagicHaskeller hiding (listP)
import Test.QuickCheck
import Monad(liftM, liftM2)
import Text.Printf
import Control.Exception
import System.CPUTime
import Generics.Pointless.Combinators
import Generics.Pointless.Functors
import Generics.Pointless.RecursionPatterns

-- ===================================================
-- Data type declarations
-- ===================================================

data Nat = Z | S Nat
     deriving (Eq,Ord,Typeable,Show)
data List a = NilL | Cons a (List a)
     deriving (Eq,Ord,Typeable,Show)
data NTree a  = NilT | Node a (NTree a)(NTree a)
     deriving (Eq,Ord,Typeable,Show)
data LTree a = Leaf a | Branch (LTree a) (LTree a)
     deriving (Eq,Ord,Typeable,Show)
data Rose a = Forest a [Rose a]
     deriving  (Eq,Ord,Typeable,Show)
data Object = O1 | O2 | O3
     deriving  (Eq,Ord,Typeable,Show)
data Cargo = NOCARGO | IN Object Cargo
     deriving  (Eq,Ord,Typeable,Show)
data State = START | LOD Object State
           | UNL Object State | FLY State
     deriving  (Eq,Ord,Typeable,Show)
data Disc = D0 | D Disc
     deriving  (Eq,Ord,Typeable,Show)
data Action = NOOP | MV Disc Peg Peg Action
     deriving  (Eq,Ord,Typeable,Show)
```

```
data Peg = PegA | PegB | PegC
     deriving  (Eq,Ord,Typeable,Show)
data Color = Purple | Yellow
     deriving  (Eq,Ord,Typeable,Show)
data Size = Large | Small
     deriving  (Eq,Ord,Typeable,Show)
data Act = Dip | Stretch
     deriving  (Eq,Ord,Typeable,Show)
data Age = Adult | Child
     deriving  (Eq,Ord,Typeable,Show)
data Inflate = FF | TT
     deriving  (Eq,Ord,Typeable,Show)
data Weather = Sunny | Rain | Overcast | Hot | Cool
             | Mild | Warm | Cold | High |  Normal
             | Weak |Strong | Change | Same
     deriving  (Eq,Ord,Typeable,Show)
data LAge = Young | PrePresbyopic | Presbyopic
     deriving  (Eq,Ord,Typeable,Show)
data LPrescription = Myope | Hypermetrope
     deriving  (Eq,Ord,Typeable,Show)
data LAstigmatic = No | Yes
     deriving  (Eq,Ord,Typeable,Show)
data LTears = Reduced | Norml
     deriving  (Eq,Ord,Typeable,Show)
data LCLType = None | Hard | Soft
     deriving  (Eq,Ord,Typeable,Show)

-- ===================================================
-- MagicHaskeller component library, mostly taken from
-- MakicHaskeller.LibTH
-- ===================================================

----------------------------------------------
-- natural numbers
----------------------------------------------

-- data Nat = Z | S Nat
--              deriving (Eq, Ord, Typeable, Show)

natC = $(p [|( Z        :: Nat
             , S        :: Nat -> Nat
             , nat_cata :: Nat -> (a -> a) -> a -> a
            )|])

natP = $(p [|( Z        :: Nat
             , S        :: Nat -> Nat
             , nat_para :: Nat -> (Nat -> a -> a) -> a -> a
             )|])

nat_cata           :: Nat -> (a -> a) -> a -> a
nat_cata Z _ v      = v
nat_cata (S x) f v = f (nat_cata x f v)
```

```haskell
nat_para             :: Nat -> (Nat -> a -> a) -> a -> a
nat_para Z _ v       = v
nat_para (S x) f v = f (S x) (nat_para x f v)

-- background knowledge
nadd = $(p [|(natadd :: Nat -> Nat -> Nat) |])
nmlt = $(p [|(natmlt :: Nat -> Nat -> Nat) |])

natadd :: Nat -> Nat -> Nat
natadd Z x     = x
natadd x Z     = x
natadd (S n) x = natadd n (S x)

natmlt :: Nat -> Nat -> Nat
natmlt Z _     = Z
natmlt _ Z     = Z
natmlt (S n) x = natadd x (natmlt n x)

----------------------------------------------
-- lists
----------------------------------------------

listC = $(p [|( []          :: [a]
              , (:)         :: a -> [a] -> [a]
              , foldr       ::
                   (b -> a -> a) -> a -> (->) [b] a)
             |])
listP = $(p [|( []          :: [a]
              , (:)         :: a -> [a] -> [a]
              , list_para ::
                   (->) [b] (a -> (b -> [b] -> a -> a) -> a)
              )|])

list_para  :: [b] -> a -> (b -> [b] -> a -> a) -> a
list_para []     x f = x
list_para (y:ys) x f = f y ys (list_para ys x f)

-- background knowledge
lapp =  $(p  [|( (++) :: [a] -> [a] -> [a])|])
lsnc =  $(p  [|( snc  :: [a] -> a -> [a])|])
llst =  $(p  [|( last :: [a] -> a)|])

snc :: [a] -> a -> [a]
snc = (. return) . (++)

----------------------------------------------
-- Maybe
----------------------------------------------

mb = $(p [| ( Nothing :: Maybe a
            , Just    :: a -> Maybe a
```

```
            , maybe   :: a -> (b->a) -> (->) (Maybe b) a
            )|])

----------------------------------------------
-- Booleans
----------------------------------------------

bool = $(p [|( True  :: Bool
             , False :: Bool
             , iF    :: (->) Bool (a -> a -> a)
             )|])

iF             :: Bool -> a -> a -> a
iF True  t f = t
iF False t f = f

----------------------------------------------
-- Node Trees
----------------------------------------------

-- data NTree a  = NilT | Node a (NTree a)(NTree a)
ntree_para :: NTree a -> r
           -> (a -> NTree a -> NTree a -> r -> r -> r) -> r
ntree_para NilT     d _     = d
ntree_para (Node v l r) d f =
  f v l r (ntree_para l d f) (ntree_para r d f)

ntree_cata :: NTree a -> r -> (a -> r -> r -> r) -> r
ntree_cata NilT     d _     = d
ntree_cata (Node v l r) d f =
  f v (ntree_cata l d f) (ntree_cata r d f)

ntreeC = $(p [|( NilT       :: NTree a
               , Node       ::
                   a -> NTree a -> NTree a -> NTree a
               , ntree_cata :: NTree a -> r
                            -> (a -> r -> r -> r) -> r
               )|])
ntreeP = $(p [|( NilT       :: NTree a
               , Node       ::
                   a -> NTree a -> NTree a -> NTree a
               , ntree_para ::
                   NTree a -> r  ->
                   (a -> NTree a -> NTree a -> r ->
                    r -> r) -> r
               )|])

----------------------------------------------
-- pairs
----------------------------------------------

pair = $(p [| ( (,) :: a -> b -> ((,) a b)
```

```
              , fst :: ((,) a b) -> a
              , snd :: ((,) a b) -> b
              )|])

----------------------------------------------
-- Rocket types
----------------------------------------------

rockC = $(p [|( O1 :: Object, O2 :: Object, O3 :: Object
             , NOCARGO :: Cargo
             , IN :: Object -> Cargo -> Cargo
             , START :: State
             , LOD :: Object -> State -> State
             , UNL :: Object -> State -> State
             , FLY :: State -> State
             , cargo_cata ::
                 Cargo -> a -> (Object -> a -> a) -> a
             , state_cata ::
               State -> a -> (Object -> a -> a)
                     -> (Object -> a -> a) -> (a -> a) -> a
             )|] )

rockP = $(p [|( O1 :: Object, O2 :: Object, O3 :: Object
             , NOCARGO :: Cargo
             , IN :: Object -> Cargo -> Cargo
             , START :: State
             , LOD :: Object -> State -> State
             , UNL :: Object -> State -> State
             , FLY :: State -> State
             , cargo_para ::
                 Cargo -> a ->
                 (Object -> Cargo -> a -> a) -> a
             , state_para ::
                 State -> a ->
                 (Object -> State -> a -> a) ->
                 (Object -> State -> a -> a) ->
                 (State -> a -> a) -> a
             )|] )

cargo_cata :: Cargo -> a -> (Object -> a -> a) -> a
cargo_cata NOCARGO     x f = x
cargo_cata (IN o c)    x f = f o (cargo_cata c x f)

cargo_para :: Cargo -> a -> (Object -> Cargo -> a -> a) -> a
cargo_para NOCARGO     x f = x
cargo_para (IN o c)    x f = f o c (cargo_para c x f)

state_para :: State -> a -> (Object -> State -> a -> a)
           -> (Object -> State -> a -> a)
           -> (State -> a -> a) -> a
state_para START      x l u f = x
state_para (LOD o s)  x l u f =
```

```
  l o s (state_para s x l u f)
state_para (UNL o s)  x l u f =
  u o s (state_para s x l u f)
state_para (FLY s)    x l u f =
  f s (state_para s x l u f)

state_cata :: State -> a -> (Object -> a -> a)
           -> (Object -> a -> a) -> (a -> a) -> a
state_cata START      x l u f = x
state_cata (LOD o s)  x l u f =
  l o (state_cata s x l u f)
state_cata (UNL o s)  x l u f =
  u o (state_cata s x l u f)
state_cata (FLY s)    x l u f =
  f (state_cata s x l u f)

----------------------------------------------
-- Hanoi types
----------------------------------------------

hanC = $(p [|( PegA :: Peg, PegB :: Peg, PegC :: Peg
            , D0 :: Disc
             , D :: Disc -> Disc
            , NOOP :: Action
            , MV :: Disc -> Peg -> Peg -> Action -> Action
            , disc_cata :: Disc -> (a -> a) -> a -> a
            , action_cata ::
                Action ->
                    a -> (Disc -> Peg -> Peg -> a -> a) -> a
            )|])

hanP = $(p [|( PegA :: Peg, PegB :: Peg, PegC :: Peg
            , D0 :: Disc, D :: Disc -> Disc
            , NOOP :: Action
             , MV :: Disc -> Peg -> Peg -> Action -> Action
            , disc_para ::
                Disc -> (Disc -> a -> a) -> a -> a
            , action_para ::
                Action -> a ->
                (Disc -> Peg -> Peg -> Action ->
                 a -> a) -> a
            )|])


disc_cata   :: Disc -> (a -> a) -> a -> a
disc_cata D0 _ v      = v
disc_cata (D x) f v   = f (disc_cata x f v)

disc_para   :: Disc -> (Disc -> a -> a) -> a -> a
disc_para D0 _ v      = v
disc_para (D x) f v   = f (D x) (disc_para x f v)
```

```
action_cata ::
  Action -> a -> (Disc -> Peg -> Peg -> a -> a) -> a
action_cata NOOP           x f  = x
action_cata (MV d p1 p2 a) x f  =
    f d p1 p2 (action_cata a x f)

action_para ::
  Action -> a -> (Disc -> Peg -> Peg -> Action -> a -> a) -> a
action_para NOOP            x f = x
action_para (MV d p1 p2 a) x f  =
    f d p1 p2 a (action_para a x f)

----------------------------------------------
-- Balloons types
----------------------------------------------

bals = $(p [|( Purple :: Color, Yellow :: Color
             , Large :: Size, Small :: Size
             , Dip :: Act, Stretch :: Act
             , Adult :: Age, Child :: Age
             , FF :: Inflate, FF :: Inflate
             )|] )

----------------------------------------------
-- Sports types
----------------------------------------------

sports = $(p [|( Sunny :: Weather, Rain :: Weather
               , Overcast :: Weather, Hot :: Weather
               , Cold :: Weather, Mild :: Weather
               , Warm :: Weather, Cold :: Weather
               , High :: Weather, Normal :: Weather
               , Weak :: Weather, Strong :: Weather
               , Change :: Weather, Same :: Weather
               )|])

----------------------------------------------
-- Lenses types
----------------------------------------------

lens = $(p [|( Young :: LAge, PrePresbyopic :: LAge
             , Presbyopic :: LAge
             , Myope :: LPrescription
             , Hypermetrope :: LPrescription
             , No :: LAstigmatic, Yes :: LAstigmatic
             , Reduced :: LTears, Norml :: LTears
             , None :: LCLType, Hard :: LCLType
             , Soft :: LCLType
             )|] )
```

```
-- ======================================================
-- Helper functions to run and time MagicHaskeller
-- ======================================================

-- Haskeller does not allow variables, so we bind them
a = 'a'; b = 'b'; c = 'c'; d = 'd'; e = 'e'; f = 'f';
g = 'g'; h = 'h'; i = 'i'; j = 'j'; k = 'k'; l = 'l';
m = 'm'; n = 'n'; o = 'o'; s = 's'; t = 't'; u = 'u';
v = 'v'; w = 'w'; x = 'x'; y = 'y'; z = 'z'; xs = "xs";
ys = "ys"; zs = "zs"; a11 = "a11"; a12 = "a12";
a13 = "a13"; a21 = "a21"; a22 = "a22"; a23 = "a23";
a31 = "a31"; a32 = "a32"; a33 = "a33"

-- timing helper
time :: IO t -> IO t
time a = do
  start <- getCPUTime
  v <- a
  end   <- getCPUTime
  let diff = (fromIntegral (end - start)) / (10^12)
  printf "Time  : %0.3f sec\n" (diff :: Double)
  return v

runTest  :: (Typeable a) => (a -> Bool) -> IO ()
runTest = time . printOne

-- print name and background knowledge
putNam s b =
  putStr $ "Name  : " ++ s ++ "\nBack  : " ++ b
-- test would run out of memory, just print dummy
-- result and time
putOoM = putStr "Result: OoM\nTime  : NaN\n"
-- test would run out of memory in batch mode, do
-- it again manually
putMan =
  putStr "Result: OoM, run alone\nTime  : NaN\n"

-- set Haskellers program generator and momoization
initialize lb  =  setPG .  mkMemo075 $ lb

-- shortcuts for libraries
-- Use with suffix 'P' for para-, with suffix 'C'
-- for catamorphisms
llib = listP  -- listC
nlib = natP   -- natC
tlib = ntreeP -- ntreeC
rock = rockP  --rockC
han  = hanP   -- hanC
blib = bool
mlib = mb
plib = pair
ulib = (bals ++ sports ++ lens)
```

```haskell
alib = (rock ++ han ++
        $(p [|( 'D' :: Char, 'N' :: Char
              ,'V' :: Char)|]) )
lib = (llib ++ nlib ++ tlib)

-- ===================================================
-- Calling the tests
-- ===================================================

main = do
  initialize lib

  functionsonnaturalnumbers
  predicatesfunctionsonbooleans
  functionsonlists
  functionsonlistsoflists
  functionsonnaturalsandlists
  functionsontrees
  functionsonmixedinputs
  functionsonotherdatatypes
  functionsforUCIclassificationproblems

functionsonnaturalnumbers = do
   putStrLn "------functions on natural numbers------"
   initialize nlib          >> putNam "ack" "<none>"
     >> putMan -- >> runTest testACK
   initialize nlib          >> putNam "add" "<none>"
     >> runTest testADD
   initialize nlib          >> putNam "eveN" "<none>"
     >> runTest testEVEN
   initialize nlib          >> putNam "eq" "<none>"
     >> putOoM --  >> runTest testEQ
   initialize(nlib ++ nadd)
     >> putNam "gaussSum" "add"
     >> runTest testGAUSSSUM
   initialize (nlib ++ nmlt) >> putNam "fact" "mult"
     >> runTest testFACT
   initialize (nlib ++ nadd) >> putNam "fib" "add"
     >> putOoM -- >>  runTest testFIB
   initialize nlib           >> putNam "geq" "<none>"
     >> putOoM --  >> runTest testGEQ
   initialize nlib           >> putNam "mod" "<none>"
     >> putOoM --" >> runTest testMOD
   initialize nlib
     >> putNam "mult" "<none>"
     >> runTest testMULT
   initialize (nlib ++ nadd) >> putNam "mult" "add"
     >> runTest testMULT
   initialize nlib           >> putNam "odD" "<none>"
     >> runTest testODD
   initialize nlib           >> putNam "sub" "<none>"
     >> putOoM -- >> runTest testSUB
```

```
predicatesfunctionsonbooleans = do
   putStrLn "---predicates, functions on booleans---"
   initialize (blib ++ llib)
     >> putNam "andL" "<none>"
     >> runTest testANDL
   initialize (blib ++ llib) >> putNam "anD" "<none>"
     >> runTest testAND
   initialize (blib ++ llib)
     >> putNam "evenParity" "<none>"
     >> runTest testEVENPARITY
   initialize (blib ++ llib)
     >> putNam "negateAll" "<none>"
     >> runTest testNEGATEALL
   initialize (blib ++ llib)
     >> putNam "nandL" "<none>"
     >> runTest testNANDL
   initialize (blib ++ llib)
     >> putNam "norL" "<none>"
     >> runTest testNORL
   initialize (blib ++ llib)
     >> putNam "or" "<none>"
     >> runTest testOR
   initialize (blib ++ llib)
     >> putNam "orL" "<none>"
     >> runTest testORL


functionsonlists = do
   putStrLn "----------functions on lists----------"
   initialize lib  >>  putNam "appenD" "<none>"
                   >> runTest testAPPEND
   initialize llib >> putNam "concaT" "<none>"
                   >> runTest testCONCAT
   initialize llib
                   >> putNam "evenLength" "<none>"
                   >> runTest testEVENLENGTH
   initialize llib >> putNam "evenpos" "<none>"
                   >> runTest testEVENPOS
   initialize llib >> putNam "halves" "<none>"
                   >> putOoM --  >> runTest testHALVES
   initialize llib >> putNam "iniT" "<none>"
                   >> runTest testINIT
   initialize llib >> putNam "initS" "<none>"
                   >> putOoM --  >> runTest testINITS
   initialize llib
                   >> putNam "interspersE" "<none>"
                   >> runTest testINTERSPERSE
   initialize llib >> putNam "lasT" "<none>"
                   >> putOoM --  >> runTest testLAST
   initialize llib >> putNam "lastM" "<none>"
                   >> runTest testLASTM
```

```
   initialize llib >> putNam "lasts" "<none>"
                   >> runTest testLASTS
   initialize llib >> putNam "mapCons" "<none>"
                   >> runTest testMAPCONS
   initialize llib >> putNam "multfst" "<none>"
                   >> runTest testMULTFST
   initialize llib >> putNam "multlst" "<none>"
                   >> runTest testMULTLST
   initialize llib >> putNam "oddpos" "<none>"
                   >> runTest testODDPOS
   initialize llib >> putNam "pack" "<none>"
                   >> runTest testPACK
   initialize (llib ++ lapp)
     >> putNam "subseqs" "append"
     >> putOoM -- >> runTest testSUBSEQS
   initialize llib >> putNam "reversE" "<none>"
                   >> runTest testREVERSE
   initialize llib >> putNam "shiftl" "<none>"
                   >> runTest testSHIFTL
   initialize llib >> putNam "shiftr" "<none>"
                   >> putMan -- >> runTest testSHIFTR
   initialize llib >> putNam "snoc" "<none>"
                   >> runTest testSNOC
   initialize llib >> putNam "swap" "<none>"
                   >> putOoM --  >> runTest testSWAP
   initialize llib >> putNam "switch" "<none>"
                   >> putOoM --  >> runTest testSWITCH
   initialize (llib ++ plib)
     >> putNam "split" "<none>"
     >> putOoM -- >> runTest testSPLIT
   initialize llib >> putNam "taiL" "<none>"
                   >> runTest testTAIL
   initialize llib >> putNam "tailS" "<none>"
                   >> runTest testTAILS
   initialize (llib ++ plib)
     >> putNam "unzip" "<none>"
     >> putOoM --  >> runTest testUNZIP
   initialize llib
     >> putNam "weave" "<none>"
     >> putOoM --  >> runTest testWEAVE
   initialize (llib ++ plib)
     >> putNam "ziP" "<none>"
     >> putOoM -- >> runTest testZIP


functionsonlistsoflists = do
   putStrLn "------functions on lists of lists------"
   initialize llib >> putNam "mapTail" "<none>"
     >> runTest testMAPTAIL
   initialize llib >> putNam "transpose" "<none>"
     >> putOoM -- >> runTest testTRANSPOSE
   initialize llib >> putNam "weaveL" "<none>"
```

```
    >> putOoM -- >> runTest testWEAVEL


functionsonnaturalsandlists = do
  putStrLn "---functions on naturals and lists---"
  initialize (nlib ++ llib)
    >> putNam "addN" "<none>"
    >> runTest testADDN
  initialize (nlib ++ llib)
    >> putNam "alleven" "<none>"
    >> runTest testALLEVEN
  initialize (nlib ++ llib)
    >> putNam "allodd" "<none>"
    >> runTest testALLODD
  initialize (nlib ++ llib)
    >> putNam "evens" "<none>"
    >> putOoM --  >> runTest testEVENS
  initialize (nlib ++ llib)
    >> putNam "incr" "<none>"
    >> runTest testINCR
  initialize (nlib ++ llib)
    >> putNam "lengtH" "<none>"
    >> runTest testLENGTH
  initialize (nlib ++ llib)
    >> putNam "lengths" "<none>"
    >> runTest testLENGTHS
  initialize (nlib ++ llib)
    >> putNam "nthElem" "<none>"
    >> putOoM -- >> runTest testNTHELEM
  initialize (nlib ++ llib)
    >> putNam "oddslist" "<none>"
    >> runTest testODDSLIST
  initialize (nlib ++ llib)
    >> putNam "odds" "<none>"
    >> putOoM --  >> runTest testODDS
  initialize (nlib ++ llib)
    >> putNam "droP" "<none>"
    >> runTest testDROP
  initialize (plib ++ nlib ++ llib)
    >> putNam "splitAt" "<none>"
    >> putOoM -- >> runTest testSPLITAT
  initialize (nlib ++ llib)
    >> putNam "suM" "<none>"
    >> runTest testSUM
  initialize (nlib ++ llib)
    >> putNam "replicate" "<none>"
    >> runTest testREPLICATE
  initialize (nlib ++ llib)
    >> putNam "takE" "<none>"
    >> runTest testTAKE
  initialize (nlib ++ llib)
    >> putNam "zeros" "<none>"
```

```
    >> runTest testZEROS


functionsontrees = do
   putStrLn "----------functions on trees----------"
   initialize (tlib ++ llib ++ lapp)
     >> putNam "preorder" "append"
     >> runTest testPREORDER
   initialize (tlib ++ llib ++ lapp)
     >> putNam "inorder" "append"
     >> runTest testINORDER
   initialize (tlib ++ llib ++ lapp ++ lsnc)
     >> putNam "postorder" "append, snoc"
     >> runTest testPOSTORDER
   initialize tlib
     >> putNam "mirror" "<none>"
     >> runTest testMIRROR


functionsonmixedinputs = do
   putStrLn "------functions on mixed inputs------"
   initialize (mlib ++ plib ++ llib ++ nlib)
     >> putNam "pepper" "<none>"
     >> putOoM -- >> runTest testPEPPER
   initialize (mlib ++ plib ++ llib ++ nlib)
     >> putNam "pepperF" "<none>"
     >> putOoM -- >> runTest testPEPPERF


functionsonotherdatatypes = do
   putStrLn "---functions on other data types---"
   initialize alib >> putNam "rocket" "<none>"
     >> runTest testROCKET
   initialize alib >> putNam "hanoi" "<none>"
     >> putOoM -- >> runTest testHANOI
   initialize (nlib ++ alib ++ llib)
     >> putNam "sentence" "<none>"
     >> putOoM -- >> runTest testSENTENCE


functionsforUCIclassificationproblems = do
   putStrLn
     "---functions forUCI classification problems---"
   initialize (blib ++ plib ++ ulib)
     >> putNam "balloons" "<none>"
     >> putOoM --  >> runTest testBALLOONS
   initialize (blib ++ plib ++ ulib)
     >> putNam "playTennis" "<none>"
     >> putOoM --  >> runTest testPLAYTENNIS
   initialize (blib ++ plib ++ ulib)
     >> putNam "enjoySport" "<none>"
     >> putOoM --  >> runTest testENJOYSPORT
```

```
  initialize (plib ++ ulib)
    >> putNam "lenses" "<none>"
    >> putOoM --  >> runTest testLENSES


-- ====================================================
-- Test definitions
-- ====================================================

testACK fun = (fun (Z) (Z) == (S Z)) && (fun (Z) (S Z) == (S(S Z))) &&
     (fun (Z) (S(S Z)) == (S(S(S Z)))) && (fun (Z) (S(S(S Z))) == (S(S(
    S(S Z))))) && (fun (Z) (S(S(S(S Z)))) == (S(S(S(S(S Z)))))) && (fun
     (Z) (S(S(S(S(S Z))))) == (S(S(S(S(S(S Z))))))) && (fun (Z) (S(S(S(
    S(S(S Z)))))) == (S(S(S(S(S(S(S Z)))))))) && (fun (S Z) (Z) == (S(S
     Z))) && (fun (S Z) (S Z) == (S(S(S Z)))) && (fun (S Z) (S(S Z)) ==
     (S(S(S(S Z))))) && (fun (S Z) (S(S(S Z))) == (S(S(S(S(S Z)))))) &&
     (fun (S Z) (S(S(S(S Z)))) == (S(S(S(S(S(S Z))))))) && (fun (S Z) (
    S(S(S(S(S Z))))) == (S(S(S(S(S(S(S Z)))))))) && (fun (S (S Z)) (Z)
    == (S(S(S Z)))) && (fun (S (S Z)) (S Z) == (S(S(S(S(S Z)))))) && (
    fun (S (S Z)) (S(S Z)) == (S(S(S(S(S(S(S Z)))))))) && (fun (S(S(S Z
    ))) (Z) == (S(S(S(S(S Z))))))
testADD fun = (fun Z Z == Z) && (fun Z (S Z) == (S Z)) && (fun Z (S(S
    Z)) == (S(S Z))) && (fun Z (S(S(S Z))) == (S(S(S Z)))) && (fun (S Z
    ) Z == (S Z)) && (fun (S Z) (S Z) == (S(S Z))) && (fun (S Z) (S(S Z
    )) == (S(S(S Z)))) && (fun (S Z) (S(S(S Z))) == (S(S(S(S Z))))) &&
    (fun (S(S Z)) (S Z) == (S(S(S Z)))) && (fun (S(S Z)) Z == (S(S Z)))
     && (fun (S(S Z)) (S(S Z)) == (S(S(S(S Z))))) && (fun (S(S Z)) (S(S
    (S Z))) == (S(S(S(S(S Z))))))
testEVEN fun = (fun Z == True) && (fun (S Z) == False) && (fun (S (S Z
    )) == True) && (fun (S (S (S Z))) == False) && (fun (S (S (S (S Z))
    )) == True) && (fun (S (S (S (S (S Z))))) == False)
testEQ fun = (fun Z Z == True) && (fun Z (S Z) == False) && (fun Z (S
    (S Z)) == False) && (fun (S Z) Z == False) && (fun (S Z) (S Z) ==
    True) && (fun (S Z) (S (S Z)) == False) && (fun (S (S Z)) Z ==
    False) && (fun (S (S Z)) (S Z) == False) && (fun (S (S Z)) (S (S Z)
    ) == True)
testGAUSSSUM fun = (fun Z == Z) && (fun (S Z) == (S Z)) && (fun (S(S Z
    )) == (S(S(S Z)))) && (fun (S(S(S Z))) == (S(S(S(S(S(S Z))))))) &&
    (fun (S(S(S(S Z)))) == (S(S(S(S(S(S(S(S(S(S Z)))))))))))
testFACT fun = (fun Z == S(Z)) && (fun (S(Z)) == S(Z)) && (fun (S(S(Z)
    )) == S(S(Z))) && (fun (S(S(S(Z)))) == S(S(S(S(S(S(Z))))))) && (fun
     (S(S(S(S(Z))))) == S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S
    (Z))))))))))))))))))))))))) && (fun (S(S(S(S(S(Z)))))) == S(S(S(S(S
    (S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(
    S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S
    (S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(
    S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(Z))))))))))))))))))))))))))))))))))))
    )))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
    ))))))))))))))))))
testFIB fun = (fun Z == Z) && (fun (S(Z)) == S(Z)) && (fun (S(S(Z)))
    == S(Z)) && (fun (S(S(S(Z)))) == S(S(Z))) && (fun (S(S(S(S(Z)))))
    == S(S(S(Z)))) && (fun (S(S(S(S(S(Z)))))) == S(S(S(S(S(Z))))))
```

```
testGEQ fun = (fun Z Z == True) && (fun (S Z) Z == True) && (fun (S(S
   Z)) Z == True) && (fun Z (S Z) == False) && (fun (S Z) (S Z) ==
   True) && (fun (S(S Z)) (S Z) == True) && (fun Z (S(S Z)) == False)
   && (fun (S Z) (S(S Z)) == False) && (fun (S(S Z)) (S(S Z)) == True)
testMOD fun = (fun Z (S Z) == Z) && (fun (S Z) (S Z) == Z) && (fun (S(
   S Z)) (S Z) == Z) && (fun (S(S(S Z))) (S Z) == Z) && (fun Z (S(S Z)
   ) == Z) && (fun (S Z) (S(S Z)) == (S Z)) && (fun (S(S Z)) (S(S Z))
   == Z) && (fun (S(S(S Z))) (S(S Z)) == (S Z)) && (fun Z (S(S(S Z)))
   == Z) && (fun (S Z) (S(S(S Z))) == (S Z)) && (fun (S(S Z)) (S(S(S Z
   ))) == (S(S Z))) && (fun (S(S(S Z))) (S(S(S Z))) == Z) && (fun Z (S
   (S(S(S Z)))) == Z) && (fun (S Z) (S(S(S(S Z)))) == (S Z)) && (fun (
   S(S Z)) (S(S(S(S Z)))) == (S(S Z))) && (fun (S(S(S Z))) (S(S(S(S Z)
   ))) == (S(S(S Z))))
testMULT fun = (fun Z Z == Z) && (fun Z (S Z) == Z) && (fun Z (S(S Z))
    == Z) && (fun Z (S(S(S Z))) == Z) && (fun (S Z) Z == Z) && (fun (S
    Z) (S Z) == (S Z)) && (fun (S Z) (S(S Z)) == (S(S Z))) && (fun (S
   Z) (S(S(S Z))) == (S(S(S Z)))) && (fun (S(S Z)) Z == Z) && (fun (S(
   S Z)) (S Z) == (S(S Z))) && (fun (S(S Z)) (S(S Z)) == (S(S(S(S Z)))
   )) && (fun (S(S Z)) (S(S(S Z))) == (S(S(S(S(S(S Z))))))) && (fun (S
   (S(S Z))) Z == Z) && (fun (S(S(S Z))) (S Z) == (S(S(S Z)))) && (fun
    (S(S(S Z))) (S(S Z)) == (S(S(S(S(S(S Z))))))) && (fun (S(S(S Z)))
   (S(S(S Z))) == (S(S(S(S(S(S(S(S(S Z))))))))))
testODD fun = (fun Z == False) && (fun (S Z) == True) && (fun (S (S Z)
   ) == False) && (fun (S (S (S Z))) == True) && (fun (S (S (S (S Z)))
   ) == False) && (fun (S (S (S (S (S Z))))) == True)
testANDL fun = (fun [] == True) && (fun [True] == True) && (fun [False
   ] == False) && (fun [True,True] == True) && (fun [True,False] ==
   False) && (fun [False,True] == False) && (fun [False,False] ==
   False) && (fun [True,True,True] == True) && (fun [False,True,True]
   == False) && (fun [True,False,True] == False) && (fun [True,True,
   False] == False) && (fun [True,False,False] == False) && (fun [
   False,True,False] == False) && (fun [False,False,True] == False) &&
    (fun [False,False,False] == False)
testSUB fun = (fun Z Z == Z) && (fun Z (S Z) == (S Z)) && (fun Z (S(S
   Z)) == (S(S Z))) && (fun Z (S(S(S Z))) == (S(S(S Z)))) && (fun (S Z
   ) Z == Z) && (fun (S Z) (S Z) == Z) && (fun (S Z) (S(S Z)) == (S Z)
   ) && (fun (S Z) (S(S(S Z))) == (S(S Z))) && (fun (S(S Z)) Z == Z)
   && (fun (S(S Z)) (S Z) == Z) && (fun (S(S Z)) (S(S Z)) == Z) && (
   fun (S(S Z)) (S(S(S Z))) == (S Z)) && (fun (S(S(S Z))) Z == Z) && (
   fun (S(S(S Z))) (S Z) == Z) && (fun (S(S(S Z))) (S(S Z)) == Z) && (
   fun (S(S(S Z))) (S(S(S Z))) == Z)
testAND fun = (fun True True == True) && (fun True False == False) &&
   (fun False True == False) && (fun False False == False)
testEVENPARITY fun = (fun [] == True) && (fun [False] == True) && (fun
    [True] == False) && (fun [False, False] == True) && (fun [False,
   True] == False) && (fun [True, False] == False) && (fun [True, True
   ] == True) && (fun [False, False, False] == True) && (fun [False,
   False, True] == False) && (fun [False, True, False] == False) && (
   fun [False, True, True] == True) && (fun [True, False, False] ==
   False) && (fun [True, False, True] == True) && (fun [True, True,
   False] == True)
```

```
testNEGATEALL fun = (fun [] == []) && (fun [True] == [False]) && (fun
    [False] == [True]) && (fun [False,False] == [True,True]) && (fun [
    False,True] == [True,False]) && (fun [True,False] == [False,True])
    && (fun [True,True] == [False,False])
testNANDL fun = (fun [] == False) && (fun [True] == False) && (fun [
    False] == True) && (fun [True,True] == False) && (fun [True,False]
    == True) && (fun [False,True] == True) && (fun [False,False] ==
    True) && (fun [True,True,True] == False) && (fun [False,True,True]
    == True) && (fun [True,False,True] == True) && (fun [True,True,
    False] == True) && (fun [True,False,False] == True) && (fun [False,
    True,False] == True) && (fun [False,False,True] == True) && (fun [
    False,False,False] == True)
testNORL fun = (fun [] == True) && (fun [True] == False) && (fun [
    False] == True) && (fun [True,True] == False) && (fun [True,False]
    == False) && (fun [False,True] == False) && (fun [False,False] ==
    True) && (fun [True,True,True] == False) && (fun [False,True,True]
    == False) && (fun [True,False,True] == False) && (fun [True,True,
    False] == False) && (fun [True,False,False] == False) && (fun [
    False,True,False] == False) && (fun [False,False,True] == False) &&
     (fun [False,False,False] == True)
testOR fun = (fun True True == True) && (fun True False == True) && (
    fun False True == True) && (fun False False == False)
testORL fun = (fun [] == False) && (fun [True] == True) && (fun [False
    ] == False) && (fun [True,True] == True) && (fun [True,False] ==
    True) && (fun [False,True] == True) && (fun [False,False] == False)
     && (fun [True,True,True] == True) && (fun [False,True,True] ==
    True) && (fun [True,False,True] == True) && (fun [True,True,False]
    == True) && (fun [True,False,False] == True) && (fun [False,True,
    False] == True) && (fun [False,False,True] == True) && (fun [False,
    False,False] == False)
testAPPEND fun = (fun [][] == []) && (fun [][c] == [c]) && (fun [][c,d
    ] == [c,d]) && (fun [] [a,b,c] == [a,b,c]) && (fun [][a,b,c,d] == [
    a,b,c,d]) && (fun [a][] == [a]) && (fun [a][c] == [a,c]) && (fun [a
    ,b][] == [a,b]) && (fun [a][c,d] == [a,c,d]) && (fun [a,b][d] == [a
    ,b,d]) && (fun [a,c,d][] == [a,c,d]) && (fun [a,b][c,d] == [a,b,c,d
    ]) && (fun [a,b,c][d] == [a,b,c,d]) && (fun [a,b,c,d][] == [a,b,c,d
    ])
testCONCAT fun = (fun [] == []) && (fun [[]] == []) && (fun [[],[]] ==
     []) && (fun [[],[a]] == [a]) && (fun [[],[a,b]] == [a,b]) && (fun
    [[a]] == [a]) && (fun [[a],[]] == [a]) && (fun [[a],[b]] == [a,b])
    && (fun [[a],[c,d]] == [a,c,d]) && (fun [[c,d]]== [c,d]) && (fun [[
    a,b],[]] == [a,b]) && (fun [[a,b],[c]] == [a,b,c]) && (fun [[a,b],[
    c,d]] == [a,b,c,d])
testEVENLENGTH fun = (fun [] == True) && (fun [a] == False) && (fun [a
    ,b] == True) && (fun [a,b,c] == False) && (fun [a,b,c,d] == True)
    && (fun [a,b,c,d,e] == False) && (fun [a,b,c,d,e,f] == True)
testEVENPOS fun = (fun [] == []) && (fun [a] == []) && (fun [a,b] == [
    b]) && (fun [a,b,c] == [b]) && (fun [a,b,c,d] == [b,d]) && (fun [a,
    b,c,d,e] == [b,d]) && (fun [a,b,c,d,e,f] == [b,d,f])
testHALVES fun = (fun [] == ([], [])) && (fun [a] == ([a], [])) && (
    fun [a,b] == ([a],[b])) && (fun [a,b,c] == ([a,b],[c])) && (fun [a,
    b,c,d] == ([a,b],[c,d])) && (fun [a,b,c,d,e] == ([a,b,c],[d,e]))
```

```
testINIT fun = (fun [a] == []) && (fun [a,b] == [a]) && (fun [a,b,c]
   == [a,b]) && (fun [a,b,c,d] == [a,b,c])
testINITS fun = (fun [] == [[]]) && (fun [a] == [[],[a]]) && (fun [a,b
   ] == [[],[a],[a,b]]) && (fun [a,b,c] == [[],[a],[a,b],[a,b,c]]) &&
   (fun [a,b,c,d] == [[],[a],[a,b],[a,b,c,d]])
testINTERSPERSE fun = (fun x [] == []) && (fun x [y] == [y]) && (fun x
    [y,z] == [y,x,z]) && (fun x [y,z,v] == [y,x,z,x,v])
testLAST fun = (fun [a] == a) && (fun [a,b] == b) && (fun [a,b,c] == c
   ) && (fun [a,b,c,d] == d)
testLASTM fun = (fun [] == Nothing) && (fun [a] == Just a) && (fun [a,
   b] == Just b) && (fun [a,b,c] == Just c) && (fun [a,b,c,d] == Just
   d)
testLASTS fun = (fun [] == []) && (fun [[a]] == [a]) && (fun [[a,b]]
   == [b]) && (fun [[a,b,c]] == [c]) && (fun [[b],[a]] == [b,a]) && (
   fun [[c],[a,b]] == [c,b]) && (fun [[a,b],[c,d]] == [b,d]) && (fun
   [[c,d],[b]] == [d,b]) && (fun [[c],[d,e],[f]] == [c,e,f]) && (fun
   [[c,d],[e,f],[g]] == [d,f,g])
testMAPCONS fun = (fun a [] == []) && (fun a [[]] == [[a]]) && (fun a
   [xs] == [(a:xs)]) && (fun a [xs,ys] == [(a:xs),(a:ys)]) && (fun a [
   xs,ys,zs] == [(a:xs),(a:ys),(a:zs)])
testMULTFST fun = (fun [] == []) && (fun [a] == [a]) && (fun [a,b] ==
   [a,a]) && (fun [a,b,c] == [a,a,a]) && (fun [a,b,c,d] == [a,a,a,a])
testMULTLST fun = (fun [] == []) && (fun [a] == [a]) && (fun [a,b] ==
   [b,b]) && (fun [a,b,c] == [c,c,c]) && (fun [a,b,c,d] == [d,d,d,d])
testODDPOS fun = (fun [] == []) && (fun [a] == [a]) && (fun [a,b] == [
   a]) && (fun [a,b,c] == [a,c]) && (fun [a,b,c,d] == [a,c]) && (fun [
   a,b,c,d,e] == [a,c,e])
testPACK fun = (fun [] == [[]]) && (fun [a] == [[a]]) && (fun [a,b] ==
    [[a],[b]]) && (fun [a,b,c] == [[a],[b],[c]])
testSUBSEQS fun = (fun [] == [[]]) && (fun [a] == [[a],[]]) && (fun [a
   ,b] == [[a,b],[a],[b],[]]) && (fun [a,b,c] == [[a,b,c],[a,b],[a,c
   ],[a],[b,c],[b],[c],[]])
testREVERSE fun = (fun [] == []) && (fun [a] ==[a]) && (fun [a,b] == [
   b,a]) && (fun [a,b,c] == [c,b,a]) && (fun [a,b,c,d] == [d,c,b,a])
testSHIFTL fun = (fun [] == []) && (fun [a] == [a]) && (fun [a,b] == [
   b,a]) && (fun [a,b,c] == [b,c,a]) && (fun [a,b,c,d] == [b,c,d,a])
testSHIFTR fun = (fun [] == []) && (fun [a] == [a]) && (fun [a,b] == [
   b,a]) && (fun [a,b,c] == [c,a,b]) && (fun [a,b,c,d] == [d,a,b,c])
testSNOC fun = (fun a [] == [a]) && (fun b [a] == [a,b]) && (fun c [a,
   b] == [a,b,c]) && (fun d [a,b,c] == [a,b,c,d])
testSWAP fun = (fun [] == []) && (fun [a] == [a]) && (fun [a,b] == [b,
   a]) && (fun [a,b,c] == [b,a,c]) && (fun [a,b,c,d] == [b,a,d,c]) &&
   (fun [a,b,c,d,e] == [b,a,d,c,e]) && (fun [a,b,c,d,e,f] == [b,a,d,c,
   f,e])
testSWITCH fun = (fun [] == []) && (fun [a] == [a]) && (fun [a,b] == [
   b,a]) && (fun [a,b,c] == [c,b,a]) && (fun [a,b,c,d] == [d,b,c,a])
   && (fun [a,b,c,d,e] == [e,b,c,d,a]) && (fun [a,b,c,d,e,f] == [f,b,c
   ,d,e,a])
testSPLIT fun = (fun [] == ([],[])) && (fun [x] == ([x],[])) && (fun [
   x,y] == ([x],[y])) && (fun [x,y,z] == ([x,z],[y])) && (fun [x,y,z,v
   ] == ([x,z],[y,v]))
```

```
testTAIL fun = (fun [a] == []) && (fun [a,b] == [b]) && (fun [a,b,c]
    == [b,c]) && (fun [a,b,c,d] == [b,c,d])
testTAILS fun = (fun [] == [[]]) && (fun [a] == [[a],[]]) && (fun [a,b
    ] == [[a,b],[b],[]]) && (fun [a,b,c] == [[a,b,c],[b,c],[c],[]])
testUNZIP fun = (fun [(a,b)] == ([a],[b])) && (fun [(a,b),(c,d)] == ([
    a,c],[b,d])) && (fun [(a,b),(c,d),(e,f)] == ([a,c,e],[b,d,f]))
testWEAVE fun = (fun [] [] == []) && (fun [a][] == [a]) && (fun [][c]
    == [c]) && (fun [a][c] == [a,c]) && (fun [a,b][] == [a,b]) && (fun
    [][c,d] == [c,d]) && (fun [a,b][c] == [a,c,b]) && (fun [a][c,d] ==
    [a,c,d]) && (fun [a,b][c,d] == [a,c,b,d])
testZIP fun = (fun [] [] == []) && (fun [a] [] == []) && (fun [] [a]
    == []) && (fun [a] [b] == [(a,b)]) && (fun [a,b] [c] == [(a,c)]) &&
     (fun [a] [b,c] == [(a,b)]) && (fun [a,b] [c,d] == [(a,c),(b,d)])
testMAPTAIL fun = (fun [] == []) && (fun [(x:xs)] == [xs]) && (fun [(x
    :xs),(y:ys)] == [xs,ys]) && (fun [(x:xs),(y:ys),(z:zs)] == [xs,ys,
    zs])
testTRANSPOSE fun = (fun [[a11]] == [[a11]]) && (fun [[a11,a12]] == [[
    a11],[a12]]) && (fun [[a11,a12,a13]] == [[a11],[a12],[a13]]) && (
    fun [[a11],[a21]] == [[a11,a21]]) && (fun [[a11,a12],[a21,a22]] ==
    [[a11,a21],[a12,a22]]) && (fun [[a11,a12,a13],[a21,a22,a23]] == [[
    a11,a21],[a12,a22],[a13,a23]]) && (fun [[a11],[a21],[a31]] ==[[a11,
    a21,a31]]) && (fun [[a11,a12],[a21,a22],[a31,a32]] == [[a11,a21,a31
    ],[a12,a22,a32]]) && (fun [[a11,a12,a13],[a21,a22,a23],[a31,a32,a33
    ]] == [[a11,a21,a31],[a12,a22,a32],[a13,a23,a33]])
testWEAVEL fun = (fun [] == []) && (fun [[a11]] == [a11]) && (fun [[
    a11,a12]] == [a11,a12]) && (fun [[a11,a12,a13]] == [a11,a12,a13])
    && (fun [[a11],[a21]] == [a11,a21]) && (fun [[a11 ,a12],[a21]] == [
    a11 ,a21 ,a12]) && (fun [[a11],[a21,a22]] == [a11 ,a21 ,a22]) && (
    fun [[a11,a12],[a21 ,a22]] == [a11 ,a21 ,a12 ,a22]) && (fun [[a11 ,
    a12 ,a13],[a21 ,a22]] == [a11 ,a21 ,a12 ,a22 ,a13]) && (fun [[a11
    ],[a21],[a31]] == [a11 ,a21 ,a31]) && (fun [[a11,a12],[a21],[a31]]
    == [a11 ,a21 ,a31 ,a12])
testADDN fun = (fun Z [] == []) && (fun (S Z) [] == []) && (fun (S(S Z
    )) [] == []) && (fun Z [Z] == [Z]) && (fun Z [S Z] == [S Z]) && (
    fun Z [S(S Z)] == [S(S Z)]) && (fun Z [Z,(S Z)] == [Z,(S Z)]) && (
    fun Z [(S Z),Z] == [(S Z),Z]) && (fun (S Z) [Z,(S Z)] == [S Z,S(S Z
    )]) && (fun (S Z) [(S Z),Z] == [S(S Z),S Z]) && (fun (S Z) [Z] == [
    S Z]) && (fun (S Z) [S Z] == [S(S Z)]) && (fun (S Z) [S(S Z)] == [S
    (S(S Z))]) && (fun (S(S Z)) [Z] == [S(S Z)]) && (fun (S(S Z)) [S Z]
     == [S(S(S Z))]) && (fun (S(S Z)) [S(S Z)] == [S(S(S(S Z)))]) && (
    fun (S(S Z)) [Z,(S Z)] == [S(S Z),S(S(S Z))]) && (fun (S(S Z)) [S(S
     Z),Z] == [S(S(S(S Z))),S(S Z)])
testALLEVEN fun = (fun [] == True) && (fun [Z] == True) && (fun [S Z]
    == False) && (fun [S(S Z)] == True) && (fun [S(S(S Z))] == False)
    && (fun [Z, Z] == True) && (fun [Z, S Z] == False) && (fun [Z, S(S
    Z)] == True) && (fun [Z, S(S(S Z))] == False) && (fun [S Z, Z] ==
    False) && (fun [S Z, S Z] == False) && (fun [S Z, S(S Z)] == False)
     && (fun [S Z, S(S(S Z))] == False) && (fun [S(S Z), Z] == True) &&
     (fun [S(S Z), S Z] == False) && (fun [S(S Z), S(S Z)] == True) &&
    (fun [S(S Z), S(S(S Z))] == False) && (fun [S(S(S Z)), Z] == False)
     && (fun [S(S(S Z)), S Z] == False) && (fun [S(S(S Z)), S(S Z)] ==
    False) && (fun [S(S(S Z)), S(S(S Z))] == False)
```

```
testALLODD fun = (fun [] == True) && (fun [Z] == False) && (fun [S Z]
    == True) && (fun [S(S Z)] == False) && (fun [S(S(S Z))] == True) &&
     (fun [Z, Z] == False) && (fun [Z, S Z] == False) && (fun [Z, S(S Z
    )] == False) && (fun [Z, S(S(S Z))] == False) && (fun [S Z, Z] ==
    False) && (fun [S Z, S Z] == True) && (fun [S Z, S(S Z)] == False)
    && (fun [S Z, S(S(S Z))] == True) && (fun [S(S Z), Z] == False) &&
    (fun [S(S Z), S Z] == False) && (fun [S(S Z), S(S Z)] == False) &&
    (fun [S(S Z), S(S(S Z))] == False) && (fun [S(S(S Z)), Z] == False)
     && (fun [S(S(S Z)), S Z] == True) && (fun [S(S(S Z)), S(S Z)] ==
    False) && (fun [S(S(S Z)), S(S(S Z))] == True)
testEVENS fun = (fun [] == []) && (fun [Z] == [Z]) && (fun [S Z] ==
    []) && (fun [S(S Z)] == [S(S Z)]) && (fun [S(S(S Z))] == []) && (
    fun [Z,Z] == [Z,Z]) && (fun [Z,S Z] == [Z]) && (fun [Z,S (S Z)] ==
    [Z,S (S Z)]) && (fun [Z,S(S(S Z))] == [Z]) && (fun [S Z, Z] == [Z])
     && (fun [S Z, S Z] == []) && (fun [S Z, S (S Z)] == [S (S Z)]) &&
    (fun [S Z, S(S(S Z))] == []) && (fun [S (S Z), Z] == [S (S Z),Z])
    && (fun [S (S Z), S Z] == [S (S Z)]) && (fun [S (S Z), S (S Z)] ==
    [S (S Z),S (S Z)]) && (fun [S (S Z), S(S(S Z))] == [S (S Z)]) && (
    fun [S( S (S Z)), Z] == [Z]) && (fun [S( S (S Z)), S Z] == []) && (
    fun [S( S (S Z)), S (S Z)] == [S (S Z)]) && (fun [S( S (S Z)), S( S
     (S Z))] == [])
testINCR fun = (fun [] == []) && (fun [Z] == [S Z]) && (fun [S Z] == [
    S(S Z)]) && (fun [Z,S Z] == [S Z,S(S Z)]) && (fun [S Z,Z] == [S(S Z
    ),S Z])
testLENGTH fun = (fun [] == Z) && (fun [a] == S Z) && (fun [a,b] == S(
    S Z)) && (fun [a,b,c] == S(S(S Z)))
testLENGTHS fun = (fun [] == []) && (fun [[]] == [Z]) && (fun [[a]] ==
     [S Z]) && (fun [[b,a]] == [S(S Z)]) && (fun [[c,b,a]] == [S(S(S Z)
    )]) && (fun [[],[]] == [Z, Z]) && (fun [[],[a]] == [Z,S Z]) && (fun
     [[],[b,a]] == [Z,S(S Z)]) && (fun [[a],[]] == [S Z, Z]) && (fun [[
    b],[a]] == [S Z,S Z]) && (fun [[c],[b,a]] == [S Z,S(S Z)]) && (fun
    [[c,a],[]] == [S(S Z), Z]) && (fun [[b,a],[c]] == [S(S Z),S Z]) &&
    (fun [[c,d],[b,a]] == [S(S Z),S(S Z)]) && (fun [[a],[b],[c]] == [S
    Z, S Z, S Z])
testNTHELEM fun = (fun (x:xs) Z == x) && (fun (x:y:xs) (S Z) == y) &&
    (fun (x:y:z:xs) (S (S Z)) == z) && (fun (x:y:z:u:xs) (S(S (S Z)))
    == u) && (fun (x:y:z:u:v:xs) (S(S(S (S Z)))) == v)
testODDSLIST fun = (fun [] == True) && (fun [Z] == False) && (fun [S Z
    ] == True) && (fun [S(S Z)] == False) && (fun [S(S(S Z))] == True)
    && (fun [Z, Z] == False) && (fun [Z, S Z] == False) && (fun [Z, S(S
     Z)] == False) && (fun [Z, S(S(S Z))] == False) && (fun [S Z, Z] ==
     False) && (fun [S Z, S Z] == True) && (fun [S Z, S(S Z)] == False)
     && (fun [S Z, S(S(S Z))] == True) && (fun [S(S Z), Z] == False) &&
     (fun [S(S Z), S Z] == False) && (fun [S(S Z), S(S Z)] == False) &&
     (fun [S(S Z), S(S(S Z))] == False) && (fun [S(S(S Z)), Z] == False
    ) && (fun [S(S(S Z)), S Z] == True) && (fun [S(S(S Z)), S(S Z)] ==
    False) && (fun [S(S(S Z)), S(S(S Z))] == True)
testODDS fun = (fun [] == []) && (fun [Z] == []) && (fun [S Z] == [S Z
    ]) && (fun [S(S Z)] == []) && (fun [S(S(S Z))] == [(S(S(S Z)))]) &&
     (fun [Z,Z] == []) && (fun [Z,S Z] == [S Z]) && (fun [Z,S (S Z)] ==
     []) && (fun [Z,S(S(S Z))] == [S(S(S Z))]) && (fun [S Z, Z] == [S Z
    ]) && (fun [S Z, S Z] == [S Z, S Z]) && (fun [S Z, S (S Z)] == [(S
```

```
    Z)]) && (fun [S Z, S(S(S Z))] == [S Z, S(S(S Z))]) && (fun [S (S Z)
    , Z] == []) && (fun [S (S Z), S Z] == [(S Z)]) && (fun [S (S Z), S
    (S Z)] == []) && (fun [S (S Z), S(S(S Z))] == [(S(S (S Z)))]) && (
    fun [S( S (S Z)), Z] == [S( S (S Z))]) && (fun [S( S (S Z)), S Z]
    == [S( S (S Z)), S Z]) && (fun [S( S (S Z)), S (S Z)] == [(S(S(S Z)
    ))]) && (fun [S( S (S Z)), S( S (S Z))] == [S( S (S Z)), S( S (S Z)
    )])
testDROP fun = (fun Z [] == []) && (fun Z [a] == [a]) && (fun (S Z) []
     == []) && (fun (S (S Z)) [] == []) && (fun Z [a,b] == [a,b]) && (
    fun Z [a,b,c] == [a,b,c]) && (fun (S Z) [a] == []) && (fun (S Z) [a
    ,b] == [b]) && (fun (S Z) [a,b,c] == [b,c]) && (fun (S (S Z)) [a]
    == []) && (fun (S (S Z)) [a,b] == []) && (fun (S (S Z)) [a,b,c] ==
    [c]) && (fun (S (S (S Z))) [] == []) && (fun (S (S (S Z))) [a] ==
    []) && (fun (S (S (S Z))) [a,b] == []) && (fun (S (S (S Z))) [a,b,c
    ] == [])
testSPLITAT fun = (fun Z [a] == ([],[a])) && (fun Z [a,b] == ([],[a,b
    ])) && (fun Z [a,b,c] == ([],[a,b,c])) && (fun (S Z) [a] == ([a
    ],[])) && (fun (S Z) [a,b] == ([a],[b])) && (fun (S Z) [a,b,c] ==
    ([a],[b,c])) && (fun (S(S Z)) [a] == ([a],[])) && (fun (S(S Z)) [a,
    b] == ([a,b],[])) && (fun (S(S Z)) [a,b,c] == ([a,b],[c]))
testSUM fun = (fun [] == Z) && (fun [Z] == Z) && (fun [S Z] == S Z) &&
     (fun [S(S Z)] == S(S Z)) && (fun [Z,Z] == Z) && (fun [Z,S Z] == S
    Z) && (fun [Z,S(S Z)] == S(S Z)) && (fun [S Z,Z] == S Z) && (fun [S
     Z,S Z] == S(S Z)) && (fun [S Z,S(S Z)] == S(S(S Z))) && (fun [S(S
    Z),Z] == S(S Z)) && (fun [S(S Z),S Z] == S(S(S Z))) && (fun [S(S Z)
    ,S(S Z)] == S(S(S(S Z))))
testREPLICATE fun = (fun a Z == []) && (fun a (S Z) == [a]) && (fun a
    (S (S Z)) == [a,a]) && (fun a (S (S (S Z))) == [a,a,a]) && (fun a (
    S (S (S (S Z)))) == [a,a,a,a])
testTAKE fun = (fun Z [] == []) && (fun Z [a] == []) && (fun Z [b,c]
    == []) && (fun (S Z) [] == []) && (fun (S Z) [d] == [d]) && (fun (S
     Z) [e,f] == [e]) && (fun (S (S Z)) [] == []) && (fun (S (S Z)) [g]
     == [g]) && (fun (S (S Z)) [h,i] == [h,i]) && (fun (S (S (S Z))) []
     == []) && (fun (S (S (S Z))) [j] == [j]) && (fun (S (S (S Z))) [k,
    l] == [k,l])
testZEROS fun = (fun [] == []) && (fun [Z] == [Z]) && (fun [S Z] ==
    []) && (fun [S(S Z)] == []) && (fun [S(S(S Z))] == []) && (fun [Z,S
     Z] == [Z]) && (fun [Z,S(S Z)] == [Z]) && (fun [Z,S(S(S Z))] == [Z
    ]) && (fun [S Z,Z] == [Z]) && (fun [S(S Z),Z] == [Z]) && (fun [S(S(
    S Z)),Z] == [Z]) && (fun [S Z,S Z] == []) && (fun [S(S Z),S Z] ==
    []) && (fun [S(S(S Z)),S Z] == []) && (fun [S Z,S(S Z)] == []) && (
    fun [S(S Z),S(S Z)] == []) && (fun [S(S(S Z)),S(S Z)] == []) && (
    fun [S Z,S(S(S Z))] == []) && (fun [S(S Z),S(S(S Z))] == []) && (
    fun [S(S(S Z)),S(S(S Z))] == []) && (fun [Z,Z] == [Z,Z])
testPREORDER fun = (fun NilT == []) && (fun (Node a NilT NilT) == [a])
     && (fun (Node a(Node b NilT NilT)(Node c NilT NilT)) == [a,b,c])
    && (fun (Node a(Node b(Node c NilT NilT)(Node d NilT NilT))(Node e(
    Node f NilT NilT)(Node g NilT NilT))) == [a,b,c,d,e,f,g])
testINORDER fun = (fun NilT == []) && (fun (Node a NilT NilT) == [a])
    && (fun (Node a(Node b NilT NilT)(Node c NilT NilT)) == [b,a,c]) &&
     (fun (Node a(Node b(Node c NilT NilT)(Node d NilT NilT))(Node e(
    Node f NilT NilT)(Node g NilT NilT))) == [c,b,d,a,f,e,g])
```

```
testPOSTORDER fun = (fun NilT == []) && (fun (Node a NilT NilT) == [a
   ]) && (fun (Node a(Node b NilT NilT)(Node c NilT NilT)) == [b,c,a])
    && (fun (Node a(Node b(Node c NilT NilT)(Node d NilT NilT))(Node e
   (Node f NilT NilT)(Node g NilT NilT))) == [c,d,b,f,g,e,a])
testMIRROR fun = (fun NilT == NilT) && (fun (Node a NilT NilT) == (
   Node a NilT NilT)) && (fun (Node b(Node a NilT NilT)(Node c NilT
   NilT)) == (Node b(Node c NilT NilT)(Node a NilT NilT))) && (fun (
   Node d(Node b(Node a NilT NilT)(Node c NilT NilT))(Node f(Node e
   NilT NilT)(Node g NilT NilT))) == (Node d(Node f(Node g NilT NilT)(
   Node e NilT NilT))(Node b(Node c NilT NilT)(Node a NilT NilT))))
testPEPPER fun = (fun Z [] == [(Z,Nothing)]) && (fun Z [a] == [(Z,
   Just (a, S Z)),(S Z, Nothing)]) && (fun Z [a,b] == [(Z, Just (a, S
   Z)),(S Z, Just (b, S(S Z))),(S(S Z), Nothing)]) && (fun Z [a,b,c]
   == [(Z, Just (a, S Z)),(S Z, Just (b, S(S Z))),(S(S Z), Just (c, S(
   S(S Z)))),(S(S(S Z)), Nothing)]) && (fun Z [a,b,c,d] == [(Z, Just (
   a, S Z)),(S Z, Just (b, S(S Z))),(S(S Z), Just (c, S(S(S Z)))),(S(S
   (S Z)), Just (d, S(S(S(S Z))))),(S(S(S(S Z))), Nothing)])
testPEPPERF fun = (fun Z [] == [(Z,Nothing)]) && (fun Z [a] == [(Z,
   Just a),(S Z, Nothing)]) && (fun Z [a,b] == [(Z, Just a),(S Z, Just
    b),(S(S Z), Nothing)]) && (fun Z [a,b,c] == [(Z, Just a),(S Z,
   Just b),(S(S Z), Just c),(S(S(S Z)), Nothing)]) && (fun Z [a,b,c,d]
    == [(Z, Just a),(S Z, Just b),(S(S Z), Just c),(S(S(S Z)), Just d)
   ,(S(S(S(S Z))), Nothing)])
testROCKET fun = (fun NOCARGO START == FLY START) && (fun (IN O1
   NOCARGO) START == UNL O1 (FLY (LOD O1 START))) && (fun (IN O1 (IN
   O2 NOCARGO)) START == UNL O1 (UNL O2 (FLY (LOD O2 (LOD O1 START))))
   ) && (fun (IN O1 (IN O2 (IN O3 NOCARGO))) START == UNL O1 (UNL O2 (
   UNL O3 (FLY (LOD O3 (LOD O2 (LOD O1 START)))))))
testHANOI fun = (fun D0 PegA PegB PegC NOOP == MV D0 PegA PegC NOOP)
   && (fun (D D0) PegA PegB PegC NOOP == MV D0 PegB PegC (MV (D D0)
   PegA PegC (MV D0 PegA PegB NOOP))) && (fun (D(D D0)) PegA PegB PegC
    NOOP == MV D0 PegA PegC (MV (D D0) PegB PegC(MV D0 PegB PegA (MV (
   D(D D0)) PegA PegC (MV D0 PegC PegB(MV (D D0) PegA PegB (MV D0 PegA
    PegC NOOP )))))))
testSENTENCE fun = (fun Z == ['D', 'N', 'V', 'D', 'N']) && (fun (S Z)
   == ['D', 'N', 'V', 'D', 'N', 'V', 'D', 'N']) && (fun (S( S Z)) ==
   ['D', 'N', 'V', 'D', 'N', 'V', 'D', 'N', 'V', 'D', 'N'])
testBALLOONS fun = (fun (Yellow,(Small,(Stretch,Adult))) == True) && (
   fun (Yellow,(Small,(Stretch,Child))) == True) && (fun (Yellow,(
   Small,(Dip,Adult))) == True) && (fun (Yellow,(Small,(Dip,Child)))
   == True) && (fun (Yellow,(Large,(Stretch,Adult))) == False) && (fun
    (Yellow,(Large,(Stretch,Child))) == False) && (fun (Yellow,(Large
   ,(Dip,Adult))) == False) && (fun (Yellow,(Large,(Dip,Child))) ==
   False) && (fun (Purple,(Small,(Stretch,Adult))) == False) && (fun (
   Purple,(Small,(Stretch,Child))) == False) && (fun (Purple,(Small,(
   Dip,Adult))) == False) && (fun (Purple,(Small,(Dip,Child))) ==
   False) && (fun (Purple,(Large,(Stretch,Adult))) == False) && (fun (
   Purple,(Large,(Stretch,Child))) == False) && (fun (Purple,(Large,(
   Dip,Adult))) == False) && (fun (Purple,(Large,(Dip,Child))) ==
   False)
testPLAYTENNIS fun = (fun (Sunny,(Hot,(High,Weak))) == False) && (fun
   (Sunny,(Hot,(High,Strong))) == False) && (fun (Overcast,(Hot,(High,
```

```
    Weak))) == True) && (fun (Rain,(Mild,(High,Weak))) == True) && (fun
     (Rain,(Cool,(Normal,Weak))) == True) && (fun (Rain,(Cool,(Normal,
    Strong))) == False) && (fun (Overcast,(Cool,(Normal,Strong))) ==
    True) && (fun (Sunny,(Mild,(High,Weak))) == False) && (fun (Sunny,(
    Cool,(Normal,Weak))) == True) && (fun (Rain,(Mild,(Normal,Weak)))
    == True) && (fun (Sunny,(Mild,(Normal,Strong))) == True) && (fun (
    Overcast,(Mild,(High,Strong))) == True) && (fun (Overcast,(Hot,(
    Normal,Weak))) == True) && (fun (Rain,(Mild,(High,Strong))) ==
    False)
testENJOYSPORT fun = (fun (Sunny,(Warm,(Normal,(Strong,(Warm,Same)))))
     == True) && (fun (Sunny,(Warm,(High,(Strong,(Warm,Same))))) ==
    True) && (fun (Rain,(Cold,(High,(Strong,(Warm,Change))))) == False)
     && (fun (Sunny,(Warm,(High,(Strong,(Cool,Change))))) == True)
testLENSES fun = (fun (Young,(Myope,(No,Reduced))) == None) && (fun (
    Young,(Myope,(No,Norml))) == Soft) && (fun (Young,(Myope,(Yes,
    Reduced))) == None) && (fun (Young,(Myope,(Yes,Norml))) == Hard) &&
     (fun (Young,(Hypermetrope,(No,Reduced))) == None) && (fun (Young,(
    Hypermetrope,(No,Norml))) == Soft) && (fun (Young,(Hypermetrope,(
    Yes,Reduced))) == None) && (fun (Young,(Hypermetrope,(Yes,Norml)))
    == Hard) && (fun (PrePresbyopic,(Myope,(No,Reduced))) == None) && (
    fun (PrePresbyopic,(Myope,(No,Norml))) == Soft) && (fun (
    PrePresbyopic,(Myope,(Yes,Reduced))) == None) && (fun (
    PrePresbyopic,(Myope,(Yes,Norml))) == Hard) && (fun (PrePresbyopic
    ,(Hypermetrope,(No,Reduced))) == None) && (fun (PrePresbyopic,(
    Hypermetrope,(No,Norml))) == Soft) && (fun (PrePresbyopic,(
    Hypermetrope,(Yes,Reduced))) == None) && (fun (PrePresbyopic,(
    Hypermetrope,(Yes,Norml))) == None) && (fun (Presbyopic,(Myope,(No,
    Reduced))) == None) && (fun (Presbyopic,(Myope,(No,Norml))) == None
    ) && (fun (Presbyopic,(Myope,(Yes,Reduced))) == None) && (fun (
    Presbyopic,(Myope,(Yes,Norml))) == Hard) && (fun (Presbyopic,(
    Hypermetrope,(No,Reduced))) == None) && (fun (Presbyopic,(
    Hypermetrope,(No,Norml))) == Soft) && (fun (Presbyopic,(
    Hypermetrope,(Yes,Reduced))) == None) && (fun (Presbyopic,(
    Hypermetrope,(Yes,Norml))) == None)
```

D. IgorII$_H$ Specification

Listing D.1: IGOR II$_\mathrm{H}$ specification

```
{-# OPTIONS_GHC
 -XTypeOperators -XTypeFamilies -XDeriveDataTypeable
#-}
module Specifications where

import Generics.Pointless.Combinators
import Generics.Pointless.Functors
import Generics.Pointless.RecursionPatterns

import Data.List (nub, sort)
import Data.Typeable

-- ==================================================
-- data type definitions
-- ==================================================

-- Peano's Natural Numbers
data Nat = Z | S Nat
 deriving (Eq,Ord,Typeable,Show)

-- Con Lists
data List a = NilL | Cons a (List a)
 deriving (Eq,Ord,Typeable,Show)

-- Binary Node Trees
data NTree a  = NilT | Node a (NTree a)(NTree a)
 deriving (Eq,Ord,Typeable,Show)

-- Binary Leaf Trees
data LTree a = Leaf a | Branch (LTree a) (LTree a)
 deriving (Eq,Ord,Typeable,Show)

-- Rose Trees
data Rose a = Forest a [Rose a]
 deriving  (Eq,Ord,Typeable,Show)

-- ==================================================
-- Defining pattern functor and initial algebras for
-- inductive types
-- ==================================================

type instance PF Nat = Const One :+: Id
```

```
instance Mu Nat where
    inn (Left  _) = Z
    inn (Right p) = S p
    out Z         = Left _L
    out (S p)     = Right p

type instance PF (List a) =
  Const One :+: (Const a :*: Id)
instance Mu (List a) where
    inn (Left _)       = NilL
    inn (Right (a,l)) = Cons a l
    out NilL           = Left _L
    out (Cons a l)     = Right (a,l)

-- Ordinary lists are built-in but semantically their
-- definition and instance declarations for PF and MU
-- would be as follows

-- data [a] = [] | a : [a]
-- type instance PF [a] =
--     Const One :+: (Const a :*: Id)
-- instance Mu [a] where
--     inn (Left _)       = []
--     inn (Right (a,l)) = (a:l)
--     out []             = Left _L
--     out (a:l)          = Right (a,l)

type instance PF (NTree a) =
  Const One :+: (Const a :*: (Id :*: Id))
instance Mu (NTree a) where
    inn (Left _)           = NilT
    inn (Right (a,(l,r))) = Node a l r
    out NilT               = Left _L
    out (Node a l r)       = Right (a,(l,r))

type instance PF (LTree a) = Const a :+: (Id :*: Id)
instance Mu (LTree a) where
    inn (Left a)       = Leaf a
    inn (Right (l,r)) = Branch l r
    out (Leaf a)       = Left a
    out (Branch l r)   = Right (l,r)

type instance PF (Rose a) = Const a :*: ( [] :@: Id )
instance Mu (Rose a) where
    inn (a,rs)         = Forest a rs
    out (Forest a rs) = (a,rs)

-- ================================================
-- functions on natural numbers
-- ================================================

-- The Ackermann function.
```

```
ack :: Nat -> Nat -> Nat
ack (Z) (Z)                   = (S Z)
ack (Z) (S Z)                 = (S(S Z))
ack (Z) (S(S Z))              = (S(S(S Z)))
ack (Z) (S(S(S Z)))           = (S(S(S(S Z))))
ack (Z) (S(S(S(S Z))))        = (S(S(S(S(S Z)))))
ack (Z) (S(S(S(S(S Z)))))     = (S(S(S(S(S(S Z))))))
ack (Z) (S(S(S(S(S(S Z)))))) = (S(S(S(S(S(S(S Z)))))))
ack (S Z) (Z)                 = (S(S Z))
ack (S Z) (S Z)               = (S(S(S Z)))
ack (S Z) (S(S Z))            = (S(S(S(S Z))))
ack (S Z) (S(S(S Z)))         = (S(S(S(S(S Z)))))
ack (S Z) (S(S(S(S Z))))      = (S(S(S(S(S(S Z))))))
ack (S Z) (S(S(S(S(S Z)))))  = (S(S(S(S(S(S(S Z)))))))
ack (S(S Z)) (Z)              = (S(S(S Z)))
ack (S(S Z)) (S Z)            = (S(S(S(S(S Z)))))
ack (S(S Z)) (S(S Z))         = (S(S(S(S(S(S(S Z)))))))
ack (S(S(S Z))) (Z)           = (S(S(S(S(S Z)))))

-- Addition on natural numbers.
-- (using variables for fewer examples)
-- __HASKELLER_IGNORE__
add :: Nat -> Nat -> Nat
add      x          Z    =            x
add      Z    y =  y
add   (S Z)      (S Z)   =      (S(S Z))
add   (S Z)    (S(S Z))  =    (S(S(S Z)))
add   (S Z)  (S(S(S Z))) =  (S(S(S(S Z))))
add (S(S Z))     (S Z)   =    (S(S(S Z)))
add (S(S Z))   (S(S Z))  =  (S(S(S(S Z))))
add (S(S Z)) (S(S(S Z))) = (S(S(S(S(S Z)))))

-- Is the number even?
even :: Nat -> Bool
even Z                  = True
even (S Z)              = False
even (S(S Z))           = True
even (S(S(S Z)))        = False
even (S(S(S(S Z))))     = True
even (S(S(S(S(S Z))))) = False

-- Equality on natural numbers.
eq :: Nat -> Nat -> Bool
eq       Z          Z  = True
eq       Z       (S Z) = False
eq       Z    (S(S Z)) = False
eq    (S Z)         Z  = False
eq    (S Z)      (S Z) = True
eq    (S Z)   (S(S Z)) = False
eq (S(S Z))        Z   = False
eq (S(S Z))     (S Z)  = False
eq (S(S Z)) (S(S Z))  = True
```

```
-- Sum of all natural numbers from 0 to $n$.
gaussSum :: Nat -> Nat
gaussSum Z                                  = Z
gaussSum (S Z)                              = (S Z)
gaussSum (S(S Z))                           =
  (S(S(S Z)))
gaussSum (S(S(S Z)))                        =
  (S(S(S(S(S(S Z))))))
gaussSum (S(S(S(S Z))))                     =
  (S(S(S(S(S(S(S(S(S(S Z))))))))))

-- BK for gaussSum
gaussAdd :: Nat -> Nat -> Nat
gaussAdd (S Z) Z                            = (S Z)
gaussAdd (S(S Z))(S Z)                      =
  (S(S(S Z)))
gaussAdd (S(S(S Z)))(S(S(S Z)))             =
  (S(S(S(S(S(S Z))))))
gaussAdd (S(S(S(S Z))))(S(S(S(S(S(S Z)))))) =
  (S(S(S(S(S(S(S(S(S(S Z))))))))))

-- The fatcorial function.
fact :: Nat -> Nat
fact Z                  = S(Z)
fact (S(Z))             = S(Z)
fact (S(S(Z)))          = S(S(Z))
fact (S(S(S(Z))))       = S(S(S(S(S(S(Z))))))
fact (S(S(S(S(Z)))))    =
  S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(Z)))))))))))))))))))
     )))))
fact (S(S(S(S(S(Z)))))) =
  S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(
     S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S
     (S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(
     S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(Z))))))))))))))))))))))
     )))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
     )))))))))))))))))))))))))))))))))

-- The $n^{th}$ number in the Fibonacci sequence.
fib :: Nat -> Nat
fib Z                   = Z
fib (S(Z))              = S(Z)
fib (S(S(Z)))           = S(Z)
fib (S(S(S(Z))))        = S(S(Z))
fib (S(S(S(S(Z)))))     = S(S(S(Z)))
fib (S(S(S(S(S(Z)))))) = S(S(S(S(S(Z)))))

-- BK for fib
-- __HASKELLER_IGNORE__
fibAdd :: Nat -> Nat -> Nat
fibAdd Z (S Z)               = (S Z)
```

```haskell
fibAdd (S Z) (S Z)          = S(S Z)
fibAdd (S Z) (S(S Z))       = S(S(S Z))
fibAdd (S(S Z)) (S(S(S Z))) = S(S(S(S(S Z))))

-- Greater-or-equal.
geq :: Nat -> Nat -> Bool
geq Z           Z        = True
geq (S Z)       Z        = True
geq (S(S Z))    Z        = True
geq Z           (S Z)    = False
geq (S Z)       (S Z)    = True
geq (S(S Z))    (S Z)    = True
geq Z           (S(S Z)) = False
geq (S Z)       (S(S Z)) = False
geq (S(S Z))    (S(S Z)) = True

-- Multiplication on natural numbers.
mult :: Nat -> Nat -> Nat
mult        Z           Z    =              Z
mult        Z        (S Z)   =              Z
mult        Z      (S(S Z))  =              Z
mult        Z    (S(S(S Z))) =              Z
mult     (S Z)          Z    =              Z
mult     (S Z)       (S Z)   =           (S Z)
mult     (S Z)     (S(S Z))  =         (S(S Z))
mult     (S Z)   (S(S(S Z))) =       (S(S(S Z)))
mult   (S(S Z))         Z    =              Z
mult   (S(S Z))      (S Z)   =         (S(S Z))
mult   (S(S Z))    (S(S Z))  =     (S(S(S(S Z))))
mult   (S(S Z))  (S(S(S Z))) = (S(S(S(S(S(S Z))))))
mult (S(S(S Z)))        Z    =              Z
mult (S(S(S Z)))     (S Z)   =       (S(S(S Z)))
mult (S(S(S Z)))   (S(S Z))  = (S(S(S(S(S(S Z))))))
mult (S(S(S Z))) (S(S(S Z))) =
    (S(S(S(S(S(S(S(S(S Z)))))))))

-- BK for mult
-- __HASKELLER_IGNORE__
multAdd :: Nat -> Nat -> Nat
multAdd (Z) (Z)                                 =
  Z
multAdd (S (Z)) (Z)                             =
  S Z
multAdd (S (Z)) (S (Z))                         =
  S(S Z)
multAdd (S (Z)) (S(S (Z)))                      =
  S(S(S Z))
multAdd (S(S (Z))) (Z)                          =
  S(S Z)
multAdd (S(S (Z))) (S(S (Z)))                   =
  S(S(S(S Z)))
multAdd (S(S (Z))) (S(S(S(S (Z)))))             =
```

```
  S(S(S(S(S(S(S Z))))))
multAdd (S(S(S (Z)))) (Z)                          =
  S(S(S Z))
multAdd (S(S(S (Z)))) (S(S(S (Z))))                =
  S(S(S(S(S(S(S Z))))))
multAdd (S(S(S (Z)))) (S(S(S(S(S(S(S (Z)))))))) =
  S(S(S(S(S(S(S(S(S Z)))))))))

-- Check whether the input is an odd number.
odd :: Nat -> Bool
odd Z                 = False
odd (S Z)             = True
odd (S(S Z))          = False
odd (S(S(S Z)))       = True
odd (S(S(S(S Z))))    = False
odd (S(S(S(S(S Z))))) = True

-- Subtraction on natural numbers.
sub :: Nat -> Nat -> Nat
sub Z Z                     = Z
sub Z (S Z)                 = (S Z)
sub Z (S(S Z))              = (S(S Z))
sub Z (S(S(S Z)))           = (S(S(S Z)))
sub (S Z) Z                 = Z
sub (S Z) (S Z)             = Z
sub (S Z) (S(S Z))          = (S Z)
sub (S Z) (S(S(S Z)))       = (S(S Z))
sub (S(S Z)) Z              = Z
sub (S(S Z)) (S Z)          = Z
sub (S(S Z)) (S(S Z))       = Z
sub (S(S Z)) (S(S(S Z)))    = (S Z)
sub (S(S(S Z))) Z           = Z
sub (S(S(S Z))) (S Z)       = Z
sub (S(S(S Z))) (S(S Z))    = Z
sub (S(S(S Z))) (S(S(S Z))) = Z

-- ====================================================
-- predicates, functions on booleans
-- ====================================================

-- Conjunction of a lists of booleans.
andL :: [Bool] -> Bool
andL []                 = True
andL [True]             = True
andL [False]            = False
andL [True,True]        = True
andL [True,False]       = False
andL [False,True]       = False
andL [False,False]      = False
andL [True,True,True]   = True
andL [False,True,True]  = False
andL [True,False,True]  = False
```

```
andL [True,True,False]   = False
andL [True,False,False]  = False
andL [False,True,False]  = False
andL [False,False,True]  = False
andL [False,False,False] = False

-- Conjunction of two boolean values.
and :: Bool -> Bool -> Bool
and True True   = True
and True False  = False
and False True  = False
and False False = False

-- Check whether the number og \lstln{True} elements
-- is even.
evenParity :: [Bool] -> Bool
evenParity []                     =  True
evenParity [False]                =  True
evenParity [True]                 =  False
evenParity [False, False]         =  True
evenParity [False, True]          =  False
evenParity [True, False]          =  False
evenParity [True, True]           =  True
evenParity [False, False, False] =  True
evenParity [False, False, True]  =  False
evenParity [False, True, False]  =  False
evenParity [False, True, True]   =  True
evenParity [True, False, False]  =  False
evenParity [True, False, True]   =  True
evenParity [True, True, False]   =  True

-- The complement of all booleans in a list.
negateAll :: [Bool] -> [Bool]
negateAll []            = []
negateAll [True]        = [False]
negateAll [False]       = [True]
negateAll [False,False] = [True,True]
negateAll [False,True]  = [True,False]
negateAll [True,False]  = [False,True]
negateAll [True,True]   = [False,False]

-- Negated conjunction of a lists of booleans.
nandL :: [Bool] -> Bool
nandL []                  = False
nandL [True]              = False
nandL [False]             = True
nandL [True,True]         = False
nandL [True,False]        = True
nandL [False,True]        = True
nandL [False,False]       = True
nandL [True,True,True]    = False
nandL [False,True,True]   = True
```

```
nandL [True,False,True]   = True
nandL [True,True,False]   = True
nandL [True,False,False]  = True
nandL [False,True,False]  = True
nandL [False,False,True]  = True
nandL [False,False,False] = True

-- Negated disjunction of a lists of booleans.
norL :: [Bool] -> Bool
norL []                  = True
norL [True]              = False
norL [False]             = True
norL [True,True]         = False
norL [True,False]        = False
norL [False,True]        = False
norL [False,False]       = True
norL [True,True,True]    = False
norL [False,True,True]   = False
norL [True,False,True]   = False
norL [True,True,False]   = False
norL [True,False,False]  = False
norL [False,True,False]  = False
norL [False,False,True]  = False
norL [False,False,False] = True

-- Disjunction of two booleans.
or :: Bool -> Bool -> Bool
or True  True  = True
or True  False = True
or False True  = True
or False False = False

-- Disjunction of a list of booleans.
orL :: [Bool] -> Bool
orL []                  = False
orL [True]              = True
orL [False]             = False
orL [True,True]         = True
orL [True,False]        = True
orL [False,True]        = True
orL [False,False]       = False
orL [True,True,True]    = True
orL [False,True,True]   = True
orL [True,False,True]   = True
orL [True,True,False]   = True
orL [True,False,False]  = True
orL [False,True,False]  = True
orL [False,False,True]  = True
orL [False,False,False] = False

-- ====================================================
-- functions on lists
```

```haskell
-- ====================================================

-- Appending two lists.
append :: [a] -> [a] -> [a]
append [][]        = []
append [][c]       = [c]
append [a][]       = [a]
append [][c,d]     = [c,d]
append [a][c]      = [a,c]
append [a,b][]     = [a,b]
append [] [a,b,c]  = [a,b,c]
append [a][c,d]    = [a,c,d]
append [a,b][d]    = [a,b,d]
append [a,c,d][]   = [a,c,d]
append [][a,b,c,d] = [a,b,c,d]
append [a][b,c,d]  = [a,b,c,d]
append [a,b][c,d]  = [a,b,c,d]
append [a,b,c][d]  = [a,b,c,d]
append [a,b,c,d][] = [a,b,c,d]

-- Is the length of the list even?
evenLength :: [a] -> Bool
evenLength []            = True
evenLength [a]           = False
evenLength [a,b]         = True
evenLength [a,b,c]       = False
evenLength [a,b,c,d]     = True
evenLength [a,b,c,d,e]   = False
evenLength [a,b,c,d,e,f] = True

-- Select all elements at even positions.
evenpos :: [a] -> [a]
evenpos []            = []
evenpos [a]           = []
evenpos [a,b]         = [b]
evenpos [a,b,c]       = [b]
evenpos [a,b,c,d]     = [b,d]
evenpos [a,b,c,d,e]   = [b,d]
evenpos [a,b,c,d,e,f] = [b,d,f]

-- Split a list in two halves.
halves :: [a] -> ([a], [a])
halves []                    = ([], [])
halves [a]                   = ([a], [])
halves [a,b]                 = ([a],[b])
halves [a,b,c]               = ([a,b],[c])
halves [a,b,c,d]             = ([a,b],[c,d])
halves [a,b,c,d,e]           = ([a,b,c],[d,e])
halves [a,b,c,d,e,f]         = ([a,b,c],[d,e,f])

-- Remove the last element from a list.
init :: [a] -> [a]
```

```
init [a]       = []
init [a,b]     = [a]
init [a,b,c]   = [a,b]
init [a,b,c,d] = [a,b,c]

-- All initial segments of the argument,
-- shortest first.
inits :: [a] -> [[a]]
inits []        = [[]]
inits [a]       = [[],[a]]
inits [a,b]     = [[],[a],[a,b]]
inits [a,b,c]   = [[],[a],[a,b],[a,b,c]]
inits [a,b,c,d] = [[],[a],[a,b],[a,b,c],[a,b,c,d]]

-- Intersperse the given element between all two
-- consecutive elements in the list
intersperse :: a -> [a] -> [a]
intersperse x  []     = []
intersperse x [y]     = [y]
intersperse x [y,z]   = [y,x,z]
intersperse x [y,z,v] = [y,x,z,x,v]

-- The last element of a list.
last :: [a] -> a
last [a] = a
last [a,b] = b
last [a,b,c] = c
last [a,b,c,d] = d

-- Last, but defined as total function.
lastM :: [a] -> Maybe a
lastM []        = Nothing
lastM [a]       = Just a
lastM [a,b]     = Just b
lastM [a,b,c]   = Just c
lastM [a,b,c,d] = Just d

-- Replace all elements by the first.
multfst :: [a] -> [a]
multfst []              = []
multfst [a]             = [a]
multfst [a,b]           = [a,a]
multfst [a,b,c]         = [a,a,a]
multfst [a,b,c,d]       = [a,a,a,a]

-- Replace all elements by the last.
multlst :: [a] -> [a]
multlst []              = []
multlst [a]             = [a]
multlst [a,b]           = [b,b]
multlst [a,b,c]         = [c,c,c]
multlst [a,b,c,d]       = [d,d,d,d]
```

```
-- Select all elements at even positions.
oddpos :: [a] -> [a]
oddpos []          = []
oddpos [a]         = [a]
oddpos [a,b]       = [a]
oddpos [a,b,c]     = [a,c]
oddpos [a,b,c,d]   = [a,c]
oddpos [a,b,c,d,e] = [a,c,e]

-- Wraps all elements into a singleton list.
pack :: [a] -> [[a]]
pack []      = [[]]
pack [a]     = [[a]]
pack [a,b]   = [[a],[b]]
pack [a,b,c] = [[a],[b],[c]]

-- All subsequences of a list, aka powerset on lists.
subseqs :: [a] -> [[a]]
subseqs []          = [[]]
subseqs [a]         = [[a],[]]
subseqs [a,b]       = [[a,b],[a],[b],[]]
subseqs [a,b,c]     =
  [[a,b,c],[a,b],[a,c],[a],[b,c],[b],[c],[]]

-- BK for subseqs
-- __HASKELLER_IGNORE__
subseqapp  :: [a] -> [a] -> [a]
subseqapp [a][c]               = [a,c]
subseqapp [a,b][c,d]           = [a,b,c,d]
subseqapp [a,b,c,d][e,f,g,h]   = [a,b,c,d,e,f,g,h]

-- Reverse of a list.
reverse :: [a] -> [a]
reverse []               = []
reverse [a]              =[a]
reverse [a,b]            = [b,a]
reverse [a,b,c]          = [c,b,a]
reverse [a,b,c,d]        = [d,c,b,a]

-- Shift all elements to the left, by inserting
-- the first at the end.
shiftl :: [a] -> [a]
shiftl []                   = []
shiftl [a]                  = [a]
shiftl [a,b]                = [b,a]
shiftl [a,b,c]              = [b,c,a]
shiftl [a,b,c,d]            = [b,c,d,a]

-- Shift all elements to the right, by inserting
-- the last at the front.
shiftr :: [a] -> [a]
```

```
shiftr []            = []
shiftr [a]           = [a]
shiftr [a,b]         = [b,a]
shiftr [a,b,c]       = [c,a,b]
shiftr [a,b,c,d]     = [d,a,b,c]

-- Inserts an element at the end.
snoc :: a -> [a] -> [a]
snoc a []           = [a]
snoc b [a]          = [a,b]
snoc c [a,b]        = [a,b,c]
snoc d [a,b,c]      = [a,b,c,d]

-- Swaps every two concequitive elements.
swap :: [a] -> [a]
swap []            = []
swap [a]           = [a]
swap [a,b]         = [b,a]
swap [a,b,c]       = [b,a,c]
swap [a,b,c,d]     = [b,a,d,c]
swap [a,b,c,d,e]   = [b,a,d,c,e]
swap [a,b,c,d,e,f] = [b,a,d,c,f,e]

-- Switches the first with the last element.
switch :: [a] -> [a]
switch []            = []
switch [a]           = [a]
switch [a,b]         = [b,a]
switch [a,b,c]       = [c,b,a]
switch [a,b,c,d]     = [d,b,c,a]
switch [a,b,c,d,e]   = [e,b,c,d,a]
switch [a,b,c,d,e,f] = [f,b,c,d,e,a]

-- Computes the lists of elements at odd and even
-- positions
split :: [a] -> ([a],[a])
split []            = ([],[])
split [x]           =  ([x],[])
split [x,y]         = ([x],[y])
split [x,y,z]       = ([x,z],[y])
split [x,y,z,v]     = ([x,z],[y,v])
split [x,y,z,v,w]   = ([x,z,w],[y,v])
split [x,y,z,v,w,u] = ([x,z,w],[y,v,u])

-- Removes the first element
tail :: [a] -> [a]
tail [a]       = []
tail [a,b]     = [b]
tail [a,b,c]   = [b,c]
tail [a,b,c,d] = [b,c,d]

-- \lstln{map tail}
```

```
tails :: [a] -> [[a]]
tails []      = [[]]
tails [a]     = [[a],[]]
tails [a,b]   = [[a,b],[b],[]]
tails [a,b,c] = [[a,b,c],[b,c],[c],[]]

-- Compute the list first and second projections.
unzip :: [(a,a)] -> ([a], [a])
unzip [(a,b)]                     = ([a],[b])
unzip [(a,b),(c,d)]               = ([a,c],[b,d])
unzip [(a,b),(c,d),(e,f)]         = ([a,c,e],[b,d,f])
unzip [(a,b),(c,d),(e,f),(g,h)] =
  ([a,c,e,g],[b,d,f,h])

-- Combines two lists by interleaving their elements.
weave :: [a] -> [a] -> [a]
weave [] []      = []
weave [a][]      = [a]
weave [][c]      = [c]
weave [a][c]     = [a,c]
weave [a,b][]    = [a,b]
weave [][c,d]    = [c,d]
weave [a,b][c]   = [a,c,b]
weave [a][c,d]   = [a,c,d]
weave [a,b][c,d] = [a,c,b,d]

-- Computes the list of corresponding pairs.
zip :: [a] -> [a] -> [(a,a)]
zip [] []       = []
zip [a] []      = []
zip [] [a]      = []
zip [a] [b]     = [(a,b)]
zip [a,b] [c]   = [(a,c)]
zip [a] [b,c]   = [(a,b)]
zip [a,b] [c,d] = [(a,c),(b,d)]

-- =====================================================
-- functions on lists of lists
-- =====================================================

-- Concatenate all lists.
concat :: [[a]] -> [a]
concat []            = []
concat [[]]          = []
concat [[],[]]       = []
concat [[],[a]]      = [a]
concat [[],[a,b]]    = [a,b]
concat [[a]]         = [a]
concat [[a],[]]      = [a]
concat [[a],[b]]     = [a,b]
concat [[a],[c,d]]   = [a,c,d]
concat [[c,d]]       = [c,d]
```

```
concat [[a,b],[]]    = [a,b]
concat [[a,b],[c]]   = [a,b,c]
concat [[a,b],[c,d]] = [a,b,c,d]

-- \lstln{map last}
lasts :: [[a]] -> [a]
lasts []                      = []
lasts [[a]]                   = [a]
lasts [[a,b]]                 = [b]
lasts [[a,b,c]]               = [c]
lasts [[b],[a]]               = [b,a]
lasts [[c],[a,b]]             = [c,b]
lasts [[a,b],[c,d]]           = [b,d]
lasts [[c,d],[b]]             = [d,b]
lasts [[c],[d,e],[f]]         = [c,e,f]
lasts [[c,d],[e,f],[g]]       = [d,f,g]

-- Inerts the element at front of each list.
mapCons :: a -> [[a]] -> [[a]]
mapCons a []            = []
mapCons a [[]]          = [[a]]
mapCons a [xs]          = [(a:xs)]
mapCons a [xs,ys]       = [(a:xs),(a:ys)]
mapCons a [xs,ys,zs]    = [(a:xs),(a:ys),(a:zs)]
mapCons a [xs,ys,zs,ws] =
  [(a:xs),(a:ys),(a:zs),(a:ws)]

-- \lstln{map tail}
mapTail :: [[a]] -> [[a]]
mapTail []                        = []
mapTail [(x:xs)]                  = [xs]
mapTail [(x:xs),(y:ys)]           = [xs,ys]
mapTail [(x:xs),(y:ys),(z:zs)] = [xs,ys,zs]

-- Transpose a matrix.
transpose :: [[a]] -> [[a]]
-- one row
transpose [[a11]]          = [[a11]]
transpose [[a11,a12]]      = [[a11],[a12]]
transpose [[a11,a12,a13]] = [[a11],[a12],[a13]]
-- two rows
transpose [[a11]
          ,[a21]]          = [[a11,a21]]
transpose [[a11,a12]
          ,[a21,a22]]      = [[a11,a21]
                             ,[a12,a22]]
transpose [[a11,a12,a13]
          ,[a21,a22,a23]] = [[a11,a21]
                             ,[a12,a22]
                             ,[a13,a23]]
-- three rows
transpose [[a11]
```

```
          ,[a21]
          ,[a31]]          =[[a11,a21,a31]]
transpose [[a11,a12]
          ,[a21,a22]
          ,[a31,a32]]      = [[a11,a21,a31]
                             ,[a12,a22,a32]]
transpose [[a11,a12,a13]
          ,[a21,a22,a23]
          ,[a31,a32,a33]] = [[a11,a21,a31]
                             ,[a12,a22,a32]
                             ,[a13,a23,a33]]

-- Turns a matrix into a list, by appending
-- its columns.
weaveL :: [[a]] -> [a]
weaveL []                 = []
weaveL [[a11]]            = [a11]
weaveL [[a11,a12]]        = [a11,a12]
weaveL [[a11,a12,a13]]    = [a11,a12,a13]
weaveL [[a11]
       ,[a21]]            = [a11,a21]
weaveL [[a11 ,a12]
       ,[a21]]            = [a11 ,a21 ,a12]
weaveL [[a11]
       ,[a21,a22]]        = [a11 ,a21 ,a22]
weaveL [[a11,a12]
       ,[a21 ,a22]]       = [a11 ,a21 ,a12 ,a22]
weaveL [[a11 ,a12 ,a13]
       ,[a21 ,a22]]       = [a11 ,a21 ,a12 ,a22 ,a13]
weaveL [[a11 ,a12]
       ,[a21 ,a22, a23]] = [a11 ,a21 ,a12 ,a22 ,a23]
weaveL [[a11]
       ,[a21]
       ,[a31]]            = [a11 ,a21 ,a31]
weaveL [[a11,a12]
       ,[a21]
       ,[a31]]            = [a11 ,a21 ,a31 ,a12]
weaveL [[a11]
       ,[a21,a22]
       ,[a31]]            = [a11 ,a21 ,a31 ,a22]
weaveL [[a11]
       ,[a21]
       ,[a31,a32]]        = [a11 ,a21 ,a31 ,a32]
weaveL [[a11]
       ,[a21,a22]
       ,[a31,a32]]        = [a11 ,a21 ,a31 ,a22,a32]
weaveL [[a11,a12]
       ,[a21]
       ,[a31,a32]]        = [a11 ,a21 ,a31 ,a12,a32]
weaveL [[a11,a12]
       ,[a21,a22]
       ,[a31]]            = [a11 ,a21 ,a31 ,a12,a22]
```

```
-- ====================================================
-- functions on naturals and lists
-- ====================================================

-- Increment all elements by a given number.
addN :: Nat -> [Nat] -> [Nat]
addN      Z    []          = []
addN    (S Z)  []          = []
addN (S(S Z))  []          = []
addN      Z    [Z]         = [Z]
addN      Z    [S Z]       = [S Z]
addN      Z    [S(S Z)]    = [S(S Z)]
addN      Z    [Z,(S Z)]   = [Z,(S Z)]
addN      Z    [(S Z),Z]   = [(S Z),Z]
addN    (S Z)  [Z,(S Z)]   = [S Z,S(S Z)]
addN    (S Z)  [(S Z),Z]   = [S(S Z),S Z]
addN    (S Z)  [Z]         = [S Z]
addN    (S Z)  [S Z]       = [S(S Z)]
addN    (S Z)  [S(S Z)]    = [S(S(S Z))]
addN (S(S Z))  [Z]         = [S(S Z)]
addN (S(S Z))  [S Z]       = [S(S(S Z))]
addN (S(S Z))  [S(S Z)]    = [S(S(S(S Z)))]
addN (S(S Z))  [Z,(S Z)]   = [S(S Z),S(S(S Z))]
addN (S(S Z))  [S(S Z),Z]  = [S(S(S(S Z))),S(S Z)]

-- Are all numbers even?
alleven :: [Nat] -> Bool
alleven []                           = True
alleven [Z]                          = True
alleven [S Z]                        = False
alleven [S(S Z)]                     = True
alleven [S(S(S Z))]                  = False
alleven [Z, Z]                       = True
alleven [Z, S Z]                     = False
alleven [Z, S(S Z)]                  = True
alleven [Z, S(S(S Z))]               = False
alleven [S Z, Z]                     = False
alleven [S Z, S Z]                   = False
alleven [S Z, S(S Z)]                = False
alleven [S Z, S(S(S Z))]             = False
alleven [S(S Z), Z]                  = True
alleven [S(S Z), S Z]                = False
alleven [S(S Z), S(S Z)]             = True
alleven [S(S Z), S(S(S Z))]          = False
alleven [S(S(S Z)), Z]               = False
alleven [S(S(S Z)), S Z]             = False
alleven [S(S(S Z)), S(S Z)]          = False
alleven [S(S(S Z)), S(S(S Z))]       = False

-- Are all numbers odd?
allodd :: [Nat] -> Bool
```

```
allodd []                        = True
allodd [Z]                       = False
allodd [S Z]                     = True
allodd [S(S Z)]                  = False
allodd [S(S(S Z))]               = True
allodd [Z, Z]                    = False
allodd [Z, S Z]                  = False
allodd [Z, S(S Z)]               = False
allodd [Z, S(S(S Z))]            = False
allodd [S Z, Z]                  = False
allodd [S Z, S Z]                = True
allodd [S Z, S(S Z)]             = False
allodd [S Z, S(S(S Z))]          = True
allodd [S(S Z), Z]               = False
allodd [S(S Z), S Z]             = False
allodd [S(S Z), S(S Z)]          = False
allodd [S(S Z), S(S(S Z))]       = False
allodd [S(S(S Z)), Z]            = False
allodd [S(S(S Z)), S Z]          = True
allodd [S(S(S Z)), S(S Z)]       = False
allodd [S(S(S Z)), S(S(S Z))]    = True

-- Select all even numbers.
evens :: [Nat] -> [Nat]
evens []                           = []
evens [Z]                          = [Z]
evens [S Z]                        = []
evens [S(S Z)]                     = [S(S Z)]
evens [S(S(S Z))]                  = []
evens [Z,Z]                        = [Z,Z]
evens [Z,S Z]                      = [Z]
evens [Z,S(S Z)]                   = [Z,S(S Z)]
evens [Z,S(S(S Z))]                = [Z]
evens [S Z, Z]                     = [Z]
evens [S Z, S Z]                   = []
evens [S Z, S(S Z)]                = [S(S Z)]
evens [S Z, S(S(S Z))]             = []
evens [S(S Z), Z]                  = [S(S Z),Z]
evens [S(S Z), S Z]                = [S(S Z)]
evens [S(S Z), S(S Z)]             = [S(S Z),S(S Z)]
evens [S(S Z), S(S(S Z))]          = [S(S Z)]
evens [S( S(S Z)), Z]              = [Z]
evens [S( S(S Z)), S Z]            = []
evens [S( S(S Z)), S(S Z)]         = [S(S Z)]
evens [S( S(S Z)), S( S(S Z))]     = []

-- Increment al numbers in a list by one
incr :: [Nat] -> [Nat]
incr []            = []
incr [Z]           = [S Z]
incr [S Z]         = [S(S Z)]
incr [Z,S Z]       = [S Z,S(S Z)]
```

```
incr [S Z,Z]      = [S(S Z),S Z]

-- The length of a list.
length :: [a] -> Nat
length []      = Z
length [a]     = S Z
length [a,b]   = S(S Z)
length [a,b,c] = S(S(S Z))

-- \lstln{map length}
lengths :: [[a]] -> [Nat]
-- 0 sublist
lengths []            = []
lengths [[]]          = [Z]
-- 1 sublist
lengths [[a]]         = [S Z]
lengths [[b,a]]       = [S(S Z)]
lengths [[c,b,a]]     = [S(S(S Z))]
-- 2 0/n sublists
lengths [[],[]]       = [Z, Z]
lengths [[],[a]]      = [Z,S Z]
lengths [[],[b,a]]    = [Z,S(S Z)]
-- 2 1/n sublists
lengths [[a],[]]      = [S Z, Z]
lengths [[b],[a]]     = [S Z,S Z]
lengths [[c],[b,a]]   = [S Z,S(S Z)]
-- 2 2/n sublists
lengths [[c,a],[]]    = [S(S Z), Z]
lengths [[b,a],[c]]   = [S(S Z),S Z]
lengths [[c,d],[b,a]] = [S(S Z),S(S Z)]
-- 3 1/1/1 sublists
lengths [[a],[b],[c]] = [S Z, S Z, S Z]

-- Return the $n^{th}$ element.
nthElem :: [a] -> Nat -> a
nthElem (x:xs) Z                       = x
nthElem (x:y:xs) (S Z)                 = y
nthElem (x:y:z:xs) (S(S Z))            = z
nthElem (x:y:z:u:xs) (S(S(S Z)))       = u
nthElem (x:y:z:u:v:xs) (S(S(S(S Z)))) = v

-- Are all elements odd?
oddslist :: [Nat] -> Bool
oddslist []                 = True
oddslist [Z]                = False
oddslist [S Z]              = True
oddslist [S(S Z)]           = False
oddslist [S(S(S Z))]        = True
oddslist [Z, Z]             = False
oddslist [Z, S Z]           = False
oddslist [Z, S(S Z)]        = False
oddslist [Z, S(S(S Z))]     = False
```

```
oddslist [S Z, Z]               = False
oddslist [S Z, S Z]             = True
oddslist [S Z, S(S Z)]          = False
oddslist [S Z, S(S(S Z))]       = True
oddslist [S(S Z), Z]            = False
oddslist [S(S Z), S Z]          = False
oddslist [S(S Z), S(S Z)]       = False
oddslist [S(S Z), S(S(S Z))]    = False
oddslist [S(S(S Z)), Z]         = False
oddslist [S(S(S Z)), S Z]       = True
oddslist [S(S(S Z)), S(S Z)]    = False
oddslist [S(S(S Z)), S(S(S Z))] = True

-- Select all odd elements.
odds :: [Nat] -> [Nat]
odds []                        = []
odds [Z]                       = []
odds [S Z]                     = [S Z]
odds [S(S Z)]                  = []
odds [S(S(S Z))]               = [(S(S(S Z)))]
odds [Z,Z]                     = []
odds [Z,S Z]                   = [S Z]
odds [Z,S(S Z)]                = []
odds [Z,S(S(S Z))]             = [S(S(S Z))]
odds [S Z, Z]                  = [S Z]
odds [S Z, S Z]                = [S Z, S Z]
odds [S Z, S(S Z)]             = [(S Z)]
odds [S Z, S(S(S Z))]          = [S Z, S(S(S Z))]
odds [S(S Z), Z]               = []
odds [S(S Z), S Z]             = [(S Z)]
odds [S(S Z), S(S Z)]          = []
odds [S(S Z), S(S(S Z))]       = [(S(S(S Z)))]
odds [S( S(S Z)), Z]           = [S( S(S Z))]
odds [S( S(S Z)), S Z]         = [S( S(S Z)), S Z]
odds [S( S(S Z)), S(S Z)]      = [(S(S(S Z)))]
odds [S( S(S Z)), S( S(S Z))] =
  [S( S(S Z)), S( S(S Z))]

-- Drop the first \lstln{n} elements of a list
drop :: Nat -> [a] -> [a]
drop Z           []      = []
drop Z           [a]     = [a]
drop (S Z)       []      = []
drop (S(S Z))   []       = []
drop Z           [a,b]   = [a,b]
drop Z           [a,b,c] = [a,b,c]
drop (S Z)       [a]     = []
drop (S Z)       [a,b]   = [b]
drop (S Z)       [a,b,c] = [b,c]
drop (S(S Z))   [a]      = []
drop (S(S Z))   [a,b]    = []
drop (S(S Z))   [a,b,c]  = [c]
```

```
drop (S(S(S Z))) []        = []
drop (S(S(S Z))) [a]       = []
drop (S(S(S Z))) [a,b]     = []
drop (S(S(S Z))) [a,b,c]   = []

-- Split a list before at given position.
splitAt :: Nat -> [a] -> ([a],[a])
splitAt Z [a]                  = ([],[a])
splitAt Z [a,b]                = ([],[a,b])
splitAt Z [a,b,c]              = ([],[a,b,c])
splitAt Z [a,b,c,d]            = ([],[a,b,c,d])
splitAt (S Z) [a]              = ([a],[])
splitAt (S Z) [a,b]            = ([a],[b])
splitAt (S Z) [a,b,c]          = ([a],[b,c])
splitAt (S Z) [a,b,c,d]        = ([a],[b,c,d])
splitAt (S(S Z)) [a]           = ([a],[])
splitAt (S(S Z)) [a,b]         = ([a,b],[])
splitAt (S(S Z)) [a,b,c]       = ([a,b],[c])
splitAt (S(S Z)) [a,b,c,d]     = ([a,b],[c,d])
splitAt (S(S(S Z))) [a]        = ([a],[])
splitAt (S(S(S Z))) [a,b]      = ([a,b],[])
splitAt (S(S(S Z))) [a,b,c]    = ([a,b,c],[])
splitAt (S(S(S Z))) [a,b,c,d] = ([a,b,c],[d])

-- The sum of a list of integers.
sum :: [Nat] -> Nat
sum []                       = Z
sum [Z]                      = Z
sum [S Z]                    = S Z
sum [S(S Z)]                 = S(S Z)
sum [Z,Z]                    = Z
sum [Z,S Z]                  = S Z
sum [Z,S(S Z)]               = S(S Z)
sum [S Z,Z]                  = S Z
sum [S Z,S Z]                = S(S Z)
sum [S Z,S(S Z)]             = S(S(S Z))
sum [S(S Z),Z]               = S(S Z)
sum [S(S Z),S Z]             = S(S(S Z))
sum [S(S Z),S(S Z)]          = S(S(S(S Z)))

-- A list of length of length \lstln{n} containing
-- only the given element.
replicate :: a -> Nat -> [a]
replicate a Z              = []
replicate a (S Z)          = [a]
replicate a (S(S Z))       = [a,a]
replicate a (S(S(S Z)))    = [a,a,a]
replicate a (S(S(S(S Z)))) = [a,a,a,a]

-- Take the first $n$ elements.
take :: Nat -> [a] -> [a]
take Z              [] = []
```

```
take Z              [a] = []
take Z            [b,c] = []
take (S Z)          [] = []
take (S Z)         [d] = [d]
take (S Z)       [e,f] = [e]
take (S(S Z))       [] = []
take (S(S Z))      [g] = [g]
take (S(S Z))    [h,i] = [h,i]
take (S(S(S Z)))    [] = []
take (S(S(S Z)))   [j] = [j]
take (S(S(S Z))) [k,l] = [k,l]

-- Remove all non-zero integers from a list
zeros :: [Nat] -> [Nat]
zeros []        = []
zeros [Z]       = [Z]
zeros [S x]     = []
zeros [Z,S x]   = [Z]
zeros [S x,Z]   = [Z]
zeros [S x,S y] = []
zeros [Z,Z]     = [Z,Z]

-- ==================================================
-- functions on trees
-- ==================================================

-- Preorder traversal of a binary tree.
preorder :: (NTree a) -> [a]
preorder NilT                    = []
preorder (Node a NilT NilT)      = [a]
preorder (Node a
          (Node b NilT NilT)
          (Node c NilT NilT))    = [a,b,c]
preorder (Node a
          (Node b
           (Node c NilT NilT)
           (Node d NilT NilT))
          (Node e
           (Node f NilT NilT)
           (Node g NilT NilT))) = [a,b,c,d,e,f,g]

-- BK of preorder
-- __HASKELLER_IGNORE__
preapp :: [a] -> [a] -> [a]
preapp [] []          = []
preapp [a] [b]        = [a,b]
preapp [a,b] [c,d]    = [a,b,c,d]
preapp [a,b,c][d,e,f] = [a,b,c,d,e,f]

-- Inorder traversal of a binary tree.
inorder :: (NTree a) -> [a]
inorder NilT                     = []
```

```
inorder (Node a NilT NilT)       = [a]
inorder (Node a
          (Node b NilT NilT)
          (Node c NilT NilT))    = [b,a,c]
inorder (Node a
          (Node b
           (Node c NilT NilT)
           (Node d NilT NilT))
          (Node e
           (Node f NilT NilT)
           (Node g NilT NilT))) = [c,b,d,a,f,e,g]

-- BK of inorder
-- __HASKELLER_IGNORE__
inapp :: [a] -> [a] -> [a]
inapp [] []              = []
inapp [x] []             = [x]
inapp [a] [b,c]          = [a,b,c]
inapp [a,b] [c,d,e]      = [a,b,c,d,e]
inapp [a,b,c] [d,e,f,g] = [a,b,c,d,e,f,g]

-- Postorder traversal of a binary tree.
postorder :: (NTree a) -> [a]
postorder NilT                   = []
postorder (Node a NilT NilT)     = [a]
postorder (Node a
          (Node b NilT NilT)
          (Node c NilT NilT))    = [b,c,a]
postorder (Node a
          (Node b
           (Node c NilT NilT)
           (Node d NilT NilT))
          (Node e
           (Node f NilT NilT)
           (Node g NilT NilT))) = [c,d,b,f,g,e,a]

-- BK of postrder
-- __HASKELLER_IGNORE__
postapp :: [a] -> [a] -> [a]
postapp [] []           = []
postapp [a] [b]         = [a,b]
postapp [a,b] [c,d]     = [a,b,c,d]
postapp [a,b,c][d,e,f] = [a,b,c,d,e,f]

-- Mirror a binary tree by swapping all its subtrees.
mirror :: (NTree a) -> (NTree a)
mirror NilT                    = NilT
mirror (Node a NilT NilT)      = (Node a NilT NilT)
mirror (Node b
        (Node a NilT NilT)
        (Node c NilT NilT))    = (Node b
                                   (Node c NilT NilT)
```

```
                                    (Node a NilT NilT))
mirror (Node d
        (Node b
         (Node a NilT NilT)
         (Node c NilT NilT))
        (Node f
         (Node e NilT NilT)
         (Node g NilT NilT))) =
        (Node d
         (Node f
          (Node g NilT NilT)
          (Node e NilT NilT))
         (Node b
          (Node c NilT NilT)
          (Node a NilT NilT)))

-- ==================================================
-- functions on mixed inputs
-- ==================================================


-- pepper i []     = [(i,Nothing)]
-- pepper i (x:xs) =
--    (i,Just (x,i+1)):(pepper' (i+1) xs)
-- Annotate each element with an index and the index
-- of its predecessor.
pepper :: Nat -> [a] -> [(Nat, Maybe (a,Nat))]
pepper p []        = [(p,Nothing)]
pepper p [a]       = [(p, Just (a, S p)),(S p, Nothing)]
pepper p [a,b]     = [(p, Just (a, S p))
                     ,(S p, Just (b, S(S p)))
                     ,(S(S p), Nothing)]
pepper p [a,b,c]   = [(p, Just (a, S p))
                     ,(S p, Just (b, S(S p)))
                     ,(S(S p), Just (c, S(S(S p))))
                     ,(S(S(S p)), Nothing)]
pepper p [a,b,c,d] = [(p, Just (a, S p))
                     ,(S p, Just (b, S(S p)))
                     ,(S(S p), Just (c, S(S(S p))))
                     ,(S(S(S p)), Just (d, S(S(S(S p)))))
                     ,(S(S(S(S p))), Nothing)]

-- Index all elements starting by the given number.
pepperF :: Nat -> [a] -> [(Nat, Maybe a)]
pepperF p []        = [(p,Nothing)]
pepperF p [a]       = [(p, Just a),(S p, Nothing)]
pepperF p [a,b]     = [(p, Just a),(S p, Just b)
                      ,(S(S p), Nothing)]
pepperF p [a,b,c]   = [(p, Just a),(S p, Just b)
                      ,(S(S p), Just c),(S(S(S p)), Nothing)]
pepperF p [a,b,c,d] = [(p, Just a),(S p, Just b)
                      ,(S(S p), Just c),(S(S(S p)), Just d)
```

```
                       ,(S(S(S(S p))), Nothing)]

-- ====================================================
-- functions on other data types
-- ====================================================

data Object = O1 | O2 | O3 | O4
 deriving  (Eq,Ord,Typeable,Show)
data Cargo = NOCARGO | IN Object Cargo
 deriving  (Eq,Ord,Typeable,Show)
data State = START | LOD Object State
           | UNL Object State | FLY State
 deriving  (Eq,Ord,Typeable,Show)

type instance PF Cargo =
    Const One :+: (Const Object :*: Id)
instance Mu Cargo where
  inn (Left _)      = NOCARGO
  inn (Right (o,c)) = IN o c
  out NOCARGO       = Left _L
  out (IN o c)      = Right (o,c)

type instance PF State =
    Const One :+: (Const Object :*: Id)
instance Mu State where
  inn (Left _)      = START
  inn (Right (o,s)) = LOD o s
  out START       = Left _L
  out (LOD o s)      = Right (o,s)

-- The planning problem of loading a rocket and
-- flying it to the moon.
rocket :: Cargo -> State -> State
rocket                         NOCARGO     s = FLY s
rocket                   (IN x NOCARGO)    s =
  UNL x (FLY (LOD x s))
rocket             (IN x (IN y NOCARGO))   s =
  UNL x
   (UNL y
    (FLY
     (LOD y
      (LOD x s))))
rocket       (IN x (IN y (IN z NOCARGO)))  s =
  UNL x
  (UNL y
   (UNL z
    (FLY
     (LOD z
      (LOD y
       (LOD x s))))))
rocket (IN w (IN x (IN y (IN z NOCARGO)))) s =
  UNL w
```

```
  (UNL x
   (UNL y
    (UNL z
     (FLY
      (LOD z
       (LOD y
        (LOD x
         (LOD w s))))))))

data Disc = D0 | D Disc
 deriving  (Eq,Ord,Typeable,Show)
data Action = NOOP | MV Disc Peg Peg Action
 deriving  (Eq,Ord,Typeable,Show)
data Peg = PegA | PegB | PegC
 deriving  (Eq,Ord,Typeable,Show)

type instance PF Disc = Const One :+: Id
instance Mu Disc where
  inn (Left  _) = D0
  inn (Right d) = D d
  out D0         = Left _L
  out (D d)     = Right d

type instance PF Action =
  Const One :+: (Const Disc :*:
               (Const Peg :*: (Const Peg :*: Id)))
instance Mu Action where
  inn (Left _)                  = NOOP
  inn (Right (d,(p1,(p2,a)))) = MV d p1 p2 a
  out NOOP                      = Left _L
  out (MV d p1 p2 a)            = Right (d,(p1,(p2,a)))

-- Recursive definition of The Tower of Hanoi problem.
hanoi :: Disc -> Peg -> Peg -> Peg -> Action -> Action
hanoi D0 src aux dst s              = MV D0 src dst s
hanoi (D D0) src aux dst s          =
  MV D0 aux dst
   (MV (D D0) src dst
    (MV D0 src aux s))
hanoi (D(D D0)) src aux dst s       =
  MV D0 src dst
   (MV (D D0) aux dst
    (MV D0 aux src
     (MV (D(D D0)) src dst
      (MV D0 dst aux
       (MV (D D0) src aux
        (MV D0 src dst s ))))))
hanoi (D(D(D D0))) src aux dst NOOP =
  MV D0 aux dst
  (MV (D D0) src dst
   (MV D0 src aux
    (MV (D (D D0)) aux dst
```

```
      (MV D0 dst src
       (MV (D D0) aux src
        (MV D0 aux dst
         (MV (D (D (D D0))) src dst
          (MV D0 src aux
           (MV (D D0) dst aux
            (MV D0 dst src
             (MV (D (D D0)) src aux
              (MV D0 aux dst
               (MV (D D0) src dst
                (MV D0 src aux NOOP))))))))))))))

-- Enumerating all sentences of a grammar
-- Intended Grammar
-- S  := NP VP
-- NP := D N
-- VP := V NP | V S
sentence :: Nat -> [Char]
sentence Z          = ['D', 'N', 'V', 'D', 'N']
sentence (S Z)      = ['D', 'N', 'V', 'D', 'N'
                      , 'V', 'D', 'N']
sentence (S( S Z)) = ['D', 'N', 'V', 'D', 'N'
                      , 'V', 'D', 'N', 'V', 'D', 'N']

-- ======================================================
-- functions forUCI classification problems
-- ======================================================

data Color = Purple | Yellow
 deriving  (Eq,Ord,Typeable,Show)
data Size = Large | Small
 deriving  (Eq,Ord,Typeable,Show)
data Act = Dip | Stretch
 deriving  (Eq,Ord,Typeable,Show)
data Age = Adult | Child
 deriving  (Eq,Ord,Typeable,Show)
data Inflate = FF | TT
 deriving  (Eq,Ord,Typeable,Show)

-- UCI classification problem
balloons :: (Color,Size,Act,Age) -> Inflate
balloons(Yellow,Small,Stretch,Adult) = TT
balloons(Yellow,Small,Stretch,Child) = TT
balloons(Yellow,Small,Dip,Adult)     = TT
balloons(Yellow,Small,Dip,Child)     = TT
balloons(Yellow,Large,Stretch,Adult) = FF
balloons(Yellow,Large,Stretch,Child) = FF
balloons(Yellow,Large,Dip,Adult)     = FF
balloons(Yellow,Large,Dip,Child)     = FF
balloons(Purple,Small,Stretch,Adult) = FF
balloons(Purple,Small,Stretch,Child) = FF
balloons(Purple,Small,Dip,Adult)     = FF
```

```
balloons(Purple,Small,Dip,Child)      = FF
balloons(Purple,Large,Stretch,Adult) = FF
balloons(Purple,Large,Stretch,Child) = FF
balloons(Purple,Large,Dip,Adult)      = FF
balloons(Purple,Large,Dip,Child)      = FF

data Weather = Sunny | Rain | Overcast | Hot
             | Cool | Mild | Warm | Cold | High
             |  Normal | Weak |Strong | Change | Same
 deriving  (Eq,Ord,Typeable,Show)

-- Classification poblem (Mitchell)
playTennis :: (Weather, Weather, Weather, Weather)
           -> Bool
playTennis (Sunny,Hot,High,Weak)          = False
playTennis (Sunny,Hot,High,Strong)        = False
playTennis (Overcast,Hot,High,Weak)       = True
playTennis (Rain,Mild,High,Weak)          = True
playTennis (Rain,Cool,Normal,Weak)        = True
playTennis (Rain,Cool,Normal,Strong)      = False
playTennis (Overcast,Cool,Normal,Strong) = True
playTennis (Sunny,Mild,High,Weak)         = False
playTennis (Sunny,Cool,Normal,Weak)       = True
playTennis (Rain,Mild,Normal,Weak)        = True
playTennis (Sunny,Mild,Normal,Strong)     = True
playTennis (Overcast,Mild,High,Strong)    = True
playTennis (Overcast,Hot,Normal,Weak)     = True
playTennis (Rain,Mild,High,Strong)        = False

-- Classification poblem (Mitchell)
enjoySport :: ( Weather, Weather, Weather
             , Weather, Weather, Weather)
           -> Bool
enjoySport(Sunny,Warm,Normal,Strong,Warm,Same) = True
enjoySport(Sunny,Warm,High,Strong,Warm,Same)   = True
enjoySport(Rain,Cold,High,Strong,Warm,Change)  = False
enjoySport(Sunny,Warm,High,Strong,Cool,Change) = True

data LAge = Young | PrePresbyopic | Presbyopic
 deriving  (Eq,Ord,Typeable,Show)
data LPrescription = Myope | Hypermetrope
 deriving  (Eq,Ord,Typeable,Show)
data LAstigmatic = No | Yes
 deriving  (Eq,Ord,Typeable,Show)
data LTears = Reduced | Norml
 deriving  (Eq,Ord,Typeable,Show)
data LCLType = None | Hard | Soft
 deriving  (Eq,Ord,Typeable,Show)

-- UCI classification problem
lenses :: (LAge, LPrescription, LAstigmatic, LTears)
       -> LCLType
```

```
lenses (Young,Myope,No,Reduced)                      = None
lenses (Young,Myope,No,Norml)                        = Soft
lenses (Young,Myope,Yes,Reduced)                     = None
lenses (Young,Myope,Yes,Norml)                       = Hard
lenses (Young,Hypermetrope,No,Reduced)               = None
lenses (Young,Hypermetrope,No,Norml)                 = Soft
lenses (Young,Hypermetrope,Yes,Reduced)              = None
lenses (Young,Hypermetrope,Yes,Norml)                = Hard
lenses (PrePresbyopic,Myope,No,Reduced)              = None
lenses (PrePresbyopic,Myope,No,Norml)                = Soft
lenses (PrePresbyopic,Myope,Yes,Reduced)             = None
lenses (PrePresbyopic,Myope,Yes,Norml)               = Hard
lenses (PrePresbyopic,Hypermetrope,No,Reduced)       = None
lenses (PrePresbyopic,Hypermetrope,No,Norml)         = Soft
lenses (PrePresbyopic,Hypermetrope,Yes,Reduced)      = None
lenses (PrePresbyopic,Hypermetrope,Yes,Norml)        = None
lenses (Presbyopic,Myope,No,Reduced)                 = None
lenses (Presbyopic,Myope,No,Norml)                   = None
lenses (Presbyopic,Myope,Yes,Reduced)                = None
lenses (Presbyopic,Myope,Yes,Norml)                  = Hard
lenses (Presbyopic,Hypermetrope,No,Reduced)          = None
lenses (Presbyopic,Hypermetrope,No,Norml)            = Soft
lenses (Presbyopic,Hypermetrope,Yes,Reduced)         = None
lenses (Presbyopic,Hypermetrope,Yes,Norml)           = None
```

E. IgorII$_H$ Manual

Listing E.1: IGOR II$_H$ manual

```
  .___                 ._ ._
  |   | ____  ____ ._____| || |  ._
  |   |/ __ \/  _ \|  __ \ || | _| |__
  |   / /_/  ) <_> )  | \/ || |/_   _/
  |___\___  /\____/|__|  |_||_|  |_|
     /_____/                v0.8.0

Welcome to IgorII.
Type :h to get help.
IgorII > :v

Interactive commands (may be abbreviated):
==========================================
:quit
      Quit program.
:help
      Show this help.
:verboseHelp
      Show verbose help.
:load <path/to/file>
      Load a spec file into context.
:reset
      Reset the current context.
:batch <path/to/file>
      Run a batch file
:yell "something"
      Yell on command line.
:set <option>
      Set options.
:info
      Show current settings.
:generalise <tgts> [with <bgks>]
      Start generalisation.
:test [<i>] <tgts> [with <bgks>] on "expr"
       Test a generalised program.

Example Igor2 session:
==========================================

A typical Igor2 session would be as follows:

          IgorII > :load expl/Examples.hs
```

```
          File loaded in 1.0201s
          IgorII > :s +enhanced; :s +simplify
          IgorII > :g lasT

          - - - - START SYNTHESIS WITH - - - -

          Targets 'lasT'
          Background <none>
          Simplified True
          Greedy rule-splitting False
          Enhanced True
          Use paramorphisms False
          Compare rec args AWise
          DumpLog False
          Debug False
          Maximal tiers 0
          Maximal loops -1

          - - - - - - - FINISHED - - - - - - -

          lasT in 2 loops
          CPU: 0.0080s

          HYPOTHESIS 1 of 1

          lasT [a0] = a0
          lasT (_ : (a1 : a2)) = lasT (a1 : a2)

          IgorII > :quit
          Bye.

Explanation of Igor2's commands:
==========================================

Generally commands can be abbreviated with the
first character of the long command preceeded by
a colon, e.g. ':h' for ':help'. Multiple commands
can be entered either one at a time, or together
separated by a semicolon. Passing commands to Igor2
in batch mode or at command-line is similar. Except
that you need to escape string quotes at command
line with '\"'. For an example see the batch file
 'expl/batch.txt'

:quit
   Quit the program.

:help
   A short help text.

:verboseHelp
   This longer helpt text.
```

```
:load <path/to/file>
   Load a specification file into Igor2's context.
   The file is required to be a correct Haskell
   module and therefore has to type check. The I/O
   for a target function must be non-recursive, of
   course. All Prelude data types ([a], Bool,
   Maybe a, Either a b) can be used and further data
   types can be defined in this module. Imports are
   not supported, yet. Note that the path is not
   expanded, so avoid '~' for your home directory.
   Relativ paths do work, however use absolute paths
   to be on the safe side.

:reset
   Resets the current context, and sets all options
   to their default values.

:batch <path/to/file>
   Run all commands in the batch file until EOF, or
   command is not exectuable, or ':quit'. Note that
   the path is not expanded, so avoid '~'.

:yell "something"
   Print "something" one the prompt.

:set <option>
   Set an option, where <option> is one of the
   following.

     +debug              enable debug mode
     -debug              disable debug mode [default]
     +verbose            chatty ouput
     -verbose            normal output [default]
     +typeCheck          force typecheck of specifi-
                         cation [default]
     -tyepcheck          accepts _ANY_ specification,
                         (use at own risk)
     +greedySplt         Split rules greedily, i.e.
                         split one rule at all
                         possible pivot positions.
                         Results in one successor
                         with many patterns.
     -greedySplt         Split at only one pivot
                         position per rule, but make
                         one split for each pivot
                         position, results in
                         multiple successor
                         hypotheses with different
                         patterns. [default]
     +enhanced           Enhanced mode, using type
                         morphisms as program schemes
```

```
-enhanced           normal mode, no morphisms
                    [default]
+para               if in enhanced mode, use
                    paramorphisms instead of
                    catamorphisms
-para               use catamorphisms [default]
+dumpLog            write a log file
-dumpLog            do not write a log file
                    [default]
+simplify           simplify the result by
                    replacing constant function
                    calls
-simplify           show Igor2's original
                    solution [default]
colWidth=N          set the width of your
                    display to N columns
                    [default 80]
maxLoops=N          stop the synthesis cycle
                    after N loops  with a
                    (probably) partial result
                    [default (-1)]
maxTiers=N          A tiers is a set of
                    hypotheses with the same
                    heuristic value. This option
                    determines, how many tiers
                    are finished at most. N=0
                    just stops when one solution
                    has been found. For N>0 all
                    unfinished hypotheses which
                    are as good as the current
                    best are tried to finish if
                    they do not deteriorate.
                    Thus, the N-best hypotheses
                    are returned. However they
                    may be partial. For N<0
                    search continues unitil the
                    search space is exhausted.
                    [default (0)]
dumpDir="dir"       This is the directory, where
                    the log files are dumped to
                    [default "."]
redOrder=<mode>     Sets the reduction order to
                    ensure termination of the
                    final program, arguments of
                    calling/called functions
                    have to be compared.
                    <mode> states how this is
                    done. Use "Linear" to
                    compare the total number of
                    constructor symbols in the
                    arguments. To compare the
                    number of constructor symbols
```

```
                          argumentwise use "AWise"
                          [default]. Given two
                          left-hand sides a = (a1 a2
                          ..) and b = (b1 b2 ..), then
                           a < b if a1 < b1 or a1 ==
                          b1 and a2 < b2 or a1 == b1
                          and a2 == b2 ... . This is
                          necessary, if the size of
                          arguments does not change
                          over all I/O examples, e.g.
                          if accumulator variables are
                          involved.
:info
   Shows the current context with the I/O examples of
   the functions in scope, if 'verbose' is on, or the
   number of the examples, otherwise. Furthermore,
   the values of all options and flags are displayed
   as well as specification details from the current
   context.

:generalise <tgts> [with <bgks>]
   Start Igor2 with one or more target functions to
   generalise and optional functions as background
   knowledge. <tgts> and <bgks> are list of names in
   scope separated by commas.

:test [<i>] <tgts> [with <bgks>] on "cmd"
   Test a previously generalised program. If there
   were multiple  hypotheses, <i> is the one you want
   to test. If it is ommitted, all are tested. Please
   note, that you have to exactly repeat the names of
   target functions and backgroud knowledge of the
   run you want to test. If the background knowledge
   functions have already been synthesised, their
   results are taken from the history, other wise the
   I/O examples of them are used. Note, that this may
   cause the interpreter to end with "Non-exhaustive
   patterns"-errors. If the  background functions had
   multiple results, all combinations are tested.
```

F. All Results and Synthesised Programs

F.1. Programs for `ack`

Igor $\mathrm{II_H}$

```
ack (Z) (Z)         = S Z
ack (Z) (S a0)      = S (S a0)
ack (S a0) (Z)      = ack a0 (S Z)
ack (S a0) (S a1) = ack a0 (ack (S a0) a1)
```

Igor II^+ with catamorphism

```
no result
```

Igor II^+ with paramorphism

```
ack a0 a1                 = para ⊥ (fun1 a0 ⊕ fun2 a0) a1
fun1 a0 a1                = para ⊥ (fun3 a1 ⊕ fun4 a1) a0
fun2 a0 (S _, a2)         = ack a0 (S a2)
fun3 _ _                  = S Z
fun4 _ (S (Z), Z)         = S (S Z)
fun4 _ (S (S _), S a1) = ack a1 (S (S a1))
```

MagicHaskeller with paramorphism

```
λa b → nat_para a (λc d e → nat_para e (λf g → d (S g)) (S Z)) (λc →
    c) (S b)
```

MagicHaskeller with catamorphism

```
λa b → nat_cata a (λc d → nat_cata d (λe → c (S e)) (S Z)) (λc → c) (
    S b)
```

F.2. Programs for `add`

Igor II_{H}

```
add a0 (Z)          = a0
add (Z) a0          = a0
add (S a0) (S a1) = S (add (S a0) a1)

-- alternative solution
add a0 (Z)          = a0
add (Z) a0          = a0
add (S a0) (S a1) = S (add a0 (S a1))
```

Igor II^{+} with catamorphism

```
add a0 a1           = cata ⊥ (fun1 a0 ⊕ fun2 a0) a1
fun1 a0 _           = a0
fun2 (S _) (S a1) = S (S a1)
```

Igor II^{+} with paramorphism

```
add a0 a1               = para ⊥ (fun1 a0 ⊕ fun2 a0) a1
fun1 a0 _               = a0
fun2 (S _) (S a1, _) = S (S a1)
```

MagicHaskeller with paramorphism

```
λa b → nat_para b (λc d e → S (d e)) (λc → c) a
```

MagicHaskeller with catamorphism

```
λa b → nat_cata b (λc d → S (c d)) (λc → c) a
```

F.3. Programs for `addN`

Igor $\mathrm{II_H}$

```
addN _ []              = []
addN a0 (a1 : a2)      = fun1 a0 (a1 : a2) : addN a0 a2
fun1 a0 ((Z) : _)      = a0
fun1 a0 ((S a1) : _) = S (fun1 a0 [a1])
```

Igor II^+ with catamorphism

```
addN a0 a1 = map (fun1 a0) a1
fun1 a0 a1 = cata ⊥ (fun2 a0 ⊕ fun3 a0) a1
fun2 a0 _  = a0
fun3 _ a1  = S a1
```

Igor II^+ with paramorphism

```
addN a0 a1                = para ⊥ (fun1 a0 ⊕ fun2 a0) a1
fun1 _ _                  = []
fun2 a0 (a1, (a2, a3)) = fun3 a0 (a1, (a2, a3)) : a2
fun3 a0 (Z, (_, _))       = a0
fun3 a0 (S a1, (_, _)) = S (fun3 a0 (a1, ([], [])))
```

MagicHaskeller with paramorphism

```
λa b → nat_para a (λc d → list_para d [] (λe f g → S e : g)) b
```

MagicHaskeller with catamorphism

```
λa b → nat_cata a (λc → foldr (λd e → S d : e) [] c) b
```

F.4. Programs for `alleven`

Igor $\mathrm{II_H}$

```
alleven []                    = True
alleven ((Z) : a0)            = alleven a0
alleven ((S (Z)) : _)         = False
alleven ((S (S a0)) : a1) = alleven (a0 : a1)
```

Igor II^+ with catamorphism

```
alleven a0            = foldr fun1 True a0
fun1 a0 a1            = cata ⊥ (fun2 a1 ⊕ fun3 a1) a0
fun2 a0 _             = a0
fun3 a0 (False)       = a0
fun3 (True) (True) = False
```

Igor II^+ with paramorphism

```
alleven a0                   = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                       = True
fun2 (Z, (True, []))         = True
fun2 (Z, (True, [_]))        = True
fun2 (S a0, (True, []))      = alleven [S a0]
fun2 (Z, (False, [S _]))     = False
fun2 (S a0, (True, [_]))     = alleven [S a0]
fun2 (S _, (False, [S _])) = False

-- alternative solution
alleven a0                   = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                       = True
fun2 (Z, (True, []))         = True
fun2 (Z, (True, [_]))        = True
fun2 (S a0, (True, []))      = alleven [S a0]
fun2 (Z, (False, [S _]))     = False
fun2 (S a0, (True, [_]))     = fun2 (S a0, (True, []))
fun2 (S _, (False, [S _])) = False
```

MagicHaskeller with paramorphism

```
λa → list_para a True (λb c d → nat_para b (λe f g h → f h g) (λe f →
    f) False d)
```

MagicHaskeller with catamorphism

```
no result
```

F.5. Programs for `allodd`

Igor $\mathrm{II_H}$

```
no result
```

Igor II^+ with catamorphism

```
allodd a0            = foldr fun1 True a0
fun1 a0 a1           = cata ⊥ (fun2 a1 ⊕ fun3 a1) a0
fun2 _ _             = False
fun3 a0 (False)      = a0
fun3 (True) (True)   = False
```

Igor II^+ with paramorphism

```
allodd a0                       = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                          = True
fun2 (Z, (True, []))            = False
fun2 (Z, (False, [_]))          = False
fun2 (S a0, (True, []))         = allodd [S a0]
fun2 (Z, (True, [S _]))         = False
fun2 (S _, (False, [_]))        = False
fun2 (S a0, (True, [S _]))      = allodd [S a0]

-- alternative solution
allodd a0                       = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                          = True
fun2 (Z, (True, []))            = False
fun2 (Z, (False, [_]))          = False
fun2 (S a0, (True, []))         = allodd [S a0]
fun2 (Z, (True, [S _]))         = False
fun2 (S _, (False, [_]))        = False
fun2 (S a0, (True, [S _]))      = fun2 (S a0, (True, []))
```

MagicHaskeller with paramorphism

```
λa → list_para a True (λb c d → nat_para b (λe f g h → f h g) (λe f →
     f) d False)
```

MagicHaskeller with catamorphism

```
no result
```

F.6. Programs for `and`

Igor II_{H}

```
and (False) _ = False
and (True) a0 = a0

-- alternative solution
and _ (False) = False
and a0 (True) = a0
```

Igor II^{+} with catamorphism

```
and (False) _ = False
and (True) a0 = a0

-- alternative solution
and _ (False) = False
and a0 (True) = a0
```

Igor II^{+} with paramorphism

```
and (False) _ = False
and (True) a0 = a0

-- alternative solution
and _ (False) = False
and a0 (True) = a0
```

MagicHaskeller with paramorphism

```
λa b → iF (iF b True False) a False
```

MagicHaskeller with catamorphism

```
λa b → iF (iF b True False) a False
```

F.7. Programs for andL

Igor II_{H}

```
andL []              = True
andL ((False) : _) = False
andL ((True) : a0) = andL a0
```

Igor II^{+} with catamorphism

```
andL a0          = foldr fun1 True a0
fun1 (False) _ = False
fun1 (True) a0 = a0

-- alternative solution
andL a0          = foldr fun1 True a0
fun1 _ (False) = False
fun1 a0 (True) = a0
```

Igor II^{+} with paramorphism

```
andL a0                 = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                  = True
fun2 (False, (_, _)) = False
fun2 (True, (a0, _)) = a0

-- alternative solution
andL a0                     = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                      = True
fun2 (a0, (True, _))      = a0
fun2 (_, (False, _ : _)) = False
```

MagicHaskeller with paramorphism

```
λa → list_para a (λb → True) (λb c d e → iF b (d e) e) False
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c → iF c b False) True a
```

F.8. Programs for `append`

Igor $\mathrm{II_H}$

```
append [] a0         = a0
append (a0 : a1) a2 = a0 : append a1 a2
```

Igor II^+ with catamorphism

```
append [] a0         = a0
append (a0 : a1) a2 = a0 : append a1 a2
```

Igor II^+ with paramorphism

```
append a0 a1          = para ⊥ (fun1 a1 ⊕ fun2 a1) a0
fun1 a0 _             = a0
fun2 _ (a1, (a2, _)) = a1 : a2
```

MagicHaskeller with paramorphism

```
λa b → list_para a (λc → c) (λc d e f → c : e f) b
```

MagicHaskeller with catamorphism

```
λa b → foldr (λc d → c : d) b a
```

F.9. Programs for balloons

Igor II_{H}

```
balloons (_, Large, _, _)        = FF
balloons (Purple, Small, _, _) = FF
balloons (Yellow, Small, _, _) = TT

-- alternative solution
balloons (Purple, _, _, _)       = FF
balloons (Yellow, Large, _, _) = FF
balloons (Yellow, Small, _, _) = TT
```

Igor II^{+} with catamorphism

```
balloons (_, Large, _, _)        = FF
balloons (Purple, Small, _, _) = FF
balloons (Yellow, Small, _, _) = TT

-- alternative solution
balloons (Purple, _, _, _)       = FF
balloons (Yellow, Large, _, _) = FF
balloons (Yellow, Small, _, _) = TT
```

Igor II^{+} with paramorphism

```
balloons (_, Large, _, _)        = FF
balloons (Purple, Small, _, _) = FF
balloons (Yellow, Small, _, _) = TT

-- alternative solution
balloons (Purple, _, _, _)       = FF
balloons (Yellow, Large, _, _) = FF
balloons (Yellow, Small, _, _) = TT
x
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.10. Programs for `concat`

Igor $\mathrm{II_H}$

```
concat []             = []
concat [a0]           = a0
concat [[], a0]       = a0
concat [a0 : a1, a2] = a0 : concat [a1, a2]
```

Igor II^+ with catamorphism

```
concat a0          = foldr fun1 [] a0
fun1 [] a0         = a0
fun1 (a0 : a1) a2 = a0 : fun1 a1 a2
```

Igor II^+ with paramorphism

```
concat a0                          = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                             = []
fun2 (a0, ([], _))                 = a0
fun2 (a0, (_ : _, [a1 : a2])) = concat [a0, a1 : a2]
```

MagicHaskeller with paramorphism

```
λa → list_para a [] (λb c d → list_para b (λe → e) (λe f g h → e : g
   h) d)
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c → foldr (λd e → d : e) c b) [] a
```

F.11. Programs for `drop`

Igor II_{H}

```
no result
```

Igor II^{+} with catamorphism

```
drop a0 a1             = cata ⊥ (fun1 a1 ⊕ fun2 a1) a0
fun1 a0 _              = a0
fun2 _ []              = []
fun2 (_ : _) (_ : a3) = a3
```

Igor II^{+} with paramorphism

```
drop a0 a1                 = para ⊥ (fun1 a0 ⊕ fun2 a0) a1
fun1 _ _                   = []
fun2 (Z) (a0, (_, a1))     = a0 : a1
fun2 (S a0) (_, (_, a3)) = drop a0 a3
```

MagicHaskeller with paramorphism

```
λa b → nat_para a (λc d → list_para d [] (λe f g → f)) b
```

MagicHaskeller with catamorphism

```
no result
```

F.12. Programs for `enjoySport`

Igor II_H

```
enjoySport (Rain, Cold, High, Strong, Warm, Change) = False
enjoySport (Sunny, Warm, _, Strong, _, _)          = True

-- alternative solution
enjoySport (Rain, Cold, High, Strong, Warm, Change) = False
enjoySport (Sunny, Warm, _, Strong, _, _)          = True
```

Igor II^+ with catamorphism

```
enjoySport (Rain, Cold, High, Strong, Warm, Change) = False
enjoySport (Sunny, Warm, _, Strong, _, _)          = True

-- alternative solution
enjoySport (Rain, Cold, High, Strong, Warm, Change) = False
enjoySport (Sunny, Warm, _, Strong, _, _)          = True
```

Igor II^+ with paramorphism

```
enjoySport (Rain, Cold, High, Strong, Warm, Change) = False
enjoySport (Sunny, Warm, _, Strong, _, _)          = True

-- alternative solution
enjoySport (Rain, Cold, High, Strong, Warm, Change) = False
enjoySport (Sunny, Warm, _, Strong, _, _)          = True
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.13. Programs for eq

Igor $\mathrm{II_H}$

```
eq (Z) (Z)           = True
eq (Z) (S _)         = False
eq (S a0) a1         = eq a0 (fun72 (S a0) a1)
fun72 (S a0) (Z)     = S a0
fun72 (S _) (S a1)   = a1

-- alternative solution
eq (Z) (Z)           = True
eq (Z) (S _)         = False
eq (S a0) a1         = eq a0 (fun312 (S a0) a1)
fun312 (S _) (Z)     = S (S Z)
fun312 (S _) (S a1) = a1
```

Igor II^+ with catamorphism

```
no result
```

Igor II^+ with paramorphism

```
eq a0 a1             = para ⊥ (fun1 a0 ⊕ fun2 a0) a1
fun1 a0 a1           = para ⊥ (fun3 a1 ⊕ fun4 a1) a0
fun2 (Z) (_, _)      = False
fun2 (S a0) (_, a2) = eq a0 a2
fun3 _ _             = True
fun4 _ (_, _)        = False

-- alternative solution
eq a0 a1             = para ⊥ (fun1 a0 ⊕ fun2 a0) a1
fun1 a0 a1           = para ⊥ (fun3 a1 ⊕ fun4 a1) a0
fun2 (Z) (_, _)      = False
fun2 (S a0) (_, a2) = eq a2 a0
fun3 _ _             = True
fun4 _ (_, _)        = False
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.14. Programs for `even`

Igor $\mathrm{II_H}$

```
even (Z)          = True
even (S (Z))      = False
even (S (S a0)) = even a0
```

Igor II^+ with catamorphism

```
even a0       = cata ⊥ (fun1 ⊕ fun2) a0
fun1 _        = True
fun2 (False) = True
fun2 (True)  = False
```

Igor II^+ with paramorphism

```
even a0            = para ⊥ (fun1 ⊕ fun2) a0
fun1 _             = True
fun2 (True, _)     = False
fun2 (False, S _) = True
```

MagicHaskeller with paramorphism

```
λa → nat_para a (λb c → iF c False True) True
```

MagicHaskeller with catamorphism

```
no result
```

F.15. Programs for `even, odd`

Igor II_{H}

```
even (Z)     = True
even (S a0) = odd a0
odd (Z)      = False
odd (S a0)  = even a0
```

Igor II^{+} with catamorphism

```
even a0       = cata ⊥ (fun2 ⊕ fun3) a0
odd a0        = cata ⊥ (fun4 ⊕ fun5) a0
fun2 _        = True
fun3 (False) = True
fun3 (True)  = False
fun4 _        = False
fun5 (False) = True
fun5 (True)  = False
```

Igor II^{+} with paramorphism

```
even a0       = para ⊥ (fun2 ⊕ fun3) a0
odd a0        = para ⊥ (fun4 ⊕ fun5) a0
fun2 _        = True
fun3 (_, a1) = odd a1
fun4 _        = False
fun5 (_, a1) = even a1
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.16. Programs for evenLength

Igor II_{H}

```
evenLength []              = True
evenLength [_]             = False
evenLength (_ : (_ : a2)) = evenLength a2
```

Igor II^{+} with catamorphism

```
evenLength a0  = foldr fun1 True a0
fun1 _ (False) = True
fun1 _ (True)  = False
```

Igor II^{+} with paramorphism

```
evenLength a0            = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                   = True
fun2 (_, (True, _))      = False
fun2 (_, (False, _ : _)) = True
```

MagicHaskeller with paramorphism

```
λa → list_para a True (λb c d → iF d False True)
```

MagicHaskeller with catamorphism

```
no result
```

F.17. Programs for `evenParity`

Igor $\mathrm{II_H}$

```
no result
```

Igor II^+ with catamorphism

```
evenParity a0         = foldr fun1 True a0
fun1 (False) a0       = a0
fun1 (True) (False) = True
fun1 (True) (True)  = False

-- alternative solution
evenParity a0         = foldr fun1 True a0
fun1 a0 (False)       = a0
fun1 (False) (True) = True
fun1 (True) (True)  = False
```

Igor II^+ with paramorphism

```
evenParity a0           = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                  = True
fun2 (False, (a0, _)) = a0
fun2 (True, (a0, _))  = evenParity [a0]

-- alternative solution
evenParity a0           = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                  = True
fun2 (False, (a0, _)) = a0
fun2 (True, (_, a1))  = evenParity (True : a1)
```

MagicHaskeller with paramorphism

```
λa → list_para a (λb → True) (λb c d e → iF (d e) (iF b False e) b)
   True
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c → iF c (iF b False True) b) True a
```

F.18. Programs for evenpos

Igor $\mathrm{II_H}$

```
evenpos []                = []
evenpos [_]               = []
evenpos (_ : (a1 : a2)) = a1 : evenpos a2
```

Igor II^+ with catamorphism

```
evenpos []                = []
evenpos [_]               = []
evenpos (_ : (a1 : a2)) = a1 : evenpos a2
```

Igor II^+ with paramorphism

```
evenpos a0                           = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                               = []
fun2 (_, ([], a1))                   = a1
fun2 (_, (_ : _, a3 : (_ : a4))) = evenpos (a3 : (a3 : (a3 : a4)))
```

MagicHaskeller with paramorphism

```
λa → list_para a (λb → []) (λb c d e → iF e (d False) (b : d True))
    True
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c d → foldr (λe f → b : c []) (c a) d) (λb → []) a []
```

F.19. Programs for `evens`

Igor II_{H}

```
evens []                          = []
evens ((Z) : a0)                  = Z : evens a0
evens ((S (Z)) : a0)              = evens a0
evens ((S (S (Z))) : a0)          = S (S Z) : evens a0
evens ((S (S (S (Z)))) : a0)      = evens a0

-- alternative solution
evens []                          = []
evens ((Z) : a0)                  = Z : evens a0
evens ((S (Z)) : a0)              = evens a0
evens ((S (S (Z))) : a0)          = S (S Z) : evens a0
evens ((S (S (S (Z)))) : a0)      = evens a0
```

Igor II^{+} with catamorphism

```
evens a0     = filter fun1 a0
fun1 a0      = cata ⊥ (fun2 ⊕ fun3) a0
fun2 _       = False
fun3 (False) = True
fun3 (True)  = False
```

Igor II^{+} with paramorphism

```
evens a0                            = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                              = []
fun2 (Z, ([], []))                  = [Z]
fun2 (S (Z), ([], []))              = []
fun2 (Z, ([], [S _]))               = [Z]
fun2 (Z, ([_], [a0]))               = [Z, a0]
fun2 (S (S (Z)), ([], []))          = [S (S Z)]
fun2 (S a0, ([], [S _]))            = fun2 (S a0, ([], []))
fun2 (S a0, ([_], [a1]))            = evens [S a0, a1]
fun2 (S (S (S (Z))), ([], []))      = []
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.20. Programs for `fact` with `mult`

Igor $\mathrm{II_H}$

```
fact (Z)                   = S Z
fact (S (Z))               = S Z
fact (S (S (Z)))           = S (S Z)
fact (S (S (S (Z))))       = S (S (S (S (S (S Z)))))
fact (S (S (S (S (Z)))))   =
    S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S
     (S (S (S (S (S (S (S (S Z)))))))))))))))))))))))
fact (S (S (S (S (S (Z)))))) =
    S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S
     (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S
      (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S
       (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S
        (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S
         (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S
          (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S
           (S (S (S (S (S (S (S (S Z))))))))))))))))))))))
            )))))))))))))))))))))))))))))))))))))))))))))))
             )))))))))))))))))))))))))))))))))))))))))))))))
              )))
```

Igor II^+ with catamorphism

```
fact (Z)                   = S Z
fact (S (Z))               = S Z
fact (S (S (Z)))           = S (S Z)
fact (S (S (S (Z))))       = S (S (S (S (S (S Z)))))
fact (S (S (S (S (Z)))))   =
    S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S
     (S (S (S (S (S (S (S (S Z)))))))))))))))))))))))
fact (S (S (S (S (S (Z)))))) =
    S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S
     (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S
      (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S
       (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S
        (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S
         (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S
          (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S
           (S (S (S (S (S (S (S (S Z))))))))))))))))))))))
            )))))))))))))))))))))))))))))))))))))))))))))))
             )))))))))))))))))))))))))))))))))))))))))))))))
              )))
```

Igor II^+ with paramorphism

```
fact a0         = para ⊥ (fun2 ⊕ fun3) a0
fun2 _          = S Z
fun3 (S _, a1) = fact (S a1)
```

MagicHaskeller with paramorphism

```
λa → nat_para a (λb c → natmlt c b) (S Z)
```

MagicHaskeller with catamorphism

```
λa → nat_cata a (λb c → natmlt c (b (S c))) (λb → S Z) (S Z)
```

F.21. Programs for `fib` with `add`

Igor $\mathrm{II_H}$

```
fib (Z)            = Z
fib (S a0)         = add (fun2 (S a0)) (fun3 (S a0))
fun2 (S (Z))       = Z
fun2 (S (S a0))    = fib a0
fun3 (S (Z))       = S Z
fun3 (S (S a0))    = fib (S a0)
```

Igor $\mathrm{II^+}$ with catamorphism

```
fib (Z)            = Z
fib (S a0)         = add (fun2 (S a0)) (fun3 (S a0))
fun2 (S (Z))       = Z
fun2 (S (S a0))    = fib a0
fun3 (S (Z))       = S Z
fun3 (S (S a0))    = fib (S a0)
```

Igor $\mathrm{II^+}$ with paramorphism

```
fib a0             = para ⊥ (fun2 ⊕ fun3) a0
fun2 _             = Z
fun3 (Z, Z)        = S Z
fun3 (S _, S a1)   = fib (S (S a1))
```

MagicHaskeller with paramorphism

```
λa → nat_para a (λb c d e → c e (natadd e d)) (λb c → b) Z (S Z)
```

MagicHaskeller with catamorphism

```
λa → nat_cata a (λb c d → b d (natadd d c)) (λb c → b) Z (S Z)
```

F.22. Programs for gaussSum with add

Igor II_{H}

```
gaussSum (Z)     = Z
gaussSum (S a0) = add (S a0) (fun3 (S a0))
fun3 (S (Z))     = Z
fun3 (S (S a0)) = add (S a0) (fun3 (S a0))
```

Igor II^{+} with catamorphism

```
gaussSum a0            = cata ⊥ (fun2 ⊕ fun3) a0
fun17 (S (Z))          = Z
fun17 (S (S (S a0))) = fun4 a0
fun2 _                 = Z
fun3 a0                = add (fun4 a0) a0
fun4 a0                = S (fun7 a0)
fun7 (Z)               = Z
fun7 (S a0)            = fun4 (fun17 (S a0))
```

Igor II^{+} with paramorphism

```
gaussSum a0    = para ⊥ (fun2 ⊕ fun3) a0
fun2 _         = Z
fun3 (a0, a1) = add (S a1) a0
```

MagicHaskeller with paramorphism

```
λa → nat_para a (λb c → natadd c b) Z
```

MagicHaskeller with catamorphism

```
λa → nat_para a (λb c → natadd c b) Z
```

F.23. Programs for geq

Igor II_{H}

```
geq _ (Z)          = True
geq (Z) (S _)      = False
geq (S a0) (S a1) = geq a0 a1
```

Igor II^{+} with catamorphism

```
no result
```

Igor II^{+} with paramorphism

```
geq a0 a1                = para ⊥ (fun1 a0 ⊕ fun2 a0) a1
fun1 _ _                 = True
fun2 (Z) (_, _)          = False
fun2 (S a0) (True, a1) = geq a0 a1
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.24. Programs for `halves`

Igor $\mathrm{II_H}$

```
halves []                    = ([], [])
halves [a0]                  = ([a0], [])
halves [a0, a1]              = ([a0], [a1])
halves [a0, a1, a2]          = ([a0, a1], [a2])
halves [a0, a1, a2, a3]      = ([a0, a1], [a2, a3])
halves (a0 : (a1 : (a2 :
        (a3 : (a4 : a5))))) = ([a0, a1, a2], a3 : (a4 : a5))
```

Igor II^+ with catamorphism

```
no result
```

Igor II^+ with paramorphism

```
halves a0                          = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                             = ([], [])
fun2 (a0, (([], []), []))          = ([a0], [])
fun2 (a0, ((_ : _, _), a1 : a4)) = halves (a0 : (a1 : a4))
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.25. Programs for `hanoi`

Igor II_{H}

```
hanoi (D0) a0 _ a2 a3    = MV D0 a0 a2 a3
hanoi (D a0) a1 a2 a3 a4 =
     hanoi a0 a2 a1 a3
      (MV (D a0) a1 a3
       (hanoi a0 a1 a3 a2 a4))
```

Igor II^{+} with catamorphism

```
no result
```

Igor II^{+} with paramorphism

```
no result
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.26. Programs for `incr`

Igor II_{H}

```
incr []         = []
incr (a0 : a1) = S a0 : incr a1
```

Igor II^{+} with catamorphism

```
incr a0 = map fun1 a0
fun1 a0 = S a0
```

Igor II^{+} with paramorphism

```
incr a0              = para ⊥ (fun1 ⊕ fun2) a0
fun1 _               = []
fun2 (a0, (a1, _)) = S a0 : a1
```

MagicHaskeller with paramorphism

```
λa → list_para a [] (λb c d → S b : d)
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c → S b : c) [] a
```

F.27. Programs for `init`

Igor $\mathrm{II_H}$

```
init [_]                = []
init (a0 : (a1 : a2)) = a0 : init (a1 : a2)
```

Igor II^+ with catamorphism

```
init [_]                = []
init (a0 : (a1 : a2)) = a0 : init (a1 : a2)
```

Igor II^+ with paramorphism

```
init [_]                = []
init (a0 : (a1 : a2)) = a0 : init (a1 : a2)
```

MagicHaskeller with paramorphism

```
λa → list_para a [] (λb c d → list_para c (λe → []) (λe f g h → h : g
    e) b)
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c d → foldr (λe f → b : c d) d (c a)) (λb → []) a []
```

F.28. Programs for `init, last`

Igor II_{H}

```
init [_]               = []
init (a0 : (a1 : a2)) = a0 : init (a1 : a2)
last [a0]              = a0
last (_ : (a1 : a2))  = last (a1 : a2)
```

Igor II^{+} with catamorphism

```
init [_]               = []
init (a0 : (a1 : a2)) = a0 : init (a1 : a2)
last [a0]              = a0
last (_ : (a1 : a2))  = last (a1 : a2)
```

Igor II^{+} with paramorphism

```
init [_]               = []
init (a0 : (a1 : a2)) = a0 : init (a1 : a2)
last [a0]              = a0
last (_ : (a1 : a2))  = last (a1 : a2)
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.29. Programs for `inits`

Igor II_{H}

```
inits []                = [[]]
inits [a0]              = [[], [a0]]
inits [a0, a1]          = [[], [a0], [a0, a1]]
inits [a0, a1, a2]      = [[], [a0], [a0, a1], [a0, a1, a2]]
inits [a0, a1, a2, a3] =
     [[], [a0], [a0, a1], [a0, a1, a2], [a0, a1, a2, a3]]
```

Igor II^{+} with catamorphism

```
inits a0           = foldr fun1 [[]] a0
fun1 a0 ([] : a1) = [] : fun2 a0 ([] : a1)
fun2 a0 ([] : a1) = map (fun3 a0) ([] : a1)
fun3 a0 a1         = a0 : a1
```

Igor II^{+} with paramorphism

```
inits a0                        = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                          = [[]]
fun2 (a0, ([[]], []))           = [[], [a0]]
fun2 (a0, ( [] : ([_] : _)
          , a1 : a3))           = inits (a0 : (a1 : a3))
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.30. Programs for `inorder` with `append`

Igor $\mathrm{II_H}$

```
inorder (NilT)                                  = []
inorder (Node a0 a1 a2)                         =
   append (fun6 (Node a0 a1 a2)) (fun7 (Node a0 a1 a2))
fun6 (Node a0 (NilT) (NilT))                    = [a0]
fun6 (Node _ (Node a1 a2 a3) (Node _ _ _))      =
   inorder (Node a1 a2 a3)
fun7 (Node _ (NilT) (NilT))                     = []
fun7 (Node a0 (Node _ _ _) (Node a4 a5 a6))     =
   a0 : inorder (Node a4 a5 a6)
```

Igor II^+ with catamorphism

```
inorder a0                     = cata ⊥ (fun2 ⊕ fun3) a0
fun2 _                         = []
fun3 (a0, ([], []))            = [a0]
fun3 (a0, (a1 : a2, a3 : a4))  = append (a1 : a2) (a0 : (a3 : a4))
```

Igor II^+ with paramorphism

```
inorder a0                              =
   para ⊥ (fun2 ⊕ fun3) a0
fun2 _                                  = []
fun3 (a0, (([], NilT), ([], NilT)))     = [a0]
fun3 (a0, ( (_ : _, Node a3 a4 a5)
          , (_ : _, Node a8 a9 a10))) =
   inorder (Node a0 (Node a3 a4 a5) (Node a8 a9 a10))
```

MagicHaskeller with paramorphism

```
λa → ntree_para a [] (λb c d e f → e ++ (b : f))
```

MagicHaskeller with catamorphism

```
λa → ntree_cata a [] (λb c d → c ++ (b : d))
```

F.31. Programs for `intersperse`

Igor $\mathrm{II_H}$

```
intersperse _ []           = []
intersperse a0 (a1 : a2) = a1 : fun1 a0 (a1 : a2)
fun1 _ [_]                 = []
fun1 a0 (_ : (a2 : a3))  = a0 : intersperse a0 (a2 : a3)
```

Igor $\mathrm{II^+}$ with catamorphism

```
intersperse a0 a1     = foldr (fun1 a0) [] a1
fun1 _ a1 []          = [a1]
fun1 a0 a1 (a2 : a3) = a1 : (a0 : (a2 : a3))
```

Igor $\mathrm{II^+}$ with paramorphism

```
intersperse a0 a1                 = para ⊥ (fun1 a0 ⊕ fun2 a0) a1
fun1 _ _                          = []
fun2 _ (a1, ([], []))             = [a1]
fun2 a0 (a1, (_ : a3, a2 : _)) = a1 : (a0 : (a2 : a3))

-- alternative solution
intersperse a0 a1                 = para ⊥ (fun1 a0 ⊕ fun2 a0) a1
fun1 _ _                          = []
fun2 _ (a1, ([], []))             = [a1]
fun2 a0 (a1, (_ : a3, a2 : _)) = a1 : (a0 : (a2 : a3))
```

MagicHaskeller with paramorphism

```
λa b → list_para (list_para b [] (λc d e → a : (c : e))) b (λc d e → d
    )
```

MagicHaskeller with catamorphism

```
λa b → foldr (λc d → c : foldr (λe f → a : d) d d) [] b
```

F.32. Programs for `last`

Igor $\mathrm{II_H}$

```
last [a0]              = a0
last (_ : (a1 : a2)) = last (a1 : a2)
```

Igor II^+ with catamorphism

```
last [a0]              = a0
last (_ : (a1 : a2)) = last (a1 : a2)
```

Igor II^+ with paramorphism

```
last [a0]              = a0
last (_ : (a1 : a2)) = last (a1 : a2)
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.33. Programs for `lastM`

Igor $\mathrm{II_H}$

```
lastM []                = Nothing
lastM [a0]              = Just a0
lastM (_ : (a1 : a2)) = lastM (a1 : a2)
```

Igor II^+ with catamorphism

```
lastM a0                = foldr fun1 Nothing a0
fun1 a0 (Nothing)       = Just a0
fun1 _ (Just a1)        = Just a1
```

Igor II^+ with paramorphism

```
lastM a0                        = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                          = Nothing
fun2 (a0, (Nothing, []))        = Just a0
fun2 (_, (Just a1, _ : _)) = Just a1

-- alternative solution
lastM a0                        = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                          = Nothing
fun2 (a0, (Nothing, []))        = Just a0
fun2 (_, (Just a1, _ : _)) = Just a1
```

MagicHaskeller with paramorphism

```
λa → list_para a Nothing (λb c d → list_para c (Just b) (λe f g → d))
```

MagicHaskeller with catamorphism

```
no result
```

F.34. Programs for `lasts`

Igor II_H

```
lasts []                       = []
lasts ([a0] : a1)              = a0 : lasts a1
lasts ((_ : (a1 : a2)) : a3) = lasts ((a1 : a2) : a3)
```

Igor II^+ with catamorphism

```
lasts a0               = map fun1 a0
fun1 [a0]              = a0
fun1 (_ : (a1 : a2)) = fun1 (a1 : a2)
```

Igor II^+ with paramorphism

```
lasts a0                          = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                            = []
fun2 ([a0], (a1, _))              = a0 : a1
fun2 (_ : (a1 : a2), (_, a4)) = lasts ((a1 : a2) : a4)
```

MagicHaskeller with paramorphism

```
λa → list_para a [] (λb c d → list_para b (λe f → f) (λe f g h i → g
    h (e : h)) d d)
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c → foldr (λd e → foldr (λf g → e) (d : c) e) [] b) []
     a
```

F.35. Programs for `length`

Igor II_H

```
length []        = Z
length (_ : a1) = S (length a1)
```

Igor II^+ with catamorphism

```
length a0 = foldr fun1 Z a0
fun1 _ a1 = S a1
```

Igor II^+ with paramorphism

```
length a0           = para ⊥ (fun1 ⊕ fun2) a0
fun1 _              = Z
fun2 (_, (a1, _)) = S a1
```

MagicHaskeller with paramorphism

```
λa → list_para a Z (λb c d → S d)
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c → S c) Z a
```

F.36. Programs for `lengths`

Igor II_{H}

```
lengths []                  = []
lengths (a0 : a1)           = fun1 (a0 : a1) : fun2 (a0 : a1)
fun1 ([] : _)               = Z
fun1 [_ : a1]               = S (fun1 [a1])
fun1 ((_ : a1) : (_ : a3)) = S (fun1 (a1 : a3))
fun2 [_]                    = []
fun2 [[], a0]               = [fun741 [[], a0]]
fun2 ((_ : _) : (a2 : a3)) = lengths (a2 : a3)
fun741 [[], []]             = Z
fun741 [[], _ : a1]         = S (fun741 [[], a1])
```

Igor II^{+} with catamorphism

```
lengths a0 = map fun1 a0
fun1 a0    = foldr fun2 Z a0
fun2 _ a1  = S a1
```

Igor II^{+} with paramorphism

```
lengths a0             = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                 = []
fun2 (a0, (a1, a2))    = fun3 (a0, (a1, a2)) : a1
fun3 ([], (_, _))      = Z
fun3 (_ : a1, (_, _)) = S (fun3 (a1, ([], [])))
```

MagicHaskeller with paramorphism

```
λa → list_para a [] (λb c d → list_para b Z (λe f g → S g) : d)
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c → foldr (λd e → S e) Z b : c) [] a
```

F.37. Programs for `lenses`

Igor II_{H}

```
lenses (_, _, _, Reduced)                        = None
lenses (_, Hypermetrope, No, Norml)              = Soft
lenses (_, Myope, Yes, Norml)                    = Hard
lenses (PrePresbyopic, Hypermetrope, Yes, Norml) = None
lenses (PrePresbyopic, Myope, No, Norml)         = Soft
lenses (Presbyopic, Hypermetrope, Yes, Norml)    = None
lenses (Presbyopic, Myope, No, Norml)            = None
lenses (Young, Hypermetrope, Yes, Norml)         = Hard
lenses (Young, Myope, No, Norml)                 = Soft

-- alternative solution
lenses (_, _, _, Reduced)                        = None
lenses (_, Hypermetrope, No, Norml)              = Soft
lenses (_, Myope, Yes, Norml)                    = Hard
lenses (PrePresbyopic, Hypermetrope, Yes, Norml) = None
lenses (PrePresbyopic, Myope, No, Norml)         = Soft
lenses (Presbyopic, Hypermetrope, Yes, Norml)    = None
lenses (Presbyopic, Myope, No, Norml)            = None
lenses (Young, Hypermetrope, Yes, Norml)         = Hard
lenses (Young, Myope, No, Norml)                 = Soft

-- alternative solution
lenses (_, _, _, Reduced)                        = None
lenses (_, Myope, Yes, Norml)                    = Hard
lenses (PrePresbyopic, _, No, Norml)             = Soft
lenses (PrePresbyopic, Hypermetrope, Yes, Norml) = None
lenses (Presbyopic, Hypermetrope, No, Norml)     = Soft
lenses (Presbyopic, Hypermetrope, Yes, Norml)    = None
lenses (Presbyopic, Myope, No, Norml)            = None
lenses (Young, _, No, Norml)                     = Soft
lenses (Young, Hypermetrope, Yes, Norml)         = Hard
```

Igor II^{+} with catamorphism

```
lenses (_, _, _, Reduced)                        = None
lenses (_, Hypermetrope, No, Norml)              = Soft
lenses (_, Myope, Yes, Norml)                    = Hard
lenses (PrePresbyopic, Hypermetrope, Yes, Norml) = None
lenses (PrePresbyopic, Myope, No, Norml)         = Soft
lenses (Presbyopic, Hypermetrope, Yes, Norml)    = None
lenses (Presbyopic, Myope, No, Norml)            = None
lenses (Young, Hypermetrope, Yes, Norml)         = Hard
lenses (Young, Myope, No, Norml)                 = Soft

-- alternative solution
lenses (_, _, _, Reduced)                        = None
lenses (_, Hypermetrope, No, Norml)              = Soft
lenses (_, Myope, Yes, Norml)                    = Hard
lenses (PrePresbyopic, Hypermetrope, Yes, Norml) = None
```

```
lenses (PrePresbyopic, Myope, No, Norml)          = Soft
lenses (Presbyopic, Hypermetrope, Yes, Norml)     = None
lenses (Presbyopic, Myope, No, Norml)             = None
lenses (Young, Hypermetrope, Yes, Norml)          = Hard
lenses (Young, Myope, No, Norml)                  = Soft

-- alternative solution
lenses (_, _, _, Reduced)                         = None
lenses (_, Myope, Yes, Norml)                     = Hard
lenses (PrePresbyopic, _, No, Norml)              = Soft
lenses (PrePresbyopic, Hypermetrope, Yes, Norml) = None
lenses (Presbyopic, Hypermetrope, No, Norml)      = Soft
lenses (Presbyopic, Hypermetrope, Yes, Norml)     = None
lenses (Presbyopic, Myope, No, Norml)             = None
lenses (Young, _, No, Norml)                      = Soft
lenses (Young, Hypermetrope, Yes, Norml)          = Hard
```

Igor II^{+} with paramorphism

```
lenses (_, _, _, Reduced)                         = None
lenses (_, Hypermetrope, No, Norml)               = Soft
lenses (_, Myope, Yes, Norml)                     = Hard
lenses (PrePresbyopic, Hypermetrope, Yes, Norml) = None
lenses (PrePresbyopic, Myope, No, Norml)          = Soft
lenses (Presbyopic, Hypermetrope, Yes, Norml)     = None
lenses (Presbyopic, Myope, No, Norml)             = None
lenses (Young, Hypermetrope, Yes, Norml)          = Hard
lenses (Young, Myope, No, Norml)                  = Soft

-- alternative solution
lenses (_, _, _, Reduced)                         = None
lenses (_, Hypermetrope, No, Norml)               = Soft
lenses (_, Myope, Yes, Norml)                     = Hard
lenses (PrePresbyopic, Hypermetrope, Yes, Norml) = None
lenses (PrePresbyopic, Myope, No, Norml)          = Soft
lenses (Presbyopic, Hypermetrope, Yes, Norml)     = None
lenses (Presbyopic, Myope, No, Norml)             = None
lenses (Young, Hypermetrope, Yes, Norml)          = Hard
lenses (Young, Myope, No, Norml)                  = Soft

-- alternative solution
lenses (_, _, _, Reduced)                         = None
lenses (_, Myope, Yes, Norml)                     = Hard
lenses (PrePresbyopic, _, No, Norml)              = Soft
lenses (PrePresbyopic, Hypermetrope, Yes, Norml) = None
lenses (Presbyopic, Hypermetrope, No, Norml)      = Soft
lenses (Presbyopic, Hypermetrope, Yes, Norml)     = None
lenses (Presbyopic, Myope, No, Norml)             = None
lenses (Young, _, No, Norml)                      = Soft
lenses (Young, Hypermetrope, Yes, Norml)          = Hard
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.38. Programs for mapCons

Igor $\mathrm{II_H}$

```
mapCons _ []          = []
mapCons a0 (a1 : a2) = (a0 : a1) : fun1 a0 (a1 : a2)
fun1 _ [[]]           = []
fun1 a0 (_ : a2)      = fun1 a0 ([] : a2)

-- alternative solution
mapCons _ []          = []
mapCons a0 (a1 : a2) = (a0 : a1) : fun1 a0 (a1 : a2)
fun1 _ [[]]           = []
fun1 a0 (_ : a2)      = mapCons a0 a2
```

Igor II^+ with catamorphism

```
mapCons a0 a1 = map (fun1 a0) a1
fun1 a0 a1    = a0 : a1
```

Igor II^+ with paramorphism

```
mapCons a0 a1            = para ⊥ (fun1 a0 ⊕ fun2 a0) a1
fun1 _ _                 = []
fun2 a0 (a1, (a2, _)) = (a0 : a1) : a2
```

MagicHaskeller with paramorphism

```
λa b → list_para b [] (λc d e → (a : c) : e)
```

MagicHaskeller with catamorphism

```
λa b → foldr (λc d → (a : c) : d) [] b
```

F.39. Programs for `mapTail`

Igor II_{H}

```
mapTail []                = []
mapTail ((_ : a1) : a2) = a1 : mapTail a2
```

Igor II^{+} with catamorphism

```
mapTail a0    = map fun1 a0
fun1 (_ : a1) = a1
```

Igor II^{+} with paramorphism

```
mapTail a0               = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                   = []
fun2 (_ : a1, (a2, _)) = a1 : a2
```

MagicHaskeller with paramorphism

```
λa → list_para a [] (λb c d → list_para b d (λe f g → f : d))
```

MagicHaskeller with catamorphism

```
no result
```

F.40. Programs for `mirror`

Igor II_{H}

```
mirror (NilT)            = NilT
mirror (Node a0 a1 a2) = Node a0 (mirror a2) (mirror a1)
```

Igor II^{+} **with catamorphism**

```
mirror a0              = cata ⊥ (fun1 ⊕ fun2) a0
fun1 _                 = NilT
fun2 (a0, (a1, a2)) = Node a0 a2 a1
```

Igor II^{+} **with paramorphism**

```
mirror a0                        = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                           = NilT
fun2 (a0, ((a1, _), (a3, _))) = Node a0 a3 a1
```

MagicHaskeller with paramorphism

```
λa → ntree_para a NilT (λb c d e f → Node b f e)
```

MagicHaskeller with catamorphism

```
λa → ntree_cata a NilT (λb c d → Node b d c)
```

F.41. Programs for `mult`

Igor $\mathrm{II_H}$

```
no result
```

Igor II^+ with catamorphism

```
no result
```

Igor II^+ with paramorphism

```
mult a0 a1                = para ⊥ (fun1 a0 ⊕ fun2 a0) a1
fun1 _ _                  = Z
fun2 a0 (Z, _)            = a0
fun2 (S a0) (S _, S a2) = mult (S a0) (S (S a2))
```

MagicHaskeller with paramorphism

```
λa b → nat_para b (λc d e → nat_para (d e) (λf g → S g) e) (λc → Z) a
```

MagicHaskeller with catamorphism

```
λa b → nat_cata b (λc d → nat_cata (c d) (λe → S e) d) (λc → Z) a
```

F.42. Programs for `mult` with `add`

Igor II_{H}

```
no result
```

Igor II^{+} **with catamorphism**

```
no result
```

Igor II^{+} **with paramorphism**

```
mult a0 a1                = para ⊥ (fun2 a0 ⊕ fun3 a0) a1
fun2 _ _                  = Z
fun3 a0 (Z, _)            = a0
fun3 (S a0) (S _, S a2) = mult (S a0) (S (S a2))
```

MagicHaskeller with paramorphism

```
λa b → nat_para (natadd b Z) (λc d → natadd d a) Z
```

MagicHaskeller with catamorphism

```
λa b → nat_cata (natadd b Z) (λc → natadd c a) Z
```

F.43. Programs for `multfst`

Igor $\mathrm{II_H}$

```
multfst []             = []
multfst (a0 : a1)      = a0 : multfst (fun2 (a0 : a1))
fun2 [_]               = []
fun2 (a0 : (_ : a2)) = a0 : a2
```

Igor II^+ with catamorphism

```
multfst a0         = foldr fun1 [] a0
fun1 a0 a1         = foldr fun2 [a0] a1
fun2 _ (a1 : a2) = a1 : (a1 : a2)
```

Igor II^+ with paramorphism

```
multfst a0                    = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                        = []
fun2 (a0, (a1, a2))          = a0 : multfst (fun4 (a0, (a1, a2)))
fun4 (_, ([], []))           = []
fun4 (a0, (_ : _, _ : a3)) = a0 : a3
```

MagicHaskeller with paramorphism

```
λa → list_para a [] (λb c d → list_para d [b] (λe f g → b : g))
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c → foldr (λd e → d : c) c a) [] a
```

F.44. Programs for `multlst`

Igor $\mathrm{II_H}$

```
multlst []            = []
multlst (a0 : a1)     = fun1 (a0 : a1) : multlst a1
fun1 [a0]             = a0
fun1 (_ : (a1 : a2)) = fun1 (a1 : a2)
```

Igor II^{+} with catamorphism

```
multlst a0        = foldr fun1 [] a0
fun1 a0 a1        = foldr fun2 [a0] a1
fun2 a0 (_ : a2) = a0 : (a0 : a2)
```

Igor II^{+} with paramorphism

```
multlst a0                    = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                        = []
fun2 (a0, ([], []))           = [a0]
fun2 (_, (a1 : a2, _ : _)) = a1 : (a1 : a2)

-- alternative solution
multlst a0                    = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                        = []
fun2 (a0, ([], []))           = [a0]
fun2 (_, (a1 : a2, _ : _)) = a1 : (a1 : a2)
```

MagicHaskeller with paramorphism

```
λa → list_para a [] (λb c d → list_para d b (λe f g → e) : d)
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c → foldr (λd e → d) b c : c) [] a
```

F.45. Programs for nandL

Igor $\mathrm{II_H}$

```
nandL []              = False
nandL ((False) : _) = True
nandL ((True) : a0) = nandL a0
```

Igor II^+ with catamorphism

```
nandL a0          = foldr fun1 False a0
fun1 (False) _ = True
fun1 (True) a0 = a0
```

Igor II^+ with paramorphism

```
nandL a0                 = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                   = False
fun2 (False, (_, _)) = True
fun2 (True, (a0, _)) = a0
```

MagicHaskeller with paramorphism

```
λa → list_para a (λb → False) (λb c d e → iF b (d e) e) True
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c → iF b c True) False a
```

F.46. Programs for `negateAll`

Igor II_{H}

```
negateAll []         = []
negateAll (a0 : a1) = fun1 (a0 : a1) : negateAll a1
fun1 ((False) : _)  = True
fun1 ((True) : _)   = False
```

Igor II^{+} with catamorphism

```
negateAll a0 = map fun1 a0
fun1 (False) = True
fun1 (True)  = False
```

Igor II^{+} with paramorphism

```
negateAll a0           = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                 = []
fun2 (False, (a0, _)) = True : a0
fun2 (True, (a0, _))  = False : a0
```

MagicHaskeller with paramorphism

```
λa → list_para a [] (λb c d → iF b False True : d)
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c → iF b False True : c) [] a
```

F.47. Programs for `norL`

Igor II_{H}

```
norL []              = True
norL ((False) : a0) = norL a0
norL ((True) : _)   = False
```

Igor II^{+} with catamorphism

```
norL a0          = foldr fun1 True a0
fun1 (False) a0 = a0
fun1 (True) _   = False
```

Igor II^{+} with paramorphism

```
norL a0                = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                 = True
fun2 (False, (a0, _)) = a0
fun2 (True, (_, _))   = False
```

MagicHaskeller with paramorphism

```
λa → list_para a (λb → True) (λb c d e → iF b False (d True)) True
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c → iF b False c) True a
```

F.48. Programs for `nthElem`

Igor $\mathrm{II_H}$

```
nthElem (a0 : _) (Z)            = a0
nthElem (_ : (a1 : a2)) (S a3) = nthElem (a1 : a2) a3
```

Igor II^{+} with catamorphism

```
nthElem (a0 : a1) a2                       =
        cata ⊥ (fun1 (a0 : a1) ⊕ fun2 (a0 : a1)) a2
fun1 (a0 : _) _                            = a0
fun2 (_ : (a1 : _)) _                      = a1
fun2 (_ : (_ : (a2 : _))) _                = a2
fun2 (_ : (_ : (_ : (a3 : _)))) _          = a3
fun2 (_ : (_ : (_ : (_ : (a4 : _))))) _ = a4
```

Igor II^{+} with paramorphism

```
nthElem (a0 : a1) a2          =
         para ⊥ (fun1 (a0 : a1) ⊕ fun2 (a0 : a1)) a2
fun1 (a0 : _) _               = a0
fun2 (_ : (a1 : a2)) (_, a4) = nthElem (a1 : a2) a4
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.49. Programs for `odd`

Igor II_{H}

```
odd (Z)         = False
odd (S (Z))     = True
odd (S (S a0)) = odd a0
```

Igor II^{+} with catamorphism

```
odd a0        = cata ⊥ (fun1 ⊕ fun2) a0
fun1 _        = False
fun2 (False) = True
fun2 (True)  = False
```

Igor II^{+} with paramorphism

```
odd a0             = para ⊥ (fun1 ⊕ fun2) a0
fun1 _             = False
fun2 (False, _)  = True
fun2 (True, S _) = False
```

MagicHaskeller with paramorphism

```
λa → nat_para a (λb c → iF c False True) False
```

MagicHaskeller with catamorphism

```
no result
```

F.50. Programs for oddpos

Igor II_{H}

```
oddpos []           = []
oddpos (a0 : a1)    = a0 : oddpos (fun4 (a0 : a1))
fun4 [_]            = []
fun4 (_ : (_ : a2)) = a2
```

Igor II^{+} with catamorphism

```
oddpos []           = []
oddpos (a0 : a1)    = a0 : oddpos (fun4 (a0 : a1))
fun4 [_]            = []
fun4 (_ : (_ : a2)) = a2
```

Igor II^{+} with paramorphism

```
oddpos a0                   = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                      = []
fun2 (a0, (a1, a2))         = a0 : oddpos (fun6 (a0, (a1, a2)))
fun6 (_, ([], []))          = []
fun6 (_, (_ : _, _ : a3))   = a3

-- alternative solution
oddpos a0                   = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                      = []
fun2 (a0, (a1, a2))         = a0 : oddpos (fun6 (a0, (a1, a2)))
fun6 (_, ([], []))          = []
fun6 (_, (_ : _, _ : a3))   = a3
```

MagicHaskeller with paramorphism

```
λa → list_para a (λb → []) (λb c d e → iF e (d False) (b : d True))
   False
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c d → foldr (λe f → b : c []) (c a) d) (λb → []) a a
```

F.51. Programs for `odds`

Igor II_{H}

```
odds []                          = []
odds ((Z) : a0)                  = odds a0
odds ((S (Z)) : a0)              = S Z : odds a0
odds ((S (S (Z))) : a0)          = odds a0
odds ((S (S (S (Z)))) : a0)      = S (S (S Z)) : odds a0

-- alternative solution
odds []                          = []
odds ((Z) : a0)                  = odds a0
odds ((S (Z)) : a0)              = S Z : odds a0
odds ((S (S (Z))) : a0)          = odds a0
odds ((S (S (S (Z)))) : a0)      = S (S (S Z)) : odds a0
```

Igor II^{+} with catamorphism

```
odds a0       = filter fun1 a0
fun1 a0       = cata ⊥ (fun2 ⊕ fun3) a0
fun2 _        = True
fun3 (False)  = True
fun3 (True)   = False
```

Igor II^{+} with paramorphism

```
odds a0                           = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                            = []
fun2 (Z, (a0, _))                 = a0
fun2 (S (Z), (a0, _))             = S Z : a0
fun2 (S (S (Z)), (a0, _))         = a0
fun2 (S (S (S (Z))), (a0, _))     = S (S (S Z)) : a0
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.52. Programs for `oddslist`

Igor $\mathrm{II_H}$

```
no result
```

Igor II^+ with catamorphism

```
oddslist a0         = foldr fun1 True a0
fun1 a0 a1          = cata ⊥ (fun2 a1 ⊕ fun3 a1) a0
fun2 _ _            = False
fun3 a0 (False)     = a0
fun3 (True) (True) = False
```

Igor II^+ with paramorphism

```
oddslist a0                   = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                        = True
fun2 (Z, (True, []))          = False
fun2 (Z, (False, [_]))        = False
fun2 (S a0, (True, []))       = oddslist [S a0]
fun2 (Z, (True, [S _]))       = False
fun2 (S _, (False, [_]))      = False
fun2 (S a0, (True, [S _])) = fun2 (S a0, (True, []))

-- alternative solution
oddslist a0                   = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                        = True
fun2 (Z, (True, []))          = False
fun2 (Z, (False, [_]))        = False
fun2 (S a0, (True, []))       = oddslist [S a0]
fun2 (Z, (True, [S _]))       = False
fun2 (S _, (False, [_]))      = False
fun2 (S a0, (True, [S _])) = oddslist [S a0]
```

MagicHaskeller with paramorphism

```
λa → list_para a True (λb c d → nat_para b (λe f g h → f h g) (λe f →
    f) d False)
```

MagicHaskeller with catamorphism

```
no result
```

F.53. Programs for `or`

Igor $\mathrm{II_H}$

```
or (False) a0 = a0
or (True) _   = True

-- alternative solution
or a0 (False) = a0
or _ (True)   = True
```

Igor II^{+} with catamorphism

```
or (False) a0 = a0
or (True) _   = True

-- alternative solution
or a0 (False) = a0
or _ (True)   = True
```

Igor II^{+} with paramorphism

```
or (False) a0 = a0
or (True) _   = True

-- alternative solution
or a0 (False) = a0
or _ (True)   = True
```

MagicHaskeller with paramorphism

```
λa b → iF (iF b True False) True a
```

MagicHaskeller with catamorphism

```
λa b → iF (iF b True False) True a
```

F.54. Programs for orL

Igor II_{H}

```
orL []               = False
orL ((False) : a0) = orL a0
orL ((True) : _)   = True
```

Igor II^{+} with catamorphism

```
orL a0            = foldr fun1 False a0
fun1 (False) a0 = a0
fun1 (True) _    = True

-- alternative solution
orL a0            = foldr fun1 False a0
fun1 a0 (False) = a0
fun1 _ (True)    = True
```

Igor II^{+} with paramorphism

```
orL a0                      = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                      = False
fun2 (False, (a0, _))    = a0
fun2 (True, (_, _))      = True

-- alternative solution
orL a0                      = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                      = False
fun2 (a0, (False, _))    = a0
fun2 (_, (True, _ : _)) = True
```

MagicHaskeller with paramorphism

```
λa → list_para a (λb → False) (λb c d e → iF b True (d True)) True
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c → iF c True b) False a
```

F.55. Programs for `pack`

Igor II_{H}

```
pack []                = [[]]
pack [a0]              = [[a0]]
pack (a0 : (a1 : a2)) = [a0] : pack (a1 : a2)
```

Igor II^{+} with catamorphism

```
pack a0               = foldr fun1 [[]] a0
fun1 a0 [[]]          = [[a0]]
fun1 a0 ([a1] : a2) = [a0] : ([a1] : a2)
```

Igor II^{+} with paramorphism

```
pack a0                              = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                               = [[]]
fun2 (a0, ([[]], []))                = [[a0]]
fun2 (a0, ([_] : a2, a1 : _)) = [a0] : ([a1] : a2)

-- alternative solution
pack a0                              = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                               = [[]]
fun2 (a0, ([[]], []))                = [[a0]]
fun2 (a0, ([_] : a2, a1 : _)) = [a0] : ([a1] : a2)
```

MagicHaskeller with paramorphism

```
λa → list_para a (λb → b) (λb c d e → [b] : d []) [a]
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c d → [b] : c []) (λb → b) a [a]
```

F.56. Programs for pepper

Igor II_{H}

```
pepper a0 a1           = fun1 a0 a1 : fun2 a0 a1
fun1 a0 []             = (a0, Nothing)
fun1 a0 (a1 : _)       = (a0, Just (a1, S a0))
fun2 _ []              = []
fun2 a0 (a1 : a2)      =
   fun7 a0 (a1 : a2) : fun2 (fun28 a0 (a1 : a2)) a2
fun28 a0 [_]           = a0
fun28 a0 (_ : (_ : _)) = S a0
fun7 a0 [_]            = (S a0, Nothing)
fun7 a0 (_ : (a2 : _)) = (S a0, Just (a2, S (S a0)))
```

Igor II^{+} with catamorphism

```
pepper a0 a1                = foldr fun1 [(a0, Nothing)] a1
fun1 a0 ((a1, a2) : a3)     =
   (a1, Just (a0, S a1)) : fun4 a0 ((a1, a2) : a3)
fun4 a0 ((a1, a2) : a3)     = map (fun5 a0) ((a1, a2) : a3)
fun5 _ (a1, Nothing)        = (S a1, Nothing)
fun5 _ (_, Just (a2, S a1)) = (S a1, Just (a2, S (S a1)))
```

Igor II^{+} with paramorphism

```
pepper a0 a1                      = para ⊥ (fun1 a0 ⊕ fun2 a0) a1
fun1 a0 _                         = [(a0, Nothing)]
fun2 a0 (a1, ((a0, _) : _, a4))   = pepper a0 (a1 : a4)
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.57. Programs for pepperF

Igor $\mathrm{II_H}$

```
pepperF a0 a1            = fun1 a0 a1 : fun2 a0 a1
fun1 a0 []               = (a0, Nothing)
fun1 a0 (a1 : _)         = (a0, Just a1)
fun2 _ []                = []
fun2 a0 (a1 : a2)        =
   fun7 a0 (a1 : a2) : fun2 (fun28 a0 (a1 : a2)) a2
fun28 a0 [_]             = a0
fun28 a0 (_ : (_ : _)) = S a0
fun7 a0 [_]              = (S a0, Nothing)
fun7 a0 (_ : (a2 : _)) = (S a0, Just a2)
```

Igor II^+ with catamorphism

```
pepperF a0 a1            = foldr fun1 [(a0, Nothing)] a1
fun1 a0 ((a1, a2) : a3) =
   (a1, Just a0) : fun4 a0 ((a1, a2) : a3)
fun4 a0 ((a1, a2) : a3) = map (fun5 a0) ((a1, a2) : a3)
fun5 _ (a1, a2)          = (S a1, a2)
```

Igor II^+ with paramorphism

```
pepperF a0 a1                         = para ⊥ (fun1 a0 ⊕ fun2 a0) a1
fun1 a0 _                             = [(a0, Nothing)]
fun2 a0 (a1, ((a0, _) : _, a4)) = pepperF a0 (a1 : a4)
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.58. Programs for `playTennis`

Igor II_{H}

```
playTennis (Overcast, _, _, _)   = True
playTennis (Rain, _, _, Strong)  = False
playTennis (Rain, _, _, Weak)    = True
playTennis (Sunny, _, High, _)   = False
playTennis (Sunny, _, Normal, _) = True
```

Igor II^{+} with catamorphism

```
playTennis (Overcast, _, _, _)   = True
playTennis (Rain, _, _, Strong)  = False
playTennis (Rain, _, _, Weak)    = True
playTennis (Sunny, _, High, _)   = False
playTennis (Sunny, _, Normal, _) = True
```

Igor II^{+} with paramorphism

```
playTennis (Overcast, _, _, _)   = True
playTennis (Rain, _, _, Strong)  = False
playTennis (Rain, _, _, Weak)    = True
playTennis (Sunny, _, High, _)   = False
playTennis (Sunny, _, Normal, _) = True
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.59. Programs for `postorder` with `append`, `snoc`

Igor II_{H}

```
postorder (NilT)                        = []
postorder (Node a0 (NilT) (NilT))       = [a0]
postorder (Node a0
            (Node a1 (NilT) (NilT))
            (Node a2 (NilT) (NilT)))    = [a1, a2, a0]
postorder (Node a0
            (Node a1
             (Node a2 (NilT) (NilT))
             (Node a3 (NilT) (NilT)))
            (Node a4
             (Node a5 (NilT) (NilT))
             (Node a6 (NilT) (NilT)))) = [a2, a3, a1, a5, a6, a4, a0]

-- alternative solution
postorder (NilT)                        = []
postorder (Node a0 (NilT) (NilT))       = [a0]
postorder (Node a0
            (Node a1 (NilT) (NilT))
            (Node a2 (NilT) (NilT)))    = [a1, a2, a0]
postorder (Node a0
            (Node a1
             (Node a2 (NilT) (NilT))
             (Node a3 (NilT) (NilT)))
            (Node a4
             (Node a5 (NilT) (NilT))
             (Node a6 (NilT) (NilT)))) = [a2, a3, a1, a5, a6, a4, a0]

-- alternative solution
postorder (NilT)                        = []
postorder (Node a0 (NilT) (NilT))       = [a0]
postorder (Node a0
            (Node a1 (NilT) (NilT))
            (Node a2 (NilT) (NilT)))    = [a1, a2, a0]
postorder (Node a0
            (Node a1
             (Node a2 (NilT) (NilT))
             (Node a3 (NilT) (NilT)))
            (Node a4
             (Node a5 (NilT) (NilT))
             (Node a6 (NilT) (NilT)))) = [a2, a3, a1, a5, a6, a4, a0]

-- alternative solution
postorder (NilT)                        = []
postorder (Node a0 (NilT) (NilT))       = [a0]
postorder (Node a0
            (Node a1 (NilT) (NilT))
            (Node a2 (NilT) (NilT)))    = [a1, a2, a0]
postorder (Node a0
```

```
           (Node a1
            (Node a2 (NilT) (NilT))
            (Node a3 (NilT) (NilT)))
           (Node a4
            (Node a5 (NilT) (NilT))
            (Node a6 (NilT) (NilT)))) = [a2, a3, a1, a5, a6, a4, a0]

-- alternative solution
postorder (NilT)                     = []
postorder (Node a0 (NilT) (NilT))    = [a0]
postorder (Node a0
           (Node a1 (NilT) (NilT))
           (Node a2 (NilT) (NilT)))  = [a1, a2, a0]
postorder (Node a0
           (Node a1
            (Node a2 (NilT) (NilT))
            (Node a3 (NilT) (NilT)))
           (Node a4
            (Node a5 (NilT) (NilT))
            (Node a6 (NilT) (NilT)))) = [a2, a3, a1, a5, a6, a4, a0]

-- alternative solution
postorder (NilT)                     = []
postorder (Node a0 (NilT) (NilT))    = [a0]
postorder (Node a0
           (Node a1 (NilT) (NilT))
           (Node a2 (NilT) (NilT)))  = [a1, a2, a0]
postorder (Node a0
           (Node a1
            (Node a2 (NilT) (NilT))
            (Node a3 (NilT) (NilT)))
           (Node a4
            (Node a5 (NilT) (NilT))
            (Node a6 (NilT) (NilT)))) = [a2, a3, a1, a5, a6, a4, a0]

-- alternative solution
postorder (NilT)                     = []
postorder (Node a0 (NilT) (NilT))    = [a0]
postorder (Node a0
           (Node a1 (NilT) (NilT))
           (Node a2 (NilT) (NilT)))  = [a1, a2, a0]
postorder (Node a0
           (Node a1
            (Node a2 (NilT) (NilT))
            (Node a3 (NilT) (NilT)))
           (Node a4
            (Node a5 (NilT) (NilT))
            (Node a6 (NilT) (NilT)))) = [a2, a3, a1, a5, a6, a4, a0]

-- alternative solution
postorder (NilT)                     = []
postorder (Node a0 (NilT) (NilT))    = [a0]
```

```
postorder (Node a0
            (Node a1 (NilT) (NilT))
            (Node a2 (NilT) (NilT)))   = [a1, a2, a0]
postorder (Node a0
            (Node a1
             (Node a2 (NilT) (NilT))
             (Node a3 (NilT) (NilT)))
            (Node a4
             (Node a5 (NilT) (NilT))
             (Node a6 (NilT) (NilT)))) = [a2, a3, a1, a5, a6, a4, a0]
```

Igor II^{+} with catamorphism

```
postorder a0                   = cata ⊥ (fun3 ⊕ fun4) a0
fun3 _                         = []
fun4 (a0, ([], []))            = [a0]
fun4 (a0, (a1 : a2, a3 : a4)) =
   a1 : append (snoc a3 a2) (snoc a0 a4)
```

Igor II^{+} with paramorphism

```
postorder a0                          = para ⊥ (fun3 ⊕ fun4) a0
fun3 _                                = []
fun4 (a0, (([], NilT), ([], NilT)))  = [a0]
fun4 (a0, ((_ : _, Node a3 a4 a5)
          ,(_ : _, Node a8 a9 a10))) =
   postorder (Node a0 (Node a3 a4 a5) (Node a8 a9 a10))
```

MagicHaskeller with paramorphism

```
λa → ntree_para a [] (λb c d e f → snc (e ++ f) b)
```

MagicHaskeller with catamorphism

```
λa → ntree_cata a [] (λb c d → snc (c ++ d) b)
```

F.60. Programs for `preorder` with `append`

Igor II_{H}

```
preorder (NilT)           = []
preorder (Node a0 a1 a2) = a0 : append (preorder a1) (preorder a2)
```

Igor II^{+} **with catamorphism**

```
preorder a0          = cata ⊥ (fun2 ⊕ fun3) a0
fun2 _               = []
fun3 (a0, (a1, a2)) = a0 : append a1 a2
```

Igor II^{+} **with paramorphism**

```
preorder a0                    = para ⊥ (fun2 ⊕ fun3) a0
fun2 _                         = []
fun3 (a0, ((a1, _), (a3, _))) = a0 : append a1 a3
```

MagicHaskeller with paramorphism

```
λa → ntree_para a [] (λb c d e f → b : (e ++ f))
```

MagicHaskeller with catamorphism

```
λa → ntree_cata a [] (λb c d → b : (c ++ d))
```

F.61. Programs for `replicate`

Igor II_{H}

```
replicate _ (Z)     = []
replicate a0 (S a1) = a0 : replicate a0 a1
```

Igor II^{+} with catamorphism

```
replicate a0 a1 = cata ⊥ (fun1 a0 ⊕ fun2 a0) a1
fun1 _ _        = []
fun2 a0 a1      = a0 : a1
```

Igor II^{+} with paramorphism

```
replicate a0 a1 = para ⊥ (fun1 a0 ⊕ fun2 a0) a1
fun1 _ _        = []
fun2 a0 (a1, _) = a0 : a1
```

MagicHaskeller with paramorphism

```
λa b → nat_para b (λc d → a : d) []
```

MagicHaskeller with catamorphism

```
λa b → nat_cata b (λc → a : c) []
```

F.62. Programs for `reverse`

Igor $\mathrm{II_H}$

```
reverse []             = []
reverse (a0 : a1)      = fun1 (a0 : a1) : reverse (fun5 (a0 : a1))
fun1 [a0]              = a0
fun1 (_ : (a1 : a2))   = fun1 (a1 : a2)
fun5 [_]               = []
fun5 (a0 : (a1 : a2))  = a0 : fun5 (a1 : a2)
```

Igor $\mathrm{II^+}$ with catamorphism

```
reverse a0          = foldr fun1 [] a0
fun1 a0 a1          = foldr fun2 [a0] a1
fun2 a0 (a1 : a2)   = a0 : (a1 : a2)
```

Igor $\mathrm{II^+}$ with paramorphism

```
reverse a0                    = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                        = []
fun2 (a0, ([], []))           = [a0]
fun2 (a0, (_ : _, a3 : a4))   = reverse (a0 : (a3 : a4))
```

MagicHaskeller with paramorphism

```
λa → list_para a (λb → b) (λb c d e → d (b : e)) []
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c d → c (b : d)) (λb → b) a []
```

F.63. Programs for `rocket`

Igor II_{H}

```
rocket (NOCARGO) a0  = FLY a0
rocket (IN a0 a1) a2 = UNL a0 (rocket a1 (LOD a0 a2))
```

Igor II^{+} with catamorphism

```
rocket a0 a1                         = cata ⊥ (fun1 a1 ⊕ fun2 a1) a0
fun1 a0 _                            = FLY a0
fun14 a0 (a1, UNL a2 a3)             =
   fun3 a0 (a1, fun20 a0 (a1, UNL a2 a3))
fun2 a0 (a1, a2)                     = UNL a1 (fun3 a0 (a1, a2))
fun20 a0 (_, UNL _ (UNL a3 a4))      =
    UNL a3
     (fun14 a0 (a3, UNL a3 (fun20 (LOD a3 a0) (a3, UNL a3 a4))))
fun20 a0 (_, UNL _ (FLY (LOD _ a0))) = FLY a0
fun3 a0 (a1, FLY a0)                 = FLY (LOD a1 a0)
fun3 a0 (a1, UNL a2 a3)              =
    fun2 (LOD a1 a0) (a2, fun14 a0 (a1, UNL a2 a3))
```

Igor II^{+} with paramorphism

```
rocket a0 a1                       = para ⊥ (fun1 a1 ⊕ fun2 a1) a0
fun1 a0 _                          = FLY a0
fun2 a0 (a1, (a2, a3))             = UNL a1 (fun3 a0 (a1, (a2, a3)))
fun3 a0 (a1, (FLY a0, NOCARGO))    = FLY (LOD a1 a0)
fun3 a0 (a1, (UNL _ _, IN a2 a4)) =
   UNL a2 (rocket a4 (LOD a2 (LOD a1 a0)))
```

MagicHaskeller with paramorphism

```
λa b → cargo_para a (λc → FLY c) (λc d e f → UNL c (e (LOD c f))) b
```

MagicHaskeller with catamorphism

```
λa b → cargo_para a (λc → FLY c) (λc d e f → UNL c (e (LOD c f))) b
```

F.64. Programs for `sentence`

Igor $\mathrm{II_H}$

```
sentence (Z)    = ['D', 'N', 'V', 'D', 'N']
sentence (S a0) = 'D' : ('N' : ('V' : sentence a0))
```

Igor II^+ **with catamorphism**

```
sentence a0                  = cata ⊥ (fun1 ⊕ fun2) a0
fun1 _                       = ['D', 'N', 'V', 'D', 'N']
fun2 ('D' : ('N' : ('V' :
       ('D' : ('N' : a0))))) =
     'D' : ('N' : ('V' : ('D' : ('N' : ('V' : ('D' : ('N' : a0)))))))
```

Igor II^+ **with paramorphism**

```
sentence a0                      = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                           = ['D', 'N', 'V', 'D', 'N']
fun2 ('D' : ('N' : ('V' :
       ('D' : ('N' : a0))))), _) =
     'D' : ('N' : ('V' : ('D' : ('N' : ('V' : ('D' : ('N' : a0)))))))
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.65. Programs for `shiftl`

Igor $\mathrm{II_H}$

```
shiftl []                 = []
shiftl [a0]               = [a0]
shiftl (a0 : (a1 : a2)) = a1 : shiftl (a0 : a2)
```

Igor II^{+} with catamorphism

```
shiftl a0                  = foldr fun1 [] a0
fun1 a0 a1                 = foldr fun2 [a0] a1
fun2 a0 [a1]               = [a0, a1]
fun2 a0 (a1 : (a2 : a3)) = a1 : (a0 : (a2 : a3))
```

Igor II^{+} with paramorphism

```
shiftl a0                    = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                       = []
fun2 (a0, ([], []))          = [a0]
fun2 (a0, (_ : _, a3 : a4)) = shiftl (a0 : (a3 : a4))
```

MagicHaskeller with paramorphism

```
λa → list_para a [] (λb c d → list_para c [b] (λe f g → e : g))
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c d → foldr (λe f → b : c d) (c (b : d)) d) (λb → b) a
    []
```

F.66. Programs for `shiftl, shiftr`

Igor II_{H}

```
shiftl []                = []
shiftl [a0]              = [a0]
shiftl (a0 : (a1 : a2)) = a1 : shiftl (a0 : a2)
shiftr []                = []
shiftr (a0 : a1)         = fun8 (a0 : a1) : fun9 (a0 : a1)
fun8 [a0]                = a0
fun8 (_ : (a1 : a2))     = fun8 (a1 : a2)
fun9 [_]                 = []
fun9 (a0 : (a1 : a2))    = a0 : fun9 (a1 : a2)
```

Igor II^{+} with catamorphism

```
shiftl a0                 = foldr fun2 [] a0
shiftr a0                 = foldr fun6 [] a0
fun2 a0 a1                = foldr fun3 [a0] a1
fun3 a0 [a1]              = [a0, a1]
fun3 a0 (a1 : (a2 : a3)) = a1 : (a0 : (a2 : a3))
fun6 a0 a1                = foldr fun7 [a0] a1
fun7 a0 [a1]              = [a0, a1]
fun7 a0 (a1 : (a2 : a3)) = a0 : (a2 : (a1 : a3))
```

Igor II^{+} with paramorphism

```
shiftl a0                      = para ⊥ (fun2 ⊕ fun3) a0
shiftr a0                      = para ⊥ (fun9 ⊕ fun10) a0
fun10 (a0, ([], []))          = [a0]
fun10 (a0, (a1 : a2, _ : _)) = a1 : (a0 : a2)
fun2 _                         = []
fun3 (a0, ([], []))           = [a0]
fun3 (a0, (_ : _, a3 : a4))   = shiftl (a0 : (a3 : a4))
fun9 _                         = []

-- alternative solution
shiftl a0 = para ⊥ (fun2 ⊕ fun3) a0
shiftr a0                      = para ⊥ (fun9 ⊕ fun10) a0
fun10 (a0, ([], []))          = [a0]
fun10 (a0, (a1 : a2, _ : _)) = a1 : (a0 : a2)
fun2 _                         = []
fun3 (a0, ([], []))           = [a0]
fun3 (a0, (_ : _, a3 : a4))   = shiftl (a0 : (a3 : a4))
fun9 _                         = []
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.67. Programs for `shiftr`

Igor II_H

```
shiftr []               = []
shiftr (a0 : a1)        = fun1 (a0 : a1) : fun2 (a0 : a1)
fun1 [a0]               = a0
fun1 (_ : (a1 : a2))    = fun1 (a1 : a2)
fun2 [_]                = []
fun2 (a0 : (a1 : a2))   = a0 : fun2 (a1 : a2)
```

Igor II^+ with catamorphism

```
shiftr a0                     = foldr fun1 [] a0
fun1 a0 a1                    = foldr fun2 [a0] a1
fun2 a0 [a1]                  = [a0, a1]
fun2 a0 (a1 : (a2 : a3))      = a0 : (a2 : (a1 : a3))
```

Igor II^+ with paramorphism

```
shiftr a0                        = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                           = []
fun2 (a0, ([], []))              = [a0]
fun2 (a0, (a1 : a2, _ : _))      = a1 : (a0 : a2)

-- alternative solution
shiftr a0                        = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                           = []
fun2 (a0, ([], []))              = [a0]
fun2 (a0, (a1 : a2, _ : _))      = a1 : (a0 : a2)
```

MagicHaskeller with paramorphism

```
λa → list_para a [] (λb c d → list_para d [b] (λe f g → e : (b : f)))
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c d → c (foldr (λe f g → g : f e) (λe → []) a b)) (λb
    → b) a a
```

F.68. Programs for `snoc`

Igor II_{H}

```
snoc a0 []        = [a0]
snoc a0 (a1 : a2) = a1 : snoc a0 a2
```

Igor II^{+} **with catamorphism**

```
snoc a0 a1        = foldr fun1 [a0] a1
fun1 a0 (a1 : a2) = a0 : (a1 : a2)
```

Igor II^{+} **with paramorphism**

```
snoc a0 a1                 = para ⊥ (fun1 a0 ⊕ fun2 a0) a1
fun1 a0 _                  = [a0]
fun2 _ (a1, (a2 : a3, _)) = a1 : (a2 : a3)
```

MagicHaskeller with paramorphism

```
λa b → list_para b [a] (λc d e → c : e)
```

MagicHaskeller with catamorphism

```
λa b → foldr (λc d → c : d) [a] b
```

F.69. Programs for `split`

Igor $\mathrm{II_H}$

```
split a0               = (fun1 a0, fun2 a0)
fun1 []                = []
fun1 [a0]              = [a0]
fun1 (a0 : (_ : a2))   = a0 : fun1 a2
fun2 []                = []
fun2 [_]               = []
fun2 (_ : (a1 : a2))   = a1 : fun2 a2
```

Igor II^+ with catamorphism

```
split a0           = foldr fun1 ([], []) a0
fun1 a0 (a1, a2)   = (a0 : a2, a1)
```

Igor II^+ with paramorphism

```
split a0                  = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                    = ([], [])
fun2 (a0, ((a1, a2), _))  = (a0 : a2, a1)
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.70. Programs for `splitAt`

Igor II_{H}

```
no result
```

Igor II^{+} with catamorphism

```
splitAt a0 (a1 : a2)                         =
   cata ⊥ (fun1 (a1 : a2) ⊕ fun2 (a1 : a2)) a0
fun1 (a0 : a1) _                             = ([], a0 : a1)
fun11 (_ : _) ([], _ : _)                    = []
fun11 (_ : (a1 : a2)) (_ : a3, a4 : a5) =
   a1 : fun11 (a1 : a2) (a3, a4 : a5)
fun2 (a0 : a1) (a0 : a1, [])                 = (a0 : a1, [])
fun2 (a0 : a1) (a2, a3 : a4)                 =
   (a0 : fun11 (a0 : a1) (a2, a3 : a4), a4)
```

Igor II^{+} with paramorphism

```
no result
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.71. Programs for `sum`

Igor II_{H}

```
sum []             = Z
sum ((Z) : a0)     = sum a0
sum ((S a0) : a1)  = S (sum (a0 : a1))
```

Igor II^{+} with catamorphism

```
sum a0     = foldr fun1 Z a0
fun1 a0 a1 = cata ⊥ (fun2 a0 ⊕ fun3 a0) a1
fun2 a0 _  = a0
fun3 _ a1  = S a1
```

Igor II^{+} with paramorphism

```
sum a0                       = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                       = Z
fun2 (a0, (Z, _))            = a0
fun2 (a0, (S _, [S a1]))     = sum [a0, S a1]

-- alternative solution
sum a0                       = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                       = Z
fun2 (a0, (Z, _))            = a0
fun2 (a0, (S _, [S a1]))     = sum [S a1, a0]
```

MagicHaskeller with paramorphism

```
λa → list_para a Z (λb c d → nat_para d (λe f g → S (f g)) (λe → e) b
   )
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c → nat_cata c (λd → S d) b) Z a
```

F.72. Programs for `sub`

Igor $\mathrm{II_H}$

```
no result
```

Igor II^+ with catamorphism

```
sub a0 a1           = cata ⊥ (fun1 a1 ⊕ fun2 a1) a0
fun1 a0 _           = a0
fun2 _ (Z)          = Z
fun2 (S _) (S a1) = a1
```

Igor II^+ with paramorphism

```
sub a0 a1              = para ⊥ (fun1 a0 ⊕ fun2 a0) a1
fun1 _ _               = Z
fun2 (Z) (_, a0)       = S a0
fun2 (S a0) (_, a2) = sub a0 a2
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.73. Programs for `subseqs` with `append`

Igor II_{H}

```
subseqs []           = [[]]
subseqs [a0]         = [[a0], []]
subseqs [a0, a1]     = [[a0, a1], [a0], [a1], []]
subseqs [a0, a1, a2] =
   [[a0, a1, a2], [a0, a1], [a0, a2], [a0], [a1, a2], [a1], [a2], []]
```

Igor II^{+} with catamorphism

```
subseqs a0         = foldr fun2 [[]] a0
fun2 a0 (a1 : a2) = append (fun3 a0 (a1 : a2)) (fun4 a0 (a1 : a2))
fun3 a0 (a1 : a2) = map (fun8 a0) (a1 : a2)
fun4 a0 (a1 : a2) = map (fun9 a0) (a1 : a2)
fun8 a0 a1         = a0 : a1
fun9 _ a1          = a1
```

Igor II^{+} with paramorphism

```
subseqs a0                                 = para ⊥ (fun2 ⊕ fun3) a0
fun2 _                                     = [[]]
fun3 (a0, ([[]], []))                      = [[a0], []]
fun3 (a0, ((_ : _) : (_ : _), a1 : a2)) = subseqs (a0 : (a1 : a2))
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.74. Programs for `swap`

Igor II_{H}

```
swap []                = []
swap [a0]              = [a0]
swap (a0 : (a1 : a2)) = a1 : (a0 : swap a2)
```

Igor II^{+} with catamorphism

```
swap a0                                  = foldr fun1 [] a0
fun1 a0 a1                               = foldr fun2 [a0] a1
fun2 a0 [a1]                             = [a0, a1]
fun2 a0 [a1, a2]                         = [a1, a2, a0]
fun2 a0 [a1, a2, a3]                     = [a3, a2, a1, a0]
fun2 a0 (a1 : (a2 : (a3 : (a4 : a5)))) =
  a4 : (a2 : fun2 a1 (a3 : (a0 : a5)))
```

Igor II^{+} with paramorphism

```
swap a0                      = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                       = []
fun2 (a0, ([], []))          = [a0]
fun2 (a0, (_ : _, a3 : a4)) = swap (a0 : (a3 : a4))
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.75. Programs for `switch`

Igor $\mathrm{II_H}$

```
switch []               = []
switch (a0 : a1)        = fun1 (a0 : a1) : switch (fun5 (a0 : a1))
fun1 [a0]               = a0
fun1 (_ : (a1 : a2))    = fun1 (a1 : a2)
fun5 [_]                = []
fun5 (a0 : (a1 : a2))   = a0 : switch (fun5 (a1 : a2))
```

Igor II^+ with catamorphism

```
switch a0                          = foldr fun1 [] a0
fun1 a0 a1                         = foldr fun2 [a0] a1
fun2 a0 [a1]                       = [a0, a1]
fun2 a0 [a1, a2]                   = [a0, a1, a2]
fun2 a0 (a1 : (a2 : (a3 : a4)))    = a0 : (a2 : (a1 : (a3 : a4)))
```

Igor II^+ with paramorphism

```
switch a0                     = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                        = []
fun2 (a0, ([], []))           = [a0]
fun2 (a0, (_ : _, a3 : a4))   = switch (a0 : (a3 : a4))
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.76. Programs for `tail`

Igor $\mathrm{II_H}$

```
tail (_ : a1) = a1
```

Igor II^+ with catamorphism

```
tail (_ : a1) = a1
```

Igor II^+ with paramorphism

```
tail (_ : a1) = a1
```

MagicHaskeller with paramorphism

```
λa → list_para a [] (λb c d → c)
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c d → foldr (λe f → b : c d) (c a) d) (λb → []) a []
```

F.77. Programs for `tails`

Igor II_{H}

```
tails []         = [[]]
tails (a0 : a1) = (a0 : a1) : tails a1
```

Igor II^{+} with catamorphism

```
tails a0           = foldr fun1 [[]] a0
fun1 a0 (a1 : a2) = (a0 : a1) : (a1 : a2)
```

Igor II^{+} with paramorphism

```
tails a0                  = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                    = [[]]
fun2 (a0, (_ : a2, a1)) = (a0 : a1) : (a1 : a2)
```

MagicHaskeller with paramorphism

```
λa → a : list_para a [] (λb c d → c : d)
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c → foldr (λd e → (b : d) : c) c c) [[]] a
```

F.78. Programs for `take`

Igor II_{H}

```
take (Z) _            = []
take (S _) []         = []
take (S a0) (a1 : a2) = a1 : take a0 a2
```

Igor II^{+} with catamorphism

```
take a0 a1                 = cata ⊥ (fun1 a1 ⊕ fun2 a1) a0
fun1 _ _                   = []
fun2 a0 []                 = foldr (fun4 []) [] a0
fun2 (a0 : a1) (a0 : _) = a0 : a1
fun4 [] a0 _               = [a0]
```

Igor II^{+} with paramorphism

```
take a0 a1                    = para ⊥ (fun1 a0 ⊕ fun2 a0) a1
fun1 _ _                      = []
fun2 (Z) (_, ([], _))         = []
fun2 (S a0) (a1, (_, a2)) = a1 : take a0 a2
```

MagicHaskeller with paramorphism

```
λa b → nat_para a (λc d e → list_para e [] (λf g h → f : d g)) (λc →
    []) b
```

MagicHaskeller with catamorphism

```
λa b → nat_cata a (λc d → foldr (λe f → e : c f) [] d) (λc → []) b
```

F.79. Programs for `transpose`

Igor $\mathrm{II_H}$

```
no result
```

Igor II^+ with catamorphism

```
transpose ((a0 : a1) : a2)    =
   fun1 ((a0 : a1) : a2) : fun2 ((a0 : a1) : a2)
fun1 ((a0 : a1) : a2)         = map fun3 ((a0 : a1) : a2)
fun10 ((a0 : (a1 : a2)) : a3) = map fun13 ((a0 : (a1 : a2)) : a3)
fun13 (_ : (a1 : a2))         = a1 : a2
fun2 ([_] : _)                = []
fun2 ((a0 : (a1 : a2)) : a3)  =
   transpose (fun10 ((a0 : (a1 : a2)) : a3))
fun3 (a0 : _)                 = a0
```

Igor II^+ with paramorphism

```
transpose ((a0 : a1) : a2)    =
   fun1 ((a0 : a1) : a2) : fun2 ((a0 : a1) : a2)
fun1 [[a0]]                   = [a0]
fun1 ([a0] : ([a1] : a2))     = a0 : fun1 ([a1] : a2)
fun1 [a0 : (_ : _)]           = [a0]
fun1 ((a0 : (a1 : a2)) :
      ((a3 : (_ : _)) : a6))  = a0 : fun1 ((a3 : (a1 : a2)) : a6)
fun2 [[_]]                    = []
fun2 ([_] : ([_] : _))        = []
fun2 [_ : (a1 : a2)]          = transpose [a1 : a2]
fun2 ((_ : (a1 : a2)) :
      ((_ : (a4 : a5)) : a6)) =
   transpose ((a1 : a2) : transpose (fun2 ((a4 : (a4 : a5)) : a6)))
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.80. Programs for `unzip`

Igor $\mathrm{II_H}$

```
unzip ((a0, a1) : a2)              =
   (a0 : fun3 ((a0, a1) : a2), fun2 ((a0, a1) : a2))
fun2 [(_, a1)]                     = [a1]
fun2 ((a0, a1) : ((_, a3) : a4)) = a1 : fun2 ((a0, a3) : a4)
fun3 [(_, _)]                      = []
fun3 ((_, a1) : ((a2, _) : a4))  = a2 : fun3 ((a2, a1) : a4)
```

Igor II^+ with catamorphism

```
unzip ((a0, a1) : a2) = (fun1 ((a0, a1) : a2), fun2 ((a0, a1) : a2))
fun1 ((a0, a1) : a2)  = map fun3 ((a0, a1) : a2)
fun2 ((a0, a1) : a2)  = map fun6 ((a0, a1) : a2)
fun3 (a0, _)          = a0
fun6 (_, a1)          = a1
```

Igor II^+ with paramorphism

```
unzip ((a0, a1) : a2)              =
   (a0 : fun3 ((a0, a1) : a2), fun2 ((a0, a1) : a2))
fun2 [(_, a1)]                     = [a1]
fun2 ((a0, a1) : ((_, a3) : a4)) = a1 : fun2 ((a0, a3) : a4)
fun3 [(_, _)]                      = []
fun3 ((_, a1) : ((a2, _) : a4))  = a2 : fun3 ((a2, a1) : a4)
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.81. Programs for weave

Igor II_{H}

```
weave [] a0         = a0
weave (a0 : a1) a2 = a0 : weave a2 a1
```

Igor II^{+} with catamorphism

```
weave [] a0         = a0
weave (a0 : a1) a2 = a0 : weave a2 a1
```

Igor II^{+} with paramorphism

```
weave a0 a1            = para ⊥ (fun1 a1 ⊕ fun2 a1) a0
fun1 a0 _              = a0
fun2 a0 (a1, (_, a3)) = a1 : weave a0 a3
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.82. Programs for `weaveL`

Igor $\mathrm{II_H}$

```
no result
```

Igor II^+ **with catamorphism**

```
no result
```

Igor II^+ **with paramorphism**

```
no result
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

F.83. Programs for `zeros`

Igor II_{H}

```
zeros []           = []
zeros ((Z) : a0)   = Z : zeros a0
zeros ((S _) : a1) = zeros a1

-- alternative solution
zeros []           = []
zeros ((Z) : a0)   = Z : zeros a0
zeros ((S _) : a1) = zeros a1
```

Igor II^{+} with catamorphism

```
zeros a0    = filter fun1 a0
fun1 (Z)    = False
fun1 (S _)  = True
```

Igor II^{+} with paramorphism

```
zeros a0              = para ⊥ (fun1 ⊕ fun2) a0
fun1 _                = []
fun2 (Z, (a0, _))     = Z : a0
fun2 (S _, (a1, _))   = a1
```

MagicHaskeller with paramorphism

```
λa → list_para a [] (λb c d → nat_para b (λe f → d) (Z : d))
```

MagicHaskeller with catamorphism

```
λa → foldr (λb c → nat_cata b (λd → c) (Z : c)) [] a
```

F.84. Programs for zip

Igor II_{H}

```
zip [] _                 = []
zip [_] []               = []
zip (a0 : a1) (a2 : a3) = (a0, a2) : zip a1 a3
```

Igor II^{+} **with catamorphism**

```
zip [] _                 = []
zip [_] []               = []
zip (a0 : a1) (a2 : a3) = (a0, a2) : zip a1 a3
```

Igor II^{+} **with paramorphism**

```
zip [] _                 = []
zip [_] []               = []
zip (a0 : a1) (a2 : a3) = (a0, a2) : zip a1 a3
```

MagicHaskeller with paramorphism

```
no result
```

MagicHaskeller with catamorphism

```
no result
```

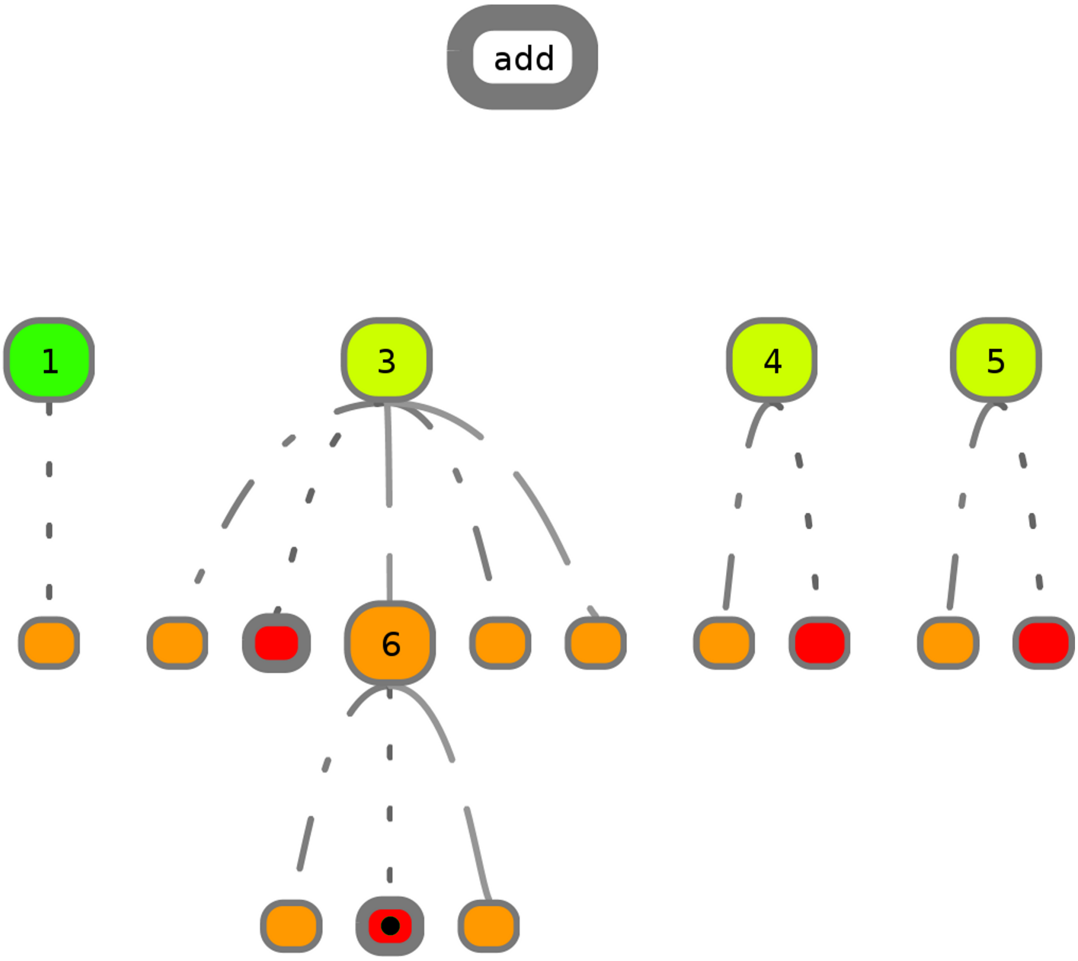
G. Search Space Visualisations

This chapter presents some examples of IGOR II's search tree visualisations created with the tool istviewer[1] which was implemented by Olga Yanenko. They give a good impression of the complexity reduction that can be achieved by the use of type morphisms.

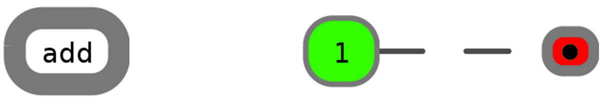
A thin node represents an unfinished hypothesis, a bold node a hypothesis where all equations are closed. The root node is the initial hypothesis and the the final solution is marked with a black dot. Different line styles represent different successor operators. The colour indicates the heuristic value of a hypothesis: green for few patterns, red for many patterns. The numbers shows the chronological order in which the hypotheses have been processed.

Note that these are only small examples containing only a few loop cycles of the IGOR II-algorithm. Many other contain just too many nodes to visualise them on one page. However, even those small examples show how the original algorithm is hampered by many equivalent hypotheses. This is especially apparent in Figure G.3 or in Figure G.2 depicting the search space of `evens` and `addN`, respectively.

[1] `http://www.cogsys.wiai.uni-bamberg.de/effalip/download.html`

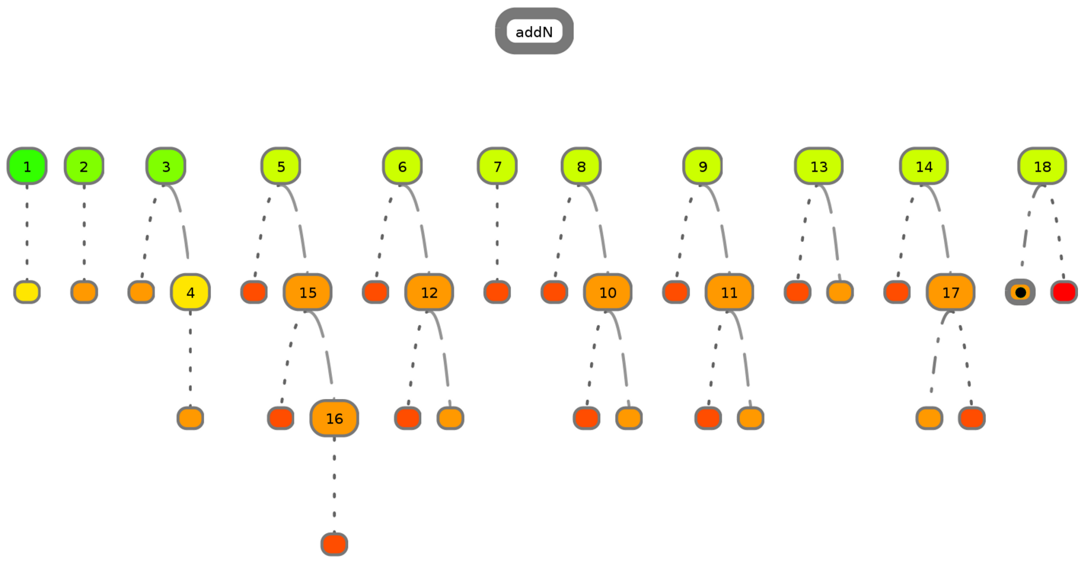


(a) IGOR II_H

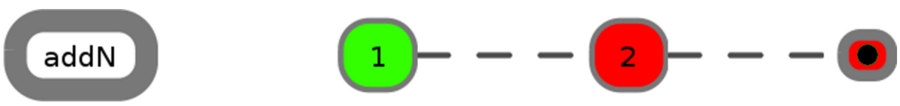


(b) IGOR II_C^+

Figure G.1.: Visualisation of the search space for `add` of IGOR II_H without (a) and of IGOR II_C^+ with catamorphisms (b).

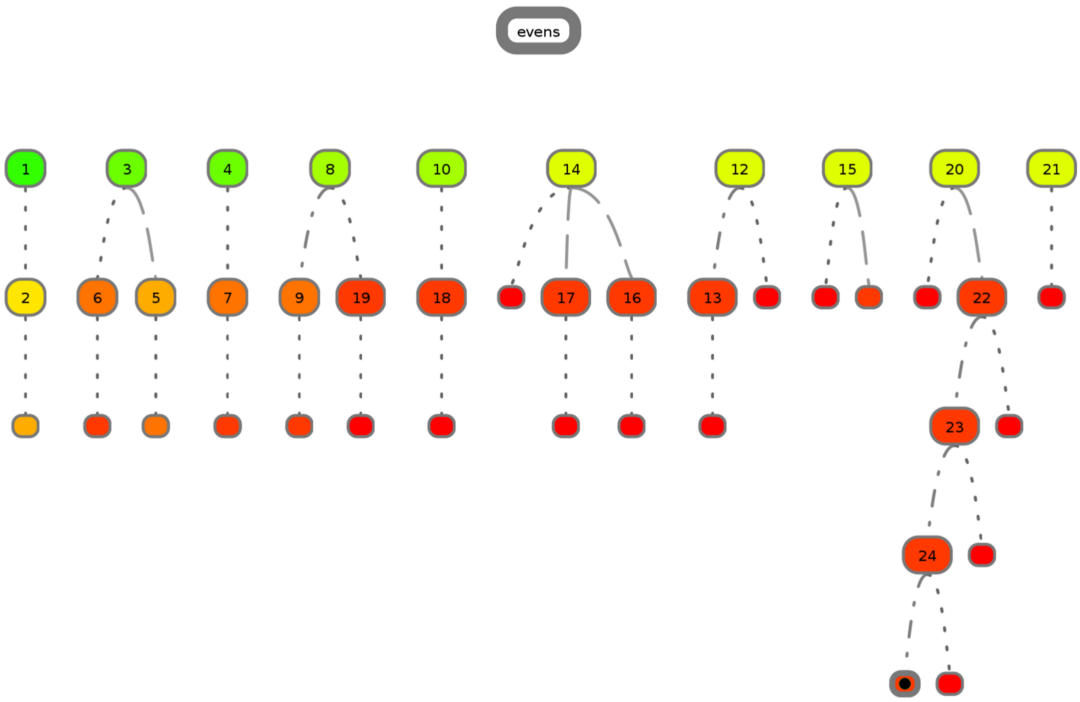


(a) IGOR II_H



(b) IGOR II_C^+

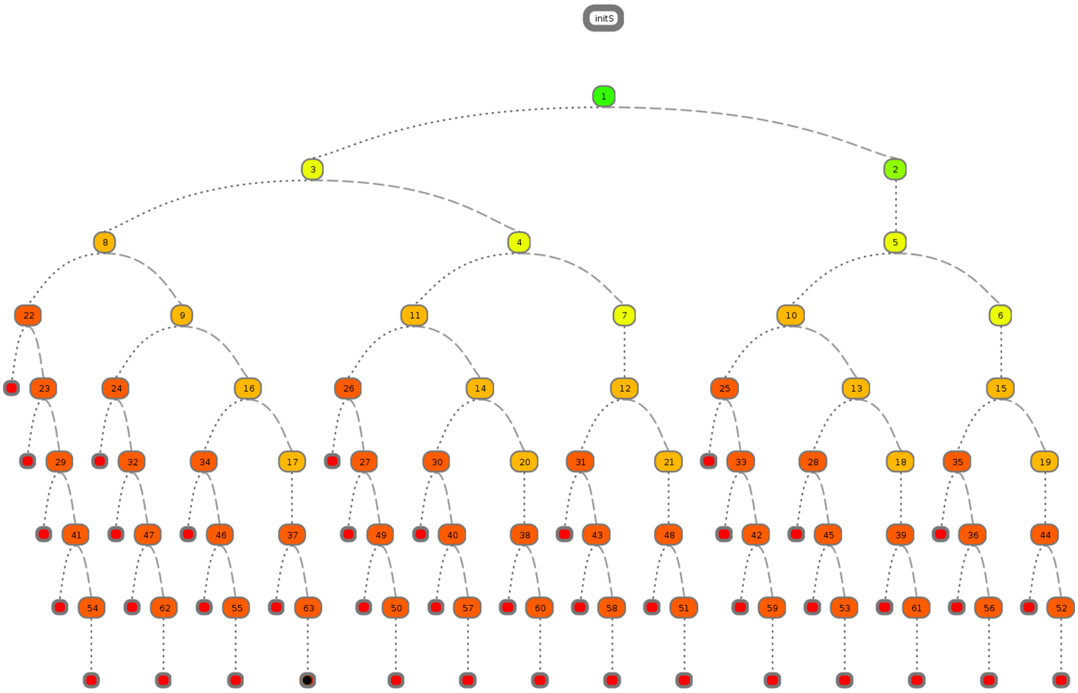
Figure G.2.: Visualisation of the search space for `addN` of IGOR II_H without (a) and of IGOR II_C^+ with catamorphisms (b).



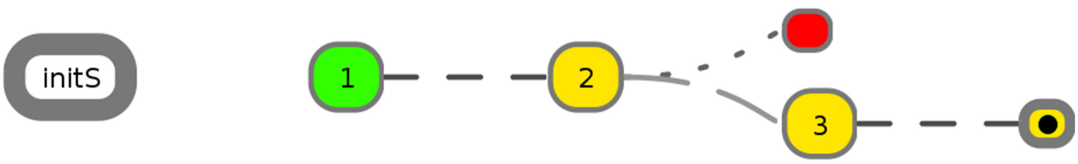
(a) IGOR II_H

(b) IGOR II_C^+

Figure G.3.: Visualisation of the search space for `evens` of IGOR II_H without (a) and of IGOR II_C^+ with catamorphisms (b).

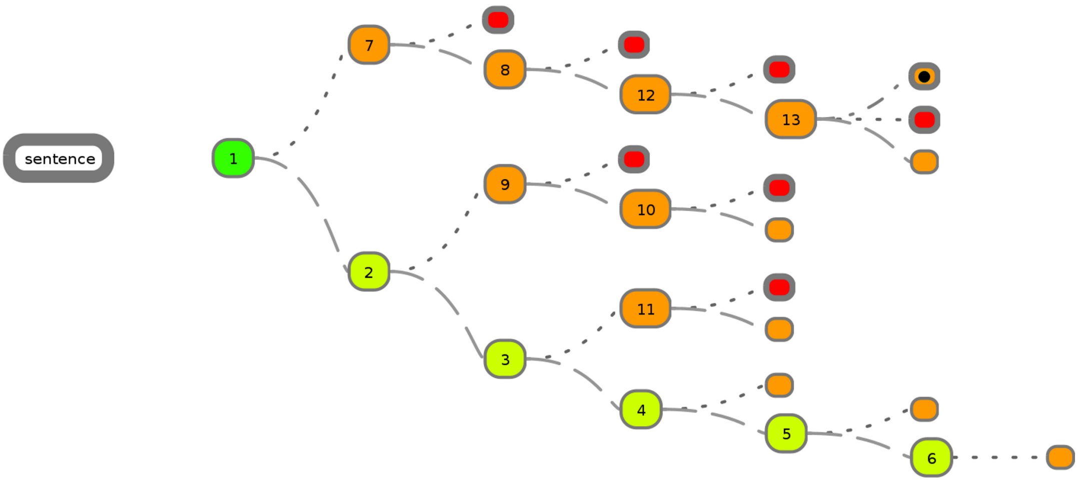


(a) IGOR $\mathrm{II_H}$



(b) IGOR II_C^+

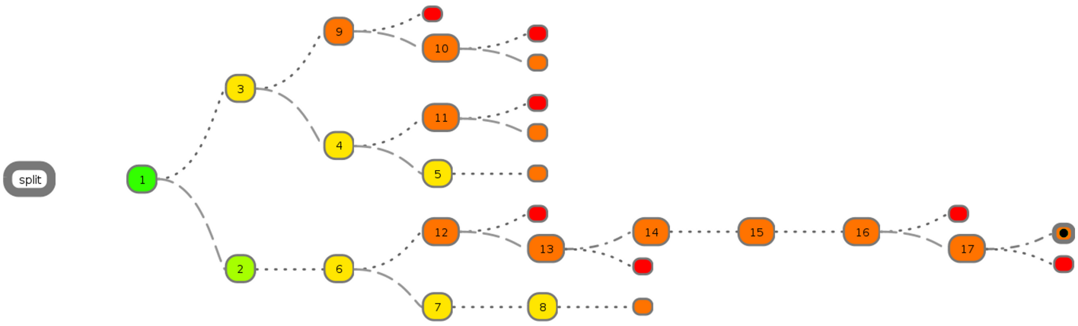
Figure G.4.: Visualisation of the search space for `inits` of IGOR $\mathrm{II_H}$ without (a) and of IGOR II_C^+ with catamorphisms (b).



(a) IGOR II$_{\mathrm{H}}$

(b) IGOR II$_C^+$

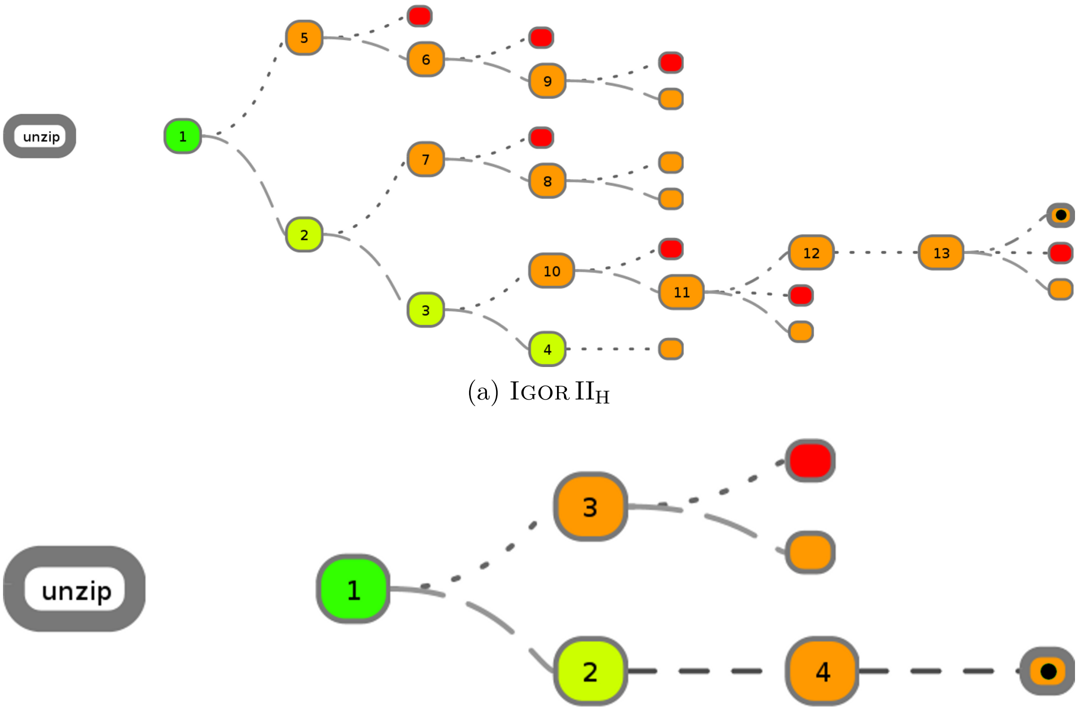
Figure G.5.: Visualisation of the search space for `sentence` of IGOR II$_{\mathrm{H}}$ without (a) and of IGOR II$_C^+$ with catamorphisms (b).



(a) IGOR II$_{\mathrm{H}}$

(b) IGOR II$_C^+$

Figure G.6.: Visualisation of the search space for `split` of IGOR II$_{\mathrm{H}}$ without (a) and of IGOR II$_C^+$ with catamorphisms (b).



(a) IGOR II_{H}

(b) IGOR II_{C}^{+}

Figure G.7.: Visualisation of the search space for `unzip` of IGOR II_{H} without (a) and of IGOR II_{C}^{+} with catamorphisms (b).

Index

:, 52
$\Theta(\alpha)$, 106
$\Phi(\rho)$, 86
$\Phi(\rho)/{\sim_p}$, 87
$\Phi(\rho)_{c_j}$, 106
$\Phi(f)$, 84
$\equiv$, 50
λ, 179
$\mathfrak{P}(S)$, 94
$\mu\mathsf{F}$, 64
$\preceq$, *see* subsumption
$|\cdot|$, 49
$\sim_p$, 87
$\square$, 50
$\subset$, 50
$\subseteq$, 50
$l = r$, *see* equation
$s[t]_p$, 49
$s|_p$, 49
`::`, 176
$\twoheadrightarrow$, 53
χ_{direct}, 91
χ_{INIT}, 84
χ_{call}, 94
χ_{cata}, 108
χ_{init}, 84
χ_{para}, 118
χ_{split}, 87
χ_{subfn}, 89
χ_{tyfunc}, 116
0, *see* initial object
$\otimes$, 186
$\oplus$, 186
`_`, 181
`()`, 176
`--`, 183
::, 48, 178
`‘_‘`, 179
$\xrightarrow{!}$, 53
$\rightarrow_{\mathcal{R}}$, 53
$\rightarrow_{\rho}$, 52
$\rightarrow$, 53
$l \rightarrow r$, *see* rewrite rule
Σ, 47
1, *see* terminal object
$\boldsymbol{t}$, 50

absorption
    coproducts, 60
    products, 58
`ack`
    description, 134
    synthesised, 245
ADATE, 36
`add`
    description, 134
    synthesised, 246
`addN`
    description, 136
    synthesised, 247
AI, *see* artificial intelligence
AI planning, 131
algebra, 64
`alleven`
    description, 136
    synthesised, 248
`allodd`
    description, 136
    synthesised, 249
`and`
    description, 134
    synthesised, 250
`andL`
    description, 134

synthesised, 251
`append`
description, 135
synthesised, 252
argument, 49
arity, 47
arrow, 55
identity, 56
artificial intelligence, 21
as-pattern, 180
ATP, *see* automated theorem proving
ATRE, 34
automated theorem proving, 24

background
knowledge, 31, 82
specification, 82
`balloons`
description, 137
synthesised, 253
bifunctor, 61
binding, 175
function, 175
group, 175
`Bool`, 176

$\mathcal{C}$, 52
cancellation
catamorphism, 65
coproduct, 59
product, 57
candidate, 85
carrier, 64
case analysis, 59
case-expression, 181
`case of`, 181
$\mathcal{C}at$, 60
category, 55
distributive, 63
of F-algebras, 64
of categories, 60
of sets, 56
theory, 55
`Char`, 177
CLAM, 25
`class`, 182
class declaration, 182

CLEAN, 34
closed, 48
codomain, 55
cognitive psychology, 131
comment, 183
commute, 56
commuting diagram, 56
compatible, 53
complete, 53
composite, 55
composition
of substitutions, 51
operator, 55
`concat`
description, 136
synthesised, 254
conditional, 181
confluent, 53
`Const`, 186
constant, 47
functor, 61
symbol, 47
constructor, 52, 175
symbol, 52
term, 52
constructor system, 52
candidate, 85
open, 82
successor, 85
unfinished, 82
constructor term rewriting
first-order, 51
system, *see* constructor system
context, 50
contractum, 52
coproduct
arrows, 59
correct w.r.t. , 82
costs
of candidates, 85
CS, *see* constructor system
extensional correct, 83
curried, 180

$\mathcal{D}$, 52
data
constructor, 175

declaration, 175
`data`, 175
definition
function, 175
derived class, 182
`deriving`, 182
diagram, 56
DIALOGS-II, 34
distributive category, 63
DJINN, 42
$\mathcal{D}om\,(\cdot)$, 50
domain, 50, 55
`drop`
description, 136
synthesised, 255

`Either` α β, 178
endofunctor, 60
`enjoySport`
description, 137
synthesised, 256
ϵ, 49
`Eq`, 182
`eq`
description, 134
synthesised, 257
equation, 54, 81
ground, 81
ESCHER, 35
evaluate, 175
`even`
description, 134
synthesised, 258
`even, odd`
synthesised, 259
`evenLength`
description, 135
synthesised, 260
`evenParity`
description, 134
synthesised, 261
`evenpos`
description, 135
synthesised, 262
`evens`
description, 136
synthesised, 263
example, 81
export declaration, 183
expression, 175

F-algebra
initial, 64
$\mathcal{A}lg_{\mathsf{F}}$, 64
F-catamorphism, 65
F-fold, 65
F-morphism, 64
`fact`
description, 134
`fact` with `mult`
synthesised, 264
$\mathsf{F}_A f$, 61
$\mathsf{F}(A, X)$, 61
$\mathsf{F}_A X$, 61
FFOIL, 34
`fib`
description, 134
`fib` with `add`
synthesised, 266
`filter`, 185
final object, 57
FLIP, 35
FOIL, 34
`foldr`, 184
fork
of functions, 57
function, 54
composition, 179
definition, 54
head, 54
set, 35
symbol, 47
symbol, defined, 52
type, 178
function call operator, 94
`Functor`, 185
functor, 60
identity, 60
fusion
catamorphism, 65
coproduct, 59
product, 57

G∀ST, 40

`gaussSum`
  description, 134
`gaussSum` with `add`
  synthesised, 267
Genetic Programming, 35
`geq`
  description, 134
  synthesised, 268
Golem, 34
GP, *see* Genetic Programming
greedy rule splitting, 128
guard, 181

`halves`
  description, 135
  synthesised, 269
`hanoi`
  description, 137
  synthesised, 270
Haskell, 34
head, 52
  symbol, 49
`hiding`, 184
HOL, 24
hole, 50
human problem solving, 131
hypotheses space, 31
hypothesis, 30
  language, 30

Id, 60
`Id`, 186
identifier, 175
identity
  arrow, 56
  functor, 60
IFLP, *see* Inductive Functional Logic Programming
`if then else`, 181
Igor I, 37
Igor II, 37
Igor II_H, 127
Igor II^+_C, 127, 138
Igor II^+_P, 127, 138
Igor II^+, 128
ILP, *see* Inductive Logic Programming
`import`, 183
import declaration, 183
`incr`
  description, 136
  synthesised, 271
indexed type families, 186
induction problem, 83
Inductive
  Functional Logic Programming, 35
  Logic Programming, 33
  Program Synthesis, 29
  Programming, 29
inductive bias, 30
in_F, 64
infix notation, 179
`init`
  description, 135
  synthesised, 272
`init, last`
  synthesised, 273
initial
  initial F-algebra, 64
  object, 57
  rule operator, 84
`inits`
  description, 135
  synthesised, 274
injection, 58
`inn`, 186
`inorder`
  description, 137
`inorder` with `append`
  synthesised, 275
input, 81
instance, 51
  declaration, 182
`instance`, 182
`Int`, 177
`Integer`, 177
`intersperse`
  description, 135
  synthesised, 276
IO
  equation, 81
  example, 81
  pattern, 81
IP, *see* Inductive Programming

IPS, *see* Inductive Program Synthesis
ISABELLE, 24

join
  of functions, 59

K., 61

lambda abstraction, 179
language bias, 30
language processing, 131
`last`
  description, 135
  synthesised, 277
`lastM`
  description, 135
  synthesised, 278
`lasts`
  description, 136
  synthesised, 279
LCF, 24
least general generalisation, 51
`Left`, 178
left-hand side, 52
left-linear, 52
`length`
  description, 136
  synthesised, 280
`lengths`
  description, 136
  synthesised, 281
`lenses`
  description, 137
  synthesised, 282
`let`, 181
LGG, *see* least general generalisation
LHS, *see* left-hand side
LHSS, *see* left-hand side
LISP, 34
list type, 177
local variable, 181

machine learning, 21
MAGICHASKELLER, 44
`map`, 185
map-reduce-filter, 184
`mapCons`
  description, 136
  synthesised, 285
mapping
  sorted, 47
`mapTail`
  description, 136
  synthesised, 286
match, 51
MAUDE, 34
`Maybe` α, 178
mediating arrow, 57, 58
MGU, *see* most general unifier
`mirror`
  description, 137
  synthesised, 287
ML, *see* machine learning
ML, 34
module, 183
  export, 183
  import, 183
`module`, 183
more general, *see* subsumption
most general unifier, 51
`Mu`, 186
`mult`
  description, 134
  synthesised, 288
`mult` with `add`
  synthesised, 289
`multfst`
  description, 135
  synthesised, 290
`multlst`
  description, 135
  synthesised, 291

`nandL`
  description, 134
  synthesised, 292
natural
  language processing, 131
  transformation, 62
natural transformation introduction, 116
`negateAll`
  description, 134
  synthesised, 293
`norL`
  description, 134

synthesised, 294
normal form, 53
normalise, 53
normalising, 53
`nthElem`
description, 136
synthesised, 295

object, 55
initial, 57
`odd`
description, 134
synthesised, 296
`oddpos`
description, 135
synthesised, 297
`odds`
description, 136
synthesised, 298
`oddslist`
description, 136
synthesised, 299
one-hole context, 50
one-step reduction relation, 52
operator
direct-call, 91
function call, 94
initial rule, 84
splitting, 87
subfunction, 89
`or`
description, 135
synthesised, 300
`Ord`, 182
`orL`
description, 135
synthesised, 301
orthogonal, 54
`out`, 186
output, 81
overlap, 53
OYSTER, 25

`pack`
description, 135
synthesised, 302
pairing
of functions, 57
paramorphism, 78
pattern, 52
matching, 180
`pepper`
description, 137
synthesised, 303
`pepperF`
description, 137
synthesised, 304
`PF`, 186
pivot position, 86
`playTennis`
description, 137
synthesised, 305
pointfree programming, 180
POLYGP, 36, 38
polymorphic type, 176
polynomial functor, 61
polytypic, 186
$\mathcal{P}os\,(\cdot)$, 49
position, 49
`postorder`
description, 137
`postorder` with `append, snoc`
synthesised, 306
preference bias, 31
prefix notation, 179
`Prelude`, 184
`preorder`
description, 137
`preorder` with `append`
synthesised, 309
primitive recursion operator, 118
problem solving, 131
product
arrows, 58
bifunctor, 61
objects, 58
of objects, 57
PROGOL, 34
projection, 57
Proof Planning, 25

qualified imports, 184

redex, 52

reduct, 53
reduction, 52, 53
    rule, 52
    step, 52
reduction order
    argument-wise, 91
    linear, 91
reduction order, 53
    default, 91
    implementation, 91
reduction sequence, 53
reflection
    catamorphism, 65
    coproduct, 59
    product, 57
renaming, 50
`replicate`
    description, 136
    synthesised, 310
restriction bias, 30
`reverse`
    description, 135
    synthesised, 311
rewrite
    rule, *see* reduction rule
    step, *see* reduction step
$R_{f,p}$, 52
RHS, *see* right-hand side
RHSS, *see* right-hand side
`Right`, 178
right-hand side, 52
rocket, 132
`rocket`
    description, 137
    synthesised, 312
root, 49
root position, 49
rule, 54
    candidate, 85
    covered, 86
    covering, 86
    extensionally correct, 83
    open, 82
    successor, 85
    unfinished, 82

search bias, 31
section, 179
sectioning, 179
`sentence`
    description, 137
    synthesised, 313
sequential covering, 34
$\mathcal{S}et$, 56
set
    sorted, 47
set,terminal, 35
`shiftl`
    description, 135
    synthesised, 314
`shiftl, shiftr`
    synthesised, 315
`shiftr`
    description, 135
    synthesised, 316
`Show`, 182
signature, 47
    many-sorted, 47
SKI calculus, 23
`snoc`
    description, 135
    synthesised, 317
solution, 83
specification, 81
    target, 82
specification subset, 85
`split`
    description, 135
    synthesised, 318
`splitAt`
    description, 136
    synthesised, 319
splitting operator, 87
STGP, *see* Strongly Typed Genetic Programming
Strongly Typed Genetic Programming, 36
structural recursion operator, 108
structure
    of an algebra, 64
`sub`
    description, 134
    synthesised, 321
subfunction, 31

subfunction operator, 89
`subseqs`
    description, 135
`subseqs` with `append`
    synthesised, 322
substitution, 50
    instance, 51
subsumes, 51
subsumption
    substitutions, 51
    terms, 51
subterm, 48–50
    at position, 49
    proper, 50
    replacing at position, 49
successor
    constructor system, 85
    rule, 85
sum
    arrows, 59
    bifunctor, 61
    objects, 58
`sum`
    description, 136
    synthesised, 320
Sussman anomaly, 132
`swap`
    description, 135
    synthesised, 323
`switch`
    description, 135
    synthesised, 324

tactic, 24
`tail`
    description, 135
    synthesised, 325
`tails`
    description, 136
    synthesised, 326
`take`
    description, 136
    synthesised, 327
target, 30, 82
    function, 82
    language, 30
    program, 30
    specification, 82
$\mathcal{T}_\mathcal{C}(\mathcal{X})$, 52
term, 47–48
    closed, 48
    ground, 48
    linear, 48
    size, 49
    syntactic identity, 50
    vector, 50
terminal
    object, 57
terminal set, 35
terminating, 53
terms, 50
THESYS, 37
Towers of Hanoi, 133
transformation, 62
    natural, 62
`transpose`
    description, 136
    synthesised, 328
tuple type, 178
$\mathcal{T}_\Sigma(\mathcal{X})$, 48
type, 175
    class, 182
    constructor, 175
    disjoint sum, 178
    families , *see* indexed type families
    functor, 74
    operator, 186
`type`, 176

Uncurried, 180
unifier, 51
unit type, 176
universal constructions, 56
`unzip`
    description, 136
    synthesised, 329

value, 175
$\mathcal{V}ar\,(\cdot)$, 49
variable, 48, 175
    occurring in term, 49
    renaming, 50
vector, 50

`weave`

description, 136
synthesised, 330
`weaveL`
description, 136
synthesised, 331
well-founded, 53
`where`, 181
wildcard, 181

$\mathcal{X}$, 48

`zeros`
description, 136
synthesised, 332
`zip`
description, 136
synthesised, 333