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

A Combined Analytical and Search-Based Approach to the Inductive Synthesis of Functional Programs

Emanuel Kitzelmann

12. Mai 2010

Promotionskommission:
Prof. Dr. Ute Schmid (1. Gutachter)
Prof. Michael Mendler, PhD (Vorsitzender)
Prof. Dr. Christoph Schlieder

Externer 2. Gutachter:
Prof. Dr. Bernd Krieg-Brückner
(Universität und DFKI Bremen)

Erklärung

Erklärung gemäß §10 der Promotionsordnung der Fakultät Wirtschaftsinformatik und Angewandte Informatik an der Otto-Friedrich-Universität Bamberg:

- Ich erkläre, dass ich die vorgelegte Dissertation selbständig, das heißt auch ohne die Hilfe einer Promotionsberaterin bzw. eines Promotionsberaters angefertigt habe und dabei keine anderen Hilfsmittel als die im Literaturverzeichnis genannten benutzt und alle aus Quellen und Literatur wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

- Ich versichere, dass die Dissertation oder wesentliche Teile derselben nicht bereits einer anderen Prüfungsbehörde zur Erlangung des Doktorgrades vorlagen.

- Ich erkläre, dass diese Arbeit noch nicht in ihrer Gesamtheit publiziert ist. Soweit Teile dieser Arbeit bereits in Konferenzbänden und Journals publiziert sind, ist dies an entsprechender Stelle kenntlich gemacht und die Beiträge sind im Literaturverzeichnis aufgeführt.

Zusammenfassung

Diese Arbeit befasst sich mit der induktiven Synthese rekursiver deklarativer Programme und speziell mit der analytischen induktiven Synthese funktionaler Programme.

Die Programmsynthese beschäftigt sich mit der (semi-)automatischen Konstruktion von Computer-Programmen aus Spezifikationen. In der *induktiven* Programmsynthese werden rekursive Programme durch das Generalisieren über unvollständige Spezifikationen, wie zum Beispiel endliche Mengen von Eingabe/Ausgabe-Beispielen (E/A-Beispielen), generiert. Klassische Methoden der induktiven Synthese funktionaler Programme sind *analytisch*; eine rekursive Funktionsdefinition wird generiert, indem rekurrente Strukturen zwischen den einzelnen E/A-Beispielen gefunden und generalisiert werden. Die meisten aktuellen Ansätze basieren hingegen auf *erzeugen und testen*, das heißt, es werden unabhängig von den bereitgestellten E/A-Beispielen solange Programme einer Klasse generiert, bis schließlich ein Programm gefunden wurde das alle Beispiele korrekt berechnet.

Analytische Methoden sind sehr viel schneller, weil sie nicht auf Suche in einem Programmraum beruhen. Allerdings müssen dafür auch die Schemata, denen die generierbaren Programme gehorchen, sehr viel beschränkter sein.

Diese Arbeit bietet zunächst einen umfassenden Überblick über bestehende Ansätze und Methoden der induktiven Programmsynthese. Anschließend wird ein neuer Algorithmus zur induktiven Synthese funktionaler Programme beschrieben, der den analytischen Ansatz generalisiert und mit Suche in einem Programmraum kombiniert. Dadurch lassen sich die starken Restriktionen des analytischen Ansatzes zu großen Teilen überwinden. Gleichzeitig erlaubt der Einsatz analytischer Techniken das Beschneiden großer Teile des Problemraums, so dass Lösungsprogramme oft schneller gefunden werden können als mit Methoden, die auf *erzeugen und testen* beruhen.

Mittels einer Reihe von Experimenten mit einer Implementation des beschriebenen Algorithmus' werden seine Möglichkeiten gezeigt.

Abstract

This thesis is concerned with the inductive synthesis of recursive declarative programs and in particular with the analytical inductive synthesis of functional programs.

Program synthesis addresses the problem of (semi-)automatically generating computer programs from specifications. In *inductive* program synthesis, recursive programs are constructed by generalizing over incomplete specifications such as finite sets of input/output examples (I/O examples). Classical methods to the induction of functional programs are *analytical*, that is, a recursive function definition is derived by detecting and generalizing recurrent patterns between the given I/O examples. Most recent methods, on the other side, are generate-and-test based, that is, they repeatedly generate programs independently from the provided I/O examples until a program is found that correctly computes the examples.

Analytical methods are much faster than generate-and-test methods, because they do not rely on search in a program space. Therefore, however, the schemas that generatable programs conform to, must be much more restricted.

This thesis at first provides a comprehensive overview of current approaches and methods to inductive program synthesis. Then we present a new algorithm to the inductive synthesis of functional programs that generalizes the analytical approach and combines it with search in a program space. Thereby, the strong restrictions of analytical methods can be resolved for the most part. At the same time, applying analytical techniques allows for pruning large parts of the problem space such that often solutions can be found faster than with generate-and-test methods.

By means of several experiments with an implementation of the described algorithm, we demonstrate its capabilities.

Acknowledgments

This thesis would not exist without support from other people.

First of all I want to thank my supervisor Prof. Ute Schmid that she awakened my interest in the topic of inductive program synthesis when I came to TU Berlin after my intermediate examination, that she encouraged me to publish and to present work at a conference when I was still a computer science student, and that she co-supervised my diploma thesis and finally became my doctoral supervisor. Ute Schmid always allowed me great latitude to comprehensively study my topic and to develop my own contributions to this field as they are now presented in this work. I also want to thank Prof. Fritz Wysotzki, the supervisor of my diploma thesis, for many discussions on the field of inductive program synthesis.

Discussions with my professors Ute Schmid and Fritz Wysotzki, with colleagues in Ute Schmid's group, with students at University of Bamberg, and—at conferences and workshops—with other researchers working on inductive programming, helped me to clarify many thoughts. Among all these people, I especially want to thank Martin Hofmann and further Neil Crossley, Thomas Hieber, Pierre Flener, and Roland Olsson.

I further want to thank Prof. Bernd Krieg-Brückner that he let me present my research in his research group at University of Bremen and that he was willing to be an external reviewer of this thesis.

I finally and especially want to thank my little family, my girlfriend Kirsten and our two children Laurin and Jonna, for their great support and their endless patience during the last years when I worked on this thesis.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

1. Introduction

1.1. Inductive Program Synthesis and Its Applications

Program synthesis research is concerned with the problem of (semi-)automatically deriving computer programs from specifications. There are two general approaches to this end: *Deduction*—reasoning from the general to the particular—and *induction*—reasoning from the particular to the general. In deductive program synthesis, starting point is an (assumed-to-be-)*complete* specification of a problem or function which is then transformed to an executable program by means of logical deduction rules (e.g., [84, 65]). In *inductive* program synthesis (or *inductive programming* for short), which is the topic of this thesis, starting point is an (assumed-to-be)*incomplete* specification. "Incomplete" means that the function to be implemented is specified only on a (small) part of its intended domain. A typical incomplete specification consists of a finite set of input/output examples (I/O examples). Such an incomplete specification is then *inductively generalized* to an executable program that is expected to compute correct outputs also for inputs that were *not* specified.

Especially in inductive program synthesis, induced programs are most often *declarative*, i.e., recursive functional or logic programs.

Example 1.1. Based on the following two equations

```
f ([x,y])        = y
f ([x,y,z,v,w])  = w ,
```

specifying that f shall return the second element of a two-element list and the fifth element of a five-element list, an inductive programming system could induce the recursive function definition

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

computing the last element of given lists of any length ≥ 1. (x and xs denote variables, _:_ denotes the usual algebraic list-constructor "cons".)

There are two general approaches to inductive program synthesis (*IPS*):

1. *Search-* or *generate-and-test based* methods repeatedly generate candidate programs from a program class and test whether they satisfy the provided specification. If a program is found that passes the test, the search stops and the solution program is returned. ADATE [82] and MagicHaskeller [45] are two representative systems of this class.

2. *Analytical* methods, in contrary, *synthesize* a solution program by inspecting a provided set of I/O examples and by detecting recurrent structures in it. Found recurrences are then inductively generalized to a recursive function definition. The classical paper of this approach is Summers' paper on his Thesys system [104]. A more recent system of this class is Igor1 [51].

Both approaches have complementary strengths and weaknesses. Classical analytical methods are fast because they construct programs almost without search. Yet they need well-chosen sets of I/O examples and can only synthesize programs that use small fixed sets of primitives and belong to restricted program schemas like linear recursion. In contrast, generate-and-test methods are in principle able to induce any program belonging to some enumerable set of programs, but due to searching in such vast problem spaces, the synthesis of all but small (toy) programs needs much time or is intractable, actually.[1]

Even though IPS is mostly basic research until now, there are several potential areas of application that have been started to be addressed, among them software-engineering, algorithm development and optimization, end-user programming, and artificial intelligence and cognitive psychology.

Software engineering. In software-engineering, IPS may be used as a tool to semi-automatically generate (prototypical) programs, modules, or single functions. Especially in *test-driven development* [7] where *test-cases* are the starting point of program development, IPS could assist the programmer by considering the test-cases as an incomplete specification and generating prototypical code from them.

Algorithm development and optimization. IPS could be used to invent new algorithms or to improve existing algorithms, for example algorithms for optimization problems where the goal is to efficiently compute approximative solutions for NP-complete problems [82, 8].

End-user programming, programming-by-example. In end-user programming, IPS may help end-users to generate their own small programs or advanced macros by demonstrating the needed functionality by means of examples [62, 36].

Artificial intelligence and cognitive psychology. In the fields of artificial intelligence and cognitive psychology, IPS can be used to model the capability of human-level cognition to obtain general declarative or procedural knowledge about inherently recursive problems from experience [95].

Especially in automated planning [32], IPS can be used to learn general problem-solving strategies in the form of recursive macros from initial planning experience in a

[1]For example, Roland Olsson reports on his homepage (`http://www-ia.hiof.no/~rolando/`), that inducing a function to transpose matrices with ADATE (with only the list-of-lists constructors available as usable primitives, i.e., without any background knowledge) takes 11.6 hours on a 200MHz Pentium Pro.

domain [96, 94]. For example, a planning or problem-solving agent may use IPS methods to derive the recursive strategy for solving arbitrary instances of the *Towers-of-Hanoi* problem from initial experience with instances including three or four discs [95].

This could be an approach to tackle the long-standing and yet open problem of scalability with respect to the number of involved objects in automated planning. When, for example, a planner is able to derive the recursive general strategy for Towers-of-Hanoi from some small problem instances, then the inefficient or even intractable *search* for plans for problem instances containing greater numbers of discs can completely be omitted and instead the plans can be generated by just executing the learned strategy.

1.2. Challenges in Inductive Program Synthesis

In general, inductive program synthesis can be considered as a *search problem*: Find a program in some program class that satisfies a provided specification. In general, the problem space of IPS is very huge—all syntactically correct programs is some computationally complete programming language or formalism, such as, for example, Turing machines, the HASKELL programming language (or a sufficient subset thereof), or term rewriting systems. In particular, the number of programs increases exponentially with respect to their size. Furthermore, it is difficult to generally calculate how changes in a program affect the computed function. Hence it is difficult to develop heuristics that work well for a wide range of domains.

To make these difficulties more clear, let us compare IPS with more standard machine learning tasks—the induction of decision trees [87] and neural networks [90]. In the case of decision trees, one has a fixed finite set of attributes and class values that can be evaluated or tested at the inner nodes and assigned to the leaves, respectively. In the case of neural networks, if the structure of the net is given, defining the net consists in defining a weight vector of fixed length of real numbers. Contrary, in IPS, the object language can in general be arbitrarily extended by defining subprograms or subfunctions or by introducing additional (auxiliary) parameters.[2]

Moreover, in decision-tree learning, statistical measures such as the information gain indicate which attributes are worth to consider at a particular node. In neural nets, the same holds for the gradient of the error function regarding the update of the weights. Even though these measures are heuristic and hence potentially misleading, they are reliable enough to be successfully used in a wide range of domains within a greedy-based search. It is much more difficult to derive such measures in the case of general programs.

Finally, different branches of a decision tree (or different rules in the case of learning *non-recursive* rules) can be developed *independently* from each other, based on their respective subsets of the training data. In the case of *recursive* rules, however, the different (base- or recursive) rules/cases generally interdepend. For example, changing a base case of a recursion not only affects the accuracy or correctness regarding instances or inputs directly covered by that base case but also those instances that are initially

[2]This is sometimes called *bias shift* [106, 101].

evaluated according to some recursive case. This is because each (terminating) evaluation eventually ends with a base case.

1.3. Related Research Fields

As we have already seen for potential application fields, inductive program synthesis has intersections with several other computer science and cognitive science subfields.

In general, IPS lies at the intersection of (declarative) programming, artificial intelligence (AI) [92], and machine learning [69]. It is related with AI by its applicability to AI problems, such as automated planning as described above, but also by the used methods: search, the need for heuristics, (inductive) reasoning to transform programs, and learning.

It is related with machine learning in that a general concept or model, in our case a recursive program, is induced or learned from examples or other kinds of incomplete information. However, there are also significant differences to standard machine learning: Typically, machine learning algorithms are applied to large data sets (e.g., in data mining), whereas the goal in inductive program synthesis is to learn from *few* examples. This is because typically a human is assumed as source of the examples. Furthermore, the training data in standard machine learning is most often noisy, i.e., contains errors and the goal is to learn a model with sufficient (but not perfect) accuracy. In contrary, in IPS the specifications are typically assumed to be error-free and the goal is to induce a program that computes all examples as specified.

By its *objects*, recursive declarative programs, it is related with functional and logic programming, program transformation, and research on computability and algorithm complexity.

Even though *learning theory*[3]—a field at the intersection of theoretical computer science and machine learning, that is concerned with questions such as which kinds of models are learnable under which conditions from which data and with which complexity—has not yet extensively studied general recursive programs as objects to be learned, it can legitimately (and *should* be) considered as a related research field.

1.4. Contributions and Organization of this Thesis

The contributions of this thesis are first, a comprehensive survey and classification of current IPS approaches, theory, and methods; second, the presentation of a new powerful algorithm, called IGOR2, for the inductive synthesis of functional programs; and third, an empirical evaluation of IGOR2 by means of several recursive problems from functional programming and artificial intelligence:

1. Though inductive program synthesis is an active area of research since the seventies, it has not become an established, unified research field since then but is

[3]The two seminal works are [33], where Gold introduces the concept of *identification in the limit* and [107], where Valiant introduces the *PAC* (probably approximately correct) learning model.

scattered over several fields such as artificial intelligence, machine learning, inductive logic programming, evolutionary computation, and functional programming. Until today, there is no uniform body of IPS theory and methods; furthermore, no survey of recent results exists. This fragmentation over different communities impedes the exchange of results and leads to redundancies.

Therefore, this thesis at first provides a comprehensive overview of existing approaches to IPS, theoretical results and methods, that have been developed in different research fields until today. We discuss strengths and weaknesses, similarities and differences of the different approaches and draw conclusions for further research.

2. We present the new IPS algorithm IGOR2 for the induction of functional programs in the framework of term rewriting. IGOR2 generalizes the classical analytical recurrence-detection approach and combines it with search in a program space in order to allow for inducing more complex programs in reasonable time. We precisely define IGOR2's synthesis operators, prove termination and completeness of its search strategy, and prove that programs induced by IGOR2 correctly compute the specified I/O examples.

3. By means of standard recursive functions on natural numbers, lists, and matrices, we empirically show IGOR2's capabilities to induce programs in the field of functional programming. Furthermore, we demonstrate IGOR2's capabilities to tackle problems from artificial intelligence and cognitive psychology at hand of learning recursive rules in some well-known domains like the *blocksworld* or the *Towers-of-Hanoi*.

The thesis is mainly organized according to the three contributions:
In the following chapter (2), we at first introduce basic concepts of algebraic specification, term rewriting, and predicate logic, as they can be found in respective introductory textbooks.

Chapter 3 then contains the overview over current approaches to inductive program synthesis. That chapter mostly summarizes research results from *other* researchers than the author of this thesis. A few exceptions are the following: In Section 3.2.3, we shortly review the IPS system IGOR1 that was co-developed by the author of this thesis. Furthermore, the arguments in the discussions at the end of each section as well as the conclusions at the end of the chapter, pointing out characteristics and relations of the different approaches, are worked out by the author of this thesis. Finally, the consideration regarding positive and negative examples in inductive logic programming and inductive functional programming (at the beginning of Section 3.3.1) is from the author of this thesis.

In Chapter 4, we present the IGOR2 algorithm, developed by the author of this thesis, that induces functional programs in the term rewriting framework. We precisely define its synthesis operators and prove some properties of the algorithm.

In Chapter 5, we evaluate a prototypical implementation of IGOR2 at hand of several recursive functions from the domains of functional programming and artificial intelligence.

In Chapter 6 we conclude.

One appendix lists the complete specification files used for the experiments of Chapter 5.

2. Foundations

In the present thesis, we are concerned with functional and logic programs. In this chapter, we define their syntax and semantics by means of concepts from algebraic specification, term rewriting, and predicate logic. Syntactically, a functional program is then a set of equations over a first-order algebraic signature; a logic program is a set of definite clauses. Denotationally, we interpret a functional program as an algebra and a logic program as a logical structure—the denoted algebra and structure are uniquely defined as the *quotient algebra* and the *least Herbrand model* of the equations and definite clauses, respectively. Operationally, the equations defining a functional program are interpreted as a *term rewriting system* and the definite clauses of a logic program are subject to *(SLD-)resolution*. Under certain conditions, denotational and operational semantics agree in both cases—the *canonical term algebra* defined by a set of equations representing a *terminating and confluent* term rewriting system is isomorphic to the quotient algebra and the ground atoms derivable by SLD-resolution from a set of definite clauses is equal to the least Herbrand model.

All introduced concepts are basic concepts from algebraic specification, term rewriting, and predicate logic and can be found more detailed in respective textbooks such as [24] (algebraic specification), [6, 105] (term rewriting), and [98] (predicate logic). We do not provide any proofs here. They can also be found in respective textbooks.

2.1. Preliminaries

We write $\mathbb{N}$ for the set of natural numbers including 0 and $\mathbb{Z}$ for the set of integers. By $[m]$ we denote the subset $\{n \in \mathbb{N} \mid 1 \leq n \leq m\}$ of all natural numbers from 1 to m.

A *family* is a mapping $I \to X : i \mapsto x_i$ from an (index) set I to a set X, written $(x_i)_{i \in I}$ or just (x_i).

Given any set X, by *id* we denote the identity function on X; $id : X \to X : x \mapsto x$.

An *equivalence relation* is a reflexive, symmetric, and transitive relation on a set X, denoted by $\sim$ or $\equiv$. One often writes $x \sim y$ instead of $(x, y) \in \sim$. By $[x]_\sim$ we denote the equivalence class of x by $\sim$, i.e., the set $\{y \in X \mid x \sim y\}$. The set of all equivalence classes of X by $\sim$ is called the *quotient set* of X by $\sim$, written $X/\sim$. It is a partition on X.

By $|\, X \,|$, we denote the cardinality of the set X. By $\mathfrak{P}(X)$, we denote the power set of the set X.

By $Dom(f)$ we denote the domain of a function f.

By $\mathcal{X}$ we denote an countable set whose elements are called *variables*.

Given a set S, we write S^* for the set of finite (including empty) sequences $s_1, \ldots, s_n$ of elements of S. If $n = 0$, $s_1, \ldots, s_n$ denotes the empty sequence, ϵ.

2.2. Algebraic Specification and Term Rewriting

2.2.1. Algebraic Specification

We shortly review some basic concepts and results (without proofs) of algebraic specification in this section, as, for example, described in [24].

Algebraic Signatures and Algebras

Algebras are sets of values, called *carrier sets* or *universes*, together with mathematical functions defined on them. The functions have names, called *function symbols*, and are collected in an *algebraic signature.*

Definition 2.1 (Algebraic signature)**.** An *algebraic signature* is a set Σ whose elements are called *function symbols.* Each function symbol $f \in \Sigma$ is associated with a natural number, called the *arity of* f, written $\alpha(f)$, which denotes the number of arguments f takes.

Function symbols of arity 0 are called *constants.* Function symbols of arity one and two are called unary and binary, respectively. In general, we speak of n-ary function symbols.

An algebraic signature Σ is interpreted by a Σ-algebra that fixes a set of data objects or values and assigns to each function symbol a function on the chosen universe.

Definition 2.2 (Σ-algebra)**.** Let Σ be an algebraic signature. A Σ-*algebra* $\mathcal{A}$ consists of

- a (possibly empty) set A, called *carrier set* or *universe*, and
- for each $f \in \Sigma$, a total function $f_{\mathcal{A}} : A^{\alpha(f)} \to A$.

Remark 2.1 (Constant functions)*.* If $\alpha(f) = 0$ for an $f \in \Sigma$, then $A^{\alpha(f)} = A^0 = \{\langle\rangle\}$. In this case, $f_{\mathcal{A}}$ is a constant function denoting the value $f_{\mathcal{A}}(\langle\rangle)$ which is simply written as $f_{\mathcal{A}}$.

Parenthesis: The many-sorted case. Typically, functional programs are *typed.* The overall universe of values is partitioned (or many-sorted) and each function is defined only on a specified subset of (a product of) the whole universe and also has values only in a specified subset.

Strong typing assures at compile-time that functions will only be called on appropriate inputs. In inductive program synthesis, typing is also useful to prune the problem space because it restricts the number of allowed expressions.

In the rest of this parenthesis we define many-sorted algebraic signatures and algebras and give an example. Afterwards we proceed with the unsorted setting because the many-sorted setting heavily bloats the notation of concepts while they essentially remain the same and are easily lifted to the many-sorted setting.

Table 2.1.: A many-sorted algebraic signature Σ and a Σ-algebra $\mathcal{A}$

Σ	$\mathcal{A}$
Sorts	**Universes**
Nat	$\mathbb{N} \cup \{\bot\}$
NatList	(Lists[a] of $\mathbb{N}$) $\cup \{\bot\}$
Function symbols	**Functions**
$z : \mathit{Nat}$	0
$s : \mathit{Nat} \to \mathit{Nat}$	$s_{\mathcal{A}}(n) = \begin{cases} n+1 & \text{if } n \in \mathbb{N} \\ \bot & \text{if } n = \bot \end{cases}$
$\mathit{nil} : \mathit{NatList}$	$()$
$\mathit{cons} : \mathit{Nat}, \mathit{NatList} \to \mathit{NatList}$	$\mathit{cons}_{\mathcal{A}}(\bot, l) = \mathit{cons}_{\mathcal{A}}(e, \bot) = \mathit{cons}_{\mathcal{A}}(\bot, \bot) = \bot$, $\mathit{cons}_{\mathcal{A}}(e_0, (e_1, \ldots, e_n)) = (e_0, e_1, \ldots, e_n)$[b]
$\mathit{Last} : \mathit{NatList} \to \mathit{Nat}$	$\mathit{Last}_{\mathcal{A}}(\bot) = \bot, \mathit{Last}_{\mathcal{A}}((e_1, \ldots, e_n)) = \begin{cases} \bot & \text{if } n = 0^b \\ e_n & \text{if } n > 0 \end{cases}$

[a]Including the empty list $()$.
[b]The sequences $e_1, \ldots, e_n$ may be empty, i.e., $n = 0$. We then have $\mathit{cons}_{\mathcal{A}}(e_0, ()) = (e_0)$ and $\mathit{Last}_{\mathcal{A}}(()) = \bot$.

Definition 2.3 (Many-sorted algebraic signature)**.** A *many-sorted algebraic signature* is a pair $\Sigma = \langle S, \mathit{OP} \rangle$ where

- S is a set whose elements are called *sorts*, and
- $\mathit{OP} = (\mathit{OP}_{\langle w;s \rangle})$ is an $(S^* \times S)$-indexed family of *sets of function symbols*.

For $f \in \mathit{OP}_{\langle s_1, \ldots, s_n; s \rangle}$ we also write $f : s_1, \ldots, s_n \to s$. If $f \in \mathit{OP}_{\langle \epsilon; s \rangle}$, we write $f : s$ and call f a *constant*.

Definition 2.4 (Many-sorted Σ-algebra)**.** Let $\Sigma = \langle S, \mathit{OP} \rangle$ be a many-sorted algebraic signature. A many-sorted Σ-*algebra* $\mathcal{A}$ consists of

- an S-indexed family of sets $A = (A_s)_{s \in S}$, where the sets A_s are called *carrier sets* or *universes*, and
- for each $f : s_1, \ldots, s_n \to s$, a total function $f_{\mathcal{A}} : A_{s_1} \times \cdots \times A_{s_n} \to A_s$.

Table 2.1 shows an example of a (many-sorted) algebraic signature Σ and a Σ-algebra $\mathcal{A}$.

We continue with the unsorted setting. In the following (throughout Section 2.2), Σ always denotes an algebraic signature and instead of *algebraic signature*, we may just say *signature*.

An algebraic signature Σ only states that a Σ-algebra includes a particular set of functions. *Terms*—words built over the signature and a set of *variables* (and some punctuation symbols)—reflect, on the syntactic side, the composition of such functions. Terms are thus the basic means to define properties of algebras.

Definition 2.5 (Terms, Herbrand universe)**.** Let Σ be a signature and $\mathcal{X}$ be an countable set whose elements are called *variables.* Then the set of Σ*-terms over* $\mathcal{X}$ (*terms* for short), denoted by $T_\Sigma(\mathcal{X})$, is defined as the smallest set satisfying the following conditions:

- Each variable $x \in \mathcal{X}$ is in $T_\Sigma(\mathcal{X})$.
- If $f \in \Sigma$ and $t_1, \dots, t_{\alpha(f)} \in T_\Sigma(\mathcal{X})$, then $f(t_1, \dots, t_{\alpha(f)}) \in T_\Sigma(\mathcal{X})$. (For constants $f \in \Sigma$ we write f instead of $f()$.)

We denote the set of variables occurring in a term t by $\mathit{Var}(t)$. Terms without variables ($\mathit{Var}(t) = \emptyset$) are called *ground terms*. The subset of $T_\Sigma(\mathcal{X})$ exactly including all ground terms is denoted by T_Σ and called the *Herbrand universe* of Σ. Ground terms only exist, if the signature contains at least one constant symbol.

Given an algebra, a ground term denotes a particular composition of functions and constants and hence a value of the universe. If a term contains variables, the denoted value depends on an assignment of values to variables. Formally:

Definition 2.6 (Term evaluation, variable assignment)**.** Let $\mathcal{A}$ be a Σ-algebra with universe A and $\mathcal{X}$ be a set of variables. The *meaning of a term* $t \in T_\Sigma(\mathcal{X})$ *in* $\mathcal{A}$ is given by a function $\beta^* : T_\Sigma(\mathcal{X}) \to A$ satisfying the following property for all $f \in \Sigma$:

$$\beta^*(f(t_1, \dots, t_n)) = f_{\mathcal{A}}(\beta^*(t_1), \dots, \beta^*(t_n)) .$$

Such a term evaluation function is uniquely determined if it is defined for all variables. A function $\beta : \mathcal{X} \to A$, uniquely determining β^*, is called *variable assignment* (or just *assignment*).

Table 2.2 shows some terms, variable assignments and evaluations according to Σ and $\mathcal{A}$ of Table 2.1.

Presentations and Models

In algebraic specification, properties of algebras are defined in terms of *equations.*

Definition 2.7 (Σ-equation, presentation)**.** A Σ-equation is a pair of two Σ-terms, $\langle t, t' \rangle \in T_\Sigma(\mathcal{X}) \times T_\Sigma(\mathcal{X})$, written $t = t'$.

A *presentation* (also called *algebraic specification*) is a pair $P = \langle \Sigma, \Phi \rangle$ of a signature Σ and a set Φ of Σ-equations, called the *axioms* of P.

A Σ-equation $t = t'$ states the requirement to Σ-algebras that for all variable assignments, both terms t and t' evaluate to the same value. Such an algebra is said to *satisfy* an equation. An algebra that satisfies all equations in a presentation is a *model* of the presentation.

Definition 2.8 (Satisfies, model, loose semantics)**.** A Σ-algebra $\mathcal{A}$ with universe A *satisfies* a Σ-equation $t = t' \in T_\Sigma(\mathcal{X}) \times T_\Sigma(\mathcal{X})$, written

$$\mathcal{A} \models t = t' ,$$

Table 2.2.: Example terms, variable assignments, and evaluations according to Σ and $\mathcal{A}$ of Table 2.1

$t \in T_\Sigma(\{x, y\})$	β^a	$\beta^*(t)$
z		0
$s(z)$		1
$s(s(s(s(z))))$		4
nil		$()$
$cons(s(s(z)), cons(z, cons(s(s(s(s(z)))), nil)))$		$(2, 0, 4)$
x	$x \mapsto 5$	5
$s(s(x))$	$x \mapsto 5$	7
$cons(z, x)$	$x \mapsto (1, 2)$	$(0, 1, 2)$
$cons(z, cons(x, cons(y, nil)))$	$x \mapsto 1, y \mapsto 2$	$(0, 1, 2)$

[a] We only display values of variables actually occurring in the particular terms.

iff for every assignment $\beta : \mathcal{X} \to A$, $\beta^*(t) = \beta^*(t')$.

A *model* of a presentation $P = \langle \Sigma, \Phi \rangle$ is a Σ-algebra $\mathcal{A}$ such that for all $\varphi \in \Phi$, $\mathcal{A} \models \varphi$; we write $\mathcal{A} \models \Phi$. The class of all models of P, denoted by $Mod(P)$, is called the *loose semantics* of P.

Remark 2.2. Note that the symbol '=' has two different roles in the previous definition. It is (i) a syntactic item to construct equations and it denotes (ii) identity on a universe.

Example 2.1. Consider the following set Φ of Σ-equations over variables $\{x, y, xs\}$ where Σ is the example signature of Table 2.1:

$$\begin{aligned} Last(cons(x, nil)) &= x\,, \\ Last(cons(x, cons(y, xs))) &= Last(cons(y, xs))\,. \end{aligned}$$

$\mathcal{A}$ of Table 2.1 is a model of $\langle \Sigma, \Phi \rangle$. Now suppose that a Σ-algebra $\mathcal{A}'$ is identical to $\mathcal{A}$ except for the following redefinition of *Last*:

$$Last_{\mathcal{A}'}(e_1, \ldots, e_n) = \begin{cases} \bot & \text{if } n = 0 \\ e_1 & \text{if } n > 0 \end{cases}.$$

I.e., $Last_{\mathcal{A}'}$ denotes the *first* element of a list instead of the last one as in $\mathcal{A}$. Then $\mathcal{A}'$ is *not* a model of $\langle \Sigma, \Phi \rangle$, because, for example,

$$\beta^*(Last(cons(x, cons(y, xs)))) = 1 \neq 2 = \beta^*(Last(cons(y, xs)))$$

with $\beta(x) = 1, \beta(y) = 2, \beta(xs) = ()$.

If an equation φ is satisfied by all models of a set of equations Φ, this means, that whenever Φ states true properties of a particular algebra, also φ does. Such an equation φ is called a *semantic consequence* of Φ.

Definition 2.9 (Semantic consequence)**.** A Σ-equation φ is a *semantic consequence* of a set of Σ-equations Φ (or, equivalently, of the presentation $\langle\Sigma, \Phi\rangle$), if for all $\mathcal{A} \in \mathit{Mod}(\langle\Sigma, \Phi\rangle)$, $\mathcal{A} \models \varphi$. We write $\Phi \models \varphi$ in this case.

Example 2.2. The equation $\mathit{Last}(\mathit{cons}(x, \mathit{cons}(y, \mathit{cons}(z, \mathit{nil})))) = \mathit{Last}(\mathit{cons}(z, \mathit{nil}))$ is a semantic consequence of the equations of Example 2.1.

Definition 2.10 (Theory)**.** A set of equations Φ is *closed under semantic consequences*, iff $\Phi \models \varphi$ implies $\varphi \in \Phi$. We may close a non-closed set of equations by adding all its semantic consequences, denoted by $\mathit{Cl}(\Phi)$.

A *theory* is a presentation $\langle\Sigma, \Phi\rangle$ where Φ is closed under semantic consequences. A presentation $\langle\Sigma, \Phi\rangle$, where Φ need not to be closed, *presents* the theory $\langle\Sigma, \mathit{Cl}(\Phi)\rangle$.

Initial Semantics

The several models of a presentation might be quite different regarding their universes and the behavior of their operations. Two critical characteristics of models are *junk* and *confusion*, defined as follows.

Definition 2.11 (Junk and confusion)**.** Let $P = \langle\Sigma, \Phi\rangle$ be a presentation and $\mathcal{A}$ be a model with universe A of P.

Junk If there are elements $a \in A$ that are not denoted by some ground term, i.e., there is no ground term t with $\beta^*(t) = a$, $\mathcal{A}$ is said to contain *junk*.

Confusion If $\mathcal{A}$ satisfies ground equations that are not in the theory presented by P, i.e., there are terms $t, t' \in T_\Sigma$ such that $\mathcal{A} \models t = t'$ but $t = t' \notin \langle\Sigma, \mathit{Cl}(\Phi)\rangle$, $\mathcal{A}$ is said to contain *confusion*.

In order to define the stronger *initial semantics*, particularly including only models without junk and confusion, we need a certain concept of function between universes of algebras to relate algebras regarding their structure as induced by their operations. A *homomorphism* is a function h between universes A and B of algebras $\mathcal{A}$ and $\mathcal{B}$, respectively, such that if h maps elements $a_1, \ldots, a_n \in A$ to elements $b_1, \ldots, b_n \in B$, then for all n-ary functions it maps $f_\mathcal{A}(a_1, \ldots, a_n)$ to $f_\mathcal{B}(b_1, \ldots, b_n)$.

Definition 2.12 (Homomorphism, Isomorphism)**.** Let $\mathcal{A}$ and $\mathcal{B}$ be two Σ-algebras with universes A and B, respectively. A Σ-*homomorphism* $h : \mathcal{A} \to \mathcal{B}$ is a function $h : A \to B$ which respects the operations of Σ, i.e., such that for all $f \in \Sigma$,

$$h(f_\mathcal{A}(a_1, \ldots, a_{\alpha(f)})) = f_\mathcal{B}(h(a_1), \ldots, h(a_{\alpha(f)}))\,.$$

A Σ-homomorphism is a Σ-*isomorphism* if it has an inverse, i.e., if there is a Σ-homomorphism $h^{-1} : \mathcal{B} \to \mathcal{A}$ such that $h \circ h^{-1} = id_A$ and $h^{-1} \circ h = id_B$. In this case, $\mathcal{A}$ and $\mathcal{B}$ are called *isomorphic*, written $\mathcal{A} \cong \mathcal{B}$.

A homomorphism $h : \mathcal{A} \to \mathcal{B}$ is an isomorphism if and only if $h : A \to B$ is bijective. If two algebras are isomorphic, the only possible difference is the particular choice of universe elements. The size of their universes as well as the behavior of their operations are identical. Hence, if two algebras are isomorphic, often each one is considered as good as the other and we say that they are identical *up to isomorphism*.

Now we are able to define the *initial semantics* of a presentation.

Definition 2.13 (Initial algebra)**.** Let $\mathcal{A}$ be a Σ-algebra and $\mathfrak{A}$ be a class of Σ-algebras. $\mathcal{A}$ *is initial in* $\mathfrak{A}$ if $\mathcal{A} \in \mathfrak{A}$ and for every $\mathcal{B} \in \mathfrak{A}$ there is a unique Σ-homomorphism $h : \mathcal{A} \to \mathcal{B}$.

Definition 2.14 (Initial semantics)**.** Let $P = \langle \Sigma, \Phi \rangle$ be a presentation and $\mathcal{A}$ be a Σ-algebra. If $\mathcal{A}$ is initial in $\mathit{Mod}(P)$ then $\mathcal{A}$ is called an *initial model* of P. The class of all initial models is called the *initial semantics* of P.

An initial model is a model which is structurally contained in each other model. The class of all initial models has two essential properties: First, all initial models are isomorphic. That is, the initial semantics appoint a unique (up to isomorphism) model of a presentation. Second, as already mentioned above, the initial models are exactly those without junk and confusion.

There is a standard initial model for presentations, which we will now construct. Though terms are per se syntactic constructs and need to be interpreted, we may take T_Σ as universe of a particular algebra $\mathcal{T}_\Sigma$, called *ground term algebra*. The functions of the ground term algebra apply function symbols to terms, hence *construct* the ground terms.

Definition 2.15 (Ground term algebra)**.** The *ground term algebra* of signature Σ, written $\mathcal{T}_\Sigma$, is defined as follows:

- The universe is the Herbrand universe, T_Σ.
- For $f \in \Sigma$, $f_{\mathcal{A}}(t_1, \dots, t_{\alpha(f)}) = f(t_1, \dots, t_{\alpha(t)})$.

The ground term algebra of signature Σ, as any other Σ-algebra, is a model of the special, trivial presentation containing *no* axioms, $P_0 = \langle \Sigma, \emptyset \rangle$.

Now reconsider the term evaluation function β^* (Definition 2.6). It is a function from $T_\Sigma(\mathcal{X})$ to the universe A of some Σ-algebra $\mathcal{A}$ that exhibits the homomorphism property. That is, β^* restricted to *ground* terms is a homomorphism from $\mathcal{T}_\Sigma$ to $\mathcal{A}$. Moreover, it is the only homomorphism from $\mathcal{T}_\Sigma$ to $\mathcal{A}$ and hence, $\mathcal{T}_\Sigma$ is an *initial* model of P_0.

If a presentation contains axioms identifying universe elements denoted by some different ground terms, then, certainly, the ground term algebra is *not* a model of that presentation. This is because in $\mathcal{T}_\Sigma$, ground terms evaluate to themselves, $\beta^*(t) = t$ for each $t \in T_\Sigma$, such that $\beta^*(t) \neq \beta^*(t')$ for any two different $t, t' \in T_\Sigma$. The solution for this case is to partition T_Σ such that all ground terms identified by the axioms are in one subset each. Taking the partition as universe and defining the functions accordingly leads to the *quotient term algebra*, the standard initial model of presentations.

Definition 2.16 (Quotient algebra)**.** A Σ-*congruence* on a Σ-algebra $\mathcal{A}$ with universe A is an equivalence $\sim$ on A which respects the operations of Σ, i.e., such that for all $f \in \Sigma$ and $a_1, a_1', \ldots, a_{\alpha(f)}, a_{\alpha(f)}' \in A$,

$$a_1 \sim a_1', \ldots a_{\alpha(f)} \sim a_{\alpha(f)}' \text{ implies } f_{\mathcal{A}}(a_1, \ldots, a_{\alpha(f)}) \sim f_{\mathcal{A}}(a_1', \ldots, a_{\alpha(f)}') \,.$$

Let $\sim$ be a Σ-congruence on $\mathcal{A}$. The *quotient algebra of $\mathcal{A}$ modulo* $\sim$, denoted by $\mathcal{A}/\!\sim$, is defined as follows:

- The universe of $\mathcal{A}/\!\sim$ is the quotient set $A/\!\sim$.
- For all $f \in \Sigma$ and $a_1, \ldots, a_{\alpha(f)} \in A$, $f_{\mathcal{A}/\sim}([a_1]_\sim, \ldots, [a_{\alpha(f)}]_\sim) = [f_{\mathcal{A}}(a_1, \ldots, a_{\alpha(f)})]_\sim$.

$\mathcal{A}/\!\sim$ is a Σ-algebra.

Definition 2.17 (Quotient term algebra)**.** Let $P = \langle \Sigma, \Phi \rangle$ be a presentation. The relation $\sim_\Phi \subseteq T_\Sigma \times T_\Sigma$ is defined by $t \sim_\Phi t'$ iff $\Phi \models t = t'$ for all $t, t' \in T_\Sigma$. $\sim_\Phi$ is a Σ-congruence on $\mathcal{T}_\Sigma$ and called the Σ-*congruence generated by* Φ. The *quotient algebra of $\mathcal{T}_\Sigma$ modulo* $\sim_\Phi$, $\mathcal{T}_\Sigma/\!\sim_\Phi$, is called the *quotient term algebra* of P.

Quotient term algebras $\mathcal{T}_\Sigma/\!\sim_\Phi$ are initial models of the corresponding presentations $P = \langle \Sigma, \Phi \rangle$.

2.2.2. Term Rewriting

The concepts of this section are described more detailed in term-rewriting textbooks such as [6, 105].

Preliminaries

A *context* is a term over an extended signature $\Sigma \cup \{\Box\}$, where $\Box$ is a special constant symbol not occurring in Σ. The occurrences of the constant $\Box$ denote empty places, or *holes*, in a context. If C is a context containing exactly n holes, and $t_1, \ldots, t_n$ are terms, then $C[t_1, \ldots, t_n]$ denotes the result of replacing the holes of C from left to right by $t_1, \ldots, t_n$. A context C containing exactly one hole is called *one-hole context* and denoted by $C[\,]$. If $t = C[s]$, then s is called a *subterm* of t. Since with the *trivial context* $C = \Box$, each term t may be written as $C[t]$, for each term t holds that t itself is a subterm of t. All subterms of t except for t itself are also called *proper* subterms.

A *position* (of a term) is a (possibly empty) sequence of positive integers. The set of positions of a term t, denoted by $Pos(t)$, is defined as follows: If $t = x \in \mathcal{X}$, i.e., t is a variable, or t is a constant, then $Pos(t) = \{\epsilon\}$, where ϵ denotes the empty sequence. If $t = f(t_1, \ldots, t_n)$, then $Pos(t) = \{\epsilon\} \cup \bigcup_{i=1}^{n} \{i.p \mid p \in Pos(s_i)\}$. Positions p of a term t denote subterms $t|_p$ of it as follows: $t|_\epsilon = t$ and $f(t_1, \ldots, t_n)|_{i.p} = s_i|_p$. By $Node(t, p)$ we refer to the root symbol of the subterm $t|_p$.

A term is called *linear*, if no variable occurs more than once in it.

The syntactic counterpart of a variable assignment and term evaluation is the replacement of variables (in a term) with terms, called *substitution.*[1] That is, a substitution is a mapping from variables to terms that is uniquely extended to a mapping from terms to terms:

Definition 2.18 (Substitution)**.** A *substitution* is a mapping from terms to terms, $\sigma : T_\Sigma(\mathcal{X}) \to T_\Sigma(\mathcal{X})$, written in postfix notation, which satisfies the property

$$f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$$

(for constants, $c\sigma = c$).

A substitution is uniquely defined by its restriction to the set $\mathcal{X}$ of variables. Application of a substitution to variables is normally written in standard prefix notation, $\sigma(x)$. Most often, we are interested in substitutions with $\sigma(x) \neq x$ for only a *finite subset* of all variables. In such a case, a substitution is determined by its restriction to this subset and typically defined extensionally, $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. By $Dom(\sigma)$ we refer to this finite subset.

A composition of two substitutions is again a substitution. Since substitutions are written postfixed, the composition of two substitutions σ and τ, $\sigma \circ \tau$, is written $\tau\sigma$. Let γ be a further substitution and t be a term. Substitutions satisfy the properties (i) $t(\tau\sigma) = (t\tau)\sigma$, i.e., applying a substitution composition $\tau\sigma$ to a term t is equivalent to applying first τ to t and then σ to the result, and (ii) $\gamma(\tau\sigma) = (\gamma\tau)\sigma$, i.e., composition of substitutions is associative. A substitution which maps distinct variables to distinct variables, i.e., which is injective and has a set of variables as range, is called *(variable) renaming.*

Definition 2.19 (Subsumption, unification)**.** If $s = t\sigma$ for two terms s, t and a substitution σ, then s is called an *instance* of t. We write $t \succeq s$ and say that t *subsumes* s, that t is *more general than* s, that, conversely, s *matches* t, and that s is *more specific than* t.

If $s\sigma = t\sigma$ for two terms s, t and a substitution σ, then we say that s and t *unify.* The substitution σ is called a *unifier.*

The relation $\succeq$ is a quasi-order on terms, called *subsumption order.* If $t \succeq s$ but not $s \succeq t$, then we write $t \succ s$, call s a *proper instance* of t, and say that t is *strictly more general than* s and that s is *strictly more specific than* t.

Definition 2.20 (Least general generalization)**.** Let $T \subseteq T_\Sigma(\mathcal{X})$ be a finite set of terms. Then there is a least upper bound with respect to the subsumption order $\succeq$ of T in $T_\Sigma(\mathcal{X})$, i.e., a least general term t such that all terms in t are instances of t. The term t is called *least general generalization (LGG)* of T, written $\mathbf{lgg}(T)$ [85].

[1] The comparison of assignments and substitutions is not perfectly appropriate, because the former assigns a particular value to a variable, which corresponds to a substitution with a *ground* term. Substitutions, though, may also be non-ground.

An LGG t of a set of terms $\{t_1, \dots, t_n\}$ is equal to each of the t_i at each position where the t_i are all equal. On positions, where at least two of the t_i differ, t contains a variable.

LGGs are unique up to variable renaming and computable. The procedure of generating LGGs is called *anti-unification*.

Example 2.3 (Least general generalization)**.** Let x_1, x_2, x_3, x_4 be variables and f, g, h, r, a, c be function symbols and constants. Let $f(a, g(h(x_1), c), h(x_1))$ and $f(a, g(r(a), x_2), r(a))$ be two terms. Their LGG is $f(a, g(x_3, x_4), x_3)$.

Term Rewriting Systems

Definition 2.21 (Rewrite rule, term rewriting system)**.** A Σ-*rewrite rule* (or just *rule*) is a pair $\langle l, r\rangle \in T_\Sigma(\mathcal{X}) \times T_\Sigma(\mathcal{X})$ of terms, written $l \to r$. We may want to name or label a rule, then we write $\rho : l \to r$. The term l is called *left-hand side (LHS)*, r is called *right-hand side (RHS)* of the rule. Typically, the set of allowed rules is restricted as follows: (i) The LHS l may not consist of a single variable; (ii) $\mathit{Var}(r) \subseteq \mathit{Var}(l)$.

A *term rewriting system (TRS)* is a pair $\langle \Sigma, R\rangle$ where R is a set of Σ-rules.

We can easily extend the concepts of substitution, subsumption, and least general generalization from terms to rules. In particular, by $(l \to r)\sigma$ we mean $l\sigma \to r\sigma$. We say that a rule r subsumes a rule r', if there is a substitution σ such that $r\sigma = r'$. And the LGG of a set R of rules is the least upper bound of R in the set of all rules with respect to the subsumption order.

Except for the two constraints regarding allowed rules, TRSs and presentations are syntactically identical—they consist of an algebraic signature Σ together with a set of pairs of Σ-terms, called rules or equations. They differ regarding their semantics. While an equation denotes *identity*, i.e., a *symmetric* relation, a rule denotes a *directed*, *non-symmetric* relation; or, while equations denotationally define functions, programs, or data types, rules define *computations*.

Rewriting or *reduction* means to repeatedly replace instances of LHSs by instances of RHSs within arbitrary contexts. The two restrictions (i) and (ii) in the definition above avoid the pathological cases of arbitrarily applicable rules and arbitrary subterms in replacements, respectively.

Definition 2.22 ((One-step) rewrite relation of a rule and a TRS)**.** Let $\rho : l \to r$ be a rewrite rule, σ be a substitution, and $C[\,]$ be a one-hole context. Then

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

is called a *rewrite step* according to ρ. The *one-step rewrite relation generated by* ρ, $\to_\rho \subseteq T_\Sigma(\mathcal{X}) \times T_\Sigma(\mathcal{X})$, is defined as the set of all rewrite steps according to ρ.

Let R be a TRS. The *one-step rewrite relation generated by* R is

$$\to_R = \bigcup_{\rho \in R} \to_\rho \ .$$

The *rewrite relation generated by* R, $\overset{*}{\rightarrow}_R$, is the reflexive, transitive closure of $\rightarrow_R$. Hence, $t_0 \overset{*}{\rightarrow}_R t_n$ if and only if $t_0 = t_n$ or $t_0 \rightarrow_R t_1 \rightarrow_R \cdots \rightarrow_R t_n$.

We may omit indexing the arrow by a rule- or TRS name if it is clear from the context or irrelevant, and just write: $\rightarrow$.

Terminology 2.1 (Instance, redex, contractum, reduct, normal form). For a rule $\rho : l \rightarrow r$ and a substitution σ, $l\sigma \rightarrow r\sigma$ is called an *instance* of ρ. Its LHS, $l\sigma$, is called *redex* (*red*ucible *ex*pression), its RHS is called *contractum.* Replacing a redex by its contractum is called *contracting* the redex.

If $t_0 \overset{*}{\rightarrow} t_n$, t_n is called a *reduct* of t_0. The (possibly infinite) concatenation of reduction steps $t_0 \rightarrow t_1 \rightarrow \ldots$ is called *reduction.* If t does not contain any redex, i.e., there is no t' with $t \rightarrow t'$, t is called *normal form.* If t_n is a reduct of t_0 and t_n is a normal form, t_n is called a normal form of t_0 and t_0 is said to have t_n as normal form.

Definition 2.23 (Termination, confluence, completeness)**.** Let R be a TRS. R is *terminating*, if there are no infinite reductions, i.e., if for every reduction $t_0 \rightarrow_R t_1 \rightarrow_R \ldots$ there is an $n \in \mathbb{N}$ such that t_n is a normal form. R is *confluent*, if each two reducts of a term t have a common reduct. R is *complete*, if it is terminating and confluent.

If a TRS is confluent, each term has at most one normal form. In this case, the unique normal form of term t, if it exists, is denoted by $t{\downarrow}$. If a TRS is terminating, all terms have normal forms. Hence, if a TRS is complete, each term t has a unique normal form $t{\downarrow}$.

An important concept with respect to termination is that of a *reduction order.*

Definition 2.24 (Reduction order)**.** A *reduction order* on terms $T_\Sigma(\mathcal{X})$ is a strict order $>$ on $T_\Sigma(\mathcal{X})$ that

1. does not admit infinite descending chains (i.e., that is a well-founded order),
2. is closed under substitutions, i.e., $t > s$ implies $t\sigma > s\sigma$ for arbitrary substitutions σ,
3. is closed under contexts, i.e., $t > s$ implies $C[t] > C[s]$ for arbitrary contexts C.

A sufficient condition for termination of a TRS R is that a reduction order $>$ exists such that for each rule $l \rightarrow r$ of R, $l > r$.

Example 2.4 (Complete TRS, reduction)**.** Reconsider the signature of Table 2.1, $\Sigma = \{z, s, \mathit{nil}, \mathit{cons}, \mathit{Last}\}$, and the equations Φ of Example 2.1. If we interpret the equations as rewrite rules, we get the following set R of two rules:

$$\begin{aligned} \rho_1 :&\quad \mathit{Last}(\mathit{cons}(x, \mathit{nil})) && \rightarrow x\,, \\ \rho_2 :&\quad \mathit{Last}(\mathit{cons}(x, \mathit{cons}(y, \mathit{xs}))) && \rightarrow \mathit{Last}(\mathit{cons}(y, \mathit{xs}))\,. \end{aligned}$$

The TRS $\langle \Sigma, R \rangle$ is terminating, because each contractum will be shorter than the corresponding redex, and confluent, because each (sub)term will match at most one of the LHSs, and hence complete.

Now consider the term (program call): $\mathit{Last}(\mathit{cons}(z, \mathit{cons}(s(s(z)), \mathit{cons}(s(z), \mathit{nil}))))$. It is reduced by R to its normal form as follows:

$$\begin{aligned} \mathit{Last}(\mathit{cons}(z, \mathit{cons}(s(s(z)), \mathit{cons}(s(z), \mathit{nil})))) &\quad \to_{\rho_2} \\ \mathit{Last}(\mathit{cons}(s(s(z)), \mathit{cons}(s(z), \mathit{nil}))) &\quad \to_{\rho_2} \\ \mathit{Last}(\mathit{cons}(s(z), \mathit{nil})) &\quad \to_{\rho_1} \\ s(z) & \end{aligned}$$

Note that the equation $\mathit{Last}(\mathit{cons}(z, \mathit{cons}(s(s(z)), \mathit{cons}(s(z), \mathit{nil})))) = s(z)$ is a semantic consequence of Φ.

2.2.3. Initial Semantics and Complete Term Rewriting Systems

A complete TRS $\langle \Sigma, R \rangle$ defines a particular Σ-algebra (a universe and functions on it), called the *canonical term algebra*, as follows: The universe is the set of all normal forms and the application of a function (to normal forms) is evaluated according to the rules in R, i.e., to its (due to the completeness of the TRS) always existing and unique normal form.

Definition 2.25 (Canonical term algebra)**.** The *canonical term algebra* $\mathcal{CT}_\Sigma(R)$ according to a complete TRS $\langle \Sigma, R \rangle$ is defined as follows:

- The universe is the set of all normal forms of $\langle \Sigma, R \rangle$ and
- for each $f \in \Sigma$, $f_{\mathcal{CT}_\Sigma}(t_1, \ldots, t_{\alpha(f)}) = f(t_1, \ldots, t_{\alpha(f)}){\downarrow}$.

A functional program, in our first-order algebraic setting, is a set of equations, which—interpreted as a set of rewrite rules—represents a complete TRS (or, in a narrower sense, a complete *constructor TRS*; see Section 2.2.4). Its denotational algebraic semantics is the quotient term algebra (Definition 2.17), its operational term rewriting semantics leads to the canonical term algebra. *Both are initial models of the functional program and hence isomorphic.*

Theorem 2.1 ([67])**.** *Let $\langle \Sigma, \Phi \rangle$ be a presentation (a set of equations representing a functional program) such that $\langle \Sigma, R \rangle$, where R are the equations of Φ interpreted from left to right as rewrite rules, is a complete TRS.*

Then the canonical term algebra according to $\langle \sigma, R \rangle$ is an initial model of $\langle \Sigma, \Phi \rangle$, hence isomorphic to the quotient term algebra:

$$\mathcal{CT}_\Sigma(R) \cong \mathcal{T}_\Sigma / {\sim_\Phi} \ .$$

2.2.4. Constructor Systems

Consider again the *Last*-TRS (Example 2.4). The LHSs have a special form: The *Last* symbol occurs only at the roots of the LHSs but not at deeper positions whereas the other function symbols only occur in the subterms but not at the roots. The *Last*-TRS has the form of a *constructor (term rewriting) system.*

Definition 2.26 (Constructor system)**.** A *constructor term rewriting system* (or just *constructor system (CS)*) is a TRS whose signature can be partitioned into two subsets, $\Sigma = \mathcal{D} \cup \mathcal{C}$, $\mathcal{D} \cap \mathcal{C} = \emptyset$, such that each LHS has the form

$$f(t_1, \ldots, t_n)$$

with $f \in \mathcal{D}$ and $t_1, \ldots, t_n \in T_\mathcal{C}(\mathcal{X})$.

The function symbols in $\mathcal{D}$ and $\mathcal{C}$ are called *defined function symbols* (or just *function symbols*) and *constructors*, respectively.

Terms in $T_\mathcal{C}(\mathcal{X})$ are called *constructor terms.* Since roots of LHSs are defined function symbols in CSs and constructor terms do not contain defined function symbols, constructor terms are normal forms.

A sufficient condition for confluence of TRSs is *orthogonality.* We do not define orthogonality here in general. However, a CS is orthogonal and thus confluent, if its LHSs are (i) linear and (ii) pairwise non-unifying.

Programs in common functional programming languages like HASKELL or SML basically have the constructor system form. The constructors in $\mathcal{C}$ correspond to the constructors of algebraic data types and the defined function symbols to the function symbols defined by equations in, e.g., a HASKELL program. The particular form of the LHSs in CSs resembles the concept of *pattern matching* in functional programming. An example of this correspondence is given in Figure 2.1.

Despite these similarities, CSs exhibit several restrictions compared to typical functional programs. First, CSs only allow for *algebraic* data types. This excludes (predefined) continuous types like real numbers. Second, functions in functional programs are first-class objects, i.e., may occur as arguments and results of (*higher-order*) functions. This is not possible for the usual case of *first-order* signatures in term rewriting. Furthermore, *partial application* (currying) is usual in functional programming but not possible in standard term rewriting. Finally, CSs consist of *sets* of rules, whereas in functional programs, the order of the equations typically matters. In particular, one condition to achieve confluence of CSs is to choose the patterns in a way such that always only one pattern is matched by a term (see above). This condition can be weakened if matches are tried in a fixed and known order, e.g., top-down through the defined functions. This allows for more flexibility in the patterns.

2.3. First-Order Logic and Logic Programming

The basic concepts of first-order logic and logic programming shortly reviewed in this section are described more detailed in textbooks such as [98]. A very thorough and consistent introduction to propositional and first-order logic, logic programming, and also the foundations of inductive logic programming (see Section 3.3) can be found in [81].

Consider again the *Last*-CS, including its signature, partitioned into $\mathcal{C}$ and $\mathcal{D}$:

$$\begin{aligned} \mathcal{C} = \{\; & z && : \mathit{Num}\,, \\ & s && : \mathit{Num} \to \mathit{Num}\,, \\ & \mathit{nil} && : \mathit{NumList}\,, \\ & \mathit{cons} && : \mathit{Num}\ \mathit{NumList} \to \mathit{NumList}\ \}\,, \\ \mathcal{D} = \{\; & \mathit{Last} && : \mathit{NumList} \to \mathit{Num}\ \}\,, \end{aligned}$$

and

$$\begin{aligned} R = \{\; & \mathit{Last}(\mathit{cons}(x, \mathit{nil})) && \to x\,, \\ & \mathit{Last}(\mathit{cons}(x, \mathit{cons}(y, \mathit{xs}))) && \to \mathit{Last}(\mathit{cons}(y, \mathit{xs}))\ \}\,. \end{aligned}$$

The corresponding Haskell program is:

$$\begin{aligned} & \textbf{data}\ \mathit{Nat} && = z \mid s\ \mathit{Nat} \\ & \textbf{data}\ \mathit{NatList} && = \mathit{nil} \mid \mathit{cons}\ \mathit{Nat}\ \mathit{NatList} \end{aligned}$$

$$\begin{aligned} & \mathit{Last} && :: \mathit{NatList} \to \mathit{Nat} \\ & \mathit{Last}(\mathit{cons}(x, \mathit{nil})) && = x \\ & \mathit{Last}(\mathit{cons}(x, \mathit{cons}(y, \mathit{xs}))) && = \mathit{Last}(\mathit{cons}(y, \mathit{xs})) \end{aligned}$$

Figure 2.1.: Correspondence between constructor systems and functional programs

2.3.1. First-Order Logic

Signatures and Structures

A signature in first-order logic extends an algebraic signature by adding *predicate symbols*. A signature is a pair of two sets $\Sigma = (OP, R)$, $OP \cap R = \emptyset$, called *function symbols* and *predicate (or relation) symbols*, respectively. Also predicate symbols have an associated arity.

A structure extends an algebra by adding relations to it according to a signature.

Definition 2.27 (Σ-structure)**.** Let Σ be a signature. A Σ-*structure* $\mathcal{A}$ consists of

- a non-empty set A, called *carrier set* or *universe*,
- for each $f \in OP$, a total function $f_{\mathcal{A}} : A^{\alpha(f)} \to A$, and
- for each $p \in R$, a relation $p_{\mathcal{A}} \subseteq A^{\alpha(f)}$.

Remark 2.3. In contrast to algebras, one typically requires non-empty universes for logical structures in order to prevent certain anomalies.

Table 2.3 shows an example of a (many-sorted) signature Σ and a Σ-structure $\mathcal{A}$.

Terms are built over function symbols and variables and evaluated as defined in Definitions 2.5 and 2.6, respectively. In particular, the set of all ground Σ-terms is called the *Herbrand universe*.

Table 2.3.: A signature Σ and a Σ-structure $\mathcal{A}$

Σ	$\mathcal{A}$
Sorts	**Universe**
Num	$\mathbb{N} \cup \{\bot\}$
NumList	(Lists[a] of $\mathbb{N}$) $\cup \bot$
Function symbols	**Functions**
$z : \mathit{Nat}$	0
$s : \mathit{Nat} \to \mathit{Nat}$	$s_{\mathcal{A}}(n) = \begin{cases} n+1 & \text{if } n \in \mathbb{N} \\ \bot & \text{if } n = \bot \end{cases}$
$\mathit{nil} : \mathit{NatList}$	$()$
$\mathit{cons} : \mathit{Nat}, \mathit{NatList} \to \mathit{NatList}$	$\mathit{cons}_{\mathcal{A}}(\bot, l) = \mathit{cons}_{\mathcal{A}}(e, \bot) = \mathit{cons}_{\mathcal{A}}(\bot, \bot) = \bot$, $\mathit{cons}_{\mathcal{A}}(e_0, (e_1, \ldots, e_n)) = (e_0, e_1, \ldots, e_n)$[b]
Predicate symbol	**Relation**
$\mathit{Last} : \mathit{NumList}, \mathit{Num}$	$\{\langle (e_1, \ldots, e_n), e_n \rangle\}$

[a]Including the empty list $()$.
[b]The sequences $e_1, \ldots, e_n$ may be empty, i.e., $n = 0$. We then have $\mathit{cons}_{\mathcal{A}}(e_0, ()) = (e_0)$.

A Σ-structure which is based on the ground term algebra (i.e., the universe is the Herbrand universe and functions are applications of function symbols to terms) is called a *Herbrand interpretation.* As ground term algebras are the basis to define unique semantics of a set of equations, in particular of functional programs represented as sets of equations or rewrite rules, Herbrand interpretations are the basis to define unique semantics of logic programs.

Definition 2.28 (Herbrand interpretation)**.** A *Herbrand interpretation* of signature Σ is defined as follows:

- The universe is the Herbrand universe, T_Σ.
- For each $f \in \Sigma$, $f_{\mathcal{A}}(t_1, \ldots, t_{\alpha(f)}) = f(t_1, \ldots, t_{\alpha(t)})$.
- For each $p \in R$, $p_{\mathcal{A}} \subseteq T_\Sigma^{\alpha(p)}$.

While there is exactly one unique ground term algebra according to any algebraic signature, Herbrand interpretations are non-unique. They vary exactly with respect to their relations $p_{\mathcal{A}}$.

Formulas and Models

Definition 2.29 (Formulas, literal, clause, Herbrand base)**.** The set of *well-formed formulas* (or just *formulas*) according to a signature $\Sigma = \langle OP, R \rangle$ is defined as follows:

- If $p \in R$ is an n-ary predicate symbol and $t_1, \ldots, t_n$ are Σ-terms, then $p(t_1, \ldots, t_n)$ is a formula, called *atom*;
- if ϕ and ψ are formulas, then $\neg\phi$ (*negation*), $\phi \wedge \psi$ (*conjunction*), $\phi \vee \psi$ (*disjunction*), and $\phi \to \psi$ (*implication*) are formulas; and
- if ϕ is a formula and x is a variable, then $\exists x\ \phi$ (existential quantification) and $\forall x\ \phi$ (universal quantification) are formulas.
- These are all formulas.

Formulas without variables are called *ground* formulas. The set of all ground atoms is called the *Herbrand base*. A *literal* is an atom (*positive literal*) or a negated atom (*negative literal*). A *clause* is a finite, possibly empty, disjunction of literals. The empty clause is denoted by $\square$.

For logic programming, only formulas of a particular form are used.

Definition 2.30 (Horn clause, definite clause)**.** A *Horn clause* is a clause with at most one positive literal. A *definite (program) clause* is a clause with exactly one positive literal.

Definition 2.31. For a signature Σ, the *first-order language* given by Σ is the set of all Σ-Formulas. The terms *clausal language* and *Horn-clause language* are defined analogously.

If a signature contains no functions symbols other than constants, the language is called *function-free.*

Notation 2.1. A definite clause C consisting of the positive literal A and the negative literals $\neg B_1, \ldots, \neg B_n$ is equivalent to the implication $B_1 \wedge \ldots \wedge B_n \to A$, typically written as

$$A \leftarrow B_1, \ldots, B_n\,.$$

A and $B_1, \ldots, B_n$ are called the *head* and *body* of C, respectively. If the body is empty, i.e., C consists of a single atom A only, it is written $A \leftarrow$ or simply A.

Definition 2.32. As between algebras and equations, there is a "satisfies" relation between structures and formulas. It is defined, first of all with respect to a particular assignment, as follows:

$$\begin{array}{lll}
(\mathcal{A}, \beta) \models p(t_1, \ldots, t_n) & \text{iff} & \langle \beta^*(t_1), \ldots, \beta^*(t_n) \rangle \in p_{\mathcal{A}}\,, \\
(\mathcal{A}, \beta) \models \neg\varphi & \text{iff} & (\mathcal{A}, \beta) \not\models \varphi\,, \\
(\mathcal{A}, \beta) \models \phi \wedge \psi & \text{iff} & (\mathcal{A}, \beta) \models \phi \text{ and } (\mathcal{A}, \beta) \models \psi\,, \\
(\mathcal{A}, \beta) \models \phi \vee \psi & \text{iff} & (\mathcal{A}, \beta) \models \phi \text{ or } (\mathcal{A}, \beta) \models \psi\,, \\
(\mathcal{A}, \beta) \models \phi \to \psi & \text{iff} & (\mathcal{A}, \beta) \not\models \phi \text{ or } (\mathcal{A}, \beta) \models \psi\,, \\
(\mathcal{A}, \beta) \models \exists x\ \phi & \text{iff} & \text{for at least one } a \in A,\ (\mathcal{A}, \beta[x \mapsto a]) \models \varphi\,, \\
(\mathcal{A}, \beta) \models \forall x\ \phi & \text{iff} & \text{for all } a \in A,\ (\mathcal{A}, \beta[x \mapsto a]) \models \varphi\,,
\end{array}$$

where $\beta[x \mapsto a](y) = \begin{cases} \beta(y) & \text{if } x \neq y \\ a & \text{if } x = y \end{cases}$.

Definition 2.33 (Satisfies, (Herbrand) model)**.** A Σ-structure $\mathcal{A}$ with universe A *satisfies* a Σ-formula φ, written $\mathcal{A} \models \varphi$, if for every assignment $\beta : \mathcal{X} \to A$, $(\mathcal{A}, \beta) \models \varphi$.

A structure $\mathcal{A}$ is a *model* of a set of formulas Φ, written $\mathcal{A} \models \Phi$, if for all $\varphi \in \Phi$, $\mathcal{A} \models \varphi$. If, furthermore, $\mathcal{A}$ is a Herbrand interpretation, then $\mathcal{A}$ is called a *Herbrand model.*

By $\mathit{Mod}_\Sigma(\Phi)$, we denote the class of all models of Φ.

A Herbrand interpretation is uniquely determined by a subset of the Herbrand base, namely the set of all ground atoms satisfied by it. This is because (i) two Herbrand interpretations only vary with respect to their relations $p_\mathcal{A}$ and (ii) $\langle t_1, \ldots, t_{\alpha(p)} \rangle \in p_\mathcal{A}$ if and only if $p(t_1, \ldots, t_{\alpha(p)})$ is satisfied. Therefore, we *identify* Herbrand interpretations and their sets of satisfied ground atoms: *A Herbrand interpretation is just a subset of the Herbrand base.*

Definition 2.34. A set of formulas Φ is said to be *satisfiable* if it has at least one model and *unsatisfiable* if it has no models.

Proposition 2.1. *Let Φ be a set of formulas and φ be a formula. $\Phi \models \varphi$ if and only if $\Phi \cup \{\neg\varphi\}$ is unsatisfiable.*

Example 2.5. Consider the following set Φ of two Σ-formulas (definite clauses), where Σ is the signature of Table 2.3:

$$\begin{aligned}&\mathit{Last}(\mathit{cons}(x, \mathit{nil}), x)\,,\\&\mathit{Last}(\mathit{cons}(x, \mathit{cons}(y, \mathit{xs})), z) \leftarrow \mathit{Last}(\mathit{cons}(y, \mathit{xs}), z)\,.\end{aligned}$$

The structure $\mathcal{A}$ of Table 2.1 is a model of Φ.

Definition 2.35 (Logical consequence, entailment)**.** A Σ-formula φ is a *logical consequence* of a set of Σ-formulas Φ, written $\Phi \models \varphi$, if for all $\mathcal{A} \in \mathit{Mod}_\Sigma(\Phi)$, $\mathcal{A} \models \varphi$. We say that Φ *entails* φ.

The problem whether $\Phi \models \varphi$ is undecidable.

Definition 2.36 (Equivalence)**.** Two Σ-formulas φ and ψ are *equivalent*, written $\varphi \equiv \psi$, if $\mathit{Mod}(\varphi) = \mathit{Mod}(\psi)$.

Resolution

Since the problem whether $\Phi \models \varphi$ is undecidable, there is no algorithm that takes a set of formulas Φ and a formula φ and, after finite time, correctly reports that either $\Phi \models \varphi$ or $\Phi \not\models \varphi$. However, calculi exist that after finite time report $\Phi \models \varphi$ if and only if in fact $\Phi \models \varphi$ and otherwise either do not terminate or correctly report $\Phi \not\models \varphi$. One such calculus restricted to clauses is *resolution* as defined in this section.

Substitutions θ (mappings from terms to terms that replace variables by terms; see Definition 2.18) are uniquely extended to atoms, literals, and clauses as follows: $p(t_1, \ldots, t_n)\theta = p(t_1\theta, \ldots, t_n\theta)$, $(\neg a)\theta = \neg(a\theta)$, where a is an atom, and $(\varphi \vee \psi)\theta = \varphi\theta \vee \psi\theta$, where φ, ψ are clauses.

By *simple expression*, we either mean a term or a literal. If $\mathcal{E} = \{e_1, \ldots, e_n\}$ is a set of simple expressions, by $\mathcal{E}\theta$ we denote the set $\{e_1\theta, \ldots, e_n\theta\}$.

Definition 2.37 ((Most general) unifier)**.** Let $\mathcal{E}$ be a finite set of simple expressions. A *unifier* for $\mathcal{E}$ is a substitution θ such that $\mathcal{E}\theta$ is a singleton, i.e., a set containing only one element. If a unifier for $\mathcal{E}$ exists, we say that $\mathcal{E}$ is *unifiable.*

A *most general unifier (MGU)* for $\mathcal{E}$ is a unifier θ for $\mathcal{E}$ such that for any unifier σ for $\mathcal{E}$ exists a substitution γ with $\sigma = \theta\gamma$.

Proposition 2.2. *Let $\mathcal{E}$ be a finite set of expressions.*

- *The problem whether $\mathcal{E}$ is unifiable is decidable.*
- *If $\mathcal{E}$ is unifiable, then there is an MGU for $\mathcal{E}$.*

There are terminating unification algorithms that take a finite set of expressions $\mathcal{E}$ and output either an MGU of $\mathcal{E}$ (if $\mathcal{E}$ is unifiable) or otherwise report that $\mathcal{E}$ is not unifiable.

Terminology 2.2. Two clauses or (two terms) are said to be *standardized apart* if they have no variables in common.

Clauses and terms can easily be standardized apart by applying a variable renaming.

Definition 2.38 (Binary resolvent)**.** Let $C = L_1 \vee \ldots \vee L_m$ and $C' = L'_1 \vee \ldots \vee L'_n$ be two clauses which are standardized apart. If the substitution θ is an MGU for $\{L_i, \neg L'_j\}$ $(1 \leq i \leq m,\ 1 \leq j \leq n)$, then the clause

$$(L_1 \vee \ldots \vee L_{i-1} \vee L_{i+1} \vee \ldots \vee L_m \vee L'_1 \vee \ldots \vee L'_{j-1} \vee L'_{j+1} \vee \ldots \vee L'_n)\theta$$

is a *binary resolvent* of C and C'. The literals L and L' are said to be the literals *resolved upon.*

Note that a binary resolvent may be the empty clause $\Box$.

Definition 2.39 (Factor)**.** Let C be a clause, $L_1, \ldots, L_n$ $(n \geq 1)$ be some unifiable literals from C, and θ be an MGU for $\{L_1, \ldots, L_n\}$. Then the clause obtained by deleting $L_2\theta, \ldots, L_n\theta$ from $C\theta$ is a *factor* of C.

Definition 2.40 (Resolvent)**.** Let C and D be two clauses. A *resolvent* R of C and D is a binary resolvent of a factor of C and a factor of D where the literals resolved upon are the literals unified by the respective factors.

C and D are called the parent clauses of R.

Definition 2.41 (Derivation, refutation)**.** Let $\mathcal{C}$ be a set of clauses and C be a clause. A *derivation* of C from $\mathcal{C}$ is a finite sequence of clauses $R_1, \ldots, R_k = C$, such that for all R_i, $1 \leq i \leq k$, $R_i \in \mathcal{C}$ or R_i is a resolvent of two clauses in $\{R_1, \ldots, R_{i-1}\}$.

Deriving the empty clause from a set of clauses $\mathcal{C}$ is a called a *refutation* of $\mathcal{C}$. If a set of clauses $\mathcal{C}$ can be refuted, then $\mathcal{C}$ is unsatisfiable.

Resolution is sound, i.e., $\Phi \models \varphi$ whenever φ is derivable be resolution from Φ. Furthermore, resolution is, due to Proposition 2.1, complete in the following sense:

Proposition 2.3 (Refutation completeness of resolution)**.** *If $\Phi \models \varphi$ for a set of clauses Φ and a clause φ, then there is a refutation of $\Phi \cup \{\neg\varphi\}$.*

2.3.2. Logic Programming

As functional programs can be regarded as a set of equations or rules of a particular form according to an algebraic signature, a *logic program* can be regarded as a set of formulas of a special form according to a signature.

Sets of arbitrary formulas or even clauses are not appropriate for programming. This is (i) because general theorem proving and also general resolution on clauses is too inefficient due to a high degree of non-determinism in each computation step, i.e., in choosing parent clauses to be resolved and literals to be resolved upon; and (ii) because for sets of arbitrary formulas or clauses one can not appoint unique models.

For logic programming, *definite programs* are used.

Definition 2.42 (Definite program)**.** A *definite program* is a finite set of definite clauses.

Proposition 2.4. *Let Π be a definite program.*

- *Π has a model iff it has a Herbrand model.*
- *Let $\mathcal{M} = \{M_1, M_2, \ldots\}$ be a possibly infinite set of Herbrand models of Π. Then the intersection $\bigcap \mathcal{M}$ is also a Herbrand model of Π.*

Definition 2.43 (Least Herbrand model)**.** Let Π be a definite program and $\mathcal{M}$ the set of all its Herbrand models. Then the intersection $\bigcap \mathcal{M}$ is called the *least Herbrand model* of Π.

Hence, if a definite program has a model, it also has a least Herbrand model, which is unique. It just consists of all ground atoms that are logical consequences of Π and is taken as its standard denotational semantics.

A program call consists of a *conjunction* of atoms, possibly containing variables. It is evaluated by adding its negation to the set of definite clauses forming the definite program and applying a particular efficient form of resolution as defined below to that set. If the set can be refuted, the corresponding substitutions of the variables are reported as output of the evaluation.

The negation of a conjunction of atoms $\neg(B_1 \wedge \cdots \wedge B_n)$ is equivalent to a disjunction of the negated atoms $\neg B_1 \vee \cdots \vee \neg B_n$. This is called a *goal clause* and written $\leftarrow B_1, \ldots, B_n$.

Definition 2.44 (SLD-resolution)**.** Let Π be a definite program and G be a goal clause. An *SLD-refutation* of $\Pi \cup \{G\}$ is a finite sequence of goal clauses $G = G_0, \ldots, G_k = \Box$, such that each G_i $(1 \leq i \leq k)$ is a binary resolvent of R_{i-1} and a clause C from Π where the head of C and a *selected literal* of R_{i-1} are the literals resolved upon.

Theorem 2.2 (Completeness of SLD-resolution with respect to $\mathcal{M}_\Pi$)**.** *Let* Π *be a definite program and* A *be a ground atom. Then* $A \in \mathcal{M}_\Pi$ *if and only if* $\Pi \cup \{\leftarrow A\}$ *has an SLD-refutation.*

Example 2.6. Consider again the definite program for *Last* from Example 2.5 and the program call $\mathit{Last}(\mathit{cons}(z, \mathit{cons}(s(s(z)), \mathit{cons}(s(z), \mathit{nil}))), X)$ or rather the corresponding goal clause $\leftarrow \mathit{Last}(\mathit{cons}(z, \mathit{cons}(s(s(z)), \mathit{cons}(s(z), \mathit{nil}))), X)$. The refutation consists of the following sequence:

$$\begin{aligned}
G_0 :& \quad \leftarrow \mathit{Last}(\mathit{cons}(z, \mathit{cons}(s(s(z)), \mathit{cons}(s(z), \mathit{nil}))), X)\,, \\
G_1 :& \quad \leftarrow \mathit{Last}(\mathit{cons}(s(s(z)), \mathit{cons}(s(z), \mathit{nil})), X)\,, \\
G_2 :& \quad \leftarrow \mathit{Last}(\mathit{cons}(s(z), \mathit{nil}), X)\,, \\
G_3 :& \quad \quad \Box\,.
\end{aligned}$$

3. Approaches to Inductive Program Synthesis

Even though research on inductive program synthesis started in the 1970s already, it has not become a unified research field since then, but is scattered over several research fields and communities such as artificial intelligence, inductive inference, inductive logic programming, evolutionary computation, and functional programming. This chapter provides a comprehensive survey of the different existing approaches, including theory and methods. A shortened version of this chapter was already published in [49]. We grouped the work into three blocks: First, the classical analytic induction of Lisp programs from examples, as introduced by Summers [104] (Section 3.2); second, inductive logic programming (Section 3.3); and third, several recent generate-and-test based approaches to the induction of functional programs (Section 3.4). In the following section (3.1), we at first introduce some general concepts.

3.1. Basic Concepts

We only consider *functions* as objects to be induced in this section. General relations, dealt with in (inductive) logic programming, fit well into these rather abstract illustrations by considering them as boolean-valued functions.

3.1.1. Incomplete Specifications and Inductive Bias

Inductive program synthesis (IPS) aims at (semi-)automatically constructing computer programs or algorithms from *(known-to-be-)incomplete specifications* of functions. We call such functions to be induced *target functions*. *Incomplete* means, that target functions are not specified on their complete domains but only on (small) parts of them. A typical incomplete specification consists of a subset of the graph of a target function f—$\{\langle i_1, o_1\rangle, \dots, \langle i_k, o_k\rangle\} \subseteq \mathit{Graph}(f)$—called *input/output examples (I/O examples)* or *input/output pairs (I/O pairs)*. The goal is then to find a program P that correctly computes the provided I/O examples, $P(i_j) = o_j$ for all $1 \leq j \leq k$, (and that also correctly computes all unspecified inputs). The concrete shape of incomplete specifications varies between different approaches to IPS and particular IPS algorithms.

If a program computes the correct specified output for each specified input then we say that the program is *correct* with respect to the specification (or that it satisfies the specification). Yet note that, due to the underspecification, correctness in this sense does not imply that the program computes the “correct” function in the sense of the *intended* function.

Having in mind that we are concerned with *inductive* program synthesis from *incomplete* specifications, we may in the following just say *specification* (instead of *incomplete specification*).

Due to the inherent underspecification in inductive reasoning, typically infinitely many (semantically) different functions or relations satisfy an incomplete specification. For example, if one specifies a function on natural numbers in terms of a finite number of I/O examples, then there are obviously infinitely many functions on natural numbers whose graphs include the provided I/O examples and hence, which are correct with respect to the provided incomplete specification. Without further information, an IPS system cannot know which of them is intended by the specifier; there is no objective criterion to decide which of the different functions or relations is the right one. This ambiguity is inherent to IPS and therefore, programs generated by IPS systems are often called *hypotheses*.

Even though (or rather: *because*) there is no objective criterion to decide which of the possible hypotheses is the intended one, returning one of them as *the* solution, or even returning all of them in a particular order, implies criteria to include, exclude, and/or rank possible solutions. Such criteria are called *inductive bias* [69]. In general, the inductive bias comprises all factors—other than the actual incomplete specification of the target function—which influence the selection or ordering of possible solutions.

There are two general kinds of inductive bias: The first one is given by the class of all programs that can in principle be generated by an IPS system. It may be fixed or problem dependent and depends on the used object language, including predefined functions that may be used, and the (search) operators to create and transform programs. It possibly already excludes particular algorithms or even computable functions (no matter *how*, by which algorithm, they are computed). As an example imagine a *finite* class of programs computing functions on natural numbers. Then, certainly, not each computable function is represented. This bias, given by the class of generatable programs, is called *language bias*, *restriction bias*, or *hard bias*.

The second kind of inductive bias is given by the order in which an IPS system explores the program class and by the acceptance criteria (if there are any except for correctness with respect to the specification). Hence it determines the selection of solutions from generated candidate programs and their ordering. This inductive bias is called *search bias*, *preference bias*, or *soft bias*. A preference bias may be modelled as a probability distribution over the program class [78].

3.1.2. Inductive Program Synthesis as Search, Background Knowledge

Inductive program synthesis is most appropriately understood as a search problem. An IPS algorithm is faced with an (implicitly) given class of programs from which it has to choose one. This is done by repeatedly generating candidate programs until one is found satisfying the specification. Typically, the search starts with an initial program and then, in each search step, some program transformation operators are applied to an already generated program to get new (successor) candidate programs.

In general, the program class is not fixed but depends on additional (amongst the

Listing 3.1: reverse with accumulator variable

```
reverse (l)        =  rev (l, [])
rev ([], ys)       =  ys
rev (x : xs, ys)   =  rev (xs, x : ys)
```

specification of the function) input to the IPS system. It is determined by *primitives*, predefined functions which can be used by induced programs, and some definition of syntactically correctness of programs.

In early approaches (Section 3.2), the primitives to be used were fixed within IPS systems and restricted to small sets of data type constructors, projection functions, and predicates. By now, usually arbitrary functions may be provided as (problem-dependent) input to an IPS system. We call such problem-dependent input of predefined functions *background knowledge*. It is well known in artificial intelligence that background knowledge—in general: knowledge, that simplifies the solution to a problem—is very important to solve complex problems. Additional primitives, though they enlarge the program class, i.e., the problem space, may help to find a solution program. This is because solutions may become more compact such that they are constructible by fewer transformations.

3.1.3. Inventing Subfunctions

Implementing a function typically includes the identification of subproblems, the implementation of solutions for them in terms of separate (sub)functions, and composing the main function from those help functions. This facilitates reuse and maintainability of code and may lead to more concise implementations. Furthermore, without subfunctions and depending on available primitives, some functions may not be representable at all, some particular algorithm may not be representable, or the function definition will become bulky and hard to understand.

Hence, introducing and inducing subfunctions that are neither (explicitly) specified nor provided as primitives—*(sub)function invention*—can be an important capability of IPS systems.

For example, consider the reverse function on lists. Typical implementations either use an accumulator variable or—to put the first element at the end of the reversed rest-list—the list-appending function $++$. Listings 3.1 and 3.2 show these two implementations. In contrast, Listing 3.3 shows an implementation with only one function definition only using the usual list constructors [] (empty list) and $\_:\_$ ("cons"ing an element to the front of a list) and the selection functions head (first element) and tail (rest-list).

For another example, consider *sorting* a list. Well-known algorithms like *quicksort*, *mergesort*, or *selection sort* define subfunctions such as partitioning, splitting, and merging lists and selecting particular elements from lists. Just for fun, Listing 3.4 shows an implementation without such subfunctions, only using the list constructors and selection functions, $\leq$, and an if−then−else conditional.

Listing 3.2: reverse with append (++)

```
reverse ([])     =  []
reverse (x : xs) =  reverse (xs) ++ (x : [])
```

Listing 3.3: reverse without help functions and variables

```
reverse ([])     =  []
reverse (x : xs) =  head(reverse(xs)) : reverse(x : reverse(tail(reverse(xs))))
```

The ability of automatically introducing subfunctions or relations is called *predicate invention* in inductive logic programming. It is a kind of *constructive induction* [68, 73]. In the context of inductive bias one speaks of *bias shift* [106, 101].

3.1.4. The Enumeration Algorithm

In this subsection, we present a very basic solution to the inductive program synthesis problem: the *enumeration algorithm*. The definitions and results are (slightly adapted) taken from Biermann [11] and go back, in their original form, to Gold [33]. We restrict ourselves to incomplete specifications in terms of I/O examples.

The problem that an IPS algorithm has to solve, is to take a program class and a set of I/O examples and to return a program P from the program class that computes the specified output for each example input. One solution to this problem is the *enumeration algorithm* (Algorithm 1), denoted by Enum.

It has some noteworthy properties.

Definition 3.1. Let $\mathcal{PC}$ be a program class. An IPS algorithm ips is

- *sound* for $\mathcal{PC}$, if for each set of I/O examples E, $\textsc{ips}(\mathcal{PC}, E) = P$ implies that P is correct with respect to E,
- *complete* for $\mathcal{PC}$, if for each program $P \in \mathcal{PC}$ exists a set of I/O examples E such that $\textsc{Enum}(\mathcal{PC}, E) = P'$ and $P'(x) = P(x)$ for all $x \in Dom(P)$,
- *stable* in $\mathcal{PC}$, if for any two disjoint sets E, E' of I/O examples, $\textsc{Enum}(\mathcal{PC}, E) = P$ and P correctly computes E' implies $\textsc{Enum}(\mathcal{PC}, E \cup E') = P$,

Listing 3.4: List-sorting without subfunctions

```
sort ([])         =  []
sort (x : [])     =  x : []
sort (x : y : xs) =  if x ≤ head(sort(y : xs))
                     then x : sort(y : xs)
                     else head(sort(y : xs)) : sort(x : tail(sort(y : xs)))
```

Algorithm 1: The enumeration algorithm Enum for inductive program synthesis

```
  Input: An enumerable program class $\mathcal{PC} = \{P_i \mid i \in \mathbb{N}\}$ with decidable
         Halting-problem
  Input: A set of I/O pairs $E = \{\langle i_1, o_1\rangle, \ldots, \langle i_k, o_k\rangle\}$
  Output: A program $P \in \mathcal{PC}$ such that $P(i_j) = o_j$ for all $1 \leq j \leq k$
1 $i \leftarrow 0$
2 while $P_i$ does not correctly compute $E$ do increment $i$
3 return $P_i$
```

- *input-optimal* for $\mathcal{PC}$ in a class $\mathcal{IPS}$ of IPS algorithms, if IPS $\in \mathcal{IPS}$ and there is no algorithm IPS$' \in \mathcal{IPS}$ such that
  - IPS$(\mathcal{PC}, E) = P$ implies IPS$'(\mathcal{PC}, E') = P$ for some $E' \subseteq E$ and
  - IPS$(\mathcal{PC}, E) \neq$ IPS$(\mathcal{PC}, E')$ for at least one such E'.

Soundness means, that induced programs are correct with respect to the specification; completeness means, that each program P (or at least a program that is equivalent to P on P's domain) in a given program class can be induced; stability means, that if IPS returns a program P based on some set of I/O pairs E, it will return P based on any set of I/O examples of P containing E; and input-optimality for a certain class of IPS algorithms means, that there is no algorithm in that class, which induces all programs based on less information.

We call enumerable program classes with decidable Halting-problem *admissible.*

Theorem 3.1 (Biermann)**.** *The enumeration algorithm is*

- *sound, complete, and stable for admissible program classes and*
- *input-optimal in the class of sound, complete, and stable IPS algorithms for admissible program classes.*

Proof. See [11, Theorems 5 and 6]. □

Clearly, for considerable program classes the enumeration algorithm is practically useless because of arbitrary large indices of the solution programs. Finding "good" program classes and methods to efficiently explore them is *the* problem of inductive program synthesis research.

3.2. The Analytical Functional Approach

A first systematic attempt to IPS was made by Summers [103, 104]. He noticed that under particular restrictions regarding allowed primitives, program schema, and choice of I/O examples, a recursive Lisp program can be directly *computed* from I/O examples

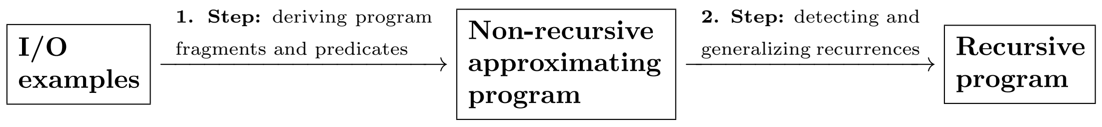
I/O examples
1. Step: deriving program fragments and predicates
Non-recursive approximating program
2. Step: detecting and generalizing recurrences
Recursive program

Figure 3.1.: The classical two-step approach for the induction of Lisp programs

instead of found by *searching* in program space. In this section we describe his original method and some extensions and variants of this approach.

The general principle of the analytic approach is this: If a function is recursively defined, then evaluating one input (that is not covered by some base case) depends on evaluating other, smaller, inputs by the same program. Hence outputs of smaller inputs go into outputs of greater inputs in a recurrent way. These recurrent relations between I/O examples are discovered and then inductively generalized to a recursive function definition.

3.2.1. Summers' Pioneering Work

Summers' approach to induce recursive Lisp functions from I/O examples includes two steps (see Figure 3.1): First, a so-called *program fragment*, an expression of one variable and the allowed primitives, is derived for each I/O-pair such that if it is applied to the input, evaluates to the specified output. Furthermore, predicates are derived to distinguish between example inputs. Integrated into a McCarthy conditional [66], these predicate/fragment pairs build a non-recursive program computing the I/O examples. It is considered as a first approximation to the target function. In a second step, recurrent relations between predicates and fragments each are identified and a recursive program generalizing them is derived.

Inputs and outputs are *S-expressions*, the fundamental data structure of the Lisp language [66].

Definition 3.2 (S-expressions). • Each atom (constant) is an S-expression;

- if a and b are S-expressions, so is $(a\ .\ b)$;
- these are all S-expressions.

S-expressions are uniquely constructed and deconstructed by the functions *cons*, *car*, and *cdr*:

- $cons(a, b) = (a\ .\ b)$
- $car((a\ .\ b)) = a$
- $cdr((a\ .\ b)) = b$

Non-atomic S-expressions $(a\,.\,b)$ are also called *cons-pairs*. *car* and *cdr* are undefined for atomic S-expressions. The predicate $atom(a)$ is true if a is an atom and false otherwise.

The set of all subexpressions of an S-expression consists of the S-expression itself and, if it is a *cons*-pair, of all subexpressions of its both components.

Remark 3.1. 1. Lists are a special form of S-expressions. A list $(a, b, c, \ldots)$ is represented by the S-expression $(a\,.\,(b\,.\,(c\,.\,(\ldots\,.\,nil)\ldots)))$. *nil* is a special atom denoting the empty list.

2. Compositions of *car* and *cdr* are abbreviated by words of the form $c\{a|d\}^*r$. For example, *caddr* abbreviates $car \circ cdr \circ cdr$.

The programs constructed by Summers' technique use the Lisp primitives *cons*, *car*, *cdr*, *nil*, *atom*, and T, the last denoting the truth value *true*. Particularly, no other predicates than *atom* and T (e.g., *eq* for testing equality of S-expressions), and no atoms except for *nil* are used. This choice of primitives is not arbitrary but crucial for Summers' methodology of deriving programs from examples without search. The McCarthy conditional and recursion are used as control structure. The McCarthy conditional takes a chain of predicate/function-pairs $p_i \to f_i$, $i \in \mathbb{N}$, and eventually evaluates that function f_i whose predicate p_i is the first one evaluating to *true*. Allowing *atom* and T as only predicates and *nil* as only atom in induced programs means that the atoms in the I/O examples, except for *nil*, are actually considered as *variables*. Renaming them does not change the meaning. This implies that any semantic information must be expressed by the *structure* of the S-expression.

1. Step: Generation of Program Fragments and Predicates

Given a set of k I/O examples, $\{\langle i_1, o_1\rangle, \ldots, \langle i_k, o_k\rangle\}$, a *program fragment* $f_j(x)$, $j = 1, \ldots, k$, composed of *cons*, *car*, and *cdr* is derived for each I/O-pair. It evaluates to the output when applied to the input: $f_j(i_j) = o_j$ for all $j \in \{1, \ldots, k\}$.

Recall that S-expressions are uniquely constructed by *cons* and decomposed by *car* and *cdr*. We call *car*-*cdr* compositions *basic functions* (cp. [100]). Together with the following two conditions, this allows for determining unique program fragments.

1. Each atom, except for *nil*, may occur only once in each input.

2. Each atom, except for *nil*, occurring in an output must also occur in the corresponding input.

Due to the first condition, each subexpression (except for *nil*) occurs exactly once in an S-expression such that subexpressions are denoted by unique basic functions.

Deriving a program fragment works as follows. All non-*nil* subexpressions of an input, together with their unique basic functions, are enumerated. Then the output is rewritten by composing the basic functions from the input subexpressions with *cons* and *nil*: If the output is *nil* then it is its own fragment, *nil*. Otherwise, if it is a subexpression of the input then the fragment becomes the associated basic function. Otherwise, program fragments for the two subexpressions of the output *cons*-pair are derived and a *cons* is applied to them. Condition 2 from above assures that always one of the three cases holds.

Example 3.1. Consider the I/O-pair $((a.b).(c.d)) \mapsto ((d.c).(a.b))$. The input contains the following subexpressions, paired with the corresponding unique basic functions:

$$\langle ((a\,.\,b)\,.\,(c\,.\,d)), I\rangle\,, \quad \langle (a\,.\,b), \mathit{car}\rangle\,, \quad \langle (c\,.\,d), \mathit{cdr}\rangle\,,$$
$$\langle a, \mathit{caar}\rangle\,, \quad \langle b, \mathit{cdar}\rangle\,, \quad \langle c, \mathit{cadr}\rangle\,, \quad \langle d, \mathit{cddr}\rangle\,.$$

Since the example output is neither a subexpression of the input nor *nil*, the program fragment becomes a *cons* of the fragments for the *car*- and the *cdr*-component, respectively, of the output. The *car*-part, $(d\,.\,c)$, again becomes a *cons*, namely of the basic functions for d: *cddr*, and c: *cadr*. The *cdr*-part, $(a\,.\,b)$, *is* a subexpression of the input, its basic function is *car*. With variable x denoting the input, the fragment for this I/O example is thus:

$$\mathit{cons}(\mathit{cons}(\mathit{cddr}(x), \mathit{cadr}(x)), \mathit{car}(x))$$

Next, predicates $p_j(x)$, $j = 1, \ldots, k$, must be determined. In order to get the correct program fragment f_j be evaluated for each input i_j, all predicates $p_{j'}$, $1 \leq j' < j$ (positioned before p_j in the conditional), must evaluate to *false* when applied to i_j and $p_j(i_j)$ must evaluate to *true*.

$$\text{For all } j \in \{1, \ldots, k\}\text{: } p_{j'}(i_j) = \mathit{false} \text{ for all } 1 \leq j' < j \text{ and } p_j(i_j) = \mathit{true}. \tag{3.1}$$

Predicates fulfilling this condition exist and are efficiently computable if the example inputs form a chain according to the following order:

$$\begin{array}{ccl} a \leq b & \text{if} & \mathit{atom}(a) \\ (a\,.\,b) \leq (c\,.\,d) & \text{if} & a \leq c \wedge b \leq d \end{array} \tag{3.2}$$

Assume the examples are ordered such that $i_1 < \ldots < i_k$ according to the defined order.[1] It is easy to see that condition (3.1) is fulfilled with predicates of the form $\mathit{atom}(b_j(x))$ (where b_j denotes a basic function) for all $j \in \{1, \ldots, k-1\}$ and $p_k(x) = \mathrm{T}$, if $b_j(i_j)$ evaluates to an atom and $b_j(i_{j+1})$ does not. Hence, an algorithm for computing such basic functions only need to compare each two consecutive inputs and identify those positions denoting atoms in the first and *cons*-pair subexpressions in the second input and return the corresponding basic functions. The predicate p_k becomes T because no input i_{k+1} exists to derive a more specific predicate.

Formal definitions of the algorithms for computing fragments and predicates can be found in [100]. Figure 3.2 shows an example for the first step. The result of the first synthesis step, a non-recursive program, correctly computes all provided I/O examples. It is considered as a first approximation to the target function.

2. Step: Identifying and Generalizing Recurrence Relations

The basic idea in Summers' generalization method is this: The fragments are assumed to be the actual computations carried out by a *recursive* program for the target function.

[1] If they are provided in a different order, they are automatically reordered.

I/O examples:

$$\begin{aligned}(a) &\mapsto nil,\\ (a,b) &\mapsto (a),\\ (a,b,c) &\mapsto (a,b),\\ (a,b,c,d) &\mapsto (a,b,c),\\ (a,b,c,d,e) &\mapsto (a,b,c,d)\,.\end{aligned}$$

Initial non-recursive approximating program:

$$\begin{aligned}F(x) = (\,atom(cdr(x)) &\rightarrow nil\\ atom(cddr(x)) &\rightarrow cons(car(x), nil)\\ atom(cdddr(x)) &\rightarrow cons(car(x), cons(cadr(x), nil))\\ atom(cddddr(x)) &\rightarrow cons(car(x), cons(cadr(x), cons(caddr(x), nil)))\\ T &\rightarrow cons(car(x), cons(cadr(x), cons(caddr(x),\\ &\qquad cons(cadddr(x), nil)))))\end{aligned}$$

Figure 3.2.: I/O examples and the corresponding first approximation

Hence fragments of greater inputs must comprise fragments of lesser inputs as subterms, with a suitable substitution of the variables and in a recurrent form along the set of fragments. The same holds analogously for the predicates. Summers calls this relation between fragments and predicates *differences*.

Definition 3.3. A *difference* exists between two terms (fragments or predicates) t, t' iff $t' = C[t\sigma]$ for some context $C[\;]$ and substitution σ.

If we have $k+1$ I/O examples, we only consider the first k fragment/predicate pairs because the last predicate is always 'T', such that no sensible difference can be derived for it.

Example 3.2. The following differences, with $\sigma = \{x \leftarrow cdr(x)\}$ and $\sigma' = \emptyset$, can be identified in the approximating program from Figure 3.2:

$$\begin{aligned}f_2 &= cons(car(x), f_1\sigma') & p_2 &= p_1\sigma\\ f_3 &= cons(car(x), f_2\sigma) & p_3 &= p_2\sigma\\ f_4 &= cons(car(x), f_3\sigma) & p_4 &= p_3\sigma\end{aligned}$$

The context $C[\,]$ and the substitution σ are equal in the differences for the fragments f_3, f_4 and predicates p_2, p_3, p_4. This allows us to rewrite the differences in terms of *recurrence relations*.

Example 3.3. The differences from Example 3.2 can be written as recurrence relations:

$$\begin{aligned} f_1 &= nil & p_1 &= atom(cdr(x)) \\ f_2 &= cons(car(x), nil) & p_2 &= atom(cddr(x)) \\ f_{i+1} &= cons(car(x), f_i\sigma) & p_{i+1} &= p_i\sigma \end{aligned}$$

with $\sigma = \{x \leftarrow cdr(x)\}$ and $i = 2, 3$.

In the general case, we have, for k fragments/predicates, a context $C[\,]$, and a substitution σ:

$$\begin{aligned} &j-1 \text{ “constant” fragments (as derived from the examples): } f_1, \dots, f_{j-1}, \\ &\text{further } n \text{ constant base cases: } f_j, \dots, f_{j+n-1}, \\ &\text{finally, remaining } k-(j+n-1) \text{ cases recurring to previous cases:} \\ &\quad f_{i+n} = C[f_i\sigma], \quad \text{for } j \le i \le k-n\,; \\ &\text{and the same for predicates: } p_1, \dots, p_{j-1}, p_j, \dots, p_{j+n-1}, p_{i+n} = p_i\sigma\ . \end{aligned} \tag{3.3}$$

Index j denotes the first predicate/fragment pair which recurs in some following predicate/fragment pair (the first base case). The precedent $j-1$ predicate/fragment pairs do not recur. n is the interval of the recurrence. In Example 3.3 we have $k = 4$, $j = 2$, and $n = 1$.

Inductive Inference. If $k - j \ge cn$ with $1 < c \in \mathbb{N}$, then we *inductively infer* that the recurrence relations hold for all $i \ge j$.

This is the only logically unsound operation in the synthesis. It is classical induction. We observe some property for some instances of a concept without counter instances and thus assume that the property is inherent to *all* instances of the concept. The constant c determines how often the regularity between the fragments and predicates have to be observed until the induction seems to be justified. Summers sets $c = 2$.

In Example 3.3 we have $k - j - 1 = 2 \ge 2 = 2n$ and hence induce that the relations hold for all $i \ge 2$.

The generalized recurrence relations lead to new approximations of the assumed target function. The mth *approximating function*, $m \ge j$, is defined as

$$F_m(x) = (p_1(x) \to f_1(x), \dots, p_m(x) \to f_m(x), \mathrm{T} \to \omega)$$

where the p_i, f_i with $j < i \le m$ are defined in terms of the generalized recurrence relations and ω means *undefined.* Consider the following complete partial order (cpo) over partial functions which is well known from denotational semantics and domain theory:

$$F(x) \le G(x) \quad \text{iff} \quad F(x) = G(x) \text{ for all } x \in Dom(F)\,.$$

Regarding this order, the set of approximating functions builds a chain.

Definition 3.4. We define the function $\mathbf{F}$ specified by the examples to be $\sup\{F_m(\boldsymbol{x})\}$, the supremum of the chain of approximations.

Now the hypothesized target function is defined, in terms of recurrence relations. In his synthesis theorem and several corollaries, Summers shows how a partial function defined in this way can be expressed by a recursive program.[2] The following theorem comprises the basic synthesis theorem and its first corollary in [104].

Theorem 3.2 ([104])**.** *If* $\mathbf{F}$ *is defined by the recurrence relations*

$$\begin{aligned} &f_1, \ldots, f_{j-1}, f_j, \ldots, f_{j+n-1}, f_{i+n} = C[f_i\sigma_1], \\ &p_1, \ldots, p_{j-1}, p_j, \ldots, p_{j+n-1}, p_{i+n} = p_i\sigma \qquad \text{for } i \geq j\,, \end{aligned}$$

then the following recursive program, with main function F*, computes* $\mathbf{F}$*:*

$$\begin{aligned} F(x) = (&p_1(x) \to f_1(x), & G(x) = (&p_j(x) \to f_j(x), \\ &\vdots & &\vdots \\ &p_{j-1}(x) \to f_{j-1}(x), & &p_{j+n-1}(x) \to f(x), \\ &\mathrm{T} \to G(x)) & &\mathrm{T} \to C[G(\sigma(x))]) \end{aligned}$$

Example 3.4. The generalized ($i \geq 2$) recurrence relations from Example 3.3 define the function $\mathbf{F}$ to be the *Init*-function, which reproduces a list of arbitrary length without its last element. The resulting recursive program according to the synthesis theorem (Theorem 3.2) is:

$$\begin{aligned} &F(x) = (\mathit{atom}(\mathit{cdr}(x)) \to \mathit{nil},\ \mathrm{T} \to G(x)) \\ &G(x) = (\mathit{atom}(\mathit{cddr}(x)) \to \mathit{cons}(\mathit{car}(x), \mathit{nil}),\ \mathrm{T} \to \mathit{cons}(\mathit{car}(x), G(\mathit{cdr}(x)))) \end{aligned}$$

In two additional corollaries, Summers extends the theorem to the more general case where the basic functions in the recurrences, leading to the substitutions in the recursive call, may be different for fragments and predicates.

Introducing Auxiliary Parameters

Recurrence relations as stated in (3.3) do not always exist such that the non-recursive program cannot immediately be inductively generalized as described. In this case a variable-addition heuristic is applied, transforming the original fragments into more general ones by replacing some common subexpression by a variable. For the resulting more general set of fragments, a recurrence relation may exist.

Example 3.5. Given the following examples for *reverse*,

$$\mathit{nil} \mapsto \mathit{nil},\ (a) \mapsto (a),\ (a,b) \mapsto (b,a),\ (a,b,c) \mapsto (c,b,a),$$

[2] This works, in a sense, reverse to interpreting a recursively expressed function by the partial function given as the fixpoint of the functional of the recursive definition. In the latter case we have a recursive program and want to have the particular partial function computed by it—here we have a partial function and want to have a recursive program computing it.

the program fragments

$$\begin{aligned}
f_1 &= \mathit{nil}, \\
f_2 &= \mathit{cons}(\mathit{car}(x), \mathit{nil}), \\
f_3 &= \mathit{cons}(\mathit{cadr}(x), \mathit{cons}(\mathit{car}(x), \mathit{nil})), \\
f_4 &= \mathit{cons}(\mathit{caddr}(x), \mathit{cons}(\mathit{cadr}(x), \mathit{cons}(\mathit{car}(x), \mathit{nil}))).
\end{aligned}$$

are derived. Differences are, with $\sigma = \emptyset$:

$$f_2 = \mathit{cons}(\mathit{car}(x), f_1\sigma),\; f_3 = \mathit{cons}(\mathit{cadr}(x), f_2\sigma),\; f_4 = \mathit{cons}(\mathit{caddr}(x), f_3\sigma)\,.$$

All differences have different contexts such that we cannot generalize them as described. Indeed, the *reverse* function cannot be implemented by only using the allowed primitives and without auxiliary recursive subprograms, an additional accumulator variable, or more complex recursive calls than linear recursion only.

We observe that *nil* is a subexpression common to all fragments, so we may replace it by a new variable yielding fragments of *two* variables:

Example 3.6.

$$\begin{aligned}
g_1 &= y, \\
g_2 &= \mathit{cons}(\mathit{car}(x), y), \\
g_3 &= \mathit{cons}(\mathit{cadr}(x), \mathit{cons}(\mathit{car}(x), y)), \\
g_4 &= \mathit{cons}(\mathit{caddr}(x), \mathit{cons}(\mathit{cadr}(x), \mathit{cons}(\mathit{car}(x), y)))\,.
\end{aligned}$$

Generally, let s denote such a common subexpression which is replaced by a variable y, leading to more general fragments g_i. Then we have $f_i = g_i\{y \leftarrow s\}$ for all i. If recurrence relations exist based on these more general fragments, then a recursive program G is synthesized from the fragments g_i (and the corresponding predicates), as described in the previous section, and $F(x)$ is set to $F(x) = G(x, s)$.

Example 3.7. For our *reverse* example, the following common differences exist, with $\sigma = \{x \leftarrow \mathit{cdr}(x), y \leftarrow \mathit{cons}(\mathit{car}(x), y)\}$: $g_2 = g_1\sigma,\; g_3 = g_2\sigma,\; g_4 = g_3\sigma$. Together with the differences identified for the predicates (not stated here) the resulting program, computing the reverse function, is:

$$\begin{aligned}
F(x) &= G(x, \mathit{nil}) \\
G(x, y) &= (\mathit{atom}(x) \rightarrow y,\ \mathrm{T} \rightarrow G(\mathit{cdr}(x), \mathit{cons}(\mathit{car}(x), y)))
\end{aligned}$$

3.2.2. Early Variants and Extensions

BMWk: Extended Forms of Recurrence Relations

In Summers' approach, the condition for deriving a recursive function from detected differences is that the differences hold—starting from an initial index j and for a particular interval n—recurrently along fragments and predicates with a constant context $C[\,]$ and a constant substitution σ for the variable x. The BMWK[3] algorithm [52, 42, 53, 43] of Kodratoff and his colleagues generalizes these conditions by allowing for contexts and substitutions that are different in each difference. Then a found sequence of differences originates a sequence of contexts and substitutions each. The new sequences are considered as fragments of new *subfunctions*. The BMWK algorithm is then recursively applied to these new fragment sequences, hence features the automatic introduction of subfunctions that are not explicitly specified. Furthermore, Summers' ad-hoc method to introduce additional variables is systematized by computing *least general generalizations (LGGs)* (see Definition 2.20) of successive fragments. The first step—deriving fragments and predicates—is identical to Summers.

Definition 3.5. A sequence of fragments $f_1, \ldots, f_k$ is a *matching sequence* iff there exists a sequence of substitutions $\sigma_2, \ldots, \sigma_k$ such that $f_i = f_{i-1}\sigma_i$ for all $2 \leq i \leq k$. For each $x \in Dom(\sigma)$, $\sigma_2(x), \ldots, \sigma_k(x)$ is a *generated sequence of fragments.*

This is similar to Summers' differencing; Definition 3.3. The extension is to consider the substitutions as new generated sequences of fragments. In Summers' recurrence relations, Equation (3.3), the substitution needs to be constant for the whole sequence, starting at index j. In contrast, the BMWK algorithm considers the, possibly non-constant, sequences of substitutions for each variable as fragments of new *subfunctions* and is recursively applied to them.

Remark 3.2. The definition exhibits two limitations compared to Summers. It only considers matchings at roots of fragments whereas Summers matches also subfragments, and it considers all and only directly consecutive fragment pairs whereas Summers considers initial non-recurring fragments and recurrence intervals $n \geq 1$. These are no inherent restrictions, the BMWK algorithm may also match (sub)fragments of any interval. We choose the limited version here in order to keep things simple and because it suffices to explain the ideas of the BMWK algorithm that extend Summers' method. If matching of subterms is allowed, then, in addition to the sequence of substitutions, also the sequence of possibly non-constant contexts constitute a new fragment sequence.

It may happen that a sequence of fragments is not a matching sequence, either because there exists a position denoting different functions in two consecutive fragments. Or because one and the same variable appears at at least two different positions in a fragment and had to be substituted by two different terms, which is not allowed, in order to get the fragment matched. In such a case, the fragment sequence is generalized. For an example, consider again the fragments for *reverse* as stated in Example 3.5. If, according to Definition 3.5, matching is allowed at root positions only, then these fragments do not form a matching sequence, because, for example, we have *nil* as root of f_1 and *cons* as root of f_2, such that f_2 cannot be expressed in terms of f_1.

[3]This abbreviates *Boyer-Moore-Wegbreit-Kodratoff*, because the algorithm has its roots in a paper of Boyer and Moore on proving theorems about LISP functions [15] and a paper of Wegbreit on goal directed program transformation [108].

$$\begin{aligned} F(x) &= F'(x, I(x)) \\ F'(x,y) &= (p_1(x) \to f_1(x,y), \\ &\qquad \vdots \\ &\qquad p_n(x) \to f_n(x,y), \\ &\qquad\quad \mathrm{T} \to H(x, F'(\sigma(x), G(x,y)))) \end{aligned}$$

Figure 3.3.: The general BMWk schema (due to [100, page 317]). H and G are either terms over the signature $\{cons, car, cdr\}$ or automatically introduced (sub)programs themselves fitting the schema and resulting from non-constant context- and/or substitution-sequences, respectively.

Definition 3.6. The sequence $g_1, \ldots, g_{k-1}$ is the *least generalized sequence* of a fragment sequence $f_1, \ldots, f_k$ iff $g_i = \mathbf{lgg}(f_i, f_{i+1})$ for all $1 \leq i \leq k-1$.

This least generalization method is a systematization of Summers' heuristic method of replacing any common expressions by a variable as described in Section 3.2.1. The least generalization of two fragments assures that systematically those expressions (from the first, to be matched, fragment) precluding the matching—due to the two possible reasons mentioned above—are replaced by new, unique variables.

The least generalized sequence for the fragments in Example 3.5 corresponds to the sequence of generalized fragments in Example 3.6, except for that the last fragment, g_4, is missing. Since each generalized fragment results from two (anti-unified) original fragments, it is inherent to this method of generalization that the generalized sequence contains one fragment less than the source sequence.

Induced programs fit the general schema shown in Figure 3.3. The different variables x, y allow for different substitutions for predicates p_i and functions f_i. I is some function initializing y.

Not every program fitting the schema in Figure 3.3 is inducible. Particularly (cp. Remark 3.2), H may only compute some *context* for the result of a recursive call of F'. For a counterexample, consider the definition for *reverse* that uses *Append* instead of an accumulator variable: of the recursive case of *reverse* that uses *Append*:

$$\begin{aligned} reverse(x) &= (atom(x) \to nil, \mathrm{T} \to Append(reverse(xs), x)), \\ Append(x,y) &= (atom(x) \to y, \mathrm{T} \to cons(car(x), Append(cdr(x), y)))\,. \end{aligned}$$

This program fits the schema with $F' = reverse$ and $H = Append$. In particular, *Append* itself fits the schema. However, *Append* does *not* compute a context for the recursive call of *reverse*. Consider for example the I/O-pair: $reverse([a,b,c]) = [c,b,a]$. Then the recursive call computes $[c,b]$ from $[b,c]$. But $[c,b]$ is not a subterm of $[c,b,a]$, i.e., there is no context for $[c,b]$ yielding $[c,b,a]$. Hence, this definition of *reverse* with *Append* as subfunction cannot be induced by the BMWk algorithm.

Biermann et al: Pruning Enumerative Search Based on Single Traces

Summers objective was to avoid search and to justify the synthesis by an explicit inductive inference step and a subsequent proven-to-be-correct program construction step. This could be achieved by a restricted program schema and the requirement of a well chosen set of I/O examples.

On the contrary, Biermann's approach [11] is to employ traces (fragments) to speed up an exhaustive enumeration of a well-defined program class, the so-called *regular* LISP *programs*. Biermann's objectives regarding the synthesis were

1. *convergence* to the class of regular LISP programs,
2. convergence on the basis of *minimal input information*,
3. robust behavior on different inputs.

Particularly 2 and 3 are contradictory to the recurrence detection method—by 2 Biermann means that no synthesis method exists which is able to synthesize every regular LISP program from fewer examples and by 3 he means that examples may be chosen randomly.

A *semiregular program* f is a finite non-empty set of *component* programs $f_i, i = 1, \dots, m$ with f_1 being the *initial* component, $f(x) = f_1(x)$. A component program has the form

$$f_i(x) = (p_{i1}(x) \to f_{i1}(x), \dots, p_{in}(x) \to f_{in}(x))$$

where the predicates form a chain and each f_{ij} is either nil or x or $f_h(car(x))$ or $f_h(cdr(x))$ or $cons(f_h(x), f_k(x))$ with f_h, f_k being component programs of f. Predicates $p_1, \dots, p_k$ form a chain iff they have the form $atom(b_i(x))$ for $1 \leq i < k$ and $p_k = T$ where the b_i are basic functions and b_i is a suffix of b_{i+1} for $1 \leq i < k-1$. If the conditional of a component program f_i consists of only one predicate/function pair $(T \to f_{i1})$ then the conditional is not needed and it is abbreviated to $f_i(x) = f_{i1}(x)$.

A *regular* program is a semiregular program satisfying some precedence criteria regarding application of the primitive functions, if different orders are possible. The class of regular programs is decidable and a property is stated [11] for deciding whether a program belongs to the class. It is strictly smaller than the class of computable functions.

The enumeration. After program fragments (Section 3.2.1) have been computed for each, possibly only one, I/O example, they are rewritten as regular programs (without recursion and conditionals). Figure 3.4 shows one I/O pair for the *Init* function, its program fragment (cp. Section 3.2.1) and the fragment rewritten as a regular program.

In the following, with *trace* we always refer to the regular program form of a program fragment. Note that each component program of a trace has an associated (example) input resulting from the input of the I/O example for f_1 and the computations of the precedent component programs in the trace. In Figure 3.4, f_1, f_2, f_3 have the original input (a, b, c), f_4 has input a, f_5 has input (b, c), and f_6, f_7, f_8 have input b.

An I/O pair for the *Init* function: $(a, b, c) \mapsto (a, b)$.
The corresponding fragment: $\mathit{cons}(\mathit{car}(x), \mathit{cons}(\mathit{car}(\mathit{cdr}(x)), \mathit{nil}))$.
The fragment rewritten as regular program:

$$\begin{aligned}
f_1(x) &= \mathit{cons}(f_2(x), f_3(x))\\
f_2(x) &= f_4(\mathit{car}(x))\\
f_3(x) &= f_5(\mathit{cdr}(x))\\
f_4(x) &= x\\
f_5(x) &= f_6(\mathit{car}(x))\\
f_6(x) &= \mathit{cons}(f_7(x), f_8(x))\\
f_7(x) &= x\\
f_8(x) &= \mathit{nil}
\end{aligned}$$

Figure 3.4.: One I/O pair for the *Init* function, its fragment, and the fragment rewritten as regular program

The further synthesis, the generalization step, is effectively an enumeration of regular programs, in ascending order regarding the number of component programs. It stops when a program reproducing all traces has been found. Only those programs reproducing at least some initial sequence of the given traces are enumerated.[4] This drastically prunes the set of considered regular programs but certainly does not exclude possible solutions from consideration.

For the following description of the generalization we assume that exactly one I/O example, i.e., one trace is provided. Biermann's method works as follows: Initially, the trace is tried to be expressed by only one component function. That is, all components f_i of the trace need to be merged together into one and the same initial component f_1. The different trace components are accounted for by integrating them into a McCarthy conditional within f_1. Therefore, a chain of predicates distinguishing the inputs associated with the different trace components is generated. This is done in a way similar to the method described in Section 3.2.1. If all different expressions can be accounted for we are done. If not, a regular program with two components is searched for.

For the trace in Figure 3.4 it is not possible to merge all component functions. To see this, observe that the components f_1, f_2, f_3 clearly carry out different computations, thus need to become functions in different predicate/function-pairs in the McCarthy conditional building the body of the resulting single component function. Yet the input of f_1, f_2, f_3 is the same, (a, b, c), such that they cannot be distinguished.

Searching for a program with two components means to try all possible partitions of the trace components into two subsets. The components of each subset are again tried to be merged together. If this does not succeed, all partitions into three subsets are tried

[4]This method is an adaptation to the Lisp domain of earlier work [12, 13] on program synthesis from traces.

and so on. It is obvious that, if no regular program with fewer components exists, the search eventually stops with the reproduced trace as solution.

For our example, merging into two components fails for the same reason as merging into one component—at least three components are necessary to properly distinguish f_1, f_2, f_3. Indeed, three components suffice.

Example 3.8. Consider the partitioning of the eight components from Figure 3.4 into three subsets representing three components of the induced program. f_1, f_2, f_3 must belong to different subsets due to their same input but different computations. Components f_4 and f_7 may be merged into one component, obviously. Again f_4, f_7, f_6, and f_8 must belong to different subsets because they carry out different computations yet have, structurally, the same input. One solution fulfilling these constraints is the partition

$$\{f_1, f_5, f_6\}, \{f_2, f_4, f_7\}, \{f_3, f_8\}\,.$$

The resulting induced program, correctly implementing the *Init* function, is

$$\begin{aligned}
f_1(x) &= (\mathit{atom}(x) \to \mathit{cons}(f_2(x), f_3(x)),\ \mathit{atom}(\mathit{cddr}(x)) \to f_1(\mathit{car}(x)),\\
&\qquad \mathrm{T} \to \mathit{cons}(f_2(x), f_3(x)))\\
f_2(x) &= (\mathit{atom}(x) \to x,\ \mathrm{T} \to f_2(\mathit{car}(x)))\\
f_3(x) &= (\mathit{atom}(x) \to \mathit{nil},\ \mathrm{T} \to f_1(\mathit{cdr}(x)))\,.
\end{aligned}$$

As we have seen, many partitions may be skipped over. This is, generally, because if one particular subset fails, i.e., its functions cannot be merged together, then all combinations of partitioning the remaining components need not be considered.

Biermann's pruned enumeration algorithm for regular LISP programs has all properties of the enumeration algorithm (with respect to the class of regular LISP programs), see Section 3.1.4.

3.2.3. Igor1: From S-expressions to Recursive Program Schemes

At the beginning of Section 3.2.1 we stated the LISP primitives as used in programs induced by Summers' method (as well as by BMWK and Biermann's method). This selection is crucial for the first step, the deterministic construction of first approximations, yet not for the generalization step. Indeed, the latter is independent from particular primitives, it rather relies on matching (sub)terms over arbitrary first-order signatures. The recent system IGOR1 [79, 94, 51], inspired by Summers' recurrence detection method, induces *recursive program schemes*, a particular kind of (constructor) term rewriting systems.

A recursive program scheme (RPS) [21] is a TRS with the following characteristics: The signature is divided into two disjoint subsets $\mathcal{F}$ and $\mathcal{G}$, called *unknown* and *basic* functions, respectively; rules have the form $F(x_1, \ldots, x_n) \to t$ where $F \in \mathcal{F}$ and the

x_i are variables, and there is exactly one rule for each $F \in \mathcal{F}$. One of the unknown functions is distinguished as the *main* function.[5]

IGOR1 automatically introduces recursive subfunctions that are not explicitly specified, if needed, as well as additional variables. Induced functions can be tree-recursive instead of only linear recursive. (Recursive) calls of defined functions can not be nested, however.

Recursive RPSs do not terminate. Their standard interpretation is the infinite term defined as the limit $\lim_{n\to\infty, F(\boldsymbol{x}) \overset{n}{\to} t} t$ where F denotes the main rule of the RPS. One gets finite approximations (in the sense of domain theory) by replacing infinite subterms by the special symbol Ω, meaning *undefined*. Certainly, such an infinite tree and its approximations contain recurrent patterns because they are generated by *repeatedly* replacing instances of LHSs of the rules by instances of their RHSs. In the case of RPSs we call this rewriting process *unfolding* (of the RPS) and the infinite term as well as its finite approximations, *(finite) unfoldings*.

IGOR1 takes a finite approximation t of some (hypothetical) infinite tree (i.e., a finite unfolding of some hypothetical RPS) as input, discovers the recurrent patterns in it, and builds an RPS R such that t is a finite unfolding of R. We call such finite approximating terms *initial terms*. We say that IGOR1 *folds* the provided initial term to an RPS. Similar to Summers inductive inference step, the technique requires that discovered patterns recur a few times such that there is some amount of evidence for the induced RPS.

Folding of Initial Terms to Recursive Program Schemes

We describe the folding method by means of an example.

Example 3.9. Consider the following RPS that takes a list of lists and computes a list of the last elements of the input lists:

$$\begin{aligned} &\mathit{Lasts}(x) \to \mathit{if}(\mathit{atom}(x), \mathit{nil}, \mathit{cons}(\mathit{car}(\mathit{Last}(\mathit{car}(x))), \mathit{Lasts}(\mathit{cdr}(x)))) \\ &\mathit{Last}(x) \;\;\to \mathit{if}(\mathit{atom}(\mathit{cdr}(x)), x, \mathit{Last}(\mathit{cdr}(x))) \end{aligned}$$

Figure 3.5 shows one of its finite approximations.

The goal of folding is to induce an RPS from an initial term. In our example, the *Lasts* RPS shall be induced from the initial tree of Figure 3.5. To this end, the initial term is considered as finite unfolding of the (unknown) RPS to be induced.

The induction of each rule is done in three steps: First, positions in the initial tree are identified where the rule has been unfolded, i.e., where (instances of) its LHS has been replaced by (instances of) its RHS. This divides the initial tree into segments. Second, the RHS (except for the recursive calls) is computed by antiunifying the segments. The

[5]Actually, an RPS is a special constructor system: The unknown and basic functions correspond to the defined functions and constructors, respectively, in a CS. Yet whereas the formal arguments of a defined function symbol in the LHS may be constructor terms in a CS, they are restricted to be single variables in an RPS.

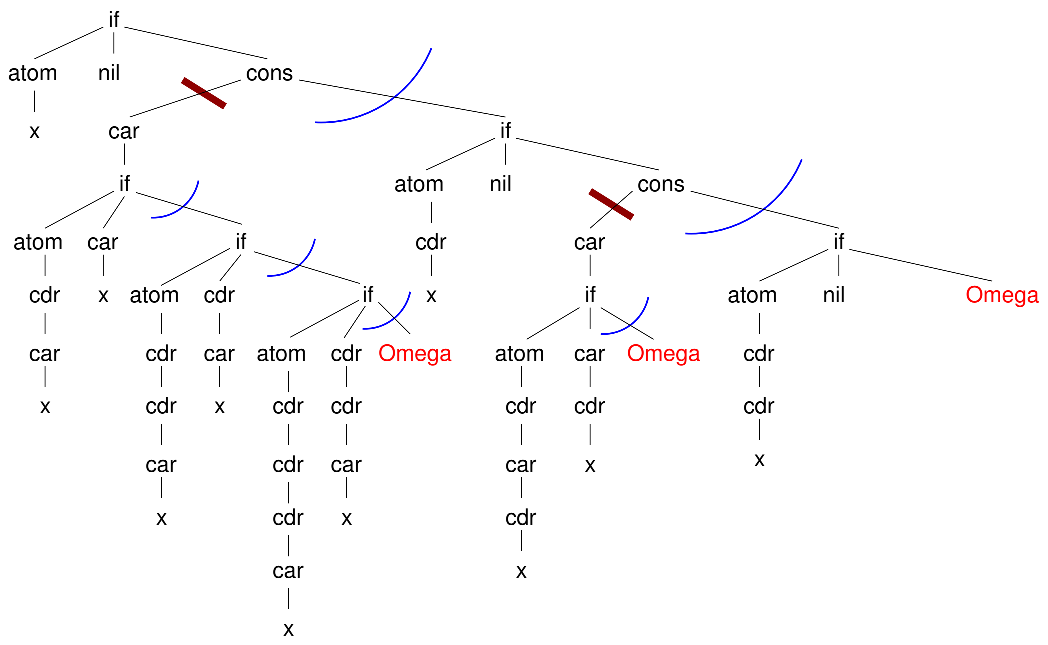

Figure 3.5.: A finite approximating tree for the *Lasts* RPS. The intended meaning of *cdr*, *car*, *cons*, and *atom* is as stated at the beginning of Section 3.2.1, that of *if* is a 3-ary function that takes the value of its second parameter if its first parameter evaluates to *true* and its third parameter otherwise.

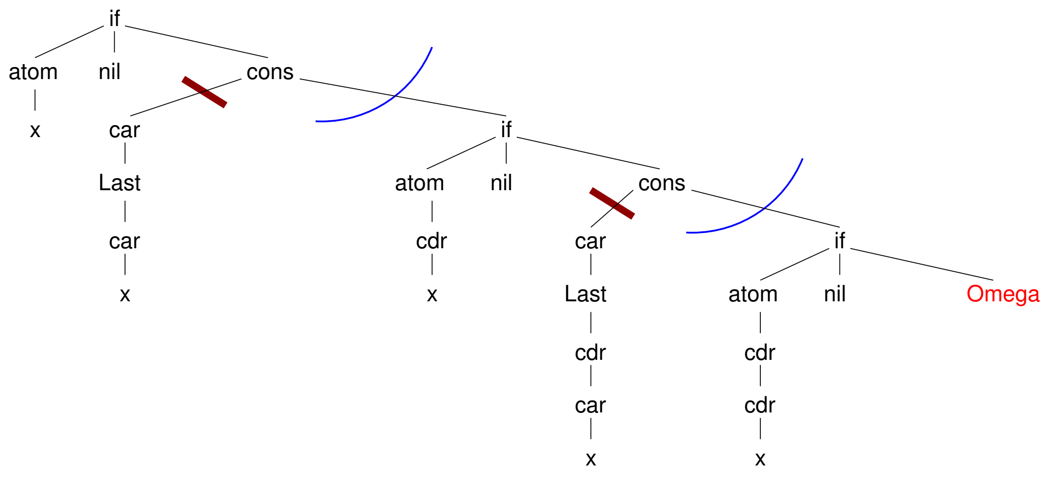

Figure 3.6.: Reduced Initial Tree for *Lasts*

variables of the resulting least general generalization of the set of segments are taken as variables of the function. And third, the substitutions of the induced variables within the recursive calls are computed from the instances of the variables regarding the different sequences.

The segmentation of the initial tree is computed as follows: Since in the *finite* unfolding process, each LHS is replaced by an Ω eventually, the unfolding positions must lie at paths to Ωs. Moreover, the sequence of symbols *between* these positions must always be the same since they belong to the same rule. Hence, we search for recurring symbol sequences from the root of the tree to the Ωs. In our example tree we find a recurring $\mathit{if} - \mathit{cons}$ sequence from the root to the rightmost Ω. The corresponding segmentation of the tree is indicated by the blue curves on that path. We do not find this sequence at paths to the other Ωs. Therefore, we assume that the remaining Ωs belong to a further *sub*function, that is called by the main function. We cut the corresponding subtrees as indicated by the brown bars and consider them as unfoldings of that subfunction.

Before steps two and three are applied, the assumed additional subfunctions, only one in our example, are induced from the subtrees by recursively applying the folding algorithm to them. If this has been done, the initial tree is reduced by replacing the subtrees by suitable calls to the induced subfunctions. For our example, this results in the tree shown in Figure 3.6.

The segments of such a reduced tree are now taken as different terms (the positions where the tree is divided become the name of the function since these are the positions of the recursive calls), each one an instantiation of the RHS of the assumed function. For our example tree this leads to the two terms

1. $\mathit{if}(\mathit{atom}(x), \mathit{nil}, \mathit{cons}(\mathit{car}(\mathit{Last}(\mathit{car}(x))), \mathit{Lasts}))$,
2. $\mathit{if}(\mathit{atom}(\mathit{cdr}(x)), \mathit{nil}, \mathit{cons}(\mathit{car}(\mathit{Last}(\mathit{car}(\mathit{cdr}(x)))), \mathit{Lasts}))$.

(The third segment is incomplete and not considered.)

$$
\begin{aligned}
nil &\mapsto nil,\\
(a) &\mapsto (a),\\
((a,b)) &\mapsto (b),\\
((a,b,c),(d)) &\mapsto (c,d),\\
((a,b,c,d),(e,f)) &\mapsto (d,f),\\
((a),(b),(c)) &\mapsto (a,b,c)
\end{aligned}
$$

Figure 3.7.: I/O examples specifying the *Lasts* function

Since the first of these two segments subsumes the second one, it is also the LGG, i.e., the induced RHS without substitution of the only recursive call. The only variable is x and the substitution of x in the second segment, i.e., in the recursive call, is $\{x \mapsto cdr(x)\}$.

This leads to the rule for *Lasts* as stated in the RPS in Example 3.9.

Achieving Finite Approximations

The folding algorithm, as described above, is, like BMWk, a generalization of Summers' second synthesis step—the inductive generalization based on discovering syntactical recurrences in an approximating program. Yet whereas BMWk deals with approximating programs as constructed in Summers' first synthesis step—a chain of predicate/fragment pairs integrated into a McCarthy conditional—, the folding algorithm deals with the more general concept of finite terms over any first-order signature. There is no straightforward adaptation of Summers' first step to this new concept and, more important, a simple adaptation would not take advantage of the generalization.

Different methods to achieve a finite approximating term have been proposed. Kitzelmann and Schmid [51] describe an extension of Summers' first step. Inputs and outputs are S-expressions, manipulated by a fixed set of primitives as in Summers' approach. Yet they need *not* be linearly ordered.

For example, the *Lasts* function could be specified by the I/O examples shown in Figure 3.7. They lead to the initial term of Figure 3.5 with the following method:

First, program fragments are generated for all given I/O examples as described in Section 3.2.1. The initial term is then constructed by going through all fragments in parallel position by position. If the same function symbol stands at the current position in all fragments, then it is introduced to the initial term at this position. If at least two fragments differ at the current position, then an if-then-else-fi-expression is introduced. Therefore, a predicate function is generated to distinguish the inputs according to the different fragments. Construction of the initial term proceeds from the partitioned inputs and fragments for the `then`- and `else`-branch, respectively.

As another approach of achieving finite approximations, Wysotzki, Schmid, and Kitzelmann [96, 94, 47] propose *universal planning* and different subsequent transformations as first step to program synthesis. In this case, starting point is not a set of I/O examples but a planning problem over a finite domain of objects. Result of universal planning is a graph whose nodes denote states and whose paths denote optimal plans from the states to some distinguished goal state. Such a universal plan must then be transformed into an initial term that can be generalized by the folding algorithm. This transformation is not satisfyingly solved yet. In particular it seems as if the transformation to an appropriate approximating term is not possible without detecting recurrences in the universal plan already such that the serial execution of plan transformation and folding contains serious redundancies. A second problem is that a classical plan describes a *linear* transformation from an initial state (an input) to the goal state (an output). As a consequence, the planning approach can only construct approximating terms of linearly recursive RPSs. These issues are analyzed in Kitzelmann's diploma thesis [47].

3.2.4. Discussion

Summers' important insights were first, how the algebraic properties of S-expressions can be exploited to construct program fragments and predicates uniquely without search and second, that fragments (and predicates) for different I/O pairs belonging to one recursively defined function share recurrent patterns that can be used to identify a recursive function definition, correctly computing the provided I/O examples.

Obviously, this recurrence detection method relies on sets of I/O examples that are not randomly chosen but "complete" in the sense that if one particular example input is given, then also all inputs that are generated by recursive calls of the (hypothetical) target function definition must be included. Hence, a set of I/O examples must contain the first $k \in \mathbb{N}$ I/O pairs with respect to the order of the domain of the target function established by the recursive call. For example, if the domain is *lists* and in the recursive call the rest list is taken, then this orders lists by their length such that all lists up to a certain length must be provided as example inputs.

To see this by means of the *Init* example, again consider the I/O examples for *Init* in Figure 3.2. If, for example, the second I/O pair would be missing, then the recurrence relation in Example 3.3 could not be established and the recursive *Init* implementation could not be derived.

Hence the recurrence detection method presupposes that the specifier has some knowledge regarding the recursive scheme of the function to be induced. This phenomenon also applies to inductive logic programming methods as described in the next section, so we delay further discussion of this point to the conclusion 3.5 of this chapter.

The unique construction of program fragments relies on the unique construction and decomposition of S-expressions by *cons*, *car*, and *cdr*. That is, this technique excludes the use of background knowledge. Furthermore, the derivation of the recursive pattern from the examples without search relies on the restricted linear-recursive program schema.

Both—no usage of background knowledge and the simple program schema—are strong restrictions regarding the class of programs that can be induced and prevent Summers' approach in its original form from being practically relevant. Especially the support of background knowledge is important from an artificial intelligence perspective—using existing knowledge to find simpler solutions or to find solutions at all—as well as from a programming perspective—re-use of code, use of libraries.

The BMWk algorithm extends Summers original method by relaxing the strongly restricted program schema. Certainly, this may lead to non-deterministic steps during the program induction—for example, computing a matching sequence with non-equal substitutions and recursively applying BMWk to the derived substitution sequence versus computing a generalized sequence and recursive applying BMWk to that sequence.

The deterministic construction of program fragments as well as the detection of syntactical recurrences is not restricted to S-expressions and the respective constructors and selection functions but can be extended to terms over arbitrary algebraic signatures. This idea, besides and extension of Summers' original schema, is realized in the relatively recent Igor1 system.

Finally, Biermann's approach shows that it is possible to use program expressions computing single I/O pairs as derived in Summers' first synthesis step to prune the exhaustive enumeration of a program class as done by the enumeration algorithm (Algorithm 1) without loosing its properties stated in Theorem 3.1.

3.3. Inductive Logic Programming

Inductive Logic Programming (ILP) [74, 78, 81] is a branch of machine learning—intensional concept descriptions are learned from examples and counterexamples, called *positive and negative examples.* The specificity of ILP lies in its basis in computational logic: First-order clausal logic is used as uniform representation language for induced programs (hypotheses), examples, and background knowledge, semantics of ILP is based on entailment, and inductive learning techniques are derived by regarding induction as the inverse of deduction.

Horn clause logic together with resolution constitutes the (Turing-complete) programming language Prolog. Program synthesis is therefore principally within the scope of ILP and has been regarded as one application field of ILP [78]. One of the first ILP systems, MIS [99], is an automatic programming/debugging system. Today, however, ILP is concerned with (relational) data-mining and knowledge discovery and program synthesis does not play a role anymore.

3.3.1. Overview

In general, hypotheses, examples, and background knowledge are sets of clauses. Hypotheses and background knowledge are finite. A common restriction, which is sufficiently adequate for our program synthesis perspective, is to restrict hypotheses and background knowledge to be definite programs and examples to be ground atoms.

On Positive and Negative Examples

In concept learning in general and in inductive logic programming in particular, it is common to have *negative examples*, i.e., examples of what the concept or relation to be learned is *not*, in addition to the positive examples of what the concept or relation *is*. This seems to be in contrast to the (only positive) I/O examples in the inductive synthesis of functional programs (Sections 3.2 and 3.4) and one might ask, why negative examples are necessary in one setting while unnecessary in the other.

We do not give answers here for the question which concepts are theoretically learnable from positive and negative or only positive examples. We rather want to give a short argument that the apparent discrepancy of only positive examples in one setting and positive and negative examples in the other setting is no discrepancy but that negative examples in the relational setting actually follow from the only positive examples in the functional setting and vice versa. The argument is based on considering functions mathematically as special relations and, on the other hand, relations as special, boolean-valued, functions.

Since we were concerned with functions until now, we first consider relations as boolean-valued functions. A relation $r \subseteq X_1 \times \cdots \times X_n$ is considered as the function $f_r : X_1 \times \cdots \times X_n \to \mathbb{B}$, with $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$, defined by:

$$f_r(x_1, \ldots, x_n) = \begin{cases} \mathit{true} & \text{if } (x_1, \ldots, x_n) \in r \\ \mathit{false} & \text{if } (x_1, \ldots, x_n) \notin r \end{cases} .$$

For example, consider the relation $\mathit{Even} \subseteq \mathbb{N}$, containing all even natural numbers and the corresponding boolean-valued function $f_{\mathit{Even}} : \mathbb{N} \to \mathbb{B}$. If we specified the function f_{Even} by I/O pairs, we reasonably would provide examples for both possible outputs. For example:

$$\{\langle 0, \mathit{true}\rangle, \langle 1, \mathit{false}\rangle, \langle 2, \mathit{true}\rangle, \langle 3, \mathit{false}\rangle\} .$$

All these examples are *positive* in that they say what the function value *is* for the specified inputs, instead of what it is *not*; e.g., $\langle 1, \mathit{true}\rangle$. The same examples as specification for the *relation Even* would yet be:

$$E^+ = \{0, 2\}, \quad E^- = \{1, 3\} ,$$

where E^+ means *positive examples* and E^- means *negative examples*.

Hence one can say that if one specifies (only positive) I/O examples for a boolean-valued function, this corresponds to provide positive as well as negative examples from the perspective of specifying a relation.

Now let us consider functions mathematically as special relations—namely as (single-valued) relations $r \subseteq X_1 \times \cdots \times X_n$ where the product set $X_1 \times \cdots \times X_n$ can be partitioned into a domain $X_1 \times \cdots \times X_k$ and a codomain $X_{k+1} \times \ldots \times X_n$ ($k \leq n$) such that $(x_1, \ldots, x_n) \in r$ implies $(x_1, \ldots, x_k, y_{k+1}, \ldots, y_n) \notin r$ if $y_i \neq x_i$ for any $i \in \{k+1, \ldots n\}$.

For example, consider the *Init* function again. Some (positive) examples for *Init*, considered as relation, would be

$$E^+ = \{([a], [\,]),\ ([a, b], [a]),\ ([a, b, c], [a, b])\} .$$

Now if an ILP system "knows" that the specified object is a *functional* (single-valued) relation, these positive examples immediately imply negative examples: all pairs with first components occurring in E^+ but different second components than in E^+ are negative examples; e.g., since $([a, b], [a]) \in E^+$, it is $([a, b], [b]) \notin Init$, hence implicitly $([a, b], [b]) \in E^-$.

Hence, the knowledge that the specified object is actually a function is a kind of inductive bias from the relational perspective that allows for omitting negative examples.

The ILP Problem

The ILP problem, restricted to the definite setting and respecting the *normal semantics*[6] is given by the following two definitions.

Definition 3.7. Let Π be a definite program and E^+, E^- be positive and negative examples. Π is

complete with respect to E^+ if $\Pi \models E^+$,

consistent with respect to E^- if $\Pi \not\models e$ for every $e \in E^-$,

correct with respect to E^+ and E^- if it is complete and consistent with respect to E^+ and E^-, respectively.

Definition 3.8 (The ILP problem)**.** Given

- a set of possible hypotheses (definite programs) $\mathcal{H}$,
- positive and negative examples E^+, E^-,
- consistent background knowledge B (i.e., $B \not\models e$ for every $e \in E^-$) such that $B \not\models E^+$,

find a hypothesis $H \in \mathcal{H}$ such that $H \cup B$ is correct with respect to E^+ and E^-.

Entailment ($\models$) is undecidable in general and for Horn clauses, definite programs, and between definite programs and single atoms in particular. Thus, in practice, different decidable (and preferably *efficiently* computable) *coverage* relations, which are sound but more or less incomplete with respect to entailment, are used. We say that a program *covers* an example if it can be proven true from it by some sound, but not necessarily complete, proof calculus. Hence, a program is correct if it covers all positive and no negative examples, but not each correct program is covered such that some actual solutions will be rejected.

Two commonly used notions of coverage are:

[6]This is also called *explanatory* setting because the induced program "explains" the truth of the positive and the non-truth of the negative examples. There is also a *non-monotonic* setting in ILP where hypotheses need not to entail the positive examples but shall only state true properties. This is useful for *data mining* and *knowledge discovery* but not for inductive programming, so we do not consider it here.

Extensional coverage. Given a hypothesis H, a finite set of ground atoms B as background knowledge, positive examples E^+, and an example e, H (together with B) *extensionally covers* e if there exists a clause $(A \leftarrow B_1, \ldots, B_n) \in H$ and a substitution θ such that $A\theta = e$ and $\{B_1, \ldots, B_n\}\theta \subseteq B \cup E^+$.

Intensional coverage. Given a program P and an example e, P *intensionally covers* e if e can be proven true from P by applying some terminating proof technique, e.g., depth-bounded SLD-resolution.

"Extensional" refers to the fact that coverage of an example mostly relies—except for one single clause—on the (incomplete) *extensional* problem description in terms of example- and background knowledge facts. On the other side, "intensional" refers to the fact, that coverage of an example relies on the *intensional* definition of a relation by a (recursive) program.

Example 3.10. Let $E^+ = \{\, \mathit{Init}([c], [\,]), \mathit{Init}([b,c], [b]), \mathit{Init}([a,b,c], [a,b]) \,\}$, $B = \emptyset$, and $P = \{\, \mathit{Init}([X], [\,]),\ \mathit{Init}([X \mid \mathit{Xs}], [X \mid \mathit{Ys}]) \leftarrow \mathit{Init}(\mathit{Xs}, \mathit{Ys}) \,\}$. P implements the *Init* function, therefore entails E^+, and also covers all examples in E^+, both extensionally and intensionally. To see this for the extensional case, consider, e.g., the positive example $\mathit{Init}([a,b,c], [a,b])$ and the recursive clause in P. The head of this clause is matched with the example by $\theta = \{X \leftarrow a, \mathit{Xs} \leftarrow [b,c], \mathit{Ys} \leftarrow [b]\}$. Applying θ to the body of the clause yields $\mathit{Init}([b,c],[b]) \in E^+$.

Both extensional and intensional coverage are sound. Extensional coverage is more efficient but less complete. For example, suppose the positive example $\mathit{Init}([c], [\,])$ is missing in E^+ in Example 3.10. Then P still intensionally covers $e = \mathit{Init}([b,c],[b])$, yet not covers e extensionally anymore. Obviously, extensional coverage requires that positive examples (and background knowledge) are complete up to some complexity as it is also required for the analytical induction of Lisp programs (cp. Section 3.2.4). Another problem with extensional coverage is that if two clauses each do not cover a negative example, both together possibly do [88].

Intensional and extensional coverage are closely related to the general ILP algorithm (Algorithm 2) and the covering algorithm (Algorithm 3) as well as to the generality models *entailment* and *θ-subsumption*, respectively, as described below in Section 3.3.2.

ILP as Search

ILP is considered as a search problem in a space of (definite) logic programs. Typically, operators to compute successor programs are based on the dual notions of *generalization* and *specialization* of programs and clauses. There exist different but related notions of generality. The basic one is the following.

Definition 3.9. A program Π is *more general than* a program Φ if $\Pi \models \Phi$. Φ is said to be *more specific than* Π.

This structure of the problem space provides a way to prune the search tree. If a program is not consistent then all generalizations are also not consistent and therefore need not be considered. This dually holds for non-completeness and specializations. Hence, if a program is not complete, it must be generalized. If it is not consistent, it must be specialized. This leads to Algorithm 2. Most ILP systems are instances of this generic algorithm.

Algorithm 2: A generic ILP algorithm

```
  Input: Positive and negative examples $E^+$, $E^-$
  Input: Background knowledge $B$
  Output: A definite program $H$ such that $H \cup B$ is correct with respect to $E^+$
          and $E^-$
1 Start with some initial (possibly empty) hypothesis $H$;  repeat
2     if $H \cup B$ is not consistent then specialize $H$
3     if $H \cup B$ is not complete then generalize $H$
4 until $H \cup B$ is correct with respect to $E^+$ and $E^-$
5 return $H$
```

A commonly used instance is the (sequential) *covering algorithm* (Algorithm 3). The individual clauses of a program are searched for independently one after the other.[7] Hence, the problem space is not the *program* space (*sets* of clauses) but the *clause* space (*single* clauses). This leads to a more efficient search. However, independent learning of clauses conflicts with intensional coverage such that ILP systems instantiating the covering algorithm normally rely on extensional coverage.

Algorithm 3: The covering algorithm (typically applying extensional coverage)

```
1 Input and Output as in Algorithm 2
2 Start with the empty hypothesis $H = \emptyset$
3 repeat
4     Add a clause $C$ not covering any $e \in E^-$ to $H$
5     Remove all $e \in E^+$ covered by $C$ from $E^+$
6 until $E^+ = \emptyset$
7 return $H$
```

Entailment ($\models$) as well as θ-subsumption (see Section 3.3.2 below) are *quasi-orders* on sets of definite programs and clauses, respectively. We associate "more general" with "greater". The operators carrying out specialization and generalization are called *refinement operators*. They map clauses to sets of (refined) clauses or programs to sets of (refined) programs. Most ILP systems explore the problem space mainly in one

[7]The procedure that computes the single clauses (line 4 in Algorithm 3) is usually called LEARN−ONE−RULE.

direction, either from general to specific (*top-down*) or the other way round (*bottom-up*). The three well-known ILP systems Foil [86] (top-down), Golem [72] (bottom-up), and Progol [77] (mixed) are instances of the covering algorithm.

Example 3.11. For an example of the covering algorithm, let E^+ and B be as in Example 3.10 and E^- all remaining instantiations for the "inputs" $[c], [b,c], [a,b,c]$, e.g., $\mathit{Init}([b,c],[c])$. Let us assume that our concrete instance of the covering algorithm uses the top-down approach. That means that clauses are generated by starting with a (too) general clause and then successively specializing it until no negative examples are covered anymore. We start with $H = \emptyset$ and construct the first clause: Suppose we start with the clause $\mathit{Init}(\mathit{Xs}, \mathit{Ys}) \leftarrow$. Due to non-compound patterns (simply variables) in the head and an empty body, it (extensionally) covers all positive and also all negative examples and hence, is too general. We may specialize it by applying the substitution $\{\mathit{Xs} \leftarrow [X]\}$. The resulting clause $\mathit{Init}([X], \mathit{Ys}) \leftarrow$ excludes all positive and negative examples from being covered except for those with input $[c]$. We may further specialize it by applying the substitution $\{\mathit{Ys} \leftarrow [\,]\}$, thereby excluding all remaining positive and negative examples from being covered except for the single positive example $\mathit{Init}([c],[\,])$. Since no negative example is covered by this clause, it is added to H and $\mathit{Init}([c],[\,])$ is removed from E^+. Since there are still uncovered positive examples, a second clause is to be constructed: We start again with $\mathit{Init}(\mathit{Xs}, \mathit{Ys}) \leftarrow$ which is too general. Now we specialize it by substitution $\{\mathit{Xs} \leftarrow [X \mid \mathit{Xs}]\}$ yielding $\mathit{Init}([X \mid \mathit{Xs}], \mathit{Ys}) \leftarrow$. This clause still covers all remaining positive examples (those with inputs $[b,c]$ and $[a,b,c]$), but also all negative examples with these inputs. Applying the substitution $\{\mathit{Ys} \leftarrow [X \mid \mathit{Ys}]\}$ specializes it to $\mathit{Init}([X \mid \mathit{Xs}], [X \mid \mathit{Ys}]) \leftarrow$. This excludes some (but not all) negative examples (e.g., $\mathit{Init}([b,c],[c])$). We specialize it by adding the literal $\mathit{Init}(\mathit{Xs}, \mathit{Ys})$ to the body and get $\mathit{Init}([X \mid \mathit{Xs}], [X \mid \mathit{Ys}]) \leftarrow \mathit{Init}(\mathit{Xs}, \mathit{Ys})$. All remaining positive examples are still covered but no negative one is covered anymore. Hence, the clause is added and the remaining positive examples are removed from E^+, such that $E^+ = \emptyset$ now. Hence, the two induced clauses are returned as solution.

All specializations in the example were refinements under θ-subsumption (see the subsection on θ-subsumption in Section 3.3.2 below). Of course, many other candidate specializations would have been generated and evaluated by a real top-down covering ILP algorithm like Foil, such that constructing each clause becomes, indeed, a *search* in clause space, rather than a straight-forward construction as in our example.

Refinement operators in general. Refinement operators compute generalizations or specializations under particular generality models (see Section 3.3.2) of programs or single clauses. In particular, a *specialization (downward refinement) operator* maps a clause or program to a set of more specific clauses or programs. A *generalization (upward refinement) operator* maps a clause or program to a set of more general clauses or programs. Before we define refinement operators for particular objects (clauses or programs) and orderings (θ-subsumption or entailment), we introduce them in an abstract way as functions on quasi-ordered sets:

Definition 3.10. Let $\langle G, \leq \rangle$ be a quasi-ordered set. A *specialization operator* for $\langle G, \leq \rangle$ is a function ρ with $\rho(C) \subseteq \{D \in G \mid C \geq D\}$ for every $C \in G$.

A *generalization operator* for $\langle G, \leq \rangle$ is a function δ with $\delta(C) \subseteq \{D \in G \mid D \geq C\}$ for every $C \in G$.

In order to achieve a computable and rather complete and efficient search through the ordered set, Nienhuys-Cheng and De Wolf [81] define and study the following properties of refinement operators, exemplarily defined for specialization operators. By $\rho^*(C)$ we denote the set of all elements that can be generated by a finite number of applications of ρ.

Definition 3.11. Let $\langle G, \leq \rangle$ be a quasi-ordered set and ρ a specialization operator for it.

- ρ is *locally finite* if for every $C \in G$, $\rho(C)$ is finite and computable.
- ρ is *complete* if for every two clauses $C, D \in G$ with $C \geq D$ there is an $E \in \rho^*(C)$ with $D \equiv E$.
- ρ is *proper* if for every $C \in G$, $\rho(C) \subseteq \{D \mid C > D\}$.
- ρ is *ideal* if it is locally finite, complete, and proper.

Analogous definitions can be stated for the dual case of generalization operators.

Local finiteness is required to consider each generated program in finite time. Completeness means that any specialization can actually be computed by ρ in finite time. If ρ is not complete, then some specializations cannot be found by a finite number of computations. Properness prevents the computation of equivalences and is thus desirable for efficiency reasons.

Muggleton and De Raedt [78] define refinement operators to compute *minimal* generalizations and *maximal* specializations only and define different properties; informally: ρ is

- *globally complete*, if all elements of the ordered set can be generated by repeated applications of ρ starting from the top element (if it exists),
- *locally complete*, if for all elements C in the ordered set, an application of ρ yields all maximal specializations of C, and
- *optimal*, if only one path exists between two elements by repeatedly applying ρ. Hence, ρ spans a *tree*.

Analogously for generalization operators.

3.3.2. Generality Models and Refinement Operators

Instead of entailment ($\models$), θ-subsumption is often used as generality model. It is incomplete with respect to entailment—$C \models D$ if C θ-subsumes D but not vice versa—but decidable, simple to implement, and efficiently computable.

If we have background knowledge B, then we are not simply interested in whether a clause C (or a program Π) is more general than a clause D (or a program Π') but in whether C (Π) together with B is more general than D (Π'). This is captured by the notions of *relative* (to background knowledge) entailment and θ-subsumption. Relative entailment is defined as:

Definition 3.12. A program Π is *more general than* a program Φ with respect to *entailment relative to background knowledge* B, written $\Pi \models_B \Phi$, if

$$\Pi \cup B \models \Phi.$$

Generalization and Specialization under (Relative) θ-subsumption

Definition 3.13 ((Relative) θ-subsumption)**.** Let C and D be clauses and B a set of clauses.

C *θ-subsumes D*, written $C \succeq D$, iff there exists a substitution θ such that $C\theta \subseteq D$.[8]

C *θ-subsumes D relative to B*, written $C \succeq_B D$, if $B \models C\theta \rightarrow D$ for a substitution θ.

The definition of relative subsumption appears reasonable as an extension of ordinary θ-subsumption if one observes that $C\theta \subseteq D$, i.e., ordinary θ-subsumption, implies that $C\theta \rightarrow D$ is a tautology, $\models C\theta \rightarrow D$. Now having background knowledge B, we do not require $C\theta \rightarrow D$ to hold unconditionally but only if B is given.

It follows that if $C \succeq D$ then also $C \succeq_B D$. The converse, of course, need not hold. For *empty* background knowledge B and non-tautologous clauses C and D we have $C \succeq D$ if and only if $C \succeq_B D$, i.e., θ-subsumption and relative subsumption coincide in this special case.

Least General Generalizations. A Horn clause language quasi-ordered by θ-subsumption with an additional bottom element is a lattice. This does not generally hold for relative subsumption. Least upper bounds are called *least general generalizations (LGG)* [85]. LGGs and greatest lower bounds are computable and hence may be used for generalization and specialization, though they do not properly fit into our general notion of refinement operators because they neither map single clauses to sets of clauses nor single programs to sets of programs.

A useful restriction is to let background knowledge be a finite set of ground literals. In this case, LGGs exist also under relative subsumption, called *relative LGGs (RLGGs)*,

[8]Confusingly, both notations, $C \succeq D$ and $C \preceq D$, are used in the literature for writing that C θ-subsumes D. The latter one comes from interpreting a subset as smaller than its superset. However, the concept to be denoted here is *logical generality* (and not the subset relation, even if θ-subsumption is defined based on this relation). Hence we prefer the first notation, which correctly expresses the generality order.

and relative subsumption and RLGGs can be reduced to ordinary θ-subsumption and LGGs:

Proposition 3.1. *Let C and D be non-tautologous clauses and B be a finite set of ground literals such that $B \cap D = \emptyset$. Then*

- *$C \succeq_B D$ iff $C \succeq D \cup \bar{B}$ and*
- *$lgg_B(\{C_1, \ldots, C_n\}) = lgg(\{C_1 \cup \bar{B}, \ldots, C_n \cup \bar{B}\}$.*

The bottom-up system Golem [72] computes RLGGs under this setting.

Refinement operators. In general, ideal refinement operators do not exist for Horn clause languages ordered by (relative) subsumption. It suffices to drop the properness property, though. Locally finite and complete refinement operators for Horn clause languages exist. For *finite* Horn clause languages, ideal refinement operators do exist.

A specialization operator refines a clause by

- applying a substitution for a (single) variable or
- adding a (most general) literal.

Dually, a generalization operator refines a clause by

- applying an inverse substitution or
- removing a literal.

Application of such refinement operators is quite common in ILP, e.g., in the well-known systems MIS [99], Foil, Golem, and Progol. A specialization search-operator on clauses with respect to θ-subsumption as described above was used in the MIS system for the first time.

Incompleteness of θ-subsumption. (Relative) θ-subsumption is sound but not complete with respect to (relative) entailment; In general, (relative) θ-subsumption is sound but not complete. If $C \succeq D$ ($C \succeq_B D$) then $C \models D$ ($C \cup B \models D$) but not vice versa. For a counter-example of completeness let $C = P(f(X)) \leftarrow P(X)$ and $D = P(f(f(X))) \leftarrow P(X)$ then $C \models D$[9] but $C \not\succeq D$. As the example indicates, the incompleteness is due to *recursive* rules and therefore especially critical for *program synthesis*.

To see this in general, consider the following theorem that shows completeness (except for tautologies) of resolution in combination with θ-subsumption:[10]

[9] D is simply the result of self-resolving C.

[10] In introductory texts on (clausal) logic, most often *refutation completeness* (Proposition 2.3) is shown for resolution. The subsumption theorem and refutation completeness are equivalent in that they can mutually be proven from each other [80].

Theorem 3.3 (Subsumption Theorem, due to [80])**.** *Let T be a set of clauses and D be a clause. $T \models D$ iff D is a tautology or there exists a clause E such that $T \vdash_r E$ and $E \succeq D$.*

From this theorem follows the following corollary relating resolution and θ-subsumption to entailment relative to background knowledge between single clauses.

Corollary 3.1. *Let C and D be clauses and B be a set of clauses. Then $C \models_B D$ iff D is a tautology or there exists a clause E such that $\{C\} \cup B \vdash_r E$ and $E \succeq D$.*

Relative subsumption (Definition 3.13) can equivalently be expressed in terms of resolution and ordinary θ-subsumption:

Definition 3.14 (Alternative definition of relative subsumption (equivalent to that in Definition 3.13)**.** Let C and D be clauses and B a set of clauses. Then $C \succeq_B D$ iff D is a tautology or there exists a clause E such that $\{C\} \cup B \vdash_r E$, *where C must not be used more than once in the derivation of E*, and $E \succeq D$.

From this definition and the corollary above follows that the incompleteness of (relative) θ-subsumption is due to recursive clauses.

Incompleteness of θ-subsumption does *not* mean that recursive clauses cannot be generated by θ-subsumption refinement operators as described above. Actually—since the empty clause $\square$ θ-subsumes every other clause and and a complete specialization operator ρ under θ-subsumption exists—for each clause a logical equivalent can be computed by a finite number of applications of ρ starting with $\square$.

However, to generate a recursive clause, it is necessary to start from a *sufficiently general* clause. Therefore, incompleteness is an issue especially for *local* search.

Inverse Resolution. Within their early ILP system CIGOL, Muggleton and Buntine [75] introduced the concept of *inverting resolution* as a generalization technique. Inverse resolution operators are highly non-deterministic in general and thus inefficient. In his seminal paper on ILP [74], Muggleton introduces most specific inverse resolution operators and shows an equivalence to RLGGs.

Specialization and Generalization under (Relative) Entailment

Due to the incompleteness of (relative) θ-subsumption regarding recursive clauses, refinement under (relative) entailment (Definitions 3.9 and 3.12) has been studied.

Neither LGGs nor greatest specializations nor optimal refinement operators exist in general for Horn clause languages ordered by (relative) entailment.

Due to the subsumption theorem (Theorem 3.3), implication is equivalent to resolution followed by θ-subsumption. Hence a first idea to achieve a locally finite and complete specialization operator ρ_{I} for single clauses C under implication might be to simply add all self-resolvents of C to ρ_{S}. But this approach does not result in a complete specialization operator since in a deduction of D from C, C possibly needs to be used in several resolution steps together with another clause E derived in an earlier step. Yet our naive ρ_{I} can only resolve clauses with itself because it works on single clauses only. The solution is to construct a specialization operator mapping *definite programs* (sets of clauses) to sets of definite programs instead of *single clauses* to sets of clauses. Hence, we now consider definite programs ordered by (relative) implication (instead of single clauses ordered by θ-subsumption).

Locally finite and complete refinement operators exist for this setting and can be derived from the subsumption theorem.

A locally finite and complete specialization operator under entailment refines a definite program Π by

- adding a resolvent of two clauses in Π to Π or
- building the union of Π with $\rho_{\mathrm{S}}(C)$ for a clause $C \in \Pi$ or
- deleting a clause from Π.

Note that a result of applying ρ_{I} as defined in the first two items is equivalent to Π under implication because some clauses which logically follow from Π are simply added to Π. (Hence ρ_{I} is improper.) An actual specialization is carried out if ρ_{I} is applied as defined in the last item and if the deleted clause does not follow from the remaining clauses.

3.3.3. General Purpose ILP Systems

Three of the best known ILP systems are FOIL, GOLEM, and PROGOL. We may call them general purpose systems because they are not specifically designed to learn *recursive* programs but are in principle able to learn recursive programs. Especially FOIL has features especially addressing recursion [17] and has been systematically tested on standard recursive logic programs on lists [86].

FOIL instantiates the covering algorithm (Algorithm 3) where extensional coverage is applied. Clauses are generated top-down (from general to specific) starting with the clause $p(X_1, \ldots, X_n) \leftarrow$, if p is to be induced. FOIL uses a θ-subsumption specialization operator within a hill-climbing search. In each specialization step, all possibilities of adding a literal to a clause are enumerated and the best-rated clause is chosen for further specialization. Clauses are rated with respect to the *foilgain*, a statistical measure based on information theory. Basically, the foilgain tries to increase the fraction of covered positive examples. A variant of FOIL, FFOIL, is specifically designed for learning *functional* relations.

GOLEM also instantiates the covering algorithm but in contrast to FOIL, its search for clauses is bottom-up (from specific to general). First, the LGG under relative subsumption of a subset of positive examples is computed. The subset is determined by starting with pairs of randomly chosen examples and then greedily growing the subset. In each step, the subset whose LGG covers the most positive and no negative examples is chosen. The process stops if no increasing coverage of positive examples is achieved by a greater subset. It is taken care for consistency of computed clauses. In order to efficiently compute finite LGGs, only particular restricted forms of definite clauses can be induced.

PROGOL searches top-down through a θ-subsumption (sub)lattice of clauses. The sublattice is bounded from below by a most specific clause that is, bottom-up, derived from one example by inverting implication. Restricting the search to a sublattice reduces search effort. Inverse implication is made tractable by mode and type declarations provided by the user.

A good overview of the kind of recursive programs that can be learned by FOIL is given in FOIL's *midterm report* [86]. In that paper, FOIL is tested on most of the list-processing predicates such as member, reverse, shift, etc., defined Chapter 3 of Bratko's textbook *Prolog Programming for Artificial Intelligence* [16]. FOIL can learn most of the predicates from example sets that are complete up to lists of three or four elements. Other examples of recursive functions that can be learned are the Ackermann function and the *quicksort* algorithm, if the partitioning and appending functions are provided as background knowledge. GOLEM also can learn *quicksort* for example with the appropriate background knowledge.

However, the strength of these systems lie in learning non-recursive concepts from large data sets, and synthesis of recursive programs is problematic due to some systematic reasons: Due to their use of the (extensional) covering algorithm, they need complete example sets and background knowledge to induce recursive programs. Since they (at least FOIL and GOLEM) explore (i) only the θ-subsumption lattice of clauses and (ii) do this greedily, correct clauses are likely to be passed. Furthermore, their objective functions in the search for clauses is to cover as many as possible positive examples. Yet base clauses typically cover only few examples such that these systems often fail to induce correct base cases.

Statistical measures like the *foilgain* require comparatively large data sets in order to sensibly guide the search. In program synthesis, however, the specification often is written by a human person such that the ability to learn from *few* examples is crucial.

Finally, none of the three systems described is able to do *predicate invention*, i.e., to introduce non-specified predicates. This might be of minor importance for non-recursive programs, since each non-recursive (sub)predicate can be eliminated by unfolding it into the body of the clause where it is called. But for recursive programs, invention of subfunctions/subpredicates is an important feature (cp. Section 3.1.3).

3.3.4. Program Synthesis Systems

As a consequence of the problems regarding recursion described above, ILP systems especially designed to learn recursive programs have been developed. They address different issues: Handling of sparse and random examples, predicate invention, usage of general programming knowledge, and usage of problem-dependent knowledge of the user, which goes beyond examples. A comprehensive overview and comparison of (general purpose as well as program synthesis) ILP systems particularly addressing program synthesis capabilities can be found in [30].

Inverting Implication by Structural Analysis

Several systems—Lopster [60], Crustacean [1], Clam [89], Tim [39], mri [31]—address the issue of inducing *recursive* programs from random or sparse examples by inverting entailment based on *structural analysis*, similar to Section 3.2, instead of *searching* in the θ-subsumption lattice. These systems also have similar restrictions regarding the general schema of learnable programs. For example, Crustacean generates programs belonging to the following schema, containing one base clause and one purely linear-recursive clause:

$$\begin{aligned} &p(a_1, \dots, a_n) \\ &p(b_1, \dots, b_n) \leftarrow p(c_1, \dots, c_n) \end{aligned}$$

The a_i, b_i, c_i are terms, each c_i is a subterm of b_i. No background knowledge can be used and no auxiliary predicates are invented.

However, some of the systems can use background knowledge; mri can find more than one recursive clause.

Even though these systems are in principle able to learn recursive programs from *incomplete* example sets, they are more sensible to the incompleteness than enumerative systems.

Top-down Induction of Recursive Programs

Top-down systems can principally—even though they explore the θ-subsumption clause-lattice only—generate any (in particular any recursive) Horn clause from the empty clause. This follows from the existence of a complete θ-subsumption top-down operator (cp. Section 3.3.2).[11] Thus, if a top-down covering system would use intensional instead of extensional coverage, it could principally induce recursive programs from *random* examples. Certainly, this would require to find clauses in a particular order—base clauses first, then recursive clauses, only depending on base clauses and themselves, then recursive clauses, only depending on base clauses, the previously generated recursive clauses, and themselves, and so on. This excludes programs with mutually interdepending clauses. The system Smart [70] is based on these ideas. It induces programs consisting of one base clause and one recursive clause. Several techniques to sensibly prune the search space allows for a more exhaustive search than the greedy search applied by Foil, such that the incompleteness issue of θ-subsumption-based search is weaken.

The system Filp [9, 10] is a covering top-down system that induces *functional* predicates only, i.e., predicates with distinguished input- and output parameters, such that for each binding of the input parameters exactly one binding of the output parameters exists. This makes negative examples unnecessary (cp. the paragraph on positive and

[11] Hence, although θ-subsumption is incomplete with respect to entailment due to recursive clauses, every clause, in particular the recursive clauses, can be generated by refinement based on θ-subsumption—if one searches top-down starting from the empty clause or some other clause general enough to θ-subsume the desired clauses.

negative examples at the beginning of Section 3.3.1). Filp can induce multiple interdependent predicates/functions where each may consist of several base- and recursive clauses. Hence, intensional coverage is not assured to work. Filp starts with a few randomly chosen examples and tries to use intensional covering as far as possible. If, during the intensional proof of some example, an instance of the input parameters of some predicate appears for which an output is neither given by an example nor can be derived intensionally, then Filp queries for this "missing" example and thereby completes the example set as far as needed.

A third top-down system especially addressing recursive programs is Atre [63]. Atre deals with the problem of interdependent (mutually) recursive clauses by searching for several clauses in parallel, applies a more general generalization model than θ-subsumption, and keeps track of the possible non-consistence union of independently consistent clauses.

Using Programming Knowledge

Flener argued for the use of program schemas that capture general program design knowledge like divide-and-conquer, generate-and-test, global-search etc. [26, 27, 28, 29], and has implemented this in several systems. He distinguishes between schema-*based* systems inducing programs of a system-inherent schema only and schema-*guided* systems, which take schemas as dynamic, problem-dependent, additional input and thus are more flexible. Flener's Dialogs [27] system uses schemas and strong queries, such as how the input is to be decomposed for recursive calls, to restrict the search space and thereby is able to efficiently induce comparatively complex algorithms, e.g., *mergesort*, including predicate invention, e.g., invention of the *merge* function.

Jorge and Brazdil [40, 41] have—besides for *clause structure grammars* defining a program class and thus similar to schemas as dynamic language-bias—argued for so called *algorithm sketches*. An algorithm sketch is problem-dependent algorithm knowledge about the target function and provided by the user in addition to examples. This idea is implemented in their SKIL system. SKIL alone is not able to induce recursive programs from random examples. However, some non-recursive generalization takes place in this case, yielding non-ground, non-recursive clauses, assumed to be true properties of the target function. SKIL is then called again and again with properties induced in one iteration as additional input for the next iteration, thus iteratively called on greater and greater (with respect to generality) specifications such that the possibility of finding a recursive solution increases. Jorge and Brazdil call this technique *iterative bootstrap induction* and have implement it in their SKILit system. Though SKILit is able to induce some recursive programs such as *Member* from random examples, it fails for other, very simple recursive programs in this case, e.g., *Last*, because no sufficiently general properties are ever induced.

3.3.5. Learnability Results

Two complementary papers from Cohen [19, 20] address the learnability and non-learnability of certain (simple) classes of recursive logic programs within a variant of the PAC-learning framework [107, 3, 4].

Theoretical results regarding the necessity and its decidability of predicate invention are reported in [102].

3.3.6. Discussion

Compared to the classical analytic functional approach as described in Section3.2, ILP has broadened the class of inducible functions/relations by allowing for background knowledge and by searching in program spaces. At the core of ILP is the structuring of the program space by generality models such as entailment or θ-subsumption. It is not only a means to prune the search but also to theoretically analyze properties like (in-)completeness of search operators.

The sequential covering approach (generating one clause after the other) applied to the clause space structured by θ-subsumption leads to an efficient search for programs. However, this gain of efficiency comes at the price of difficulties with inducing recursive (interdependent) clauses due to (i) extensional coverage and the subsequent need for examples and background knowledge that are complete up to some complexity and (ii) the incompleteness of θ-subsumption.

Shapiro [99] and Muggleton and De Raedt [78] argued for clausal logic as universal language in favor to other universal formalisms such as Turing machines or LISP. Their arguments are: (i) Syntax and semantics are closely and in a natural way related. Hence if a logic program makes errors, it is possible to identify the erroneous clause. Furthermore, there are simple and efficient operations to manipulate a logic program with predictable semantic effects (cp. Section 3.3.2; however, the simple and efficient manipulation does only hold for operators based on θ-subsumption but not for those based on entailment). Both is not possible for, say, Turing machines. (ii) It suffices to focus on the logic of the program, control is left to the interpreter. In particular, logic programs (and clauses) are *sets* of clauses (and literals), order does not matter.

The first argument carries over to other declarative formalisms such as equational logic, term rewriting, and functional logic programming (FLIP [25] is an IPS system in this formalism). The second argument also carries over to some extent, declarative programming all in all shifts the focus off control and to logic. Yet in this generality it only holds for non-recursive programs or ideal, non-practical, interpreters. For the efficient interpretation of recursive programs however, order of clauses in a program and order of literals in a clause matters. Hence we think that declarative, (clausal- and/or equational-)logic-based formalisms are principally equally well suited for IPS.

Logic programs represent general relations. (Partial) functions are special relations—their domains are distinguished into source and target (or: a functional relation has input and output parameters) and they are single-valued (each instantiation of the input parameters implies a unique instantiation of the output parameters). Regarding functional- and logic programming, there is another difference: Functional programs are typically typed, i.e., their domain is partitioned and inputs and outputs of each function must belong to specified subsets, whereas logic programs are typically untyped. Interestingly, all three "restrictions" of functions compared to relations have been shown to be advantageous from a learnable point of view in ILP. The general reason is that they restrict the problem space such that search becomes more efficient and fewer examples are needed to describe the intended function. In particular, no negative examples are needed since they are implicitly given by the positive ones (cp. “On positive and negative examples” at the beginning of Section 3.3.1).

ILP is built around the natural generality structure of the problem space. Regarding *functional* relations, we observe an “oddity” of this structure. For definite programs, “more general”, with respect to the minimal Herbrand model, means “more atoms”. If the relation is a *function*, an additional ground atom must have a different instantiation of the input parameters compared to all other included atoms. Thus, “more general” in the case of definite programs representing functions reduces to “greater domain”. In other words: *All functions with the same domain are incomparable with respect to generality.* Since most often one is interested in total functions, generality actually provides *no structure at all* of the space of possible solutions.

3.4. Generate-and-Test Based Approaches to Inductive Functional Programming

The approaches reviewed in this section have in common, that they search through a class of functional programs by repeatedly generating and testing programs. That is, the generation of programs is independent from the incomplete specification or the I/O examples and the latter is only used as acceptance criterion or to define a *fitness-* or *objective function.*

3.4.1. Program Evolution

Evolutionary algorithms are inspired by biological evolution. They maintain *populations* of candidate solutions called *individuals*, get new ones by stochastical methods like *reproduction*, *mutation*, *recombination/crossover*, and *selection*, and thereby try to increase the *fitness* of individuals. “Fitness” here simply is a measure of how “good” a candidate solution is, e.g., which fraction of I/O examples a candidate program correctly computes.

Evolutionary algorithms have advantages when the problem space is too broad to conduct an exhaustive search and simultaneously nothing or few is known about the *fitness landscape*, hence when it is hard to construct sensible heuristics. The randomness of the search care for a widespread exploration of the problem space which is guided by the fitness measure.

On the other side, this randomized and “chaotic” search in a space with unknown properties makes it difficult to give guaranties regarding solutions and typically leads to non-optimal but only approximated solutions. Due to the randomness, solving a problem by evolutionary algorithms consists of several independent runs. The fittest, i.e., best, individual out of the final populations of all runs is designated to be the solution.

Genetic Programming

The subfield of evolutionary algorithms where the evolved objects are computer programs (in contrast to strings or other simple objects) is *genetic programming (GP)* [58].

Programs are typically represented as parse trees where inner nodes represent operators (primitive non-constant functions, e.g., arithmetic or boolean operations) and leafs represent operands (variables and constants). Operators and operands allowed to be used by the program to be evolved are given in so-called *function* and *terminal sets*, respectively. Classically, these two sets must be closed which means that all functions and terminals are of the same type such that they can be composed without restrictions in evolved programs. *Strongly typed genetic programming (STGP)* has been introduced by Montana [71] to overcome this restriction to closed function and terminal sets and to restrict the search space by types associated to operators and operands.

A GP problem is specified by *fitness cases* (e.g., example inputs of the target function), a fitness function, and the function and terminal sets. The fitness function assigns a fitness value to each individual program based on evaluating it on the fitness cases. E.g., the fitness function could compute the fraction of the sample inputs for which a correct output has been computed. There are no predefined acceptance criteria or preference biases in GP systems. The search is completely guided by the fitness function that is to be optimized. If, for example, small solutions shall be preferred than the size of an individual program must explicitly be considered by the fitness function. Additionally, several parameters such as the size of the population, the probabilities of performing the different genetic operations, maximum size or depth of individual programs, and the number of runs are to be specified.

Data structures and recursion (or iteration) do not play a predominant role in GP. A typical evolved program is a non-ground arithmetic expression or a propositional formula. However, some attempts have been made to integrate recursion or iteration into GP. In the following we consider some non-problem dependent techniques to evolve recursive programs by GP methods.

Koza and his colleagues [59] and Wong and Mun [109] integrated explicit recursion into GP by naming evolved functions and calling them within the function body.[12] One of the major issues is the handling of non-terminating programs. As a generate-and-test approach, GP relies on testing evolved candidate programs against the given examples. If non-termination may appear then a runtime limit is applied. This raises two problems if non-terminating programs are frequently generated: (i) The difficulty of assigning a fitness value to an aborted program and (ii) the runtime uselessly consumed by evaluating non-terminating programs. Wong and Mun [109] deal with this problem by a meta-learning approach to decrease the possibility of evolving non-terminating programs.

Others try to avoid non-termination completely:

[12]This approach—naming and recursively calling generated functions (or relations)—of *explicit* recursion corresponds to the other two approaches (Sections 3.2 and 3.3) to inductive program synthesis described so far in this chapter.

In her strongly typed GP system PolyGP [110, 111] (which uses Haskell as implementation and object language), Yu integrates *implicit recursion* through the use of user-provided *higher-order* functions. This kind of implicit recursion by using predefined higher-order functions like *map*, *filter*, and *reduce*, that capture particular recursive schemes, is routinely used in functional programming. If PolyGP introduces a higher-order function somewhere in the program tree, this forces a function as parameter for the higher-order function to be evolved. These parameter functions are implemented in the form of anonymous lambda abstractions by PolyGP. PolyGP has been tested for general *EvenParity* [110] and *Fibonnacci* and STRSTR [111]. STRSTR is a function from the C library and scans for the first appearance of a string in another string. The tests show a current drawback of the system: The user-provided higher-order functions need to fit the particular problem to be solved. This means that the user must actually know the recursive structure of the problem in advance.

Kahrs [44] evolves *primitive recursive* functions over the natural numbers. Binard and Felty [14] evolve programs in System F, a typed lambda calculus where only total recursive functions are expressible. The primitive recursive functions are contained in System F as proper subclass. Their system succeeds to induce *Sum*, *Prod*, and *Inc* with *Add* and *Mult* as background knowledge.

ADATE: Automatic Design of Algorithms Through Evolution

The ADATE system [82] is an evolutionary system in that it maintains a population of programs and performs a greedy search guided by a fitness function. Yet unlike GP, it is especially designed to evolve recursive functional programs in a (first-order) subset of Standard ML. To this end, ADATE applies sophisticated program transformation operators, a sophisticated search strategy, and predefined program evaluation functions, instead of the standard evolutionary GP operators, unsystematic search, and only user-defined fitness functions, respectively.

A specification consists of datatype definitions, predefined functions which can be used in the function to be induced, the type declaration of the target function, and a set of sample inputs for the target function. Furthermore, the user has to provide an output evaluation function to rate individual programs. This function takes a set of I/O pairs where the inputs are the given sample inputs and the outputs are those computed by an individual program to be rated. Hence testing an evolved program involves running it on the sample inputs and applying the output evaluation function to the result. In the easiest case, the output evaluation function includes outputs for all sample inputs and simply test whether the evolved program correctly computes these specified outputs. In this case, the evaluation function actually is a set of I/O examples. However, it can be more general. Generally, the output evaluation function must decide for every pair of a user-provided input and an output computed from it by an evolved program, whether the computed output is correct, incorrect, or whether this cannot be decided ("don't know"). As an example (taken from [82]), the *mergesort* algorithm is based on a function splitting a list into two lists whose lengths differ by at most one. Which elements go into which of the two halves is not important. Thus, given a sample input $[1, 2, 3, 4]$, both $\langle[1,2],[3,4]\rangle$ and $\langle[1,3],[2,4]\rangle$ and several solutions are equally good outputs. Such a non-deterministic specification of a function is not possible with I/O examples where each input is assigned a unique output, yet it can be stated by an output evaluation function for ADATE.

Individual programs are rated according to the user-provided output evaluation function and to their syntactical and computational complexity. In addition to the correctness decision, the output evaluation function may assign a list of grades (real numbers) which are to be minimized, to I/O pairs. This allows the user to specify any number of problem-dependent preference biases in addition to the biases *correct outputs*, *low syntactical complexity*, and *low computational complexity.*

New programs are created by applying transformations to existing programs. There are four *atomic* transformations:

Replacement. Replaces a subexpression *Sub* of a program by a new synthesized expression *Syn.* Parts of *Sub* may occur in *Syn.* Replacement is the only atomic transformation which changes the semantics of a program. Expression synthesis is heuristical and not exhaustive.

Abstraction. Introduces auxiliary subfunctions by choosing a subexpression *Sub* in a program P, replacing some disjoint subexpressions $S_1, \ldots, S_n$ of *Sub* by variables $V_1, \ldots, V_n$, taking the resulting generalized subexpression as body of a new subfunction $g(V_1, \ldots, V_n)$ of the introduced variables, and replacing the chosen subexpression *Sub* by $g(S_1, \ldots, S_n)$ in P.

Case-distribution. Chooses a function with one actual parameter being a `case`-expression and transforms it to an equivalent `case`-expression with adapted function calls as subexpressions. This transformation can also be applied vice versa.

Embedding. Changes the type of an (auxiliary) function.

Atomic transformations are not applied directly but only within a compound transformation which encapsulates a chain of atomic transformations.

The initial population consists of only one (empty or undefined) program. The incremental search produces successively populations containing bigger-and-bigger and better-and-better programs.

Even if the search is far more systematic than general GP, it is greedy and therefore it is unlikely to find the best program with respect to the combined program evaluation functions. And even if the search is greedy and applies highly specialized transformations, it consumes far more runtime than the approaches described in Sections 3.2 and 3.3. The main and general reason for this is its most unrestricted program space. ADATE is, to the best of out knowledge, the most powerful currently existing inductive programming system with respect to the class of programs that can in principle be induced.

At the ADATE homepage[13], Olsson reports on a number of problems successfully solved by ADATE on a 200 MHz PentiumPro. These include general *EvenParity*[14],

[13]`http://www-ia.hiof.no/~rolando/`

[14]In 12 seconds; this is the simplest reported problem.

several non-trivial list functions such as to check whether one list is a permutation of another[15], and path finding in a directed graph[16]. In [83] Olsson and Powers report on learning the semantics of simple natural language sentences with ADATE.

Evolving Algebraic Specifications From Positive and Negative Facts

A system lying in the intersection of ILP, GP and algebraic specification has been developed by Hamel and Shen [34, 35]. Their system evolves (recursive) algebraic specifications, i.e., equational theories over many-sorted signatures, using GP search methods. Yet instead of providing a fitness function, a theory to be induced is, as in ILP, specified by positive and negative facts—ground equations in this case. Additionally a background theory, corresponding to the function set of primitive functions in GP and to a background theory in ILP, may be provided. The fitness function to be maximized is derived from such a specification. It sums the satisfied positive facts, the non-satisfied negative facts and the reciprocal of the syntactical complexity of the candidate theory. That is, candidate theories satisfying more positive facts, excluding more negative facts and being of smaller syntactical complexity are preferred.

Candidate theories may be recursive but no mechanism to prevent non-terminating theories from being generated are included to the system. It simply stops evaluation after a limit of the number of evaluation steps has been exceeded.

3.4.2. Exhaustive Enumeration of Programs

Finally, two systems generating functional programs by simply enumerating a program class, until one is found satisfying the provided specification, have recently been presented. They are more or less direct instances of the enumeration algorithm (Algorithm 1).

MagicHaskeller

Katayama [45, 46] has developed a system, called MagicHaskeller, which essentially simply enumerates *all* type-correct programs composed of some predefined functions in order from small to infinitely large until one is found which satisfies a given constraint, e.g., a set of (possibly only a single) I/O examples. Input to the system consists of only the set of predefined functions, called *component library*, and the constraint to be fulfilled by a desirable program. Particularly, no parameters need to be adapted.

The library of predefined functions may contain higher-order functions. Target language is typed lambda calculus. Recursion is implicitly provided by the predefined higher-order functions. The algorithm uses some memoization, avoids re-enumeration of programs, and excludes some syntactically different but semantically equivalent programs from being enumerated.

[15]In about 16 hours.

[16]With *Append* predefined; in about nine days.

The objective of the system is, besides from being used as a programming tool, to provide a base-line for empirical evaluating other, more sophisticated inductive programming approaches. Katayama points out that, given a reasonable but fixed, problem-independent component library, the search space of type-correct expressions spanned by his exhaustive search is surprisingly small for some simple target programs.

As first results, he shows that correct programs for *Nth*, *Length*, and *Map* are found in below two seconds on a 2 GHz PENTIUM4. By comparison, the GP system POLYGP needs different higher-order functions for each of these problems, needs several runs to find a solution, needs additional parameters to be set, and yet consumes more time to induce a solution.

Inductive Programming with G∀ST

Recently, Koopman and Plasmeijer used the software testing system G∀ST [54] to induce primitive recursive functions from I/O examples [56, 55]. G∀ST can automatically, by finding counterexamples, refute a statement $\neg\exists t : T.P(t)$ or $\forall t : T.\neg P(t)$ respectively. Therefore it enumerates the type T from smaller to greater values and tests for each instance t whether $P(t)$ holds.

This mechanism, after some adaptation, is used for function induction as follows: The class of candidate programs is defined by a grammar for syntax trees which is provided as a data type to G∀ST. Enumerating this data type means enumerating syntax trees of candidate programs. The factorial function, for example, could be specified by

$$\neg\exists f. f(2) = 2 \wedge f(4) = 24 \wedge f(6) = 720.$$

This statement is refuted by the factorial function.

One may exclude particular generated functions from being tested by providing additional predicates. Koopman and Plasmeijer use this possibility to exclude functions with undesirable subexpressions like $x + 0$ from being considered.

The Optimal Ordered Problem Solver

Schmidhuber's *Optimal Ordered Problem Solver (OOPS)* [97] is a problem solver designed to be applied to *sequences* of increasingly hard problems of one class instead of being applied to *single* problems. The search for solutions to later problems always includes exploiting solutions to previously solved problems in the sequence in a theoretically optimal way. It is based on *Universal Search* [61], an asymptotically fastest way of solving a *single* given task.

Instead of searching in the raw (problem) solution space, e.g., in a space of action sequences to solve the *Towers of Hanoi* problem for n discs, OOPS searches in *program space* to find a program that solves the currently considered problem instance as well as all previously solved (smaller) instances. In the case of *Towers of Hanoi*, OOPS eventually generates the recursive solution program to solve any instance (for any number of discs).

Whereas classical planners fail to solve *Towers of Hanoi* for instances exceeding a certain number of discs (the limit is quoted with $n = 15$ in [97])—the reason is the exponential growth of the minimal number of actions to solve the problem as the number of discs increases)—OOPS, by generating the recursive solution program, is able to solve the problem, for example, for 30 discs (if it could previously solve smaller instances).

A further feature of OOPS is that it optimizes its inductive (preference) bias during the search.

3.4.3. Discussion

One general advantage of generate-and-test methods is their greater flexibility in at least two aspects: First regarding the problem space—there are no *principle* difficulties in enumerating even very complex programs. Second regarding the form of specifications. Whereas the synthesis operators of analytical techniques depend on the specification (e.g., I/O examples) such that different forms of specifications need different synthesis techniques, the generation of programs in generate-and-test methods is independent from the specification such that more general and expressive forms of specifications can easily be integrated. In particular, fitness functions in GP or the objective function in ADATE are more expressive than I/O examples since no fixed outputs need to be provided but general *properties* to be satisfied by computed outputs can be specified. Moreover, they need not specify complete sets of example inputs up to a certain complexity as required by the analytical techniques from Section 3.2 and ILP techniques (Section 3.3) relying on extensional coverage.

The disadvantage of generate-and-test methods is that they generally generate far more candidate programs until a solution is found and hence need much more time than data-driven methods to induce programs of equal size. A further problem is non-termination. As generated programs need to be tested against the provided specification, non-termination is a serious issue. Higher-order functions or formalisms that a-priori only include total functions are helpful to circumvent this problem.

3.5. Conclusions

In the previous sections, we described several approaches and systems to the inductive synthesis of functional and logic programs and discussed pros and cons and relations between them.

One obvious general dimension to classify them is the way of how the specification is used: As basis to construct solution programs (the analytical approach, Section 3.2) or to test and evaluate independently generated candidates (the generate-and-test based approach, Section 3.4). In ILP (Section 3.3), both approaches (bottom-up and top-down) are found. The analytical approach is faster because most representable programs are a priori excluded from being generated. On the other side, since it strongly depends on the specification and the language bias, it is much less robust and flexible regarding the whole problem specification including the specification of the target function and the inductive biases, especially the language bias.

Besides further developing both general approaches separately, we think that examining ways to combine them could be helpful to achieve a satisfiable tradeoff of robustness, flexibility, expressiveness, and efficiency. In the context of relational machine learning, the ILP system PROGOL successfully combined bottom-up and top-down construction of logic programs. In the next chapter, we present our functional IPS algorithm IGOR2 that combines analytical synthesis techniques with search in a program space.

A second dimension to classify IPS systems is on whether they generate all base- and recursive rules or cases one by one and independently of each other, i.e., whether they search in rule (or clause) space, or whether they generate all rules or cases in parallel, i.e., whether they in fact search in *program* spaces. While most ILP systems, e.g., FOIL, GOLEM, PROGOL, are of the first class (they are instances of the covering algorithm; Algorithm 3), the functional generate-and-test systems, as described in Section 3.4, are mostly of the latter class.

Generating rules independently requires extensional coverage and hence examples as specifications that are complete up to some complexity. On the other side, generate-and-test based systems can induce programs from random or sparse example sets.

One important topic, that certainly has not received sufficient attention in the context of inductive program synthesis, is learning theory, including models of learning and criteria to evaluate candidate programs. PAC-learning, the predominant learning model in machine learning, is well-suited for restricted representation languages and noisy data, hence approximate solutions. Yet in program synthesis, we have rich representation languages, often assume error-free examples, and want have programs that *exactly* compute an intended function or relation. Moreover, efficiency, not only of the induction process, but of the induced program, becomes an important issue. Muggleton's *U-learning* model[17] [76] captures these needs and is probably a good model or initial point to develop learning models for inductive program synthesis.

[17]The 'U' stands for 'universal'.

4. The Igor2 Algorithm

This chapter describes our own IPS algorithm called Igor2. It induces functional programs in terms of *constructor (term rewriting) systems (CSs)* from I/O examples or I/O patterns (non-ground "I/O examples"). Relevant definitions of term rewriting and in particular of constructor systems are given in Sections 2.2.2 and 2.2.4.

4.1. Introduction

Igor2 combines and generalizes several ideas and methods of the different approaches to IPS as described in the previous chapter. In particular, it is an attempt to combine the analytical *recurrence detection* method invented by Summers with *search in program (or rule) spaces* in order to overcome the strong restrictions—no usage of background knowledge and strongly restricted program schemas—of the classical analytical approach but without falling back to a generate-and-test search.

The main idea behind Igor2 is to conduct a global search in a comparatively less restricted program space including background knowledge, complex recursion schemes, and the automatic invention of auxiliary subfunctions in order to facilitate the reliable induction of non-trivial programs in different domains. For example, Igor2 is able to induce the Ackermann function as well as a function for transposing matrices as well as the recursive solution strategy for the *Towers of Hanoi* (see Chapter 5 for several experiments with a prototypical implementation of Igor2). However, candidate programs are constructed analytically from the I/O examples or I/O patterns instead of generating them independently from the specification as in generate-and-test approaches. The analytic construction of candidate programs has two advantages: First, often large parts of the program space can be pruned from the search tree; and second, constructed programs need not be tested, i.e., evaluated on the example inputs, because they are correct by construction.

Furthermore, Igor2 can be considered as a further development of Igor1 (see Section 3.2.3) in that Igor2 picks up the insight behind Igor1 that the analytical recurrence detection method is not restricted to S-expressions, but can be applied to terms over arbitrary algebraic (first-order) signatures. As in Igor1, induced programs are expressed as term rewriting systems. Extending Igor1 in this point, programs induced by Igor2 are *constructor systems (CSs)* instead of recursive program schemes (RPSs). CSs can be considered as an extension of RPSs; in particular, each RPS is also a CS. The extension is with respect to the form of the LHSs: While the arguments of the defined/unknown function symbols that are the roots of the LHSs are single variables only in the case of RPSs, they can be general *constructor terms* in the case of CSs.

This extension corresponds to *pattern matching* in function heads in modern functional programming languages like HASKELL or SML (in contrast to early functional languages like LISP where function heads are simply a function symbol applied to a sequence of single variables). Patterns in LHSs/function heads facilitate two things: First, a pattern decomposes the input such that (sub)parts of the input can be referred to by different variables in the RHS/function body. Second, it constitutes a condition. The respective rule is only applied if the input *matches* with the pattern. Pattern matching is the standard form of conditional evaluation in programs induced by IGOR2 at the moment. However, as a second form of conditional evaluation, *conditional rules*, can also be synthesized by IGOR2.

As a consequence, I/O examples or I/O patterns themselves are (syntactically correct) programs. For example, the I/O pattern

```
f (x : y : z : []) → z
```

is a syntactically correct rule of a CS, yet neither of an RPS nor of a LISP program. In fact, the first synthesis step in the classical analytical approach was to rewrite the I/O examples to a syntactically correct LISP program—the first approximation of the target function—by first deriving a fragment for each pair, e.g.

```
f (x) → caddr (x)
```

for the I/O pair above. In contrast, by using CSs as object language, the I/O examples themselves form a correct program. The first synthesis step is omitted and the detection of recurrences is directly applied to the I/O examples.

A further extension of IGOR2 is that signatures are sorted, i.e., that the terms and functions induced by IGOR2 are typed. In particular, there is no restriction to any predefined types or data structures for the programs induced by IGOR2. Instead, the user may specify its own algebraic (first-order) data types without any restrictions.

Both, the use of CSs to represent programs and pattern matching in LHSs as well as using typed signatures, let programs induced by IGOR2 look much more like programs written in modern functional languages. Another advantage of typing is that this further prunes the search by a priori ruling out many (type-incorrect) expressions. The most essential difference to usual functional programming languages is the current lack of higher-order functions in programs induced by IGOR2.

Besides the restriction to small sets of primitives and the restricted program schemas, a third serious drawback of the purely analytical approach was the need for sets of I/O examples that are complete up to some complexity. In the current version, IGOR2 does not systematically approach this problem, i.e., also IGOR2 relies on complete example sets. However, first ideas exist to overcome this restriction, see Section 4.9.3.

Even though IGOR2 does not add one rule after the other as in the ILP covering algorithm (Algorithm 3), it has in common with that algorithm that the rules of a program (a CS in our case) are generated independently from each other. Analogously to the covering algorithm, this implies a notion of *extensional* coverage or correctness of *single* rules that must be consistent with the (intensional) correctness of a complete recursive program, where the rules interdepend.

Listing 4.1: Mutually recursive definitions of odd and even induced by Igor2

```
odd (0)     →  false
odd (S n)   →  even (n)
even (0)    →  true
even (S n)  →  odd (n)
```

A last noteworthy characteristic is the capability of inducing several interdependent target functions in parallel. For example, if the two target predicates even and odd (of type $\mathbb{N} \to \mathit{Bool}$), testing whether a natural number is even or odd, respectively, are specified together by some I/O examples each, then Igor2 induces the mutually recursive function definitions shown in Listing 4.1

The overall goal of Igor2 is to broaden the class of effectively inducible programs or functions. Until now, we only focus on *correctness* of induced programs and *not* on efficiency or other criteria.

This chapter is organized as follows:

In the following short section, we introduce some additional notation regarding constructor systems.

In Section 4.3 we precisely define the problem that is solved by Igor2, i.e., how specifications and background knowledge look like and how induced functions are related to them.

In Section 4.4 we outline the general search through the program space that Igor2 conducts and briefly sketch its analytical refinement (synthesis) operators.

In Section 4.5 we then demonstrate Igor2 by means of an example run, inducing the reverse function on lists.

In Section 4.6, we define the concept of *extensional correctness* of single rules and CSs.

In Section 4.7 we then precisely specify the synthesis operators and show that they only synthesize extensionally correct rules. Furthermore, we present algorithms for all synthesis operators.

In Section 4.8, based on the operator definitions, we at first precisely define the problem space that is searched by Igor2. In Subsection 4.8.2, we then prove that, under certain conditions, the search applied by Igor2 in this space is terminating and complete. The problem space also contains *pseudo*-CSs, i.e., CSs whose RHSs contain variables not occurring in the LHSs. We call these candidate CSs *open.* Such open CSs are not confluent and do not denote functions. Actually, the *only* goal during the search is to find a *closed*, i.e., non-open, CS. Each such CS denotes a goal node or a solution within the program space. In Subsection 4.8.3, we then finally prove that, indeed, each closed CS in the program space correctly computes the specified I/O examples. This *intensional* correctness is concluded from the extensional correctness of the single rules and the fact that CSs induced by the synthesis operators terminate on the specified example inputs.

Finally, in Section 4.9, we shortly describe some extensions to the Igor2 algorithm to extend the program class and to make the induction more efficient.

Sections 4.3, 4.4, 4.5 are partly based on [48].

4.2. Notations

By $Var(t)$ we denote the set of variables occurring in term t.

Let r be a rule (of a CS), then by $Lhs(r)$ and $Rhs(r)$ we denote its left-hand side (LHS) and right-hand side (RHS), respectively. By $Defines(r)$, we denote the defined function symbol at the root of the LHS of r.

If R is a CS, by $Lhss(R)$, we denote the set of all LHSs occurring in R. Furthermore, by $\mathcal{D}_R$ and $\mathcal{C}_R$, we denote the defined function symbols (roots of the LHSs) and constructors of R, respectively.

We frequently write $\boldsymbol{t}$ for a sequence or vector $t_1, \dots, t_n$ of terms. E.g., instead of writing $f(p_1, \dots, p_n) \to t$ for a rule of a constructor system, we write $f(\boldsymbol{p}) \to t$.

4.3. Definition of the Problem Solved by Igor2

Incomplete specifications, background knowledge, and induced programs are finite, orthogonal CSs, called *specification CSs* (or just *specifications*), *background CSs*, and *induced CSs*, respectively.

Orthogonality, i.e., linear and pairwise non-unifying LHSs, is a property that is simple to verify and to produce, hence it is the most appropriate way to assure confluence of induced CSs.

Specifications and background CSs are CSs of the same restricted form (Definition 4.1 below). The only difference is their different meaning within the induction problem. In contexts where their different meaning is irrelevant, we just say *specification* for both. Specifications (hence also background CSs) have *constructor terms as RHSs*. Hence *ground* specifications and background CSs denote I/O examples (of target functions to be induced and background functions that are assumed to be already implemented, respectively), e.g.:

```
..., add (S S 0, S S S 0) → S S S S S 0, ...
```

However, specifications and background CSs may also contain *variables*, e.g.:

```
..., add (S S 0, y) → S S y, ...
```

Each single rule that contains variables does not denote a *single* I/O example. Rather one may think of such a rule as an I/O *pattern* representing the (I/O example-)*set* of all its ground instances (where variables are instantiated by ground constructor terms). This interpretation conforms to the rewriting semantics since the rewrite relation defined by a term rewriting system contains all (ground) instances of its rules.

Definition 4.1 (Specification, specification rule, I/O pattern, I/O example)**.** Let $\mathcal{D}$ and $\mathcal{C}$ denote sets of defined function symbols and constructors, respectively. Then a rule of the form

$$f(p_1, \dots, p_n) \to t$$

with $f \in \mathcal{D}$, $p_i \in T_{\mathcal{C}}(\mathcal{X})$ (for $i = 1, \dots n$), and $t \in T_{\mathcal{C}}(\mathcal{X})$, is called *specification rule.* A *ground* specification rule is also called *I/O example.* A *non-ground* specification rule is also called *I/O pattern.*

A *specification* is an orthogonal CS consisting of specification rules, i.e., a set of specification rules whose LHSs are linear and pairwise non-unifying.

We refer to (ground-instances of) LHSs and RHSs of specifications by *(specified) inputs* and *outputs*, respectively.

Given a specification Φ and a background CS B, the problem is to find a CS P that (re-)defines the defined functions of Φ and possibly additional (sub-)functions. In the RHSs of the defined functions, functions from B may be "called". $P \cup B$ must then correctly reduce each specified input to the corresponding output.

Definition 4.2 (Correctness)**.** Let Φ be a specification and P be any CS. We say that *P is correct with respect to* Φ if and only if

$$\overset{*}{\rightarrow}_P \supseteq \rightarrow_\Phi \ .$$

Note that if P is correct with respect to Φ, this implies that also all *instances* of Φ are included in $\overset{*}{\rightarrow}_P$.

The induction problem to be solved is defined as follows:

Definition 4.3 (Induction problem)**.** Let Φ and B be two specifications with disjoint sets of defined functions; $\mathcal{D}_\Phi \cap \mathcal{D}_B = \emptyset$. We call Φ and B *specification* and *background CS*, respectively.

Find a CS P with defined functions $\mathcal{D}_P$, such that

1. *P is orthogonal*: all LHSs are linear and pairwise non-unifying,
2. *P does not (re-)define background functions:* $\mathcal{D}_P \cap \mathcal{D}_B = \emptyset$, and
3. *$P \cup B$ is correct with respect to Φ:* $\overset{*}{\rightarrow}_{P \cup B} \supseteq \rightarrow_\Phi$.

We refer to a CS P satisfying the conditions of the induction problem by *solution CS*, *solution program*, or just *solution*.

Note first that the first two conditions, orthogonality of P and disjoint sets of defined functions of P and B imply that also $P \cup B$ is orthogonal (if, as assumed, B is also orthogonal). Note further that with respect to this definition, $P = \Phi$ is a perfect solution, just as good as infinitely many other solutions. This under-determination is a matter of fact in inductive reasoning. Which solution is returned depends on additional criteria, called *inductive bias* (cp. Section 3.1.1). For example, *Occam's razor* would prefer "simpler" solutions to more complex ones.

Recall that conditional evaluation in CSs induced by IGOR2 is realized by pattern matching, i.e., by different (non-unifying) patterns in the LHSs of a CS. The bias applied by IGOR2 is now that CSs with fewer cases in terms of maximal specific patterns (i.e., minimal patterns with respect to the subsumption order on terms) in the LHSs are preferred. The idea behind this bias is that fewer cases/patterns mean that inputs are partitioned into fewer subsets and hence that CSs with fewer cases are more general. In contrast, if one would prefer a *maximal* number of patterns, then the best solution would be the specification itself—each specified input would become its own pattern in its own rule.

Listing 4.2: I/O patterns for reverse

```
reverse ([])                → []
reverse (x : [])            → x : []
reverse (x : y : [])        → y : x : []
reverse (x : y : z : [])    → z : y : x : []
reverse (x : y : z : v : []) → v : z : y : x : []
```

Listing 4.3: I/O patterns for last, provided as background CS for reverse

```
last (x : [])            → x
last (x : y : [])        → y
last (x : y : z : [])    → z
last (x : y : z : v : []) → v
```

Example 4.1. Consider the rules (I/O patterns) in Listing 4.2, forming a specification Φ with $\mathcal{D}_\Phi = \{\mathsf{reverse}\}$. Furthermore, assume the rules in Listing 4.3 are provided as background CS B with $\mathcal{D}_B = \{\mathsf{last}\}$. x, y, z, v denote variables. The constructors are $\mathcal{C}_\Phi = \mathcal{C}_B = \{\,[],\ \_:\_\}$ where [] denotes the empty list and $\_:\_$ the usual list constructor to insert one element at the front of the list, written in mixfix notation.

Then IGOR2 would induce the rule set of Listing 4.4 that implements the reverse function for lists of arbitrary length. It forms a solution CS P ($\mathcal{D}_P = \{\mathsf{reverse}, \mathsf{sub}\}$) with respect to Definition 4.3. The non-specified recursive subfunction sub computes the init function and is automatically invented by IGOR2.

4.4. Overview over the Igor2 Algorithm

4.4.1. The General Algorithm

The induction of a solution CS is organized as a uniform cost search in a space of orthogonal CSs, including *open* CSs. By *open CS*, we refer to a CS P *not* satisfying $Var(r) \subseteq Var(l)$ for each rule $l \to r$ of P (cp. Definition 2.21). We also call the respective rules and RHSs open. In the following we just say CS to denote CSs in the narrow sense as well as open CSs. We call generated CSs *candidate CSs* and their rules *candidate rules.*

Listing 4.4: Induced CS for the reverse function, with last as background knowledge and inventing init as further help function

```
reverse ([])        → []
reverse (x : xs)    → last (x : xs) : reverse (sub (x : xs))
sub ([x])           → []
sub (x1 : x2 : xs)  → x1 : sub (x2 : xs)
```

Algorithm 4 shows the general search algorithm. The *cost* of a candidate is its number of *cases* or *conditions* in terms of maximal specific patterns in the LHSs. We say that a CS with *fewer cases* is *more general* because it partitions the example inputs into fewer subsets. The search operators never decrease the cost of a candidate—they never generalize a pattern—but either increase it, by computing a set of more specific patterns from one pattern, or leave it unchanged, when only the RHSs of a candidate are manipulated. Hence, IGOR2 prefers more general CSs to more specific CSs. Accordingly, the *initial* candidate CS consists of only *one* rule (one pattern) per target function.

In contrast to generate-and-test methods, where successor candidates are computed by manipulating or refining a selected candidate *independently* from the given specification, IGOR2's refinement operators compute successor candidates *based* on the given specification; or, they *synthesize* new candidates from the given specification. In particular, it is assured that each constructed candidate CS P satisfies all three conditions of Definition 4.3: Orthogonality, disjoint sets of defined functions of P and background CS B, and correctness of $P \cup B$ with respect to the specification.

However, candidates may be open CSs and as such they will not be accepted as solutions. Open CSs are not confluent and do not denote functions because the variables in RHSs not occurring in their LHSs may arbitrarily be instantiated in a rewrite step. Hence, purpose of the search operators is to decrease the number of such variables in order to achieve a closed CS. Each generated *closed*, i.e., not open, CS is accepted and a solution CS according to Definition 4.3.

In each search step, first the subset of candidates with minimal cost is chosen from all maintained candidates $\mathcal{N}$. If one of them is closed, it is returned as solution. If all chosen candidates are open, one open rule of one of them is selected and successor-rule sets are synthesized for it. The rule is then (in parallel) replaced by all its successor sets, leading to a set of successor candidate CSs.

Refining a candidate CS, i.e., computing successors of it to replace it by them, may involve the abduction of additional specification rules if a successor CS introduces a new, originally not specified, subfunction. Therefore, each candidate CS has an own specification belonging to it and IGOR2 actually maintains a set of pairs $\langle P, \Phi \rangle$ of candidate CSs P and corresponding specifications Φ. If P is open (closed), then we also say that $\langle P, \Phi \rangle$ is open (closed) and call $\langle P, \Phi \rangle$ a solution if P is closed.

4.4.2. Initial Rules and Initial Candidate CSs

In order to generate the initial candidate CS (Function `initialCandidate`), the specification Φ is partitioned such that each subset contains all rules (I/O examples or I/O patterns) belonging to one and the same defined function. (If only one function is specified, the only subset is the specification itself.) That is, we get the partition

$$\{\Phi(f) \mid f \in \mathcal{D}_\Phi\} \ , \quad \text{where } \Phi(f) := \{\varphi \in \Phi \mid \mathit{Defines}(\varphi) = f\} \, .$$

Then *one*, possibly open, initial rule is computed for each subset $\Phi(f)$, i.e., for each specified function $f \in \mathcal{D}$, by the *initial rule operator* χ_{init}.

Algorithm 4: The general Igor2 algorithm

```
Input: a specification (of target functions) $\Phi$
Input: a background CS $B$ with $\mathcal{D}_B \cup \mathcal{D}_\Phi = \emptyset$
Output: a (maximal general) CS $P$ satisfying the conditions of Definition 4.3
1  $P \leftarrow$ initialCandidate($\Phi$) // defined below
2  $\mathcal{N} \leftarrow \{\langle P, \Phi \rangle\}$
3  while $\langle P, \Phi \rangle$ open do
4      $r \leftarrow$ an open rule of $P$
5      $S \leftarrow$ successorRuleSets($r, \Phi, B$) // defined below
6      remove $\langle P, \Phi \rangle$ from $\mathcal{N}$
7      foreach successor rule-set and corresponding new specification $\langle s, \phi_{\text{new}} \rangle \in S$
       do
8          $P' \leftarrow (P \setminus \{r\}) \cup s$
9          insert $\langle P', \Phi \cup \phi_{\text{new}} \rangle$ into $\mathcal{N}$
10     $\langle P, \Phi \rangle \leftarrow$ a maximal general CS (and the corresponding specification) in $\mathcal{N}$
11 return $P$
```

Function initialCandidate(Φ)

```
Input: a specification $\Phi$ with defined functions $\mathcal{D}_\Phi$
Output: an initial (possibly open) candidate CS $P$ of $\Phi$; one rule per defined
        function
1  $P \leftarrow \emptyset$
2  foreach $f \in \mathcal{D}_\Phi$ do  insert $\chi_{\text{init}}(\Phi(f))$ into $P$
3  return $P$
```

Initial rules are not only computed for the initial candidate from the specification subsets $\Phi(f)$, but also later on in the induction process. During the search for a solution CS, the specification Φ is further partitioned into more and more, smaller and smaller subsets. Furthermore, if new defined functions not occurring in $\mathcal{D}_\Phi$ are introduced, then specifications for them are abduced such that new sets of specification rules appear. Whenever a new specification subset occurs—either by further partitioning or by introducing subfunctions and their specifications—an initial rule is computed by χ_{init} for the new subset.

The initial-rule operator χ_{init} computes a special kind of minimal generalization under subsumption, namely *minimal left-linear generalizations (MLGs)*, of specification subsets as initial rules. An MLG of a set Φ of rules equals the least general generalization (LGG) of Φ (Definition 2.20) except for that each repeated occurrence of a variable in the LHS is replaced by a new variable. For example, the *LGG* of the two rules $f(a,a) \to a$ and $f(b,b) \to b$ (a and b are constants) is $f(x,x) \to x$ whereas the two MLGs of these rules are $f(x,y) \to x$ and $f(y,x) \to x$. We take MLGs instead of LGGs in order to get left-linear rules.

Taking *minimal* (most specific) generalizations under subsumption as initial rules has several reasons: Most important, since each rule in a set Φ of specification rules is an instance of an MLG of Φ, every LHS in Φ can be reduced to its RHS by an MLG of Φ. Second, it is efficiently computable. (These two points are not only true for *minimal*, but for *all* generalizations under subsumption of a set of rules.) Third, taking a *most specific* generalization has the advantage that it maximally decomposes the subsumed LHSs (i.e., the specified inputs) such that maximally "deep" subterms of the specified inputs can be referred to in the RHS of the generated initial rule. And finally fourth, if a generalization under subsumption of a set of specification rules Φ, that is a *closed* rule, exists at all, then in particular each MLG is also closed. In other words: *If* a specification set can be solved by simply taking a generalization under subsumption, then each MLG is such a solution. But even if all (also the minimal) generalizations under subsumption remain *open*, the MLGs are the "best" of such open generalizations because they show best which parts of the specification rules cause the absence of a closed generalization and hence need to be generalized by methods going beyond generalization under subsumption.

Example 4.2. Consider the following set of two I/O patterns of the init function:

```
init (x : y : [])        → x : []
init (x : y : z : [])    → x : y : []
```

Its initial rule (in this case the LGG) is

```
init (x : y : xs) → x : ys
```

It is open (due to the variable ys). However, it is more useful than, for example, the (very general) generalization init (x) → y. The LGG already "solves" parts of the specifying rules (especially the root and the left subtree of the RHSs) and suggests the right subtree to be further dealt with, whereas the more general generalization does not contain any information of how to further refine it.

Finally note that, since specification rules have constructor terms as RHSs, also initial rules always have constructor rules as RHSs, i.e., do not contain any (recursive) function calls.

4.4.3. Refinement (or Synthesis) Operators

Now let P be some generated candidate, Φ be the corresponding specification, and $r : f(\boldsymbol{p}) \to t$ be an open rule in P that has been selected to be refined. The *specification subset of* Φ *associated with* r is the set

$$\Phi(r) := \{\varphi \in \Phi \mid f(\boldsymbol{p}) \succeq \mathit{Lhs}(\varphi)\}$$

consisting of all specification rules whose LHSs match with the LHS of the open rule r, i.e., whose LHSs would be reduced by r. Now three out of four refinement operators (χ_{split}, χ_{sub}, χ_{smplCall}, χ_{call}) are in parallel applied to the open rule in order to eliminate its open variables (Function `successorRuleSets`). They synthesize successor rule sets based on the associated specification subset $\Phi(r)$.

```
Function successorRuleSets($r, \Phi, B$)
  Input: an open rule $r$
  Input: a specification $\Phi$
  Input: a background CS $B$
  Output: a set of pairs of successor-rule sets and corresponding new specification
          rules
1 $S_1 \leftarrow \chi_{\text{split}}(r, \Phi)$
2 $S_2 \leftarrow \chi_{\text{sub}}(r, \Phi, B)$
3 $S_3 \leftarrow \chi_{\text{smplCall}}(r, \Phi, B)$
4 if $S_3 = \emptyset$ then $S_3 \leftarrow \chi_{\text{call}}(r, \Phi, B)$
5 return $S_1 \cup S_2 \cup S_3$
```

Splitting an Open Rule into a Set of Several More Specific Rules: χ_{split}

The *rule-splitting operator* χ_{split} partitions $\Phi(r)$ and then applies χ_{init} to each subset in order replace r by a *set* of *more specific* initial rules. It is assured that the LHSs of the new initial rules are pairwise non-unifying. The idea is, that only *one* rule probably does not suffice to compute the example inputs specified by $\Phi(r)$ but that different cases/rules are needed to properly compute them. For example, each *recursive* solution CS consists of at least *two* rules—one non-recursive rule as base case, and one rule containing a recursive call in the RHS.

The partitions of $\Phi(r)$ are computed according to positions in the LHS of the open rule r that denote a variable in the LHS of r and (different) constructors in the LHSs of the subsumed specification rules $\Phi(r)$. We call such positions *pivot positions (of* $\Phi(r)$*)*. All specification rules with the *same* constructor at such a pivot position are sorted into the same subset. The MLGs of the resulting subsets then have the respective (different) constructors at the pivot position and hence are non-unifying, hence constitute different cases. E.g., consider again the two I/O patterns and the corresponding open initial rule (MLG) of the init function in Example 4.2. The position denoting variable xs in the LHS of the initial rule denotes the different constructors [] and : in the first and second I/O pattern, respectively. Hence these two I/O patterns would be sorted into different subsets.

The operator χ_{split} is the only operator that increases the cost of candidate CSs (by replacing a pattern by a set of more specific patterns).

Introducing Subfunctions to Separately Compute Subterms: χ_{sub}

The *subproblem operator* χ_{sub} deals with those subterms of the RHSs in $\Phi(r)$ that cause r to be open. Those subterms are considered as new subproblems and new subfunctions are introduced to separately compute them. The open subterms in the open RHS are then replaced by calls to these new subfunctions. E.g., consider again Example 4.2 above. For the open subterm ys in the RHS of the initial rule (the MLG) of the two specification rules for init, a new subfunction sub1 would be introduced and ys would be replaced by a call of sub1:

```
init (x : y : xs) → x : sub1 (x : y : xs)
```

For the new subfunctions, in our example for sub1, new specification rules ϕ_{new} are then abduced by taking the LHSs of $\Phi(r)$ as LHSs of ϕ_{new} (with the new defined function symbols at the roots) and the respective subterms of the RHSs in $\Phi(r)$ as RHSs in ϕ_{new}. For our init example we would abduce the two I/O patterns

```
sub1 (x : y : [])      → []
sub1 (x : y : z : []) → y : []
```

from the two I/O patterns of init. Finally, for each new subfunction, an initial rule is computed by χ_{init} from the abduced specifications. In our case we get the initial rule

```
sub1 (x : y : xs) → ys
```

for sub1.

Introducing (Recursive) Function Calls: χ_{smplCall} and χ_{call}

Both the splitting and the subproblem operator together can effectively not do more—i.e., can not synthesize (semantically) more functions—than the splitting operator alone. This is because the calls to (sub)functions introduced by χ_{sub} are always calls of *new* subfunctions that are not currently defined by the candidate CS. That is, χ_{sub} never introduces a recursive call. Yet non-recursive subfunctions can be eliminated, without changing the semantics of the program, by unfolding them. Furthermore, χ_{sub} does not introduce calls of background functions. In contrast, the *function-call operators* χ_{smplCall} and χ_{call} introduce calls of already defined functions. They replace the (open) RHS of an open initial rule r by a call of a defined function. This may be the function

Defines(r) itself, i.e., a recursive call, or another target or previously introduced subfunction (possibly leading to mutual recursive calls) or a background function. Both operators introduce different forms of function calls. Actually, at most *one* of them is effectively applied (i.e., leads to a non-empty set of successor rules).

The operator χ_{smplCall} introduces function calls of the form $f'(\boldsymbol{p'})$, where $\boldsymbol{p'}$ are constructor terms. We call this a *simple* function call due to the constructor-term arguments. In contrast, χ_{call} introduces function calls of the form $f'(g_1(\boldsymbol{p}), \ldots, g_n(\boldsymbol{p}))$ where the g_i are new subfunctions. The idea behind χ_{call} is that a constructor term possibly does not suffice as argument of a function call but that the argument itself possibly also must be computed by a (recursive) defined function, e.g., a separate new function or a background function.

The function-call operators realize a generalization of the recurrence detection method of the analytical approach to IPS as introduced by Summers 3.2.1. If, e.g., χ_{smplCall} replaces the RHS of an open rule $r: f(\boldsymbol{p}) \to t$ by a function call with the resulting rule $f(\boldsymbol{p}) \to f'(\boldsymbol{p'})$, this is done based on matching specified outputs of the functions f and f'. If each RHS in $\Phi(r)$ matches a RHS in the specification of f', then a call of f' is in general possible and $\boldsymbol{p'}$ is constructed in a way such that it maps the LHSs in $\Phi(r)$ to the respective and appropriately instantiated LHSs of f'.

Concrete examples for χ_{smplCall} and χ_{call} are contained in the exemplary induction of the reverse function in the following section.

4.5. A Sample Synthesis

In this section we show a complete synthesis of the reverse function. As specification, we provide the following I/O patterns:

```
reverse ([])                 → []
reverse (x : [])             → x : []
reverse (x : y : [])         → y : x : []
reverse (x : y : z : [])     → z : y : x : []
```

Furthermore, as background CS, we provide I/O patterns of the last function:[1]

```
last (x : [])                → x
last (x : y : [])            → y
last (x : y : z : [])        → z
```

Since only one function, reverse, is specified as target function, the specification need not be partitioned for the initial candidate CS P_0. P_0 simply consists of the initial rule for reverse, the MLG of all four reverse I/O patterns:

```
reverse (x1) → x2
```

: Initial candidate P_0

[1] Providing last as background knowledge is not necessary; we provide last here as background knowledge just to give an example of how a background CS is used by IGOR2. If last was not given as background knowledge, IGOR2 would automatically introduce it.

It is open due to the variable x2 in the RHS that does not occur in the LHS and hence needs to be refined.

1st Iteration

The initial set of maintained candidate CSs $\mathcal{N}$ contains only the initial candidate: $\mathcal{N} = \{P_0\}$.[2] The only rule of P_0 is open and selected to be refined.

Selected candidate CS: P_0.
Selected rule: reverse (x1) $\rightarrow$ x2.
Associated I/O patterns: All four reverse I/O patterns.

Rule splitting. The only pivot position is 1, denoting the variable x1 in the LHS of the selected initial rule. The first reverse input pattern has the constructor '[ ]' at position 1, the remaining reverse input patterns have the constructor ':' at position 1. Hence χ_{split} generates the partition $\{\{1\}, \{2, 3, 4\}\}$ of the reverse I/O patterns. The first successor candidate CS results from replacing the selected initial rule by the two more specific initial rules (MLGs) of the two generated subsets:

```
reverse ( [ ] )      →  [ ]
reverse (x1 : x2)  →  x3 : x4
```

$: P_0 \triangleright P_1$

Subfunctions. Since the RHS of the selected initial rule consists of a single variable, x2, instead of being rooted by a constructor, there are no proper subterms and χ_{sub} is not applicable.

Function Call. The RHS of the selected rule cannot become a call of last due to different types.[3] A recursive call (of reverse) is not possible since none of the RHSs of the reverse I/O patterns subsumes another one.

2nd Iteration

The initial CS P_0 has been replaced by the only generated successor CS P_1, i.e., we have now $\mathcal{N} = \{P_1\}$, and only its second rule is open.

Selected candidate: P_1.
Selected rule: reverse (x1 : x2) $\rightarrow$ x3 : x4.
Associated I/O patterns: reverse I/O patterns $\{2, 3, 4\}$ (because the LHS of the selected rule subsumes the LHSs of these I/O patterns but not the LHS of the first I/O pattern).

[2]We omit explicitly stating the corresponding specifications here together with the candidate CSs in $\mathcal{N}$. The specifications appear in the text.
[3]We do not explicitly stated the types of reverse and last here, but we implicitly assume that reverse is of type list to list and last is of type list to element.

Rule splitting. The only pivot position is 1.2 which denotes the variable x2 in the LHS of the selected rule and the constructors '[ ]' and ':' in the associated I/O patterns which are now further partitioned into $\{\{2\}\{3,4\}\}$. The selected open rule of P_1 is replaced by the initial rules for the two new subsets, leading to a new candidate P_2:

```
reverse ([ ])             →  [ ]
reverse (x1 : [ ])        →  x1 : [ ]
reverse (x1 : x2 : x3)    →  x4 : x5 : x6
```

: $P_0 \rhd P_1 \rhd P_2$

Subfunctions. The RHS of the selected rule of P_1 is rooted by a constructor such that χ_{sub} can be applied. The idea is to consider the open subterms as subproblems which are solved by new, separate subfunctions. That is, the open subterms in the selected rule are replaced by calls to two new (sub)functions sub1 and sub2:

```
    reverse (x1 : x2) → sub1 (x1 : x2) : sub2 (x1 : x2)
```

: Refined selected rule

This involves the abduction of specifications for the new subfunctions from the selected reverse I/O patterns. The formal parameter of the subfunction calls in the refined rule is always the same as the pattern of that rule—in this case: x1 : x2. Therefore, sub1 and sub2 are called with the same inputs as the selected rule. Hence the LHSs in the abduced specifications for sub1 and sub2 are the same as those of the selected reverse I/O patterns (only the root symbol reverse is replaced by sub1 and sub2). From these same inputs, sub1 and sub2 must compute the first and second subterms of the respective reverse RHSs.

Here are (again) the I/O patterns, associated with the selected rule, from which the new specifications are abduced:

```
2  reverse (x : [ ])           →  x : [ ]
3  reverse (x : y : [ ])       →  y : x : [ ]
4  reverse (x : y : z : [ ])   →  z : y : x : [ ]
```

: Selected I/O patterns

From these, the following I/O patterns for the new subfunctions are abduced:

```
1  sub1(x : [ ])          →  x     sub2(x : [ ])          →  [ ]
2  sub1(x : y : [ ])      →  y     sub2(x : y : [ ])      →  x : [ ]
3  sub1(x : y : z : [ ])  →  z     sub2(x : y : z : [ ])  →  y : x : [ ]
```

: Abduced specifications for sub1 and sub2

The successor candidate results from replacing the selected rule by its refined version as stated above and by adding initial rules for the new subfunctions (MLGs of their respective I/O patterns). This leads to a new candidate P_3:

```
reverse ([])     →  []
reverse (x1 : x2) → sub1 (x1 : x2) : sub2 (x1 : x2)
sub1 (x1 : x2)    → x3
sub2 (x1 : x2)    → x4
```

: $P_0 \triangleright P_1 \triangleright P_3$

Program Call. A program call is not possible for the same reasons as in the first iteration.

3rd Iteration

In iteration 2, two successor candidates of P_1 were generated. The candidate set is now $\mathcal{N} = \{P_2, P_3\}$. P_3 is more general than P_2 because of only two maximal specific patterns in P_3 ([] and x1 : x2) instead of three in P_2 ([], x1 : [], x1 : x2 : x3), hence we select P_3 and one of its open rules, say the third one, to be refined.

Selected candidate: P_3.
Selected rule: sub1 (x1 : x2) → x3 .
Associated I/O patterns: All three sub1 I/O patterns.

Rule splitting. The only pivot position is 1.2 where the LHS of the first sub1 I/O pattern has a '[]' and the remaining two a ':'. We partition the I/O patterns accordingly and replace the selected rule by the two resulting initial rules, leading to candidate P_4:

```
reverse ([])        → []
reverse (x1 : x2)    → sub1 (x1 : x2) : sub2 (x1 : x2)
sub1 (x1 : [])       → x1
sub1 (x1 : x2 : x3)  → x4
sub2 (x1 : x2)       → x4
```

: $P_0 \triangleright P_1 \triangleright P_3 \triangleright P_4$

Subfunctions. The selected rule has a single variable as RHS such that this operator is not applicable.

Program Call. Each LHS in the I/O patterns of sub1 matches a LHS of the last background CS such that the RHS of the selected sub1 rule can be replaced by a call of last. The argument of that call must then map the sub1 example LHSs to the corresponding (possibly instantiated) last LHS.

Since actually each RHS in the sub1 specification matches *each* RHS in the last specification, there are several possibilities for the argument of the call.

At first, χ_{smplCall} is tried where the formal argument of the function call is a constructor term

One possible argument for the call of the last function is a very simple one: It is the pattern of the selected rule itself, because, actually, the I/O patterns of sub1 are identical to the I/O patterns of last. We here take this simple solution leading to the successor candidate P_5:

```
reverse ([])       → []
reverse (x1 : x2)  → sub1 (x1 : x2) : sub2 (x1 : x2)
sub1 (x1 : x2)     → last (x1 : x2)
sub2 (x1 : x2)     → x4
```

: $P_0 \rhd P_1 \rhd P_3 \rhd P_5$

If χ_{smplCall} succeeds, as in this case, then χ_{call} will not be tried.

4th Iteration

We now have the candidate set $\mathcal{N} = \{P_2, P_4, P_5\}$ where P_5 is better rated than the equally rated P_4 and P_2 and thus selected. The only open rule of P_5 is the fourth one.

Selected candidate: P_5.
Selected rule: sub2 (x1 : x2) → x3 .
Associated I/O patterns: All three I/O patterns of sub2.

Rule splitting. Results in P_6:

```
reverse ([])          → []
reverse (x1 : x2)     → sub1 (x1 : x2) : sub2 (x1 : x2)
sub1 (x1 : x2)        → last (x1 : x2)
sub2 (x1 : [])        → []
sub2 (x1 : x2 : x3)   → x4 : x5
```

: $P_0 \rhd P_1 \rhd P_3 \rhd P_5 \rhd P_6$

Subfunctions. Not applicable due to a single variable as RHS.

Program Call. A call of last is not possible due to type mismatch. A recursive call of sub2 is not possible because no RHS of the I/O patterns of sub2 subsumes another one. Yet a call of reverse is possible. Each RHS of the sub2 I/O patterns matches (in this special case: is *identical* with) a RHS of the reverse specification. The following listing shows the identical RHSs in the middle and the corresponding sub2 and reverse LHSs at the left and the right, respectively:

```
sub2(x : [])          → []          ← reverse ([])
sub2(x : y : [])      → x : []      ← reverse (x : [])
sub2(x : y : z : [])  → y : x : []  ← reverse (x : y : [])
```

: Identical RHSs and corresponding sub2 and reverse specification LHSs

We replace the open RHS x3 of the selected rule sub2 (x1 : x2) → x3 by a call to reverse. The argument of the call must then map the input patterns x : [], x : y : [], x : y : z : [] of sub2 to the corresponding input patterns [], x : [], x : y : [] of reverse, respectively. It is obvious that this mapping cannot be achieved by a constructor term as argument only. Hence χ_{smplCall} fails here.

Therefore χ_{call} is invoked. χ_{call} considers the described mapping as the specification of a new subfunction sub3 for computing the function-call argument:

```
sub3 (x : [])         → []
sub3 (x : y : [])     → x : []
sub3 (x : y : z : []) → x : y : []
```

: Abduced I/O patterns for the new subfunction sub3

Since reverse calls sub2, we are going to introduce a mutual recursion here. Therefore, it is important that the input patterns of sub2 are greater than the corresponding input patterns of reverse, i.e., that the output patterns of sub3 are smaller than the respective input patterns.

In the resulting successor candidate of P_5, the RHS of the selected rule is replaced by a call to reverse with a call of sub3 as argument. Furthermore, the initial rule for sub3 is added:

```
reverse ([])      → []
reverse (x1 : x2) → sub1 (x1 : x2) : sub2 (x1 : x2)
sub1 (x1 : x2)    → last (x1 : x2)
sub2 (x1 : x2)    → reverse (sub3 (x1 : x2))
sub3 (x1 : x2)    → x3
```

: $P_0 \triangleright P_1 \triangleright P_3 \triangleright P_5 \triangleright P_7$

5th Iteration

The candidate set is now $\mathcal{N} = \{P_2, P_4, P_6, P_7\}$ where P_7 is better rated than the equally rated remaining three candidates. The only open rule of P_7 is its fifth rule.

Selected candidate: P_7.
Selected rule: sub3 (x1 : x2) → x3 .
Associated I/O patterns: All three I/O patterns of sub3.

Rule splitting. Leads to P_8:

```
reverse ([])        → []
reverse (x1 : x2)   → sub1 (x1 : x2) : sub2 (x1 : x2)
sub1 (x1 : x2)      → last (x1 : x2)
sub2 (x1 : x2)      → reverse (sub3 (x1 : x2))
sub3 (x1 : [])      → []
sub3 (x1 : x2 : x3) → x1 : x4
```

: $P_0 \triangleright P_1 \triangleright P_3 \triangleright P_5 \triangleright P_7 \triangleright P_8$

Subfunctions. Not applicable due to a single-variable RHS.

Program Call. Each sub3 output pattern matches a reverse output pattern such that, in principle, we could again introduce a call of reverse here. This would result in a new subfunction sub4 to compute the argument of the call (like sub3 computes the argument of the reverse call in the sub2 rule). In the next step, sub4 then again could call reverse and so on, ad infinitum. Therefore, the maximal depth of nested calls of defined functions is limited by some natural number which can be set by the specifier. Let us assume that this parameter prevents a further call of reverse here.

Furthermore, each sub3 output pattern equals a sub2 output pattern. Yet since (i) sub2 calls sub3 such a call of sub2 would introduce a mutual recursion and (ii), the argument of that call would not decrease, a call of sub2 is not be possible here.

A call to last or sub1 is not possible due to different types.

6th Iteration

We have now $\mathcal{N} = \{P_2, P_4, P_6, P_8\}$. All these candidates are equally rated such that it is not determined, which of them will be refined in the next step. Let us take P_8. Its only open rule is the sixth one.

Selected candidate: P_8.
Selected rule: sub3 (x1 : x2 : x3) → x1 : x4 .
Associated I/O patterns: sub3 I/O patterns $\{2, 3\}$.

Rule splitting. Leads to P_9:

```
1  reverse ([])                 → []
2  reverse (x1 : x2)            → sub1 (x1 : x2) : sub2 (x1 : x2)
3  sub1 (x1 : x2)               → last (x1 : x2)
4  sub2 (x1 : x2)               → reverse (sub3 (x1 : x2))
5  sub3 (x1 : [])               → []
6  sub3 (x1 : x2 : [])          → x1 : []
7  sub3 (x1 : x2 : x3 : [])     → x1 : x2 : []
```

: $P_0 \triangleright P_1 \triangleright P_3 \triangleright P_5 \triangleright P_7 \triangleright P_8 \triangleright P_9$

Note, that the I/O patterns for sub3 are simply reproduced in this candidate.

Subfunctions. The RHS of the selected rule is rooted by a constructor such that we may treat the subterms as new problems. The first (left) subterm is the variable x1 that also occurs in the LHS, hence need not to be replaced. The second subterm x4, however, is open and is thus replaced by a call of a new subfunction sub4. The following I/O patterns for sub4 are abduced from I/O patterns 2, 3 of sub3:

```
1  sub4 (x : y : [])        → []
2  sub4 (x : y : z : [])    → y : []
```

: Abduced I/O patterns for the new subfunction sub4

Replacing the open subterm by the call to sub4 and adding the initial rule for sub4 yields the new candidate P_{10}:

```
reverse ([])          →  []
reverse (x1 : x2)     →  sub1 (x1 : x2) : sub2 (x1 : x2)
sub1 (x1 : x2)        →  last (x1 : x2)
sub2 (x1 : x2)        →  reverse (sub3 (x1 : x2))
sub3 (x1 : [])        →  []
sub3 (x1 : x2 : x3)   →  x1 : sub4 (x1 : x2 : x3)
sub4 (x1 : x2 : x3)   →  x4
```

: $P_0 \triangleright P_1 \triangleright P_3 \triangleright P_5 \triangleright P_7 \triangleright P_8 \triangleright P_{10}$

Program Call. As in the previous iteration, a call to reverse would in principle be possible but we assume that this is prevented by a parameter bounding the maximal depth of nested program calls.

7th Iteration

The candidate set is now $\mathcal{N} = \{P_2, P_4, P_6, P_9, P_{10}\}$. P_9 is worst rated and therefore not selected; the remaining candidates are equally rated. We take P_{10} and its seventh rule to be refined in the next step.

Selected candidate: P_{10}.
Selected rule: sub4 (x1 : x2 : x3) → x4 .
Associated I/O patterns: The two I/O patterns of sub4.

Rule splitting. Yields P_{11}. We leave this as an exercise for the reader.

Subfunctions. Not applicable.

Program Call. Calls of reverse, sub2 and sub3 are in principle possible and are introduced. We here show the call of sub3. The following listing shows the two sub4 I/O patterns as well as the corresponding two (renamed) sub3 I/O patterns whose RHSs subsume the sub4 RHSs.

```
sub4 (x : y : [])      →  []          sub3 (v : [])      →  []
sub4 (x : y : z : [])  →  y : []      sub3 (v : w : [])  →  v : []
```

: Matching sub4 and (renamed) sub3 I/O patterns

The first sub4 and sub3 RHSs are equal. The second sub3 RHS subsumes the second sub4 RHS with substitution $\tau = \{v \mapsto y\}$.

We find x2 : x3 as constructor term argument, i.e., the refined selected rule is sub4 (x1 : x2 : x3) → sub3 (x2 : x3). With this call, the first sub4 LHS (from the matching I/O patterns above) is correctly mapped to the first sub3 LHS and the second sub4 LHS is correctly mapped to the second sub3 LHS.

The resulting successor candidate CS is the following:

```
reverse ([])          → []
reverse (x1 : x2)     → sub1 (x1 : x2) : sub2 (x1 : x2)
sub1 (x1 : x2)        → last (x1 : x2)
sub2 (x1 : x2)        → reverse (sub3 (x1 : x2))
sub3 (x1 : [])        → []
sub3 (x1 : x2 : x3)   → x1 : sub4 (x1 : x2 : x3)
sub4 (x1 : x2 : x3)   → sub3 (x2 : x3)
```

$: P_0 \triangleright P_1 \triangleright P_3 \triangleright P_5 \triangleright P_7 \triangleright P_8 \triangleright P_{10} \triangleright P_{12}$

8th Iteration

We now have $\mathcal{N} = \{P_2, P_4, P_6, P_9, P_{11}, P_{12}\}$. P_2, P_4, P_6, P_{12} are equally rated and better than P_9, P_{11}. Since P_{12} is closed, we have found a maximally general and closed candidate. Hence the search stops here and returns P_{12} as solution CS.

Post processing

We observe that P_{12} contains non-recursive subfunctions—sub1 and sub2. They can be eliminated by unfolding them into the RHSs where they are called. sub3 and sub4 are mutually recursive but not directly recursive either. So we can eliminate one of them. Since sub3 is also called by sub2 we keep it and eliminate sub4 by unfolding it into the second RHS of sub3.

Such simple unfoldings can be done fully automatically. The resulting solution is:[4]

```
reverse ([])          → []
reverse (x1 : x2)     → last (x1 : x2) : reverse (sub3 (x1 : x2))
sub3 (x1 : [])        → []
sub3 (x1 : x2 : x3)   → x1 : sub3 (x2 : x3)
```

4.6. Extensional Correctness

As we have seen, the rules of candidate CSs are generated independently from each other, each single rule based on the specification only. However, in an eventually induced closed CS the rules interdepend and must together correctly reduce each specified input to the corresponding output.

Example 4.3. Consider the following specification for the two target functions even and odd:

```
even (0)      → true        odd (0)      → false
even (S 0)    → false       odd (S 0)    → true
even (S S 0)  → true        odd (S S 0)  → false
```

[4] The automatically introduced subfunction sub3 is the init function that deletes the last element of an input list.

Now suppose the χ_{split} operator has already partitioned both specifications (for even and odd) into two subsets each, where one subset contains the respective first I/O example and the second one contains the respective remaining two I/O examples. The corresponding candidate would be:

```
even (0)    →  true
even (S x)  →  y
odd (0)     →  false
odd (S x)   →  y
```

Suppose the open rule for even is selected and χ_{smplCall} is applied and tries to introduce a call of odd. χ_{smplCall} then checks whether each RHS of the I/O examples associated with the selected open rule (the second and third I/O example for even in our example) is subsumed by any of the specified outputs of the function to be called (odd in our example). Indeed, the RHS of I/O example 2 for even is false and equals, e.g., the RHS of I/O example 1 for odd. And the RHS of I/O example 3 for even is true and equals the RHS of I/O example 2 for odd. Now the argument of the function call is computed such that it maps the LHSs of the two I/O examples of even to the according LHSs of odd. The resulting rule is

```
even (S x)  →  odd (x)
```

and the corresponding (still open) successor candidate CS is:

```
even (0)    →  true
even (S x)  →  odd (x)
odd (0)     →  false
odd (S x)   →  y
```

The synthesized rule even (S x) → odd (x) is *extensionally correct* with respect to the specification because if it is used to achieve one reduction step, then the specification itself further reduces the resulting term correctly. E.g., the rule even (S x) → odd (x) reduces in one step the example input even (S S 0) to odd (S 0). We cannot use the current candidate CS to uniquely reduce this term because the respective rule in the candidate CS is the still open rule. Yet we may use the specification itself to reduce odd (S 0) to true. And indeed, true is the correct result for our exemplary input even (S S 0).

The last open rule for odd is similarly refined by χ_{smplCall} to the extensionally correct rule odd (S x) → even (x). The solution CS is that from the introduction of this chapter (Listing 4.1).

Of course, we now expect that this CS is (intensionally) correct, i.e., that it reduces each example input to the corresponding output *without* using the specification itself for the reduction. Indeed, the solution CS is correct in this sense (i.e., according to Definition 4.2).

Certainly, this independent construction of single rules, which eventually interdepend in the induced CS, comes not for free: It presupposes appropriately chosen I/O examples or I/O patterns, hence impedes the synthesis from *randomly* chosen I/O examples. In particular, the specification must be complete up to some complexity of the I/O examples or I/O patterns.

For example, if the second I/O example for odd in the example above was missing, then the synthesized rule for even that calls odd would *not* be extensionally correct anymore, because if we reduced the example input even (S S 0) to odd (S 0) and this example input for odd was missing, then odd (S 0) would *not* correctly be reduced to true (the specified output for even (S S 0)) by the specification. Actually, the rule for even that calls odd would not be found by χ_{smplCall} in this case.

Hence relying on extensional correctness makes induction *incomplete* in the sense that if specifications are provided that are *not* complete up to some complexity, then (intensionally) correct CSs may not be found (cp. also Section 3.2.4).[5]

We now precisely define extensional correctness. In the next section we then precisely define the several synthesis operators and show that they synthesize extensionally correct rules. In Section 4.8, we then, besides other things, show that, indeed, extensional correctness leads to correctness according to Definition 4.2 if, furthermore, we assure that synthesized CSs terminate for the specified inputs.

Definition 4.4 (Extensional correctness of a single rule)**.** Let Φ be a specification. A (possibly open) rule $f(\boldsymbol{p}) \to t$ *is extensionally correct with respect to* Φ iff, whenever $(f(\boldsymbol{i}) \to o) \in \Phi$ and $f(\boldsymbol{i}) = f(\boldsymbol{p})\sigma$ for a substitution σ with $Dom(\sigma) = Var(f(\boldsymbol{p}))$, then there is a substitution θ with $Dom(\theta) = Var(t) \setminus Var(f(\boldsymbol{p}))$ such that $t\sigma\theta \overset{*}{\to}_{\Phi} o$.

Corollary 4.1. *Let Φ be a specification and $f(\boldsymbol{p}) \to t$ be a rule that is extensionally correct with respect to Φ. Let Φ' be another specification with $\mathcal{D}_{\Phi} \cap \mathcal{D}_{\Phi'} = \emptyset$. Then $f(\boldsymbol{p}) \to t$ is also extensionally correct with respect to $\Phi \cup \Phi'$.*

Proof. Due to $\mathcal{D}_{\Phi} \cap \mathcal{D}_{\Phi'} = \emptyset$ there is no rule $r \in \Phi'$ with $Defines(r) = f$ such that $f(\boldsymbol{p})\sigma = f(\boldsymbol{i})$ could be satisfied. For all rules $(f(\boldsymbol{i}) \to o) \in \Phi$, however, for that substitutions σ, θ exist such that $t\sigma\theta \overset{*}{\to}_{\Phi} o$, obviously also $t\sigma\theta \overset{*}{\to}_{\Phi \cup \Phi'} o$. □

Extensional correctness of (candidate) CSs simply means that all rules in a candidate CS are extensionally correct and furthermore, that each specified input matches some LHS/pattern of the candidate CS.

Definition 4.5 (Extensional correctness of a CS)**.** Let Φ be a specification. A CS P *is extensionally correct with respect to* Φ iff:

1. *Extensionally correct rules:* Each rule in P is extensionally correct with respect to Φ.

2. *Completeness:* Each LHS of Φ matches a LHS of P.

[5] It is exactly this interdependence of (mutually) recursive functions that makes the induction of (direct) recursive functions or *multiple* (mutually recursive) functions a hard problem in general, compared to, for example, decision trees, that can be modelled as *non*-recursive rules. In general—if one does not want to pay a price such as appropriately chosen I/O examples—, for *recursive* functions, all (potentially interdependent) rules must be induced in parallel. That is, the problem space consists of *sets* of rules, whereas the rules modelling a decision tree are independent from each other such that the problem space effectively only consists of *single* rules.

Corollary 4.2. *Let Φ be a specification. Then Φ is extensionally correct with respect to Φ.*

Proof. Let $(f(\boldsymbol{p}) \to t) \in \Phi$ and $(f(\boldsymbol{i}) \to o) \in \Phi$ and $f(\boldsymbol{i}) = f(\boldsymbol{p})\sigma$. Since Φ is orthogonal, $(f(\boldsymbol{p}) \to t) = (f(\boldsymbol{i}) \to o)$ and hence $t\sigma = t = o$, hence $t\sigma \overset{*}{\to}_\Phi o$. □

Corollary 4.3. *Let P be a CS and Φ be a specification such that P is extensionally correct with respect to Φ.*

1. *$\mathcal{D}_\Phi \subseteq \mathcal{D}_P$.*
2. *Let P' be a further CS and Φ' be a further specification such that P' is extensionally correct with respect to Φ' and $\mathcal{D}_P \cap \mathcal{D}_{P'} = \emptyset$. Then $P \cup P'$ is extensionally correct with respect to $\Phi \cup \Phi'$.*

Proof.
1. Follows immediately from the completeness condition of extensional correctness.
2. *Extensionally correct rules:* W.l.o.g., let r be a rule in P. By definition, r is extensionally correct with respect to Φ. With $\mathcal{D}_P \cap \mathcal{D}_{P'} = \emptyset$, hence also $\mathcal{D}_\Phi \cap \mathcal{D}_{\Phi'} = \emptyset$, and Corollary 4.1 now follows that r is also extensionally correct with respect to $\Phi \cup \Phi'$.

   Completeness: Obvious.

□

The presented notion of extensional correctness for rules and CSs and the implied need for specifications that are complete up to some level, is closely related to the notion of extensional coverage in inductive logic programming (ILP). Also in ILP, the use of the covering algorithm (Algorithm 3) which induces the single rules independently from each other based on extensional coverage, requires specifying example sets that are complete up to some level, if the induced rules are recursive or otherwise interdepend; compare Section 3.3.1.

4.7. Formal Definitions and Algorithms of the Synthesis Operators

In this section, we formally define the initial-rule operator χ_{init} and the refinement operators χ_{split}, χ_{sub}, χ_{smplCall}, and χ_{call}, that are applied to initial rules generated by χ_{init}, if such an initial rule is *open*. We prove that all synthesized rules are extensionally correct with respect to the corresponding specification and background CS. Furthermore, we present algorithms for all synthesis operators.

The operators χ_{sub} and χ_{call} introduce new subfunctions and abduce specifications for them. Hence different candidate CSs may define different sets of (sub)functions and hence may have different specifications—even though the original, user-provided specification of the target functions is a subset of the specification of *each* candidate CS.

In general, if $\langle P, \Phi \rangle$ is a candidate CS and a corresponding specification, the refinement operators χ_{split}, χ_{sub}, χ_{smplCall}, and χ_{call} are applied to an open rule $r \in P$ (and get Φ and possibly a background CS B as further input) and yield a finite, possibly empty, set $\{\langle s_1, \phi_1 \rangle, \langle s_2, \phi_1 \rangle, \ldots\}$ of successor-rule sets s_i and corresponding new specifications ϕ_i. The ϕ_i are empty in the case of χ_{split} and χ_{smplCall} and contain the abduced specifications for the new subfunctions in case of χ_{sub} and χ_{call}. The successor pair $\langle P', \Psi' \rangle$ of $\langle P, \Psi \rangle$ according to one particular generated pair $\langle s, \phi \rangle$ is then defined by

$$\begin{aligned} P' &:= (P \setminus \{r\}) \cup s \\ \Phi' &:= \Phi \cup \phi \end{aligned}$$

If $\langle P, \Phi \rangle$ is a candidate CS and a corresponding specification, then each rule, in particular each *open* rule, $r \in P$ has an associated specification subset of Φ, denoted by $\Phi(r)$. $\Phi(r)$ exactly contains all specification rules in Φ whose LHSs match the LHS of r. That is, $\Phi(r)$ exactly contains all those specified inputs (or input patterns) that could be reduced by r.

Definition 4.6 (Specification subset associated with a rule)**.** Let $\langle P, \Phi \rangle$ be a candidate CS and the corresponding specification and let $r \in P$ be a rule of P. Then by $\Phi(r) \subseteq \Phi$ we denote the *specification subset of* Φ *associated with rule* r, defined as

$$\Phi(r) := \{\varphi \in \Phi \mid \mathit{Lhs}(r) \succeq \mathit{Lhs}(\varphi)\}\,.$$

Sometimes we do not need all specification rules associated with a particular rule but all specification rules of one defined function f.

Definition 4.7 (Specification subset of a defined function)**.** Let Φ be a specification and $f \in \mathcal{D}_\Phi$ be one of the functions specified by Φ. Then by $\Phi(f)$ we denote the *specification subset of* Φ *associated with function* f, defined as

$$\Phi(f) := \{\varphi \in \Phi \mid \mathit{Defines}(\varphi) = f\}\,.$$

Extensional correctness of rules that are synthesized by χ_{split}, χ_{sub}, χ_{smplCall}, and χ_{call} as refinements of an open initial rule r that is selected from an open candidate CS P relies on three assumptions:

Claim 4.1. *Let Φ be a specification and B be a background CS such that $\mathcal{D}_\Phi \cap \mathcal{D}_B = \emptyset$ (cp. Definition 4.3). Further, let $\langle P, \Psi \rangle$ be any candidate CS and corresponding specification generated by* IGOR2 *from the initial specification Φ and background CS B.*
Then

- $\mathcal{D}_P = \mathcal{D}_\Psi$,
- $\mathcal{D}_P \cap \mathcal{D}_B = \emptyset$, *and*
- *if $r \in P$ is an open rule in P then r is an MLG of $\Psi(r)$.*

We cannot prove this claim here, because we have not yet precisely defined the synthesis operators that generate candidate CSs and corresponding specifications. In the following subsection, we show the stated claims for *initial* candidate CSs. For the remainder of this section we then just *assume* that these claims are true for a provided pair $\langle P, \Psi \rangle$ and background CS B. Then in Section 4.8, we show that, provided these claims are true for a particular pair $\langle P, \Psi \rangle$ and a background CS B, they are also true for all successor pairs $\langle P', \Psi' \rangle$ generated by the synthesis operators. Together with the fact that they are true for initial CSs this inductively shows that they are true for all generated candidate CSs and corresponding specifications.

4.7.1. Initial Rules and Candidate CSs

Initial rules are computed by χ_{init} when new specification (sub)sets appear. Especially the first candidate CS, the initial CS of the search, is computed by applying χ_{init} to the specification subsets corresponding to the specified target functions. (If only one target function is specified, then χ_{init} is applied to the complete set of specification rules.)

Also when a specification subset $\Phi(r)$, associated with an open rule r, is (further) partitioned by χ_{split}, then χ_{init} is subsequently applied to the new subsets. Furthermore, when χ_{sub} or χ_{call} introduce new subfunctions and abduce specification rules for them, then χ_{init} is subsequently applied to the new specifications in order to get initial rules for the new (sub)functions.

Initial rules are *minimal left-linear generalizations* (maximal specific left-linear generalizations), with respect to subsumption, of sets of specification rules.

Definition 4.8 (Minimal left-linear generalization)**.** Let $\langle \mathcal{R}_\Sigma(\mathcal{X}), \succeq \rangle$ be the set of all specification rules over signature Σ and variables $\mathcal{X}$, quasi-ordered by subsumption. Let Φ be any finite subset of $\mathcal{R}_\Sigma(\mathcal{X})$. Then a minimal element in the set of left-linear upper-bounds of Φ in $\langle R_\Sigma(\mathcal{X}), \succeq \rangle$ is called a *minimal left-linear generalization (MLG)* of Φ.

MLGs are equal to LGGs (least general generalizations) except for that repeated occurrences of variables in LHSs are replaced by new variables. In contrast to LGGs, MLGs need not be unique.

Example 4.4 (Minimal left-linear generalizations)**.** Consider the following two rules where a and b are constants: $f(a, a) \to a$ and $f(b, b) \to b$. Their unique (up to variable renaming) *LGG* is $f(x, x) \to x$. Their two *MLGs* are $f(x, y) \to x$ and $f(x, y) \to y$.

Obviously, if the LGG of a set of rules Φ is left-linear, then there is a *unique* MLG of Φ—the LGG of Φ. For example, the LGG and unique MLG of the two rules $f(a, b) \to b$ and $f(c, d) \to d$ is $f(x, y) \to y$.

The initial rule operator χ_{init} returns one (arbitrary) MLG for a specification.

Definition 4.9 (Initial rule operators χ_{init} and χ_{INIT})**.** For a set Φ of specification rules let G denote the set of all MLGs of Φ. Then χ_{init} is (non-deterministically) defined as

$$\chi_{\text{init}}(\Phi) \in G\,.$$

For a set $\{\Phi_1, \ldots, \Phi_n\}$ of sets of specification rules, χ_{INIT} is defined as

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

Corollary 4.4. *Let $\phi \subseteq \Phi$ be a subset of a specification and $\chi_{\text{init}}(\phi) = (f(\boldsymbol{p}) \to t)$.*

1. *The rule $f(\boldsymbol{p}) \to t$ is left-linear.*

2. *If, for all $(f(\boldsymbol{i}) \to o) \in \Phi \setminus \phi$, $f(\boldsymbol{p}) \not\succeq f(\boldsymbol{i})$, then $f(\boldsymbol{p}) \to t$ is extensionally correct with respect to Φ.*

Proof. 1. χ_{init} yields MLGs that are left-linear by definition.

2. Let $f(\boldsymbol{i}) \to o \in \phi$ and $\sigma\theta$ ($Dom(\sigma) = Var(f(\boldsymbol{p}))$, $Dom(\theta) = Var(t) \setminus Var(f(\boldsymbol{p}))$) be a (composed) substitution such that $(f(\boldsymbol{i}) \to o) = (f(\boldsymbol{p}) \to t)\sigma\theta$. (Since $\chi_{\text{init}}(\phi) \succeq \varphi$ for each specification rule $\varphi \in \phi$ such a substitution $\sigma\theta$ exists for each $\varphi \in \phi$.) We have $f(\boldsymbol{i}) = f(\boldsymbol{p})\sigma$ and $t\sigma\theta = o$, hence $t\sigma\theta \overset{*}{\to}_\Phi o$.

□

Definition 4.10 (Initial candidate CS)**.** Let Φ be a specification. Then a CS P, defined as

$$P := \chi_{\text{INIT}}(\{\Phi(f) \mid f \in \mathcal{D}_\Phi\})\,,$$

(where $\Phi(f) = \{\varphi \in \Phi \mid Defines(\varphi) = f\}$, cp. Definition 4.7) is called an *initial (candidate) CS of* Φ.

Lemma 4.1. *Let Φ be a specification and B be a background CS such that $\mathcal{D}_\Phi \cap \mathcal{D}_B = \emptyset$. Let P be an initial CS of Φ. Then*

1. *$\mathcal{D}_P = \mathcal{D}_\Phi$,*

2. *$\mathcal{D}_P \cap \mathcal{D}_B = \emptyset$,*

3. *each rule $r \in P$ is an MLG of its associated specification subset $\Phi(r)$,*

4. *P is orthogonal, and*

5. *P is extensionally correct with respect to Φ.*

Proof. 1. P contains exactly one rule for each $f \in \mathcal{D}_\Phi$, hence $\mathcal{D}_P = \mathcal{D}_\Phi$.

2. Follows from $\mathcal{D}_\Phi \cap \mathcal{D}_B = \emptyset$ and $\mathcal{D}_P = \mathcal{D}_\Phi$.

3. Let $r \in P$ be an MLG of $\Phi(f)$ for an $f \in \mathcal{D}_\Phi$. Then for each $\varphi \in \Phi(f)$, $Lhs(r) \succeq Lhs(\varphi)$. Furthermore, since all LHSs in $\Phi \setminus \Phi(f)$ are rooted by function symbols $f' \neq f$, $Lhs(r) \not\succeq Lhs(\varphi)$ for all $\varphi \in \Phi \setminus \Phi(f)$. Hence $\Phi(f) = \Phi(r)$, hence r is an MLG of $\Phi(r)$.

4. Each rule in P is generated by χ_{init}, hence left-linear. Furthermore, all LHSs in a specification subset $\Phi(f)$ are rooted by the same function symbol f. Hence also the LHS of the respective rule in P, an MLG of $\Phi(f)$, is rooted by f. On the other side, the rules in a subset $\Phi(f)$ and the rules in another subset $\Phi(f')$ are rooted by the *different* function symbols $f \neq f'$. Hence the LHSs of the respective initial rules in P are rooted by the different function symbols f, f' and hence non-unifying.

5. *Extensionally correct rules:* Let be $r \in P$ and $\mathit{Defines}(r) = f$. Then $r = \chi_{\text{init}}(\Phi(f)) = \chi_{\text{init}}(\Phi(r))$. Furthermore, by definition of $\Phi(r)$, $\mathit{Lhs}(r) \not\succeq \mathit{Lhs}(\varphi)$ for all $\varphi \in \Phi \setminus \Phi(r)$. Now from Corollary 4.4 follows that r is extensionally correct with respect to Φ.

   Completeness: Let $(f(\boldsymbol{i}) \to o) \in \Phi$ be any rule in Φ. Then $(f(\boldsymbol{i}) \to o) \in \Phi(f)$ and since P contains a rule $f(\boldsymbol{p}) \to t$ that is an MLG of $\Phi(f)$, $f(\boldsymbol{i})$ matches $f(\boldsymbol{p})$.

□

Note that by this lemma (items $1, 2, 3$), initial candidates satisfy all three assumptions of Claim 4.1 for their specifications.

4.7.2. Splitting a Rule into a Set of More Specific Rules

The splitting operator χ_{split} partitions a specification subset associated with an open rule r and then applies χ_{init} to each new subset. It is assured that the resulting LHSs are more specific than that of r, that they are pairwise non-unifying, and that each LHS in $\Phi(r)$ matches the LHS of one new rule. Thus, χ_{split} introduces conditional evaluation based on exhaustive pattern matching of the inputs specified by $\Phi(r)$. The following definition formally specifies the operator χ_{split}.

Definition 4.11 (Rule-splitting operator χ_{split})**.** Let $\langle P, \Phi\rangle$ be a candidate CS and the corresponding specification. Let r be an open rule in P and $\Phi(r)$ be the specification subset associated with r (see Definition 4.6). We assume that r is an MLG of $\Phi(r)$ (Claim 4.1).

We call a position $p \in \mathit{Pos}(\mathit{Lhs}(r))$ a *pivot position (of* $\Phi(r)$*)* if it denotes a *variable* in $\mathit{Lhs}(r)$ and a *constructor* in $\mathit{Lhs}(\varphi)$ for each $\varphi \in \Phi(r)$. Then an equivalence relation $\sim_p$ on $\Phi(r)$ is defined upon p as

$$\varphi \sim_p \varphi' \quad \text{iff} \quad \mathit{Node}(\mathit{Lhs}(\varphi), p) = \mathit{Node}(\mathit{Lhs}(\varphi'), p)\,.$$

The operator $\chi_{\text{split}}(r, \Phi)$ builds the set of all quotient sets $\Phi(r)/\sim_p$ for all pivot positions p and applies χ_{INIT} to each quotient set, yielding a set of more specific successor rules compared to r for each quotient set. Since no new subfunctions are introduced, the corresponding new specification sets are empty:

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

(where $\chi_{\text{INIT}}(\{\phi_1, \ldots, \phi_n\}) := \{\chi_{\text{init}}(\phi_1), \ldots, \chi_{\text{init}}(\phi_n)\}$, cp. Definition 4.9).

Lemma 4.2. *Let $\langle P, \Phi\rangle$ and r be defined as in Definition 4.11.*

Let p be a pivot position of $\Phi(r)$, $\langle s, \emptyset\rangle \in \chi_{\text{split}}(r, \Phi)$ be the corresponding set s of successor rules of r (together with the empty set of new specification rules), and $r' \in s$ be one of the successor rules.

Then

1. *s is orthogonal,*
2. *$r \succ r'$,*
3. *r' is extensionally correct with respect to Φ, and*
4. *r' is an MLG of its associated specification subset $\Phi(r')$.*

Proof. Let $\phi \in \Phi(r)/\sim_p$ be any subset/equivalence class in the quotient set of $\Phi(r)$ generated upon pivot position p and $r' \in s$ be the corresponding new rule, i.e., $r' = \chi_{\text{init}}(\phi)$.

1. r' is generated by χ_{init}, hence left-linear.

   Since first, r is assumed to be an MLG of $\Phi(r)$, second, p denotes a *variable* in $Lhs(r)$ and the *same* constructor c in all LHSs in $\phi \subseteq \Phi(r)$, and third, the new rule r' is an MLG of ϕ, we have $Node(Lhs(r'), p) = c$.

   Since furthermore, $\Phi(r)$ is partitioned with respect to the different constructors at position p in the LHSs, for each two $\phi_1, \phi_2 \in \Phi(r)/\sim_p$ and the corresponding new initial rules $r'_1, r'_2 \in s$ holds $Node(Lhs(r'_1), p) = c_1 \neq c_2 = Node(Lhs(r'_2), p)$ for constructors c_1, c_2.

   Hence each two successor rules $r'_1, r'_2 \in s$ are non-unifying.

2. Since r is an MLG of $\Phi(r)$ and r' is an MLG of $\phi \subseteq \Phi(r)$, $r \succeq r'$. Since, furthermore, $Node(Lhs(r), p)$ is a variable and $Node(Lhs(r'), p)$ is a constructor, $r \succ r'$.

3. By definition of $\Phi(r)$, $Lhs(r) \not\succeq Lhs(\varphi)$ for all $\varphi \in \Phi \setminus \Phi(r)$. Since furthermore, $r \succeq r'$, also $Lhs(r') \not\succeq Lhs(\varphi)$ for all $\varphi \in \Phi \setminus \Phi(r)$. Furthermore, since $Node(Lhs(r'), p) \neq Node(Lhs(\varphi), p)$ for all $\varphi \in \Phi(r) \setminus \phi$, $Lhs(r') \not\succeq Lhs(\varphi)$ for all $\varphi \in \Phi(r) \setminus \phi$. Hence $Lhs(r') \not\succeq Lhs(\varphi)$ for all $\varphi \in \Phi \setminus \phi$ and with Corollary 4.4 follows that r' is extensionally correct with respect to Φ.

4. Since $Lhs(r') \succeq Lhs(\varphi)$, $\varphi \in \Phi$, if and only if $\varphi \in \phi$, we have $\phi = \Phi(r')$. Since $r' = \chi_{\text{init}}(\phi)$ and χ_{init} generates MLGs, r' is an MLG of $\Phi(r')$.

□

Algorithm 7 computes χ_{split}.

Algorithm 7: The splitting operator χ_{split}

```
  Input: an open rule $r$
  Input: a specification $\Phi$
  Output: a finite set $S = \{\langle s_1, \emptyset\rangle, \langle s_2, \emptyset\rangle, \ldots\}$ of pairs of successor-rule sets
          $s_1, s_2, \ldots$ and empty new specification sets
1 $S \leftarrow \emptyset$
2 foreach position $p \in Pos(Lhs(r))$ do
3     if $Node(Lhs(r), p)$ is a variable and $Node(Lhs(\varphi), p)$ is a constructor for all
      $\varphi \in \Phi(r)$ then
          // $p$ is a pivot position
4         $s \leftarrow \{\chi_{\text{init}}(\phi) \mid \phi \in (\Phi(r)/\sim_p)\}$
5         insert $\langle s, \emptyset\rangle$ into $S$
6 return $S$
```

4.7.3. Introducing Subfunctions to Compute Subterms

The subproblem operator χ_{sub} introduces new subfunctions to compute subterms of RHSs of specification rules in $\Phi(r)$, if the respective subterms are open in the initial rule r. A precondition is that the initial rule is rooted by a constructor such that proper subterms exist in it.

The following definition formally specifies the operator χ_{sub}.

Definition 4.12 (Subproblem operator χ_{sub})**.** Let $\langle P, \Phi\rangle$ be a candidate CS and the corresponding specification and B be a background CS such that $\mathcal{D}_P \cap \mathcal{D}_B = \emptyset$. Let $r : f(\boldsymbol{p}) \to c(t_1, \ldots, t_n)$ be an open rule in P and $\Phi(r)$ be the specification subset associated with r.

Let $I = \{i \in [n] \mid Var(t_i) \not\subseteq Var(\boldsymbol{p})\}$ be the set of positions denoting open subterms in the open RHS $c(t_1, \ldots, t_n)$ of r. Let $(g_i)_{i \in I}$ be a family of new function symbols not occurring in P or B: $((g_i) \cap (\mathcal{D}_{P \cup B} \cup \mathcal{C}_{P \cup B} \cup \mathcal{X}) = \emptyset)$. Then

$$\chi_{\text{sub}}(r, \Phi, B) := \left\{\left\langle \{f(\boldsymbol{p}) \to c(t'_1, \ldots, t'_n)\} \cup P_{\text{new}}, \phi_{\text{new}}\right\rangle\right\}$$

where

- for all $j \in [n]$, $t'_j = \begin{cases} t_j & \text{if } j \notin I \\ g_j(\boldsymbol{p}) & \text{if } j \in I \end{cases}$,
- $\phi_{\text{new}} = \{g_j(\boldsymbol{i}) \to o_j \mid j \in I, (f(\boldsymbol{i}) \to c(o_1, \ldots, o_n)) \in \Phi(r)\}$, and
- P_{new} is an initial candidate CS of ϕ_{new}.

If the RHS of an open rule $r \in P$ is *not* rooted by a constructor, then $\chi_{\text{sub}}(r, \Phi, B) = \emptyset$.

Lemma 4.3. *Let $\langle P, \Phi\rangle$, B, $r : f(\boldsymbol{p}) \to c(t_1, \ldots, t_n)$, I and (g_i) be defined as in Definition 4.12 and $\chi_{\text{sub}}(r, \Phi, B) = \langle(\{f(\boldsymbol{p}) \to c(t'_1, \ldots, t'_n)\} \cup P_{\text{new}}), \phi_{\text{new}}\rangle$. Then*

1. *$f(\boldsymbol{p}) \to c(t'_1, \ldots, t'_n)$ is extensionally correct with respect to $\Phi \cup \phi_{\text{new}} \cup B$,*

2. *all rules in P_{new} are extensionally correct with respect to $\Phi \cup \phi_{\text{new}} \cup B$, and*

3. *the new (initial) rules $r' \in P_{\text{new}}$ are MLGs of their respective associated specification subsets $(\Phi \cup \phi_{\text{new}})(r')$.*

Proof. Due to Claim 4.1, $\mathcal{D}_P = \mathcal{D}_\Phi$. With $(g_i) \cap \mathcal{D}_P = \emptyset$ follows $\mathcal{D}_{\phi_{\text{new}}} \cap \mathcal{D}_\Phi = \emptyset$.

1. Let $(f(\boldsymbol{i}) \to o) \in \Phi \cup \phi_{\text{new}} \cup B$ and $f(\boldsymbol{i}) = f(\boldsymbol{p})\sigma$. By definition of $\Phi(r)$ and due to $\mathcal{D}_\Phi \cap (\mathcal{D}_{\phi_{\text{new}}} \cup \mathcal{D}_B) = \emptyset$, $f(\boldsymbol{p}) \not\preceq \mathit{Lhs}(\varphi)$ for all $\varphi \in (\Phi \cup \phi_{\text{new}} \cup B) \setminus \Phi(r)$. Hence $(f(\boldsymbol{i}) \to o) \in \Phi(r)$. We have to show that $c(t'_1, \ldots, t'_n)\sigma \overset{*}{\to}_{\Phi \cup \phi_{\text{new}} \cup B} o$ where $o = c(o_1, \ldots, o_n)$. (We have $\mathit{Var}(\boldsymbol{p}) = \mathit{Var}(c(t'_1, \ldots, t'_n))$, hence θ is empty.) Since r is an MLG of $\Phi(r)$ by Claim 4.1 and $f(\boldsymbol{i}) = f(\boldsymbol{p})\sigma$, we have $t'_j\sigma = t_j\sigma = o_j$ for all $j \in [n] \setminus I$. Hence it remains to show $g_j(\boldsymbol{p})\sigma \overset{*}{\to}_{\Phi \cup \phi_{\text{new}} \cup B} o_j$ for all $j \in I$. This follows from $g_j(\boldsymbol{p})\sigma = g_j(\boldsymbol{i})$ and $(g_j(\boldsymbol{i}) \to o_j) \in \phi_{\text{new}}$.

2. By Lemma 4.1, the rules in P_{new} are extensionally correct with respect to ϕ_{new}. Since $\mathcal{D}_{\phi_{\text{new}}} \cap (\mathcal{D}_\Phi \cup \mathcal{D}_B) = \emptyset$, it follows with Corollary 4.1 that the rules in P_{new} are extensionally correct with respect to $\Phi \cup \phi_{\text{new}} \cup B$.

3. Since P_{new} is an initial CS of ϕ_{new} and due to Lemma 4.1, each $r' \in P_{\text{new}}$ is an MLG of $\phi_{\text{new}}(r')$. Since $\mathcal{D}_\Phi \cap \mathcal{D}_{\phi_{\text{new}}} = \emptyset$, we have $\phi_{\text{new}}(r') = (\Phi \cup \phi_{\text{new}})(r')$.

□

Algorithm 8 computes χ_{sub}.

4.7.4. Introducing Function Calls

Two further operators, χ_{smplCall} and χ_{call}, replace an open RHS by a call of an already introduced function (a target function, a previously introduced help function, or a background function).

Let $f(\boldsymbol{p}) \to t$ be the selected open rule. Then, in the case of χ_{smplCall}, the new rule has the form $f(\boldsymbol{p}) \to f'(\boldsymbol{p'})$, where $\boldsymbol{p'}$ is a constructor term over variables from $\boldsymbol{p}$. In the case of χ_{call}, the new rule has the form $f(\boldsymbol{p}) \to f'(g_1(\boldsymbol{p}), \ldots, g_n(\boldsymbol{p}))$, where the g_i are new function symbols. The idea behind the g_i as arguments is that, if χ_{smplCall} fails, i.e., when no function call with a constructor term as argument can be found, then the argument of the function call possibly needs to be computed by another, possibly new and/or recursive, subfunction. In addition to the new rule $f(\boldsymbol{p}) \to f'(g_1(\boldsymbol{p}), \ldots, g_n(\boldsymbol{p}))$, the operator χ_{call} abduces specifications for the new subfunctions g_i and invokes χ_{init} to get initial rules for them.

Algorithm 8: The subproblem operator χ_{sub}

```
Input: an open rule $r:\ f(\boldsymbol{p}) \to t$
Input: a specification $\Phi$
Input: a background CS $B$
Output: the empty set or a set $\{\langle s, \phi_{\text{new}}\rangle\}$ containing of pair of a successor-rule
        set and the corresponding new specification subset
if $t = c(t_1, \dots, t_n)$ then
    $\phi_{\text{new}} \leftarrow \emptyset$
    foreach $j \in [n]$ do
        if $Var(t_j) \not\subseteq Var(\boldsymbol{p})$ then
            $g_j \leftarrow$ a new defined function symbol; $g_j \notin \mathcal{D}_{P\cup B} \cup \mathcal{C}_{P\cup B} \cup \mathcal{X}$
            $\phi_{\text{new}} \leftarrow \phi_{\text{new}} \cup \{g_j(\boldsymbol{i}) = o_j \mid (f(\boldsymbol{i}) \to c(o_1, \dots, o_n)) \in \Phi(r)\}$
            $t'_j \leftarrow g_j(\boldsymbol{p})$
        else $t'_j \leftarrow t_j$
    $P_{\text{new}} \leftarrow$ initialCandidate($\phi_{\text{new}}$)
    return $\{\langle \{f(\boldsymbol{p}) \to c(t'_1, \dots, t'_n)\} \cup P_{\text{new}}, \phi_{\text{new}}\rangle\}$
else return $\emptyset$
```

Concerning Termination of Solution CSs on Specified Inputs

Certainly, we expect that induced solution CSs are terminating, at least for the specified inputs. In particular, termination on specified inputs follows as necessary condition from the definition of solution CSs (Definition 4.3). The correctness property, $\to_{P\cup B} \supseteq \to_\Phi$, of a solution CS implies that the example inputs specified by Φ *terminate* in $P \cup B$. This is because example outputs are normal forms and the correctness property assures that these normal forms can be reached. Hence, the corresponding derivation terminates. Furthermore, *confluence* of solution CSs assure that *each* derivation of a specified LHS reaches the specified output. Hence, infinite derivations for specified inputs are not possible in solution CSs.[6]

However, termination of a CS does *not* follow from *extensional* correctness of its rules. Suppose we have a specification Φ for one single defined function f, then $f(\boldsymbol{p}) = f(\boldsymbol{p})$ is extensionally correct with respect to Φ, yet it does not terminate for any term (and therefore cannot be (intensionally) correct with respect to Φ). Hence, in addition to extensional correctness, we must assure termination of specified inputs.

The reason for potential *non*-termination is (mutual) recursion. That is, χ_{call} and χ_{smplCall} could potentially cause a (closed) candidate CS to be non-terminating. Intuitively, in order to assure that recursive programs terminate, it must be assured that the arguments of repeated calls of the same function during the evaluation always get strictly reduced with respect to a well-founded order, so that after finitely many calls, a

[6]Note that the correctness property does *not* assure termination *in general* of generated CSs, i.e., that also *non-specified* inputs terminate.

minimal argument is reached and no further call is possible.

Again let $f(\boldsymbol{p}) \to t$ be a selected open rule. Then it is assured that χ_{smplCall} replaces its RHS t only by those function calls $f'(\boldsymbol{p'})$, such that if $f(\boldsymbol{p})\sigma$ is a specified input for f, then $f'(\boldsymbol{p'})\sigma$ is a specified input for f' and $f(\boldsymbol{p})\sigma > f'(\boldsymbol{p'})\sigma$ with respect to a reduction order $>$. (See Definition 2.24 for *reduction order*.)

Analogously, it is assured that χ_{call} replaces the open RHS t only by those function calls $f'(g_1(\boldsymbol{p}), \dots, g_n(\boldsymbol{p}))$, such that if $f(\boldsymbol{p})\sigma$ is a specified input for f, then the $g_i(\boldsymbol{p})\sigma$ are specified inputs for the g_i with corresponding outputs o_i and $f(\boldsymbol{p})\sigma > f'(o_1, \dots, o_n)$ with respect to a reduction order $>$.

This condition is not only posed to direct recursive calls, i.e., if $f = f'$, but for each $f' \in \Phi$, where Φ is the specification of the candidate CS, because recursion can also be introduced indirectly in the form of mutual recursive calls. The only exception is when f' is a background function, i.e., $f' \in \mathcal{D}_B$ for a background CS B. In this case, the call cannot cause recursion because the background CS is defined independently from the candidate CS.

Of course, these conditions must be satisfied with respect to one and the same reduction order for the complete candidate CS. Then, these conditions, together with extensional correctness, assure termination of the specified inputs. Restricted to *linear* terms, the order $s > t$ if and only if $|\,Pos(s)\,| \; > |\,Pos(t)\,|$ is, for example, a reduction order. This reduction order is applied by IGOR2 if no other reduction order is specified.

The Simple-Call Operator χ_{smplCall}

We call function calls introduced by χ_{smplCall}, due to the constructor-term arguments, *simple*. The operator χ_{smplCall} is formally specified by the following definition.

Definition 4.13 (Simple function-call operator χ_{smplCall})**.** Let $\langle P, \Phi\rangle$ be a candidate CS and the corresponding specification and B be a background CS such that $\mathcal{D}_P \cup \mathcal{D}_B = \emptyset$ (Claim 4.1). Let $\mathcal{C}$ denote the set of constructors of $P \cup B$. Let $r : f(\boldsymbol{p}) \to t$ be an open rule in P and $\Phi(r)$ be the specification subset associated with r.

Then $\chi_{\text{smplCall}}(r, \Phi, B)$ yields a (possibly empty) set of pairs of unit rule sets and empty new specifications, where the rules have the form $f(\boldsymbol{p}) \to f'(\boldsymbol{p'})$, $f' \in \mathcal{D}_{\Phi \cup B}$ (possibly $f = f'$) and $\boldsymbol{p'} \in T_{\mathcal{C}}(Var(\boldsymbol{p}))$. The set is uniquely specified as follows:

$$\big\langle \{f(\boldsymbol{p}) \to f'(\boldsymbol{p'})\}, \emptyset \big\rangle \in \chi_{\text{smplCall}}(r, \Phi, B)$$

if and only if for each $(f(\boldsymbol{i}) \to o) \in \Phi(r)$ there is a rule $(f'(\boldsymbol{i'}) \to o') \in \Phi \cup B$ (w.l.o.g. we assume that $f(\boldsymbol{i}) \to o$ and $f'(\boldsymbol{i'}) \to o'$ do not share any variables; this can always be achieved by standardizing apart both rules) such that the following conditions are satisfied. Let σ denote the substitution that matches the selected open rule $r : f(\boldsymbol{p}) \to t$ with the respective specification rule $(f(\boldsymbol{i}) \to o) \in \Phi(r)$, i.e., $(f(\boldsymbol{i}) \to o) = (f(\boldsymbol{p})\sigma \to t\sigma)$.

1. $t\sigma = o'\tau$ for a substitution τ with $Dom(\tau) = Var(o')$. (Each output pattern $t\sigma$ specified in $\Phi(r)$ is subsumed by a specified output pattern o' of f'.)

2. $f'(\boldsymbol{p'}) \succeq f'(\boldsymbol{i'})\tau$ and $f'(\boldsymbol{p'})\sigma = f'(\boldsymbol{i'})\tau\theta$ for any substitution θ[7] with $Dom(\theta) = Var(f'(\boldsymbol{i'})) \setminus Var(o')$. (The formal argument $\boldsymbol{p'}$ of the function call leads to the "correct" inputs for f', i.e., to the inputs with corresponding required outputs $o'\tau$.)
3. If $f' \in \mathcal{D}_\Phi$, then $f(\boldsymbol{p})\sigma > f'(\boldsymbol{p'})\sigma$ with respect to a reduction order $>$. (The argument of the function call is reduced if f' is not a background function and the call can thus lead to recursion.)

If $(f(\boldsymbol{p})\sigma \to t\sigma) \in \Phi(r)$ and the new rule is $f(\boldsymbol{p}) \to f'(\boldsymbol{p'})$, then $f'(\boldsymbol{p'})\sigma$ must reduce to $t\sigma$. We check this (extensionally) at hand of the specification of f': There must be a rule $f'(\boldsymbol{i'}) \to o'$ for f' such that its output subsumes $t\sigma$ (by some substitution τ)—this is condition 1 in Definition 4.13—and $f'(\boldsymbol{p'})\sigma$ is the correspondingly (by τ) instantiated input $f'(\boldsymbol{i'})$—this is condition 2. Condition 3 additionally requires that the argument of the function call is reduced in order to assure termination, if f' is not a background function.

Especially condition 2 needs some further explanation. A naive approach would be to simply require $f'(\boldsymbol{p'})\sigma = f'(\boldsymbol{i'})\tau$. This suffices if $f'(\boldsymbol{i'}) \to o'$ is ground (hence $\tau = \emptyset$) or at least $Var(f'(\boldsymbol{i'})) = Var(o')$. Yet it is also possible that $Var(f'(\boldsymbol{i'})) \supset Var(o')$. If this is the case, the instantiation of the variables in $f'(\boldsymbol{i'})$ not occurring in o' is not determined by the substitution τ. This is the reason for the additional substitution θ in condition 2: We need $f'(\boldsymbol{p'})\sigma = f'(\boldsymbol{i'})\tau\theta$ instead of merely $f'(\boldsymbol{p'})\sigma = f'(\boldsymbol{i'})\tau$. Actually—at least if condition 3 does not apply in the case that f' is a background function—these variables may be arbitrarily instantiated by θ—the output o' and hence $t\sigma$ obviously does not depend on them. Yet simply generating *all* appropriate function calls (for all substitutions θ) is not possible because these are infinitely many. Hence we somehow need to restrict this infinite set of principally appropriate calls to a finite number. While $f'(\boldsymbol{p'})\sigma = f'(\boldsymbol{i'})\tau\theta$ implies that $f'(\boldsymbol{p'})$ is a generalization of $f'(\boldsymbol{i'})\tau\theta$ ($f'(\boldsymbol{p'}) \succeq f'(\boldsymbol{i'})\tau\theta$), we additionally require that $f'(\boldsymbol{p'})$ is already a generalization of $f'(\boldsymbol{i'})\tau$ (without the instantiation θ applied). This essentially bounds the size of $\boldsymbol{p'}$ from above and prevents an infinite set of different calls $f'(\boldsymbol{p'})$.

Lemma 4.4. *Let Φ, B, and $r : f(\boldsymbol{p}) \to t$ be defined as in Definition 4.13 and let be $\langle (f(\boldsymbol{p}) \to f'(\boldsymbol{p'})), \emptyset \rangle \in \chi_{\text{smplCall}}(r, \Phi, B)$. Then $f(\boldsymbol{p}) \to f'(\boldsymbol{p'})$ is extensionally correct with respect to $\Phi \cup B$.*

Proof. Let $(f(\boldsymbol{i}) \to o) \in \Phi \cup B$ and $f(\boldsymbol{i}) = f(\boldsymbol{p})\sigma$. By definition of $\Phi(r)$ and due to $\mathcal{D}_P \cap \mathcal{D}_B = \emptyset$, $f(\boldsymbol{p}) \not\succeq Lhs(\varphi)$ for all $\varphi \in (\Phi \cup B) \setminus \Phi(r)$. Hence $(f(\boldsymbol{i}) \to o) \in \Phi(r)$. Due to condition 2 in Definition 4.13, there is a rule $(f'(\boldsymbol{i'}) \to o') \in \Phi \cup B$ such that $f'(\boldsymbol{p'})\sigma = f'(\boldsymbol{i'})\tau\theta$ for substitutions τ and θ, where $Dom(\tau) = Var(o')$. Hence we have $f'(\boldsymbol{p'})\sigma \to_{\Phi\cup B} o'\tau$. Due to condition 1, $o'\tau = o$, hence we have $f'(\boldsymbol{p'})\sigma \overset{*}{\to}_{\Phi\cup B} o$. □

Algorithm 9 computes χ_{smplCall}. It makes use of Function `sigmaThetaGeneralizations`.

Consider Algorithm 9. The set S to be computed is initialized with the empty set. Then, in the outmost loop, it is iterated over all functions f' (for the function calls

[7]Note the explanations for this additional substitution θ below.

Algorithm 9: The simple call operator χ_{smplCall}

```
Input: an open rule $r:\ f(\boldsymbol{p}) \to t$
Input: a specification $\Phi$
Input: a background CS $B$
Output: a finite (possibly empty) set $S = \{\langle\{r'_1\},\emptyset\rangle, \langle\{r'_2\},\emptyset\rangle, \ldots\}$ of pairs of
        unit successor-rule sets $\{r'_1\},\{r'_2\},\ldots$ and empty new specification
        subsets
1  $S \leftarrow \emptyset$
2  foreach $f' \in \mathcal{D}_{\Phi\cup B}$ do
3      foreach $(f(\boldsymbol{i}) \to o) \in \Phi(r)$ do
4          $G_{f(\boldsymbol{i})} \leftarrow \emptyset$
5          $\sigma \leftarrow$ the substitution that matches $f(\boldsymbol{i})$ with $f(\boldsymbol{p})$ $(f(\boldsymbol{i}) = f(\boldsymbol{p})\sigma)$
6          foreach $(f'(\boldsymbol{i'}) \to o') \in \Phi \cup B$ do
7              if $o = o'\tau$ for any substitution $\tau$ with $Dom(\tau) = Var(o')$ then
8                  $V \leftarrow Var(f'(\boldsymbol{i'})) \setminus Var(o')$
9                  $G_{f(\boldsymbol{i})} \leftarrow G_{f(\boldsymbol{i})} \cup$ sigmaThetaGeneralizations($\sigma, f'(\boldsymbol{i'})\tau, V$)
10                 if $f' \in \Phi$ then remove all $f'(\boldsymbol{p'})$ with $f'(\boldsymbol{p'})\sigma \not< f(\boldsymbol{p})\sigma$ from $G_{f(\boldsymbol{i})}$
11     $G \leftarrow \bigcap_{(f(\boldsymbol{i})\to o)\in\Phi(r)} G_{f(\boldsymbol{i})}$
12     foreach $f'(\boldsymbol{p'}) \in G$ do insert $\langle f(\boldsymbol{p}) \to f'(\boldsymbol{p'}), \emptyset\rangle$ into $S$
13 return $S$
```

$f'(\boldsymbol{p'})$). The first inner loop then iterates over all specification rules associated with the open rule r. Within this loop (lines 4–10), all possible function calls $f'(\boldsymbol{p'})$, that satisfy the three conditions of Definition 4.13 for the current specification rule $f(\boldsymbol{i})$, are generated and collected into the set $G_{f(\boldsymbol{i})}$. Since *each* generated function call must satisfy the three conditions for *all* rules $(f(\boldsymbol{i}) \to o) \in \Phi(r)$, in line 11 the intersection of all the generated $G_{f(\boldsymbol{i})}$ is taken. In the next line, one pair $\langle f(\boldsymbol{p}) \to f'(\boldsymbol{p'}), \emptyset\rangle$ for each remaining function call is generated an inserted to the solution set.

The function calls, collected into the $G_{f(\boldsymbol{i})}$, for a single specification rule $(f(\boldsymbol{i}) \to o) \in \Phi(r)$, are generated as follows (lines 4–10): $G_{f(\boldsymbol{i})}$ is initialized with the empty set. Then the algorithm iterates (line 6) over all specification rules $f'(\boldsymbol{i'}) \to o'$ for the currently selected function f'. First it checks whether the respective RHS o' subsumes the RHS o (by some substitution τ). If this is the case, the variables of $f'(\boldsymbol{i'})$ not occurring in o' (to be instantiated by θ, cp. condition 2 of Definition 4.13 and the explanation in the text) are stored into V. Then all possible function calls according to $f(\boldsymbol{i}) \to o$ and $f'(\boldsymbol{i'}) \to o'$ are generated by the function `sigmaThetaGeneralizations` and added to $G_{f(\boldsymbol{i})}$. If $f' \in \Phi$, then in line 10 all function calls not reducing the argument are removed.

Finally, let us look to Function `sigmaThetaGeneralizations` now: It generates, as described in the following, a set T of all terms s, such that, for given term t, variables $V \subseteq \mathit{Var}(t)$, and substitution σ, $s \succeq t$ and $s\sigma = t\theta$ for arbitrary substitutions θ with $\mathit{Dom}(\theta) = V$. Obviously, since `sigmaThetaGeneralizations` is called with $t = f'(\boldsymbol{i'})\tau$ and $V = \mathit{Var}(f'(\boldsymbol{i'})) \setminus \mathit{Var}(o')$, T is then the set of all terms $f'(\boldsymbol{p'})$ satisfying condition 2 of Definition 4.13 according to the two particular rules $f(\boldsymbol{i}) \to o$ and $f'(\boldsymbol{i'}) \to o'$.

To achieve T as required, we first check each single variable assignment in σ (lines 2, 3 in Function `sigmaThetaGeneralizations`). If the term t' assigned to a variable x according to σ equals $t\theta$, then we have $x\sigma = t\theta$. (Furthermore, obviously $x \succeq t$.) Hence x satisfies the conditions and is added to T. Furthermore, in the case that t is a constant function symbol, t itself satisfies the conditions and is also inserted into T. Finally, t may be a term rooted by a symbol c and with arguments $t_1, \ldots, t_n$. Then we recursively apply `sigmaThetaGeneralizations` to each of these arguments (line 6), achieving sets of terms T_i such that for each $s_i \in T_i$, $s_i \succeq t_i$ and $s_i\sigma = t_i\theta_i$. Then we obviously have $c(s_1, \ldots, s_n) \succeq c(t_1, \ldots, t_n)$ and $c(s_1, \ldots, s_n)\sigma = c(t_1, \ldots, t_n)\theta$ for some substitution θ. Hence we insert $c(s_1, \ldots, s_n)$ for each tuple of the recursively generated s_i into T.

These are all possibilities for terms s satisfying the conditions.

If χ_{smplCall} succeeds, i.e., if at least one rule $f(\boldsymbol{p}) \to f'(\boldsymbol{p'})$ according to Definition 4.13 exists, no other successor rules of r introducing function calls will be generated. Yet if χ_{smplCall} yields the empty set, then the second function-call operator, χ_{call}, is invoked to introduce function calls.

The Function-Call Operator χ_{call}

The operator χ_{call} is formally specified as follows.

Definition 4.14 (Function call operator χ_{call})**.** Let $\langle P, \Phi\rangle$ be a candidate CS and the corresponding specification and B be a background CS such that $\mathcal{D}_P \cap \mathcal{D}_B = \emptyset$ (Claim 4.1).

Function `sigmaThetaGeneralizations`(σ, t, V)

```
Input: a substitution $\sigma$
Input: a linear term $t$
Input: a set of variables $V \subseteq Var(t)$
Output: the (possibly empty) set $T$ of all terms $s$ with $s \succeq t$ and such that
        $s\sigma = t\theta$ for arbitrary substitutions $\theta$ with $Dom(\theta) = V$
1 $T \leftarrow \emptyset$
2 foreach variable assignment $(x \leftarrow t') \in \sigma$ do
3     if $t' = t\theta$ for any substitution $\theta$ with $Dom(\theta) = V$ then insert $x$ into $T$
4 if $t$ is a constant function symbol then insert $t$ into $T$
5 if $t = c(t_1, \ldots, t_n)$ then
6     foreach $i \in [n]$ do $T_i \leftarrow$sigmaThetaGeneralizations$(\sigma, t_i, V)$
7     foreach tuple $(s_1, \ldots, s_n) \in T_1 \times \cdots \times T_n$ do insert $c(s_1, \ldots, s_n)$ into $T$
8 return $T$
```

Let $r : f(\boldsymbol{p}) \to t$ be an open rule in P and $\Phi(r)$ be the specification subset associated with r. Finally, let $(g_i)_{i \in \mathbb{N}}$ be a family of new (function) symbols not occurring in P or B; $(g_i) \cap (\mathcal{D}_{P \cup B} \cup \mathcal{C}_{P \cup B} \cup \mathcal{X}) = \emptyset$.

Then $\chi_{\text{call}}(r, \Phi, B)$ yields a (possibly empty) set of pairs of the form

$$\left\langle \{f(\boldsymbol{p}) \to f'(g_1(\boldsymbol{p}), \ldots, g_n(\boldsymbol{p}))\} \cup P_{\text{new}}, \phi_{\text{new}} \right\rangle, \text{ where } f' \in \mathcal{D}_{\Phi \cup B} \text{ and } \alpha(f') = n\,,$$

specified as follows:

$$\left\langle \{f(\boldsymbol{p}) \to f'(g_1(\boldsymbol{p}), \ldots, g_n(\boldsymbol{p}))\} \cup P_{\text{new}}, \phi_{\text{new}} \right\rangle \in \chi_{\text{call}}(r, \Phi, B)$$

if and only if there is a total mapping $\mu : \Phi(r) \to (\Phi \cup B)(f')$ such that for each $(f(\boldsymbol{i}) \to o) \in \Phi(r)$ and $\mu(f(\boldsymbol{i}) \to o) = (f'(\boldsymbol{i'}) \to o')$ (w.l.o.g. we assume that $f(\boldsymbol{i}) \to o$ and $f'(\boldsymbol{i'}) \to o'$ do not share any variables; this can always be achieved by standardizing apart both rules), the following conditions are satisfied:

1. $Var(f'(\boldsymbol{i'})) = Var(o')$.
2. $o = o'\tau$ for a some substitution τ (Each output pattern $t\sigma$ specified in $\Phi(r)$ is subsumed by a specified output pattern o' of f'.)
3. If $f' \in \mathcal{D}_\Phi$, then $f(\boldsymbol{i}) > f'(\boldsymbol{i'})\tau$ with respect to a reduction order $>$.

Let μ be such a mapping. Then

- $\phi_{\text{new}} := \{g_j(\boldsymbol{i}) \to i'_j\tau \mid j \in [n], (\varphi : f(\boldsymbol{i}) \to o) \in \Phi(r), \mu(\varphi) = (f'(i'_1, \ldots, i'_n) \to o')\}$ and
- P_{new} is an initial candidate CS of ϕ_{new},

If $(f(\boldsymbol{p})\sigma \to t\sigma) \in \Phi(r)$ and the new rule is $f(\boldsymbol{p}) \to f'(g_1(\boldsymbol{p}), \dots, g_n(\boldsymbol{p}))$, then $f'(g_1(\boldsymbol{p}), \dots, g_n(\boldsymbol{p}))\sigma$ must reduce to $t\sigma$. Hence there must be a rule $f'(i'_1, \dots, i'_n) \to o'$ for f' such that its output subsumes $t\sigma$ (by some substitution τ)—this is condition 2 in Definition 4.14. Furthermore, $f'(g_1(\boldsymbol{p}), \dots, g_n(\boldsymbol{p}))\sigma$ must reduce to the corresponding (instantiated by τ) input, $f'(i'_1, \dots, i'_n)\tau$; this is assured by introducing the new specification for the g_j appropriately: Each $g_j(\boldsymbol{i})$ must reduce to $i'_j\tau$. Yet since τ is determined by matching the example output patterns o, o', variables in $f'(\boldsymbol{i'})$ not occurring in o' would not be instantiated by τ. Actually, they could be instantiated arbitrarily; o' does not depend on them. In the current version of the IGOR2 algorithm, we deal with this case only for the simple-call operator; cp. Definition 4.13 and its explanation in the text. For χ_{call}, we currently simply exclude this case by requiring that all variables occurring in $f'(\boldsymbol{i'})$ also occur in o' and hence are (uniquely) instantiated by τ—condition 1 in Definition 4.14. Condition 3 then additionally requires that the arguments of the call are appropriately reduced, if f' is not a background function, in order to assure termination.

For each new function g_j, an initial rule is introduced by computing an initial candidate CS P_{new} for ϕ_{new}.

Lemma 4.5. *Let Φ, B, $r : f(\boldsymbol{p}) \to t$, and (g_i) be defined as in Definition 4.14. and let be $\langle\{f(\boldsymbol{p}) \to f'(g_1(\boldsymbol{p}), \dots, g_n(\boldsymbol{p}))\} \cup P_{\text{new}}, \phi_{\text{new}}\rangle \in \chi_{\text{call}}(r, \Phi, B)$.*
Then

1. *$f(\boldsymbol{p}) \to f'(g_1(\boldsymbol{p}), \dots, g_n(\boldsymbol{p}))$ is extensionally correct with respect to $\Phi \cup \phi_{\text{new}} \cup B$,*
2. *all rules in P_{new} are extensionally correct with respect to $\Phi \cup \phi_{\text{new}} \cup B$, and*
3. *all rules $r' \in P_{\text{new}}$ are MLGs of their associated specification subsets $(\Phi \cup \phi_{\text{new}})(r')$.*

Proof. Due to Claim 4.1, $\mathcal{D}_P = \mathcal{D}_\Phi$. With $(g_i) \cap \mathcal{D}_P = \emptyset$ follows $\mathcal{D}_{\phi_{\text{new}}} \cap \mathcal{D}_\Phi = \emptyset$.

1. Let $(f(\boldsymbol{i}) \to o) \in \Phi \cup \phi_{\text{new}} \cup B$ and $f(\boldsymbol{i}) = f(\boldsymbol{p})\sigma$. By definition of $\Phi(r)$ and due to $\mathcal{D}_\Phi \cap (\mathcal{D}_B \cup (g_i)) = \emptyset$, $f(\boldsymbol{p}) \not\geq Lhs(\varphi)$ for all $\varphi \in (\Phi \cup \phi_{\text{new}} \cup B) \setminus \Phi(r)$. Hence $(f(\boldsymbol{i}) \to o) \in \Phi(r)$. Let $\mu(f(\boldsymbol{i}) \to o) = (f'(i'_1, \dots, i'_n) \to o')$ and $o = o'\tau$. We have to show that $f'(g_1(\boldsymbol{i}), \dots, g_n(\boldsymbol{i})) \overset{*}{\to}_{\Phi\cup\phi_{\text{new}}\cup B} o$. We have $(g_j(\boldsymbol{i}) \to i'_j\tau) \in \phi_{\text{new}}$ for all $j \in [n]$. Therefore $f'(g_1(\boldsymbol{i}), \dots, g_n(\boldsymbol{i})) \overset{*}{\to}_{\Phi\cup\phi_{\text{new}}\cup B} f'(i'_1, \dots, i'_n)\tau$. Finally, from $(f'(i'_1, \dots, i'_n) \to o') \in \Phi \cup B$ and $o = o'\tau$ follows $f'(i'_1, \dots, i'_n)\tau \overset{*}{\to}_{\Phi\cup\phi_{\text{new}}\cup B} o$.
2. By Lemma 4.1, the rules in P_{new} are extensionally correct with respect to ϕ_{new}. Since $\mathcal{D}_{\phi_{\text{new}}} \cap (\mathcal{D}_\Phi \cup \mathcal{D}_B) = \emptyset$, it follows with Corollary 4.1 that the rules in P_{new} are extensionally correct with respect to $\Phi \cup \phi_{\text{new}} \cup B$.
3. Since P_{new} is an initial CS of ϕ_{new} and due to Lemma 4.1, each $r' \in P_{\text{new}}$ is an MLG of $\phi_{\text{new}}(r')$. Since $\mathcal{D}_\Phi \cap \mathcal{D}_{\phi_{\text{new}}} = \emptyset$, we have $\phi_{\text{new}}(r') = (\Phi \cup \phi_{\text{new}})(r')$.

□

Algorithm 11 computes χ_{call}.

Algorithm 11: The function-call operator χ_{call}

```
Input: an open rule $r:\ f(\boldsymbol{p}) \to t$
Input: a specification $\Phi$
Input: a background CS $B$
Output: a set $S = \{\langle s_1, \phi_1\rangle, \langle s_2, \phi_2\rangle, \ldots\}$ of pairs of successor-rule sets
        $s_1, s_2, \ldots$ and corresponding new specification subsets $\phi_1, \phi_2, \ldots$
1  $S \leftarrow \emptyset$
2  foreach $f' \in \mathcal{D}_{\Phi\cup B}$ do
3      $\mu' \leftarrow$ possibleMappings($r, f', \Phi \cup B$)
4      foreach mapping $\mu : \Phi(r) \to \Phi \cup B$ with $\mu(\varphi) \in \mu'(\varphi)$ for each $\varphi \in \Phi(r)$ do
5          if $f' \in \mathcal{D}_B$ or $f(\boldsymbol{i}) > f'(\boldsymbol{i'})\tau$ for each $\mu(f(\boldsymbol{i}) \to o) = (f'(\boldsymbol{i'}) \to o')$ and
           $o = o'\tau$ then
6              $\phi_{\text{new}} \leftarrow \emptyset$
7              foreach $j \in [n]$ do
8                  $g_j \leftarrow$ a new defined function symbol; $g_j \notin (\mathcal{D}_{\Phi\cup B} \cup \mathcal{C}_{\Phi\cup B} \cup \mathcal{X})$
9                  $\phi_{\text{new}} \leftarrow \phi_{\text{new}} \cup$
10                  $\{g_j(\boldsymbol{i}) \to i'_j\tau \mid (\varphi : f(\boldsymbol{i}) \to o) \in \Phi(r), \mu(\varphi) = f'(i'_1, \ldots, i'_n) \to o', o = o'\tau\}$
11             $P_{\text{new}} \leftarrow$ initialCandidate($\phi_{\text{new}}$)
12             insert $\langle\{f(\boldsymbol{p}) \to f'(g_1(\boldsymbol{p}), \ldots, g_n(\boldsymbol{p}))\} \cup P_{\text{new}}, \phi_{\text{new}}\rangle$ into $S$
13 return $S$
```

The outmost loop iterates over all $f' \in \mathcal{D}_{\Phi \cup B}$ for the possible function calls. In each iteration, at first (line 3) a mapping $\mu' : \Phi(r) \to \mathfrak{P}(\Phi \cup B)$ is generated by the help function `possibleMappings`, such that for each $f(\boldsymbol{i}) \to o \in \Phi(r)$ and $f'(\boldsymbol{i'}) \to o' \in \mu'(f(\boldsymbol{i}) \to o)$, conditions 1 and 2 from Definition 4.14 are satisfied.

Then the first inner loop (line 4) iterates over all mappings $\mu : \Phi(r) \to \Phi \cup B$ that satisfy conditions 1 and 2 from Definition 4.14. Line 5 additionally assures that condition 3 is satisfied. Then at first ϕ_{new} is generated (lines 6–10 according to the current mapping μ. After computing the initial candidate CS P_{new} for ϕ_{new} (line 11), the solution pair $\langle \{f(\boldsymbol{p}) \to f'(g_1(\boldsymbol{p}), \ldots, g_n(\boldsymbol{p}))\} \cup P_{\text{new}}, \phi_{\text{new}} \rangle$ according to μ is added to S.

Function `possibleMappings`(r, ϕ, f')

```
    Input: an open rule $r : f(\boldsymbol{p}) \to t$
    Input: a set of specification rules $\phi$
    Input: a function symbol $f' \in \mathcal{D}_\phi$
    Output: a total mapping $\mu' : \phi(r) \to \mathfrak{P}(\phi)$
1 $\mu' \leftarrow \emptyset$
2 foreach $(f(\boldsymbol{i}) \to o) \in \phi(r)$ do
3     $P_{f(\boldsymbol{i})} \leftarrow \emptyset$
4     foreach $(f'(\boldsymbol{i'}) \to o') \in \phi$ with $Var(f'(\boldsymbol{i'})) = Var(o')$ do
5         if $o = o'\tau$ for any substitution $\tau$ then insert $f'(\boldsymbol{i'}) \to o'$ into $P_{f(\boldsymbol{i})}$
6     insert $(f(\boldsymbol{i}) \to o) \mapsto P_{f(\boldsymbol{i})}$ into $\mu'$
7 return $\mu'$
```

4.7.5. The Synthesis Operators Combined

If an open initial rule is selected to be refined, then χ_{split}, χ_{sub}, and either χ_{smplCall} or χ_{call} are applied to r, a specification Φ, and possibly a background CS B, each yielding a (possibly empty) set S of pairs of successor-rule sets and (possibly empty) new specifications.

The sets S computed by the different synthesis operators are pairwise disjoint; this follows immediately from the Definitions 4.11, 4.12, 4.13, and 4.14 of the operators. By $\chi_B(r, \Phi)$ we denote this disjoint union:

Definition 4.15 (Successor candidate-rule sets and corresponding new specification rules)**.** Let $\langle P, \Phi \rangle$ be a candidate CS and the corresponding specification, $r \in P$ be an open rule in P, and B be a background CS. Then $\chi_B(r, \Phi)$—*the set of successor candidate-rule sets and corresponding new specification sets of* r *with respect to* Φ *and* B—is defined as the (disjoint) union of the single synthesis operators:

$$\chi_B(r, \Phi) = \begin{cases} \chi_{\text{split}}(r, \Phi) \mathbin{\dot\cup} \chi_{\text{sub}}(r, \Phi, B) \mathbin{\dot\cup} \chi_{\text{smplCall}}(r, \Phi, B) & \text{if } \chi_{\text{smplCall}}(r, \Phi, B) \neq \emptyset \\ \chi_{\text{split}}(r, \Phi) \mathbin{\dot\cup} \chi_{\text{sub}}(r, \Phi, B) \mathbin{\dot\cup} \chi_{\text{call}}(r, \Phi, B) & \text{if } \chi_{\text{smplCall}}(r, \Phi, B) = \emptyset \end{cases}$$

The following corollary states some simple facts regarding χ_B that immediately follow from the definitions of the synthesis operators.

Corollary 4.5. *Let $\langle P, \Phi \rangle$ be a candidate CS and a corresponding specification, B be a background CS, $r \in P$ be an open rule in P, and $\langle s, \phi_{\text{new}} \rangle \in \chi_B(r, \Phi)$.*

- *$r \notin s$.*
- *s contains a rule r' with $Lhs(r) \succeq Lhs(r')$.*
- *$\chi_B(r, \Phi)$, s, and ϕ_{new} are all finite, if P, Φ, B are finite.*

4.8. Properties of the Igor2 Algorithm

In this section, we show that, under certain conditions, the search of IGOR2 is terminating and complete with respect to the problem space determined by the initial specification, the provided background CS, and the synthesis operators. (By *initial* specification we explicitly refer to the original, user-provided specification; in contrast to the specifications belonging to candidates, that all include the initial specification but possibly specify further subfunctions that were introduced by the synthesis operators.)

Furthermore we show that closed CSs in the problem space, i.e., those candidate CSs that are accepted as solutions, are indeed correct (according to Definition 4.2) with respect to the provided specification. Together with the also proven facts that candidate CSs are orthogonal and do not redefine background functions, it follows that accepted candidate CSs are solutions of the induction problem as stated in Definition 4.3.

Furthermore, we shortly discuss the completeness of IGOR2 with respect to certain function classes and the complexity of IGOR2.

We start with a formal definition of the problem space.

4.8.1. Formalization of the Problem Space

Given a candidate CS P, a corresponding specification Φ, and a background CS B, a successor CS P' and corresponding successor specification Φ' results from selecting one (arbitrary) open rule $r \in P$, applying the synthesis operators to get $\chi_B(r, \Phi)$, choosing one $\langle s, \phi_{\text{new}} \rangle$ from it, and removing r and adding s to P to get P' and adding ϕ_{new} to Φ to get Φ'. Formally, the set of *all* possible successor CSs and successor specifications is defined as follows:

Definition 4.16 (Successor candidate CSs and specifications). Let $\langle P, \Phi \rangle$ be a candidate CS and a corresponding specification and B be a background CS. Then *the set of successor CSs and corresponding specifications of $\langle P, \Phi \rangle$ with respect to B*, denoted by $\Xi_B(\langle P, \Phi \rangle)$, is defined as

$$\Xi_B(\langle P, \Phi \rangle) := \bigcup_{\substack{r \in P \\ r \text{ open}}} \left\{ \big\langle (P \setminus \{r\}) \cup s \, , \, \Phi \cup \phi_{\text{new}} \big\rangle \mid \langle s, \phi_{\text{new}} \rangle \in \chi_B(r, \Phi) \right\} .$$

Now the problem space according to an initial, user-provided, specification Φ and a background CS B, denoted by $\mathcal{P}_{\Phi,B}$, is defined as a graph where CSs and corresponding specifications (reachable from initial CSs of Φ by repeatedly applying the synthesis operators) are nodes and an arc exists between two nodes $\langle P, \Phi\rangle$ and $\langle P', \Phi'\rangle$ if and only if $\langle P', \Phi'\rangle$ is a successor CS and successor specification of $\langle P, \Phi\rangle$.

Definition 4.17 (Problem Space)**.** Let Φ and B be an initial specification and a background CS, respectively, such that $\mathcal{D}_\Phi \cap \mathcal{D}_B = \emptyset$. Then *the problem space with respect to* Φ *and* B, denoted by $\mathcal{P}_{\Phi,B}$, is a directed graph defined as follows:

- If P is an initial CS of Φ, the pair $\langle P, \Phi\rangle$ is a node in $\mathcal{P}_{\Phi,B}$. We call this node an *initial node (of* $\mathcal{P}_{\Phi,B}$*).*
- If and only if $\langle P, \Psi\rangle$ is a node in $\mathcal{P}_{\Phi,B}$ and $\langle P', \Psi'\rangle \in \Xi_B(\langle P, \Psi\rangle)$, then $\langle P', \Psi'\rangle$ is a node in $\mathcal{P}_{\Phi,B}$ and $(\langle P, \Psi\rangle, \langle P', \Psi'\rangle)$ is a directed arc in $\mathcal{P}_{\Phi,B}$.

Definition 4.18 (Reachable nodes)**.** Let N be any node in $\mathcal{P}_{\Phi,B}$.

By $\Xi^n_B(N)$ we denote the set of all nodes in $\mathcal{P}_{\Phi,B}$ that are reachable from N by a path of length n.

By $\Xi^*_B(N)$ we denote the set of all nodes in $\mathcal{P}_{\Phi,B}$ that are reachable from N (by a path of any length).

Notation 4.1. Let $\langle P, \Psi\rangle$ be any node in $\mathcal{P}_{\Phi,B}$, $r \in P$ be an open rule in P, $\langle s, \phi_{\text{new}}\rangle \in \chi_B(r, \Phi)$ be a successor-rule set and corresponding new specification, and $\langle P', \Psi'\rangle = \langle (P \setminus \{r\}) \cup s, \Psi \cup \phi_{\text{new}}\rangle \in \Xi_B(\langle P, \Psi\rangle)$ be the corresponding successor CS and corresponding successor specification of $\langle P, \Psi\rangle$.

If $\langle s, \phi_{\text{new}}\rangle \in \chi_{\text{split}}(r, \Phi)$, we denote the corresponding arc in $\mathcal{P}_{\Phi,B}$ by

$$\langle P, \Psi\rangle \xrightarrow{\chi_{\text{split}}} \langle P', \Psi'\rangle \ .$$

We denote the arc analogously when $\langle s, \phi_{\text{new}}\rangle$ is generated by χ_{sub}, χ_{smplCall}, or χ_{call}.

Now we can restate and prove the assumptions that were made in Claim 4.1.

Lemma 4.6. *Let Φ be a specification and B be a background CS such that $\mathcal{D}_\Phi \cap \mathcal{D}_B = \emptyset$. Further, let $\langle P, \Psi\rangle \in \mathcal{P}_{\Phi,B}$ be any candidate CS and corresponding specification generated by the synthesis operators from the initial specification Φ and background CS B.*

Then

1. $\mathcal{D}_P = \mathcal{D}_\Psi$,
2. $\mathcal{D}_P \cap \mathcal{D}_B = \emptyset$, *and*
3. *if $r \in P$ is an open rule in P then r is an MLG of $\Psi(r)$.*

Proof. Regarding the second statement, note that all synthesis operators, including χ_{init}, by definition only introduce rules for defined functions $f \in \Phi$ and possibly for new defined functions g with $g \notin \mathcal{D}_B$. With $\mathcal{D}_\Phi \cap \mathcal{D}_B = \emptyset$ then follows that $\mathcal{D}_P \cap \mathcal{D}_B = \emptyset$.

We prove statements 1 and 3 by induction over the length n of a path to reach $\langle P, \Psi \rangle$ from an initial node I of $\mathcal{P}_{\Phi,B}$.

Induction base ($n = 0$): In this case, $\langle P, \Psi \rangle$ is an initial node in $\mathcal{P}_{\Phi,B}$, i.e., P is an initial CS of Φ. All three statements are then true by Lemma 4.1.

Induction step ($n \to n+1$): Let I be an initial node in $\mathcal{P}_{\Phi,B}$ and $\langle P, \Psi \rangle \in \Xi^n_B(I)$ be a node in $\mathcal{P}_{\Phi,B}$ that is reachable by a path of length n from I. By induction assumption, all three statements are true for $\langle P, \Psi \rangle$.

Let $r \in P$ be an open rule in P, $\langle s, \phi_{\text{new}} \rangle \in \chi_B(r, \Phi)$ be one corresponding successor-rule set and new specification, and $\langle P', \Psi' \rangle \in \Xi_B(\langle P, \Psi \rangle)$ be the corresponding successor CS and successor specification of $\langle P, \Psi \rangle$, hence reachable from I by a path of length $n+1$.

- *Statement* 1*:* If $\langle P, \Psi \rangle \xrightarrow{\chi_{\text{split}}} \langle P', \Psi' \rangle$ or $\langle P, \Psi \rangle \xrightarrow{\chi_{\text{smplCall}}} \langle P', \Psi' \rangle$, then $\phi_{\text{new}} = \emptyset$ and $\mathcal{D}_{P'} = \mathcal{D}_P$ and $\mathcal{D}_{\Psi'} = \mathcal{D}_{\Psi'}$, hence $\mathcal{D}_{P'} = \mathcal{D}_{\Psi'}$ by induction assumption.

  If $\langle P, \Psi \rangle \xrightarrow{\chi_{\text{sub}}} \langle P', \Psi' \rangle$ or $\langle P, \Psi \rangle \xrightarrow{\chi_{\text{call}}} \langle P', \Psi' \rangle$, then $\mathcal{D}_{P'} = \mathcal{D}_P \cup \mathcal{D}_{P_{\text{new}}}$ and $\mathcal{D}_{\Psi'} = \mathcal{D}_{\Psi'} \cup \mathcal{D}_{\phi_{\text{new}}}$ (where $s = \{r'\} \cup P_{\text{new}}$ and P_{new} is the initial CS of ϕ_{new}). Since P_{new} is an initial CS of ϕ_{new}, $\mathcal{D}_{P_{\text{new}}} = \mathcal{D}_{\phi_{\text{new}}}$, hence $\mathcal{D}_{P'} = \mathcal{D}_{\Psi'}$ by induction assumption.

- *Statement* 3*:* If $\langle P, \Psi \rangle \xrightarrow{\chi_{\text{split}}} \langle P', \Psi' \rangle$, each rule $r' \in s$ is an MLG of $\Psi'(r')$ by Lemma 4.2. Since $\Psi = \Psi'$ in this case and by the induction assumption, also each rule $r' \in P' \setminus s$ is an MLG of $\Psi'(r')$.

  If $\langle P, \Psi \rangle \xrightarrow{\chi_{\text{smplCall}}} \langle P', \Psi' \rangle$ the open rules of P' are identical to those in P, except for the selected open rule $r \in P$ that is replaced by the only (closed) successor rule in P'. Furthermore, $\Psi' = \Psi$. Hence, by induction assumption, for each open rule $r \in P'$, r is an MLG of $\Psi'(r)$.

  If $\langle P, \Psi \rangle \xrightarrow{\chi_{\text{sub}}} \langle P', \Psi' \rangle$ or $\langle P, \Psi \rangle \xrightarrow{\chi_{\text{call}}} \langle P', \Psi' \rangle$, the rules $r' \in P_{\text{new}}$ are MLGs of their associated specification subsets $\Psi'(r')$ by Lemmata 4.3 and 4.5. Since $\mathcal{D}_\Psi \cap \mathcal{D}_{\phi_{\text{new}}} = \emptyset$ and by the induction assumption, also all rules $r' \in P' \setminus s$ are MLGs of $\Psi'(r')$.

□

4.8.2. Termination and Completeness of Igor2's Search

We now first show that the problem space has no loops and cycles (Lemma 4.8) and is, under a certain restriction (Definition 4.19), finite (Lemma 4.10). From these results we conclude termination of IGOR2 (Theorem 4.1).

Furthermore we show that under another restriction (Definition 4.20) the order in which open rules in a candidate CS are replaced does not affect the resulting (closed) CS and that it is thus sufficient that IGOR2 in each search-step only computes successor CSs with respect to *one arbitrary* open rule in a CS (Lemma 4.11) instead of introducing *all* possible successor CSs (with respect to all open rules). From this result we conclude completeness of IGOR2's search strategy. (Theorem 4.2).

In order to show that $\mathcal{P}_{\Phi,B}$ is acyclic, we need the result that candidate CSs are orthogonal.

Lemma 4.7 (Orthogonality of candidate CSs)**.** *Let $\langle P, \Psi\rangle$ be a node in $\mathcal{P}_{\Phi,B}$. Then P is orthogonal.*

Proof. Linear LHSs: Only χ_{init} generates LHSs. The other synthesis operators either only replace RHSs or generate new rules (new LHSs) by invoking χ_{init}. Since χ_{init} generates MLGs, generated LHSs are linear.

Pairwise non-unifying LHSs: We show pairwise non-unifying LHSs of P by induction over the length n of a path to reach $\langle P, \Psi\rangle$ from an initial node I.

Induction base ($n = 0$): The initial CS of the initial specification Φ is orthogonal by Lemma 4.1.

Induction step ($n \to n+1$): Let $\langle P, \Psi\rangle \in \Xi^n_B(I)$ be a node that is reachable by a path of length n from I. By induction assumption, the LHSs of P are pairwise non-unifying.

Let $\langle P', \Psi'\rangle \in \Xi_B(\langle P, \Psi\rangle)$, hence reachable by a path of length $n+1$ from I.

If $\langle P, \Psi\rangle \xrightarrow{\chi_{\text{split}}} \langle P', \Psi'\rangle$ and r and s are the correspondingly selected open rule $r \in P$ and successor rules computed by χ_{split} to achieve P', then by Lemma 4.2, s is orthogonal. Furthermore, for each rule $r' \in s$, $\mathit{Lhs}(r) \succeq \mathit{Lhs}(r')$. Hence, since r is, by induction assumption non-unifying with all other rules in P, also each $r' \in s$ is non-unifying with all other rules in P. It follows that all rules in P' are pairwise non-unifying.

If $\langle P, \Psi\rangle \xrightarrow{\chi_{\text{smplCall}}} \langle P', \Psi'\rangle$, then $\mathit{Lhss}(P') = \mathit{Lhss}(P)$, hence the LHSs of P' are pairwise non-unifying by induction assumption.

If $\langle P, \Psi\rangle \xrightarrow{\chi_{\text{sub}}} \langle P', \Psi'\rangle$ or $\langle P, \Psi\rangle \xrightarrow{\chi_{\text{call}}} \langle P', \Psi'\rangle$, then additionally to the LHSs of P, P' contains the LHSs from P_{new} (cp. Definitions 4.12 and 4.14). Due to Lemma 4.1, the LHSs of P_{new} are pairwise non-unifying. Furthermore, $\mathcal{D}_P$ and $\mathcal{D}_{P_{\text{new}}}$ are disjoint. Together with the induction assumption follows that the LHSs of P' are pairwise non-unifying. □

The following lemma states that the program space does not contain loops and circles. The non-existence of loops is quite obvious: The open rule selected in a candidate CS P to compute successor CSs P' is not contained in P', hence $P \neq P'$. The idea to prove non-existence of circles is to show that P' contains a rule which is not contained in P and which (or at least any of its successor rules also not contained in P) cannot be removed again in further synthesis steps.

Lemma 4.8. *The problem space $\mathcal{P}_{\Phi,B}$ contains no loops (arcs of the form (N,N)), and no cycles, (paths of at least length 2 with the same start and end node).*

Proof. Let $(\langle P_0, \Psi_0\rangle, \langle P_1, \Psi_1\rangle)$ be any arc in $\mathcal{P}_{\Phi,B}$ and let r_0 be the corresponding selected open rule in P_0. Then, by Corollary 4.5 and Definition 4.16, $r_0 \notin P_1$, hence $P_0 \neq P_1$, hence $\mathcal{P}_{\Phi,B}$ contains no loops.

Now assume further that there is a path $P = (\langle P_1, \Psi_1\rangle, \langle P_2, \Psi_2\rangle, \ldots, \langle P_k, \Psi_k\rangle)$ with $\langle P_k, \Psi_k\rangle = \langle P_0, \Psi_0\rangle$, i.e., that $\mathcal{P}_{\Phi,B}$ contains a cycle.

First assume $\langle P_0, \Psi_0\rangle \xrightarrow{\chi_{\text{split}}} \langle P_1, \Psi_1\rangle$. Then P_1 contains a rule r' with $\mathit{Lhs}(r_0) \succ \mathit{Lhs}(r')$. It holds $r' \notin P_0$, because otherwise, since $r_0 \in P_0$, P_0 would not be orthogonal. Due to $r_0 \in P_0$ and orthogonality of P_0, there is no rule r in P_0 such that $\mathit{Lhs}(r') \succeq \mathit{Lhs}(r)$. Yet because of $r' \in P_1$, each CS that is reachable from P_1 contains such a rule r (due to Corollary 4.5 and Definition 4.17). In particular, $P_k = P_0$ must contain such a rule r. Hence we have $r \notin P_0$ and $r \in P_0$ for a rule r with $\mathit{Lhs}(r') \succeq \mathit{Lhs}(r)$—contradiction!

For all three other cases— $\langle P_0, \Psi_0\rangle \xrightarrow{\chi_{\text{sub}}} \langle P_1, \Psi_1\rangle$, $\langle P_0, \Psi_0\rangle \xrightarrow{\chi_{\text{smplCall}}} \langle P_1, \Psi_1\rangle$, or $\langle P_0, \Psi_0\rangle \xrightarrow{\chi_{\text{call}}} \langle P_1, \Psi_1\rangle$—let r' be the closed successor-rule of r_0 with $\mathit{Lhs}(r') = \mathit{Lhs}(r_0)$ (cp. Definitions 4.12, 4.13, and 4.14). Since $\mathit{Lhs}(r') = \mathit{Lhs}(r_0)$ and $r_0 \in P_0$, from orthogonality of P_0 follows $r' \notin P_0$. Yet whenever $\langle P', \Psi'\rangle \in \Xi_B(\langle P, \Psi\rangle)$, then all closed rules from P are also in P' (this follows immediately from Definition 4.16). Hence we have $r' \in P_i$ for each $i \in [k]$. In particular, $r' \in P_k (= P_0)$; hence $r' \notin P_0$ and $r' \in P_0$—contradiction!

Hence the path $P = (\langle P_0, \Psi_0\rangle, \langle P_1, \Psi_1\rangle, \ldots, \langle P_{k-1}, \Psi_{k-1}\rangle, \langle P_0, \Psi_0\rangle)$ does not exist. $\square$

Due to the operator χ_{call}, the problem space is not necessarily finite. χ_{call} introduces new functions as arguments of function calls. These new functions again may call other functions leading to another set of new argument-computing functions and so forth ad infinitum. Potentially, this leads to non-termination.

Furthermore, since introducing a function call as RHS of an open rule does not increase the cost of the candidate CS, possibly infinitely many (open) candidate CSs of one and the same cost g exist. This may lead to incompleteness (if each *closed* CS has a *higher* cost $g + g'$).

Therefore, we introduce the concept of *depth* of a rule which reflects its depth in a sequence of nested function calls. If we bound the depth from above by any natural number, then the problem space becomes finite and IGOR2 is terminating and complete.[8]

Definition 4.19 (Depth of rules)**.** Let $\langle P, \Phi\rangle$ be an initial node of $\mathcal{P}_{\Phi,B}$. Then for all rules $r \in P$, $\mathit{Depth}(r, \langle P, \Phi\rangle) = 0$.

Now let $\langle P, \Psi\rangle$ be any node in $\mathcal{P}_{\Phi,B}$, $(r : f(\boldsymbol{p}) \to t) \in P$ be an open rule, $\langle s, \psi\rangle \in \chi_B(r, \Psi)$ be some successor rule-set (and corresponding new specification), and $\langle P', \Psi'\rangle = \langle (P \setminus \{r\}) \cup s, \Psi \cup \psi\rangle$ be the corresponding successor node in $\mathcal{P}_{\Phi,B}$.

1. For all $r' \in P \cap P'$, $\mathit{Depth}(r', \langle P', \Psi'\rangle) := \mathit{Depth}(r', \langle P, \Psi\rangle)$.
2. If $\langle s, \psi\rangle \in \chi_{\text{split}}(r, \Psi) \,\dot\cup\, \chi_{\text{sub}}(r, \Psi, B) \,\dot\cup\, \chi_{\text{smplCall}}(r, \Psi, B)$, then for each rule in $r' \in s$, $\mathit{Depth}(r', \langle P', \Psi'\rangle) := \mathit{Depth}(r, \langle P, \Psi\rangle)$.
3. If $\langle \{f'(g_1(\boldsymbol{p}), \ldots, g_n(\boldsymbol{p}))\} \cup P_{\text{new}}, \psi\rangle \in \chi_{\text{call}}(r, \Psi, B)$, then

$$\mathit{Depth}(f'(g_1(\boldsymbol{p}), \ldots, g_n(\boldsymbol{p})), \langle P', \Psi'\rangle) := \mathit{Depth}(r, \langle P, \Psi\rangle)$$

[8]Probably a better, more general, solution would be to also considering the depth of a rule as a measure of the cost of a candidate CS.

and for each rule $r' \in P_{\text{new}}$,

$$Depth(r', \langle P', \Psi' \rangle) := Depth(r, \langle P, \Psi \rangle) + 1\,.$$

The following lemma shows that in each path in $\mathcal{P}_{\Phi,B}$, where the depth of all corresponding selected open rules is equal to or greater than a fixed natural number m, only finitely many of the selected rules have a depth equal to m.

Lemma 4.9. *Let $m \in \mathbb{N}$ be any natural number. Let $P = (\langle P_0, \Psi_0 \rangle, \langle P_1, \Psi_1 \rangle, \ldots)$ be a path in $\mathcal{P}_{\Phi,B}$ and $r_i \in P_i$ ($i = 0, 1, \ldots$) be the respective selected open rules, such that for all $i = 0, 1, \ldots$, $Depth(r_i, \langle P_i, \Psi_i \rangle) \geq m$. Then for only finitely many $j \in \mathbb{N}$, $Depth(r_j, \langle P_j, \Psi_j \rangle) = m$.*

Proof. For any node $\langle P_i, \Psi_i \rangle$ at path P let $\mathcal{O}_{m,i} \subseteq P_i$ denote the set of open rules of P_i with depth m:

$$\mathcal{O}_{m,i} = \{r \in P_i \mid r \text{ open and } Depth(r, \langle P_i, \Psi_i \rangle) = m\}\,.$$

For the following, we need two additional auxiliary concepts:

1. Let Φ be any specification and R be a set of rules. Then the *specification subset of Φ associated with R* is defined as the union of the specification subsets associated with the single rules in R:
$$\Phi(R) := \bigcup_{r \in R} \Phi(r)\,.$$
2. Let R be any set of rules. Then by $Rhss(R)$ we denote the *multi-set* of all RHSs in R. (That is, if any RHS appears more then once in R, then all multiple occurrences are elements of $Rhss(R)$).

Now we consider two quantities based on $\mathcal{O}_{m,i}$:

1. $Size(Rhss(\Psi_i(\mathcal{O}_{m,i})))$: The summerized number of positions in all RHSs belonging to the specification subset of Ψ_i associated with the set of open rules of depth m in P_i.
2. $\frac{|\,\Psi_i(\mathcal{O}_{m,i})\,|}{|\,\mathcal{O}_{m,i}\,|}$: The number of rules in the specification subset associated with the set of open rules of depth m in P_i divided by the number of open rules of depth m in P_i.

At first we consider how $Size(Rhss(\Psi_i(\mathcal{O}_{m,i})))$ behaves for $i = 0, 1, \ldots$:
Let $(\langle P_j, \Psi_j \rangle, \langle P_{j+1}, \Psi_{j+1} \rangle)$ be any arc in path P. We distinguish the following four (1, 2a, 2b, 3) exhaustive cases:

1. $Depth(r_j, \langle P_j, \Psi_j \rangle) > m$: Then $\mathcal{O}_{m,j} = \mathcal{O}_{m,j+1}$, hence $\Psi_j(\mathcal{O}_{m,j}) = \Psi_{j+1}(\mathcal{O}_{m,j+1})$, hence $Size(Rhss(\Psi_j(\mathcal{O}_{m,j}))) = Size(Rhss(\Psi_{j+1}(\mathcal{O}_{m,j+1})))$.
2. $Depth(r_j, \langle P_j, \Psi_j \rangle) = m$ and $\langle P_j, \Psi_j \rangle \xrightarrow{\chi_{\text{split}}} \langle P_{j+1}, \Psi_{j+1} \rangle$: Then either

a) at least one of the more specific successor rules of r_j is closed such that $|\,\Psi_j(\mathcal{O}_{m,j})\,| \;>\; |\,\Psi_{j+1}(\mathcal{O}_{m,j+1})\,|$, hence $\mathit{Size}(\mathit{Rhss}(\Psi_j(\mathcal{O}_{m,j}))) > \mathit{Size}(\mathit{Rhss}(\Psi_{j+1}(\mathcal{O}_{m,j+1})))$ or

b) all successor rules are also open such that $\Psi_j(\mathcal{O}_{m,j}) = \Psi_{j+1}(\mathcal{O}_{m,j+1})$, hence $\mathit{Size}(\mathit{Rhss}(\Psi_j(\mathcal{O}_{m,j}))) = \mathit{Size}(\mathit{Rhss}(\Psi_{j+1}(\mathcal{O}_{m,j+1})))$.

3. $\mathit{Depth}(r_j, \langle P_j, \Psi_j\rangle) = m$ and $\langle P_j, \Psi_j\rangle \xrightarrow{\chi_{\text{sub}}} \langle P_{j+1}, \Psi_{j+1}\rangle$ or $\langle P_j, \Psi_j\rangle \xrightarrow{\chi_{\text{smplCall}}} \langle P_{j+1}, \Psi_{j+1}\rangle$ or $\langle P_j, \Psi_j\rangle \xrightarrow{\chi_{\text{call}}} \langle P_{j+1}, \Psi_{j+1}\rangle$. Then $\mathit{Size}(\mathit{Rhss}(\Psi_j(\mathcal{O}_{m,j}))) > \mathit{Size}(\mathit{Rhss}(\Psi_{j+1}(\mathcal{O}_{m,j+1})))$.

   In the case of χ_{smplCall} this is quite obvious because χ_{smplCall} simply closes the selected rule r_j, hence P_{j+1} has one open rule less, and does not change the specification.

   In the case of χ_{sub}, the selected rule r_j is also closed but new initial rules for the new subfunctions are added with the same depth. For these, new specification rules are added. Yet note that the RHSs of the new specification rules are *proper subterms* of the RHSs in $\Psi_j(r_j)$ and are in total smaller than the RHSs in $\Psi_j(r_j)$.

   In the case of χ_{call}, the selected rule r_j is also closed but new specification rules for new subfunctions are added. Yet note that in this case, the initial candidate rules in P_{j+1} for the new subfunctions have a strictly increased depth. Hence their specification rules do not count.

Hence in any case $\mathit{Size}(\mathit{Rhss}(\Psi_j(\mathcal{O}_{m,j}))) \geq \mathit{Size}(\mathit{Rhss}(\Psi_{j+1}(\mathcal{O}_{m,j+1})))$ and it follows immediately that only finitely many arcs in P can be of cases 2(a) and 3.

We now consider the second quantity for particular subsequences in path P: Let, for some $l \in \mathbb{N}$, $P' = (\langle P_l, \Psi_l\rangle, \langle P_{l+1}, \Psi_{l+1}\rangle, \ldots)$ be a subsequence of P such that each arc in P' is of cases 1 or 2(b). Let $(\langle P_j, \Psi_j\rangle, \langle P_{j+1}, \Psi_{j+1}\rangle)$ be any arc in P'. If the arc is of case 1, i.e., $\mathcal{O}_{m,j} = \mathcal{O}_{m,j+1}$ and $\Psi_j(\mathcal{O}_{m,j}) = \Psi_{j+1}(\mathcal{O}_{m,j+1})$, then obviously

$$\frac{|\,\Psi(\mathcal{O}_{m,j})\,|}{|\,\mathcal{O}_{m,j}\,|} = \frac{|\,\Psi(\mathcal{O}_{m,j+1})\,|}{|\,\mathcal{O}_{m,j+1}\,|}\,.$$

If the arc is of case 2(b), then $\mathcal{O}_{m,j} < \mathcal{O}_{m,j+1}$ and $\Psi_j(\mathcal{O}_{m,j}) = \Psi_{j+1}(\mathcal{O}_{m,j+1})$, hence

$$\frac{|\,\Psi(\mathcal{O}_{m,j})\,|}{|\,\mathcal{O}_{m,j}\,|} > \frac{|\,\Psi(\mathcal{O}_{m,j+1})\,|}{|\,\mathcal{O}_{m,j+1}\,|}\,.$$

$|\,\mathcal{O}_{m,j}\,|$ cannot exceed $|\,\Psi(\mathcal{O}_{m,j})\,|$, hence P' contains only finitely many arcs of case 2(b). In general: Between each two of the finitely many arcs of cases 2(a) and 3 in path P, only finitely many arcs of case 2(b) can occur. Hence in P only finitely many arcs of cases 2 and 3, where the selected open rule is of depth m, occur. □

As a corollary we get the result that each path in $\mathcal{P}_{\Phi,B}$ only contains finitely many arcs where the selected open rules have a depth equal to or smaller than any fixed natural number n. That is, if the maximal depth of rules in $\mathcal{P}_{\Phi,B}$ is set to any natural number, then all paths in $\mathcal{P}_{\Phi,B}$ are of finite length.

Corollary 4.6. *Let $P = (\langle P_0, \Psi_0\rangle, \langle P_1, \Psi_1\rangle, \ldots)$ be any path in $\mathcal{P}_{\Phi,B}$, $r_i \in P_i$ $(i = 0, 1, \ldots)$ be the respective selected open rules, and $n \in \mathbb{N}$ be any natural number. Then for only finitely many $j \in \mathbb{N}$, $Depth(r_j, \langle P_j, \Psi_j\rangle) \leq n$.*

Proof. By induction over n.

Induction base ($n = 0$): Since the depth of a rule in a node of $\mathcal{P}_{\Phi,B}$ cannot be smaller than 0, we have to show that for only finitely many $j \in \mathbb{N}$, $Depth(r_j, \langle P_j, \Psi_j\rangle) = n$. Since the depth of all selected rules is equal to or greater than 0, this follows immediately from Lemma 4.9

Induction step ($n \to n + 1$): We assume that $Depth(r_j, \langle P_j, \Psi_j\rangle) \leq n$ for only finitely many $j \in \mathbb{N}$. Hence only finitely many subsequences P' of P exist such all selected open rules in P' have a depth equal to or greater than $n+1$. From Lemma 4.9 follows that in these subsequences P' only finitely many of the selected open rules are of depth $n + 1$. Hence also the number of arcs in P with selected open rules of depth $n + 1$ is finite. □

Lemma 4.10 ((Conditional) finiteness of the problem space). *Let Φ and B be an initial specification and a background CS, respectively. If the depth of rules cannot exceed a fixed natural number $M \in \mathbb{N}$, then $\mathcal{P}_{\Phi,B}$ is finite.*

Proof. From Lemma 4.6 follows that all paths in $\mathcal{P}_{\Phi,B}$ are finite provided such a maximal depth M is given.

Since the number of rules in an initial node of $\mathcal{P}_{\Phi,B}$ and in B is finite and if $\langle s, \phi_{\text{new}}\rangle \in \chi_B(r, \Psi)$ and Ψ is finite, then also s and ϕ_{new} are finite, for each node $\langle P, \Psi\rangle$ of $\mathcal{P}_{\Phi,B}$, P and Ψ are finite. Then, furthermore, $\chi_B(r, \Psi)$, and therefore also $\Xi_B(\langle P, \Psi\rangle)$, are finite for each node $\langle P, \Psi\rangle$ in $\mathcal{P}_{\Phi,B}$ and each open rule $r \in P$. Hence each node in $\mathcal{P}_{\Phi,B}$ has only a finite number of direct successor nodes.

Finally, each node in $\mathcal{P}_{\Phi,B}$ is reachable from some initial node and there are only finitely many initial nodes.

It follows that $\mathcal{P}_{\Phi,B}$ contains only finitely many nodes. □

Now we can conclude that IGOR2 is terminating under the condition that the depth of rules may not exceed a certain upper bound.

Theorem 4.1 (Termination of IGOR2). *If the depth of rules cannot exceed a fixed natural number $M \in \mathbb{N}$, then* IGOR2 *is terminating.*

Proof. Consider Algorithm 4. For an initial specification Φ and a background CS B, IGOR2 maintains a subset $\mathcal{N}$ of nodes of the problem space $\mathcal{P}_{\Phi,B}$. Initially, $\mathcal{N}$ contains one of the initial nodes of $\mathcal{P}_{\Phi,B}$. In each iteration of the main loop, one of the nodes is replaced by a subset (according to *one* particular selected open rule) of its direct successor nodes in $\mathcal{P}_{\Phi,B}$.

Since the problem space is a finite (provided a maximal depth of rules is given) directed acyclic and loop-free graph (Definition 4.17 and Lemmata 4.8 and 4.10), only finitely many of such replacements exist and hence, the loop has only a finite number of iterations. □

Now we consider *completeness* of IGOR2's search.

If IGOR2 initially inserted *all* initial nodes of $\mathcal{P}_{\Phi,B}$ to its set $\mathcal{N}$ of maintained nodes (possibly there are more than one initial node due to the non-uniqueness of MLGs of sets of specification rules, cp. Section 4.7.1) and in each step replaced one node by *all* direct successors, then, obviously, IGOR2 would find a closed CS, i.e., a solution, if one exists. However, IGOR2 only starts from *one* arbitrary initial node and in each step computes successor nodes according to only *one* arbitrarily selected open rule. (cp. Algorithm 4). That is, many paths of $\mathcal{P}_{\Phi,B}$ are pruned in IGOR2's search.

Only considering one initial node and, in general, only one MLG at each step, is unproblematic because different MLGs do not differ regarding whether they are closed. If one MLG is closed, then all MLGs are closed. Furthermore, each MLG leads to the same successor-rule sets.

The idea for proving that it suffices to only introduce successor CSs regarding one particular open rule instead of all open rules at each step, is that the order in which open rules are replaced is not relevant but that the crucial point is simply that all open rules are eventually replaced. That is, it does not matter with which rule we start and hence, we can indeed start with an arbitrary one and leave the remaining open rules for later steps.

However, this independence regarding the order of selecting open rules requires a further restriction with respect to the function call operators. According to Definitions 4.13 and 4.14, each function of a specification can be called. However, specifications grow along a path in the problem space due to new introduced subfunctions. That is, if an open rule is replaced at a particular arc instead of at some previous arc, then its open RHS can potentially be replaced by calls of *more* functions that were not introduced previously. Therefore, we introduce a concept that restricts the functions that can be called as replacement for an open RHS by those functions that are already introduced at the time when the open rule *itself* is introduced. Hence all functions that are introduced in later steps do not affect the set of callable functions of this rule.

Definition 4.20 (Callable functions)**.** Let $\langle P, \Phi\rangle$ be an initial node of $\mathcal{P}_{\Phi,B}$. Then $\mathit{Callable}(r, \langle P, \Phi\rangle) = \mathcal{D}_{\Phi\cup B}$ for each rule $r \in P$.

Now let $\langle P, \Psi\rangle$ be any node in $\mathcal{P}_{\Phi,B}$, $r \in P$ be an open rule, $\langle s, \psi\rangle \in \chi_B(r, \Psi)$ be some successor rule-set (and corresponding new specification), and
$\langle P', \Psi'\rangle = \langle (P \setminus \{r\}) \cup s, \Psi \cup \psi\rangle$ be the corresponding successor node in $\mathcal{P}_{\Phi,B}$.

- For all $r' \in P \cap P'$, $\mathit{Callable}(r', \langle P', \Psi'\rangle) := \mathit{Callable}(r', \langle P, \Psi\rangle)$.
- If $\langle s, \emptyset\rangle \in \chi_{\text{split}}(r, \Psi) \,\dot\cup\, \chi_{\text{smplCall}}(r, \Psi, B)$ then for each rule $r' \in s$, $\mathit{Callable}(r', \langle P', \Psi'\rangle) := \mathit{Callable}(r, \langle P, \Psi\rangle)$.
- If $\langle \{r'\} \cup P_{\text{new}}, \psi\rangle \in \chi_{\text{sub}}(r, \Phi, B) \cup \chi_{\text{call}}(r, \Phi, B)$ then
  - $\mathit{Callable}(r', \langle P', \Psi'\rangle) := \mathit{Callable}(r, \langle P, \Psi\rangle)$ and
  - for each $r'' \in P_{\text{new}}$, $\mathit{Callable}(r'', \langle P', \Psi'\rangle) := \mathit{Callable}(r, \langle P, \Psi\rangle) \cup \{\mathit{Defines}(r'')\}$.

If χ_{smplCall} and χ_{call} are restricted to $\mathit{Callable}(r, \langle P, \Psi \rangle)$, then the order of replacing open rules does not affect the resulting CS.

Lemma 4.11. *For each open rule r and each node N let χ_{smplCall} and χ_{call} be restricted to only introduce calls to functions from $\mathit{Callable}(r, N)$.*

Now let N be an open node in $\mathcal{P}_{\Phi,B}$ and r be any open rule in N. Let $C \in \Xi^*_B(N)$ be a closed node in $\mathcal{P}_{\Phi,B}$ that is reachable from N. Then there exists a direct successor $N' \in \Xi_B(N)$ of N according to rule r such that $C \in \Xi^*_B(N')$.

Proof. Let $N = \langle P_0, \Psi_0 \rangle, \dots, C = \langle P_n, \Psi_n \rangle$ be a path from N to the closed node C and let, for $i = 0, \dots, n-1$, $\langle r_i, s_i \rangle$ be the selected open rule of P_i and the successor rule set leading to the arc $(\langle P_i, \Psi_i \rangle, \langle P_{i+1}, \Psi_{i+1} \rangle)$.

For one $k \in \{0, \dots, n-1\}$, $r = r_k$. We have

$$\begin{aligned} P_{k+1} &= (((((((P_0 \setminus \{r_0\}) \cup s_0) \setminus \{r_1\}) \cup s_1) \setminus \cdots \setminus \{r_{k-1}\}) \cup s_{k-1}) \setminus \{r_k\}) \cup s_k \\ &= ((((((P_0 \setminus \{r_k\}) \setminus \{r_0\}) \cup s_0) \setminus \{r_1\}) \cup s_1) \setminus \cdots \setminus \{r_{k-1}\}) \cup s_{k-1} \cup s_k \\ &= (((((((P_0 \setminus \{r_k\}) \cup s_k) \setminus \{r_0\}) \cup s_0) \setminus \{r_1\}) \cup s_1) \setminus \cdots \setminus \{r_{k-1}\}) \cup s_{k-1} \end{aligned} \tag{4.1}$$

Furthermore, if χ_{smplCall} and χ_{call} can only introduce calls of functions in $\mathit{Callable}(r_k, \langle P_k, \Psi_k \rangle)$, then $\chi_{\text{call}}(r_k, \Psi_k, B) = \chi_{\text{call}}(r_k, \Psi_0, B)$, hence $\langle s_k, \phi_k \rangle \in \chi_B(r, \Psi_0)$. □

Theorem 4.2 (Completeness of IGOR2's search with respect to $\mathcal{P}_{\Phi,B}$)**.** *If, given a candidate CS P, a corresponding specification Ψ, and an open rule $r \in P$, χ_{smplCall} and χ_{call} only may introduce calls of functions $f' \in \mathit{Callable}(r, \langle P, \Psi \rangle)$, then, if there is a closed CS in $\mathcal{P}_{\Phi,B}$, IGOR2 returns a closed CS.*

Proof. By contraposition: If IGOR2 returns the empty set, then there is no closed CS in $\mathcal{P}_{\Phi,B}$.

In each iteration of the main loop of Algorithm 4, $\mathcal{N}$ contains a subset of nodes of $\mathcal{P}_{\Phi,B}$. IGOR2 returns the empty set if and only if $\mathcal{N}$ becomes empty. In the following we show that in each iteration of the loop, for each closed node C in $\mathcal{P}_{\Phi,B}$ there is a node N in $\mathcal{N}$, such that C (or some MLG-variant of it, that is then also closed) is reachable from N. Hence if $\mathcal{N}$ becomes empty, there cannot be a closed CS in $\mathcal{P}_{\Phi,B}$.

We show this by induction over the number n of iterations of the loop:

Induction base ($n = 1$): In the first iteration, $\mathcal{N}$ contains an initial node of $\mathcal{P}_{\Phi,B}$. Since all nodes in $\mathcal{P}_{\Phi,B}$ (or at least an MLG-variant of each node) is reachable from it follows that a closed node is reachable if one exists.

Induction step ($n \to n+1$): Let, in iteration n, $\mathcal{N}$ contain a subset of nodes of $\mathcal{P}_{\Phi,B}$ such that for each closed node C in $\mathcal{P}_{\Phi,B}$ there is a node N in $\mathcal{N}$, such that C (or an MLG-variant) is reachable from N (induction assumption).

Let now N be such a node.

If a node $N' \neq N$ is replaced in iteration N, then $N \in \mathcal{N}$ also in iteration $n+1$.

Now assume N itself is replaced. Then clearly, there is a successor N' of N such that C (or an MLG-variant) is reachable from N'. Now let r be any open rule in N. With

Lemma 4.11 follows that C (or an MLG-variant) is reachable from some successor of N according to rule r. Hence each closed node that is reachable from N is also reachable from some successor node of N' of N according to any selected open rule r in N, that is included in $\mathcal{N}$ in iteration $n+1$.

□

4.8.3. Soundness of Igor2

In this section we show that IGOR2 is sound with respect to the induction problem stated in Definition 4.3, i.e., that each generated closed candidate CS P satisfies all three conditions—orthogonality, disjoint defined functions with respect to the background CS, and correctness with respect to the specification—stated in Definition 4.3.

Orthogonality and disjoint functions with respect to the background CS of all generated candidate CSs, hence, in particular of all closed CSs, have already been shown in Lemmata 4.7 and 4.6, respectively.

In the lemmata for the single synthesis operators (Lemmata 4.2, 4.3, 4.4, 4.5) we already showed that, given a candidate CS with corresponding specification $\langle P, \Phi\rangle$ and a background CS B, all generated successor rules are extensionally correct with respect to $\Phi \cup \phi_{\text{new}} \cup B$, where ϕ_{new} denotes the new specification for possibly new introduced subfunctions.

Furthermore we already showed in Lemma 4.1 that initial candidate CSs are extensionally correct with respect to their specifications.

Based on these results, the following lemma shows that generated candidate CSs are (together with the provided background CS) extensionally correct with respect to their corresponding specifications (and the background CS).

Lemma 4.12 (Extensional correctness of candidate CSs)**.** *Let Φ and B be an initial specification and a background CS, respectively. Let $\langle P, \Psi\rangle$ be any node in $\mathcal{P}_{\Phi,B}$. Then $P \cup B$ is extensionally correct with respect to $\Psi \cup B$.*

Proof. By induction over the length n of a path from an initial node I in $\mathcal{P}_{\Phi,B}$ to the node $\langle P, \Psi\rangle$.

Induction base ($n = 0$): The node $\langle P, \Psi\rangle$ is an initial node in this case, hence P is the initial CS of $\Psi = \Phi$. By Lemma 4.1, P is extensionally correct with respect to Ψ.

With $\mathcal{D}_P \cap \mathcal{D}_B = \emptyset$ and Corollaries 4.2 and 4.3 follows that $P \cup B$ is extensionally correct with respect to $\Psi \cup B$.

Induction step ($n \to n+1$): Let $\langle P, \Psi\rangle \in \Xi^n_B(I)$ be a node reachable by a path of length n from an initial node. By induction assumption, $P \cup B$ is extensionally correct with respect to $\Psi \cup B$.

Let $\langle P', \Psi'\rangle \in \Xi_B(\langle P, \Psi\rangle)$, hence reachable by a path of length $n+1$, and let $r \in P$ be the corresponding selected open rule of P and $\langle s, \phi_{\text{new}}\rangle \in \chi_B(r, \Psi)$ be the corresponding successor candidate-rule set and the new specification that lead to $\langle P', \Psi'\rangle$.

We now separately prove for each case—$\langle s, \phi_{\text{new}}\rangle \in \chi_{\text{split}}$, $\langle s, \phi_{\text{new}}\rangle \in \chi_{\text{sub}}$, $\langle s, \phi_{\text{new}}\rangle \in \chi_{\text{smplCall}}$, $\langle s, \phi_{\text{new}}\rangle \in \chi_{\text{call}}$—(i) extensional correctness of the single rules of P' with respect to $\Psi' \cup B$ and (ii) completeness of P with respect to Ψ'. Extensional correctness of $P \cup B$ with respect to $\Psi \cup B$ then immediately follows from (i) and (ii) and $\mathcal{D}_P \cap \mathcal{D}_B = \emptyset$.

Extensionally correct rules: Assume $\langle s, \phi_{\text{new}} \rangle \in \chi_{\text{split}}(r, \Psi)$. Due to Lemma 4.2, all rules in s are extensionally correct with respect to $\Psi = \Psi'$, hence also with respect to $\Psi' \cup B$ (since $\mathcal{D}_{\Psi'} \cap \mathcal{D}_B = \emptyset$). By induction assumption and since $(P' \setminus s) \subseteq P$, all rules in $P' \setminus s$ are extensionally correct with respect to to $\Psi' \cup B$. Hence all rules in P' are extensionally correct with respect to to $\Psi' \cup B$.

If $\langle s, \phi_{\text{new}} \rangle \in \chi_{\text{smplCall}}(r, \Psi, B)$, we have $\Psi = \Psi'$. Therefore, the rules in $P \cap P' = P' \setminus s$, are extensionally correct with respect to $\Psi' \cup B$ by induction assumption. Furthermore, we have $|\, s \,| = 1$ and this only successor rule is extensionally correct with respect to $\Psi' \cup B$ by Lemma 4.4.

If $\langle s, \phi_{\text{new}} \rangle \in \chi_{\text{sub}}(r, \Psi, B)$ or $\langle s, \phi_{\text{new}} \rangle \in \chi_{\text{call}}(r, \Psi, B)$, we have $\Psi' = \Psi \cup \phi_{\text{new}}$. The function symbols of ϕ_{new} and P are disjoint. Therefore, all rules in $P \cap P'$ are extensionally correct with respect to $\Psi' \cup B$ by induction assumption. The rules s are extensionally correct with respect to $\Psi' \cup B$ by Lemmata 4.3 and 4.5.

Completeness: In the case of χ_{split}, each LHS in $\Psi(r) = \Psi'(r)$ matches a LHS in s and each LHS in $\Psi' \setminus \Psi'(r)$ matches a LHS in $P \setminus \{r\}$, hence also in $P' \setminus s$. Hence each LHSs in Ψ' matches a LHS in P'.

For χ_{sub}, χ_{smplCall}, and χ_{call} holds that all LHSs of P are also in P', hence P' is complete with respect to Ψ. In the case of χ_{smplCall}, $\Psi = \Psi'$, hence P' is complete with respect to Ψ'. For χ_{sub} and χ_{call}, the rules P_{new} are, by Lemma 4.1, complete with respect to ϕ_{new}. Hence P' is complete with respect to Ψ'.

□

From the result that candidate CSs are extensionally correct and the fact that χ_{call} and χ_{smplCall} reduce arguments of function calls with respect to a reduction order, we can conclude that candidate CSs, together with the provided background CS, are correct according to Definition 4.2 with respect to their specifications.

Theorem 4.3 (Correctness of candidate CSs). *Let Φ and B be an initial specification and a background CS, respectively. Let $\langle P, \Psi \rangle$ be any node in $\mathcal{P}_{\Phi,B}$. Then $P \cup B$ is correct with respect to Ψ according to Definition 4.2.*

Proof. By contradiction: Suppose $P \cup B$ is *not* correct with respect to Ψ. Then there is a rule $(f_0(\boldsymbol{i}_0) \to o_0) \in \Psi$ such that—due to extensional correctness of $P \cup B$ with respect to $\Psi \cup B$—$f_0(\boldsymbol{i}_0) \to_P t_0$ and $t_0 \overset{*}{\to}_{\Psi \cup B} o_0$, but—because $P \cup B$ is not correct—$t_0 \overset{*}{\not\to}_{P \cup B} o_0$.

Now let s_1 be the result of rewriting t_0 by $P \cup B$, i.e., $t_0 \overset{*}{\to}_{P \cup B} s_1$, *where only those redexes are reduced that are* correctly normalized *by $P \cup B$ (i.e., which would be reduced by $\Psi \cup B$ to the same, in $\Psi \cup B$ specified, normal form)*, until there is no further redex which is correctly normalized by $P \cup B$. We then have $s_1 \neq o_0$, because otherwise, we would have $t_0 \overset{*}{\to}_{P \cup B} o_0$, contradicting the assumption that $P \cup B$ is not correct for t_0. In particular, s_1 cannot be in normal form because t_0 was rewritten in accordance with $\Psi \cup B$ and $\Psi \cup B$ is confluent, such that, if s_1 were in normal form, we would have $s_1 = o_0$. But, again since t_0 was rewritten in accordance with $\Psi \cup B$ and $t_0 \overset{*}{\to}_{\Psi \cup B} o_0$, we also have $s_1 \overset{*}{\to}_{\Psi \cup B} o_0$.

It follows that there is a rule $(f_1(\boldsymbol{i}_1) \to o_1) \in \Psi$ (possibly $f_1 = f_0$) such that $s_1 = C[f_1(\boldsymbol{i}_1)\sigma_1]$ for some context C and substitution σ_1. Since $P \cup B$ is extensionally correct, $f_1(\boldsymbol{i}_1)\sigma_1 \to_P t_1$ and $t_1 \overset{*}{\to}_{\Psi\cup B} o_1\sigma_1$, but since $P \cup B$ is not correct, and in particular since s_1 does not contain any redex that is correctly normalized by $P \cup B$, $t_1 \overset{*}{\not\to}_{P\cup B} o_1\sigma_1$. Now—analogously to t_0 and s_1—let s_2 be the result of rewriting t_1 by $P \cup B$, i.e., $t_1 \overset{*}{\to}_{P\cup B} s_2$, where only those redexes are reduced that are *correctly* normalized by $P \cup B$, until there is no further redex which is correctly normalized by $P \cup B$.

For the same argument as for t_0, s_1 above, we then have a rule $(f_2(\boldsymbol{i}_2) \to o_2) \in \Psi$, $s_2 = C[f_2(\boldsymbol{i}_2)\sigma_2]$ for some (other) context C and a substitution σ_2, $f_2(\boldsymbol{i}_2)\sigma_2 \to_P t_2$, $t_2 \overset{*}{\to}_{\Psi\cup B} o_2\sigma_2$, and $t_2 \overset{*}{\not\to}_{P\cup B} o_2\sigma_2$.

We again rewrite, analogously to t_0 and t_1, t_2 to a term s_3 which must contain a redex $f_3(\boldsymbol{i}_3)\sigma_3$ and so forth—achieving an infinite reduction and sequence of such triples $\langle t_0, s_1, f_1(\boldsymbol{i}_1)\sigma_1\rangle, \langle t_1, s_2, f_2(\boldsymbol{i}_2)\sigma_2\rangle, \langle t_2, s_3, f_3(\boldsymbol{i}_3)\sigma_3\rangle, \ldots$. Since for all $i = 0, 1, \ldots$, $f_i \in \mathcal{D}_\Psi$, $f_i(\boldsymbol{i}_i)\sigma_i$ is a redex according to $P \cup B$, and $\mathcal{D}_\Psi$ and $\mathcal{D}_B$ are disjoint, we have $f_i \in \mathcal{D}_P$ (and $f_i \notin \mathcal{D}_B$).

We now show that such an infinite reduction by P is not possible. Let $p_1, p_2, \ldots$ be positions denoting the subterms $f_1(\boldsymbol{i}_1)\sigma_1, f_2(\boldsymbol{i}_2)\sigma_2, \ldots$ in the results $s_1, s_2, \ldots$ of rewriting the terms $t_0, t_1, \ldots$, respectively; i.e., $s_i|_{p_i} = f_i(\boldsymbol{i}_i)\sigma_i$ for all $i = 1, 2, \ldots$. Reduction steps in the reductions $t_{i-1} \overset{*}{\to}_{P\cup B} s_i$ can only occur at positions parallel to or below the p_i, i.e., at positions p with $p \parallel p_i$ or $p > p_i$, yet not at positions above the p_i ($p < p_i$). This follows from the fact that—by assumption—all reduced redexes are *correctly* reduced, hence to their correct *constructor terms* as defined by Ψ. Hence all subterms of the s_i at positions below any rewritten position are constructor terms.

It follows that $\mathit{Node}(t_0, p_1) = f_1, \mathit{Node}(t_1, p_2) = f_2$ etc., hence each f_{i+1} is already introduced by the rewrite step $f_i(\boldsymbol{i}_i) \to_P t_i$.

The synthesis operators that generate rules which have such function calls in their RHSs are χ_{sub}, χ_{smplCall}, and χ_{call}. In the case of χ_{sub}, such a rule has the form $f(\boldsymbol{p}) \to c(s_1, \ldots, s_n)$ where the s_n are either constructor terms or function calls of the form $g(\boldsymbol{p})$ ($f \neq g$). Hence if a reduction step $f_i(\boldsymbol{i}_i)\sigma_i \to_P t_i$ is based on such a rule, we have $\boldsymbol{i}_i\sigma_i = \boldsymbol{i}_{i+1}\sigma_{i+1}$.

In the case of χ_{smplCall}, synthesized rules have the form $f(\boldsymbol{p}) \to f'(\boldsymbol{p'})$, where $f(\boldsymbol{p})\sigma > f'(\boldsymbol{p'})\sigma$, if, as it is the case here, $f_i \in \mathcal{D}_P$, for each $f(\boldsymbol{p})\sigma \in \mathit{Lhss}(\Psi)$. Hence in this case we have $f_i(\boldsymbol{i}_i)\sigma_i > f_{i+1}(\boldsymbol{i}_{i+1})\sigma_{i+1}$.

Finally, in the case of χ_{call}, the synthesized rules containing function calls have the form $f(p_1, \ldots, p_n) \to f'(g_1(\boldsymbol{p}), \ldots, g_n(\boldsymbol{p}))$. The $g_i(\boldsymbol{p})\sigma_i$ are correctly reduced to constructor terms by assumption. Again, since $f_i \in \mathcal{D}_P$ and due to the definition of χ_{call}, we then have $f_i(\boldsymbol{i}_i)\sigma_i > f_{i+1}(\boldsymbol{i}_{i+1})\sigma_{i+1}$.

Since $>$ is a well-founded order, we can only have finitely many k with $\boldsymbol{i}_k > \boldsymbol{i}_{k+1}$.

The sequences *between* the finitely many indices k representing rewrite steps according to rules introduced by χ_{smplCall} or χ_{call}, are all, as we have seen, caused by rules introduced by χ_{sub}. Yet such rules do not lead to recursion, because the function symbols g in RHSs introduced by χ_{sub} are always *new* function symbols not occurring in the CS up to the invention by χ_{sub}. Since $\mathcal{D}_P$ is finite, we thus can also have only finitely many

k with $\boldsymbol{i}_k = \boldsymbol{i}_{k+1}$,i.e., only finitely many indices k representing rewrite steps according to rules introduced by χ_{sub}. Hence the complete constructed reduction must be finite which contradicts the result that it must be infinite if $P \cup B$ is *not* correct with respect to Ψ, hence $P \cup B$ is correct with respect to Ψ. □

Now that closed candidate CSs are solutions according to the induction problem (Definition 4.3) follows as a corollary.

Corollary 4.7 (Soundness of IGOR2 with respect to the induction problem)**.** *Let $\langle P, \Psi \rangle$ be a closed node in $\mathcal{P}_{\Phi,B}$. Then P is a solution CS according to Definition 4.3.*

Proof. Conjunction of Lemmata 4.7 and 4.6 and Theorem 4.3. (Note that from that $P \cup B$ is correct with respect to Ψ, as proven in Theorem 4.3, immediately follows that $P \cup B$ is correct with respect to Φ, as required by Definition 4.3, because for all specifications Ψ occurring in the nodes of $\mathcal{P}_{\Phi,B}$, $\Phi \subseteq \Psi$ because the synthesis operators only add new specification rules but never delete rules from a specification.) □

4.8.4. Concerning Completeness with Respect to Certain Function Classes

In Section 4.8.2 we showed that the search strategy of IGOR2 is complete with respect to its own program class, i.e., that if there exists a solution CS that can be reached from an initial candidate by a finite sequence of applications of the synthesis operators, then IGOR2 will find a solution CS.

Another interesting question is, which classes of computable functions are generatable at all by means of the described synthesis operators, i.e., which classes of computable functions are representable by the CSs in IGOR2's problem space $\mathcal{P}_{\Phi,B}$.

We have not yet substantially studied this question. However, we can give some first insights.

IGOR2's problem space varies with respect to the provided initial specification Φ and background CS B. It is "restricted" (i.e., excludes certain computable functions) in that it only includes functions that have the I/O examples specified in Φ as a subset of their graphs. Yet this is not a defect, but a feature compared to generate-and-test methods, where the problem space is independent from the specification and hence many functions that do *not* compute the specification correctly are enumerated until a function that passes the test is found. In particular, this "restriction" does not *in principle* rule out any computable function, because, certainly, for any computable function we may provide some subset of its graph as specification Φ.

The second parameter to bias $\mathcal{P}_{\Phi,B}$ is the background knowledge B. We consider two extremes:

1. The *target function itself* is already given by B.

2. *No* background knowledge is given; $B = \emptyset$.

Due to the possibility of the really trivial first case, one can say that IGOR2 can synthesize each computable function—provided it is already appropriately given as background knowledge. Specification Φ and background CS B are equal in this case (or B might also be a superset of Φ), except for that the (same) function specified in Φ and B has different names, say f and f', respectively, in both sets (because of the requirement $\mathcal{D}_\Phi \neq \mathcal{D}_B$). In this case, the initial candidate consists of one rule, the MLG of the I/O examples for f in Φ. In the first iteration of the search loop, χ_{smplCall} then will succeed and return the rule

```
f (x) → f' (x) ,
```

just picking the target function from the background CS.

Of course, IGOR2 has not really synthesized an (unknown) algorithm for a computable function in this case because it is presupposed, that the function *is* already implemented.

This trivial case is, however, good enough to see that it might strongly depend on the provided background knowledge whether some function is generatable or not.

Now the second extreme case is that *no* background knowledge is given, i.e., that the only (initial) primitives to be used to synthesize a program are the constructors of the data type; e.g., 0 (zero) and S_ (successor) in the case of the natural numbers $\mathbb{N}$.

Can IGOR2 in this case synthesize each computable function $f : \mathbb{N} \to \mathbb{N}$? Or each primitive recursive function? Or some other class? The question of exactly which class of functions is generatable without background knowledge is open. What we know is that (at least one) functions that are *not* primitive recursive can be synthesized, e.g., the Ackermann function (see Chapter 5). Further we know that particular functions that *are* primitive recursive can *not* be synthesized (without background knowledge), e.g., multiplication of two natural numbers.

Consider the Ackermann function (as CS with constructors 0 and S_, meaning +1):

```
1  A (0, n)      → S n
2  A (S m, 0)    → A (m, S 0)
3  A (S m, S n)  → A (m, A (S m, n))
```

From the initial candidate,

```
A (x, y) → S z
```

the above definition can be synthesized by two applications of χ_{split} to get the splitting into three rules. The first equation/rule is then simply the MLG of its specification subset. The second rule is finished by one call of χ_{smplCall}, introducing the recursive call with constructor term arguments m and S 0. Finally the third rule is synthesized by first applying χ_{call} to get the recursive call at the root. The two arguments m and the inner recursive call are at first introduced as new subfunctions:

```
A (S m, S n) → A (sub1 (S m, S n), sub2 (S m, S n)) .
```

Their initial rules are:

```
sub1 (S m, S n) → m
sub2 (S m, S n) → S z
```

The initial rule for sub1 is already closed (because the actual argument computed by it is only the constructor term m). The initial rule for sub2 is open. It can be finished by a further application of χ_{smplCall}, introducing the (recursive) call of the Ackermann function with constructor terms as arguments:

```
sub2 (S m, S n) → A (S m, n) .
```

Now consider multiplication M of two natural numbers (where A denotes addition):

```
M (0, n)   → 0
M (S m, n) → A (M (m, n), n)
A (0, n)   → n
A (S m, n) → A (m, S n)
```

If A was given as background knowledge, then M could be induced. Also if both M and A are specified they could be synthesized in parallel, leading to the above definition. Yet if A is specified neither as target- nor as background function, then it must automatically be invented as subfunction. As we have seen, IGOR2 can automatically invent new subfunctions (by χ_{sub} and χ_{call}), e.g., if one specifies the reverse function on lists, then IGOR2 automatically invents the init and last functions. However, the scheme of how calls of target, i.e., specified functions and new subfunctions are nested (or composed) is syntactically restricted due to the definitions of χ_{sub} and χ_{call}.

Consider again the induced *recursive* rule for reverse of Listing 4.4.

```
reverse (x : xs) → last (x : xs) : reverse ( init (x : xs))
```

Now compare the second (recursive) rule for M above and this (recursive) rule for reverse. In the recursive rule for M, M is called within (as an argument) of the call of A, i.e., the non-specified subfunction to be invented. In contrast, in the recursive rule for reverse, reverse is *not* called within the call of a subfunction, but, the other way round, init (an unspecified subfunction) is called within the recursive call of reverse.

The *first* form, calls of *non*-specified subfunctions within (recursive) calls of target functions, cannot be synthesized. Such subfunctions *must* rather be specified somehow, either as background knowledge or as further target function.

To see why this form cannot be synthesized, recall the definitions of χ_{sub} and χ_{smplCall} (the only two operators that introduce new subfunctions, like A in the case of M, if A is not specified somehow). The operator χ_{sub} only deals with proper *subterms*, i.e., only introduces calls to new functions as arguments of a constructor at the root of a RHS. Yet A, the subfunction to be introduced, is called at the root of the respective RHS. The operator χ_{call}, in fact, introduces function calls at the root of RHSs, but only *already known* functions. The *new* subfunctions introduced by χ_{call} are those functions that compute the *arguments* of this call. Hence the form of the recursive call in the case of reverse, reverse (init (x : xs)), can be synthesized: reverse is already known, init is the new subfunction that computes the argument. Yet the call of A in the case of M cannot be synthesized, because A is *not* known when its call must be introduced, hence it cannot be called at the root of some (composed or nested) function call introduced by χ_{call}.

4.8.5. Concerning Complexity of Igor2

Igor2 conducts a uniform cost search. At each search step, one node of $\mathcal{P}_{\Phi,B}$ is expanded yielding several successor nodes. If we assume a maximal branching factor b, a solution a depth d, and that computing one successor consumes a constant amount of time, then the time complexity is thus $O(b^d)$, i.e., exponential with respect to the depth of the solution in the search tree.

This general exponential complexity is inherent to (complete, non-greedy) *search*. The general goal must be to reduce both b and d by (i) an appropriate selection/restriction of the problem space (implicitly done by the search operators and their application preconditions) and (ii) an appropriate selection of heuristics to guide the search in it. In the case of Igor2, this is mostly subject to future research.

At the moment, unfortunately, b itself is in the worst case exponential, with respect to the size of the specification. The problematic operator is χ_{call}. To see this, recall the definition of χ_{call} (Definition 4.14) and let r, $\Phi(r)$, f, f', and $(\Phi \cup B)(f')$ be the selected open rule, its associated specification subset, the function that r defines, the function that is to be called, and the set of specification (or background) rules of f', respectively. Assume further that $\mathit{Var}(l) = \mathit{Var}(r)$ for each specification rule $l \to r$ of f' (such that condition 1 of Definition 4.14 is always satisfied for f') and that f' is a background function, actually (such that condition 3 does not apply). Then the only remaining restricting condition for the mapping $\mu : \Phi(r) \to B(f)$ is condition 2; $o = o'\tau$ for RHSs o, o' of f and f', respectively.

Now assume further that all RHSs of f' are single variables. For example, let f' be the `last` function for lists, specified by:

```
f' (x : [])       → x
f' (x : y : []) → y
...
```

Then, indeed, condition 2 is satisfied for *arbitrary* combinations of o, o'. That is, μ can be *any* mapping of $\Phi(r) \to B(f')$. The operator χ_{call}, at the moment, generates one successor CS for *each* possible mapping μ; hence, in this worst case, the number of successors equals the number of (total) mappings from $\Phi(r) \to B(f')$. This number is: $|\, B(f') \,|^{\,|\Phi(r)|}$. Hence, in the worst case, χ_{call} yields exponentially many successor candidates with respect to the number of specification rules (associated with rule r).

Recall that especially the function-call operators χ_{smplCall} and χ_{call} are inspired by the classical analytical approach to the inductive synthesis of functional programs (as described in Section 3.2) in that they synthesize (recursive) function calls based on detecting which specified outputs of the function to be induced are part of which (other) outputs of the same or another function. Yet in the described worst case, when the outputs of the function to be called, consist of single variables such that they subsume (or contain) *arbitrary* other outputs of the same type, then this generalized analytical approach degenerates to an ordinary enumeration of all programs.

4.9. Extensions

In this section we sketch three extensions to the basic IGOR2 algorithm.

The first extension is a further synthesis operator that generates—as a second form of conditional evaluation amongst pattern matching—*conditional* rules. Conditional rules are only rewritten if a specified condition—a term evaluating to a boolean value—is satisfied. A typical example for such conditions is the test of equality regarding two pattern variables. For example, a function member, testing whether a particular element is member of a given list, could be defined by conditional rules as follows:

```
member (x, [])     →  false
member (x, y : ys) →  true              if x = y
member (x, y : ys) →  member (x, ys)    if x ≠ y
```

The second and third rule are conditional rules, the conditions (test for equality) are the terms introduced by the if keyword. Section 4.9.1 shortly describes the synthesis of conditional rules.

In contrast to conditional rules, which extend the program class, the second extension aspires to increase the efficiency of the induction by replacing the rule-splitting operator by a certain variant. The idea is to combine all possible splittings—if more than one pivot position and hence more than one splitting exists—to one single splitting and hence to achieve a certain grade of splitting, that would require *several* applications of the original splitting operator, in *one single* step by the new one. We call this variant *rapid rule-splitting* and describe it in Section 4.9.2.

The third, most fundamental, extension introduces *existentially*-quantified variables into RHSs of specification rules (Section 4.9.3). This allows to leave certain parts of an output unspecified. In the extreme case, where the complete RHS is a single existentially quantified variable, the specification rule just says that the specified input is member in the domain of the function but leaves the output completely unspecified. Hence, this can also be seen as an attempt to integrate specifications, that are *not* complete up to a certain complexity, into the analytical approach to inductive program synthesis.

4.9.1. Conditional Rules

The only form of conditional evaluation in ordinary term rewriting systems is *pattern matching*—a rule is applicable if its LHS matches with some subterm of the subject term. This form does not allow for direct testing of whether, e.g., the value that matches one pattern variable is *greater* than the value matching another pattern variable. If we would allow for non-linear LHSs we could test for equality by pattern matching; yet if we require LHSs to be linear, then we cannot even test for equality directly.

Nevertheless, we *can* realize these conditions by means of pattern matching. We can define predicates such as equality or greater/smaller by ordinary rules and we can define an if−then−else−fi rule by

```
if (true, x, y)  → x
if (false, x, y) → y
```

Let equal be a defined function yielding true if two terms are identical and false otherwise. Then a definition of the member function from above in terms of ordinary pattern matching rules would read:

```
member (x, [])       →  false
member (x, y : ys)  →  if (equal (x, y), true, member (x, ys))
```

However, in order to make Igor2 able to generate functions like member, we decided to enable Igor2 to generate conditional rules as informally introduced above. This is done by adding a further synthesis operator χ_{pred}. The operator χ_{pred} is a further rule-splitting operator. In contrast to the operator χ_{split}, it does not split a rule by generating several non-unifying specializations of its pattern but rather by generating two complementary conditions/predicates depending on the pattern variables.

The conditions, however, are not analytically "synthesized" from the specification but simply enumerated until now. In addition to the specification of the target function(s) and background functions, the user may provide predicates, i.e., boolean-valued functions. Now let some open rule r be selected and p be a user-provided predicate such that for each formal parameter of the predicate there is a variable in the pattern of r with the same type (or sort). Then p applied to these pattern variables is one possible condition and its negation is the complementary condition. The selected rule is cloned and each version is endowed with one of the two complementary conditions. Finally the specification subset associated with the selected open rule r is partitioned into two subsets accordingly.

In addition to user-defined predicates, syntactical equality of terms is integrated as a standard predicate.

4.9.2. Rapid Rule-Splitting

Suppose the LHS of a selected open rule r contains more then one pivot position. Then χ_{split} introduces one partition of the associated specification subset and a corresponding set of successor rules for each pivot position.

Another possibility is to generate *one single* partition and a corresponding set of successor rules based on *all* pivot positions. In this case, one subset of the partition is not determined by one of the constructors at the chosen pivot position in the specification rules, but rather by one *combination* of constructors, one for each pivot position.

For example, if we have two pivot positions p_1, p_2 denoting constructors c_1, c'_1 and c_2, c'_2, respectively, in the specification rules, then standardly we would get two partitions with two subsets each. One partition according to p_1 with subsets according to c_1 and c_2 and analogously a further partition for p_2. With rapid rule-splitting, we get *one* partition with *four* subsets in this case: One subset for the combination c_1, c_2, one for c_1, c'_2, one for c'_1, c_2 and one for c'_1, c'_2.

4.9.3. Existentially-Quantified Variables in Specifications

This last extension allows for existentially-($\exists$-)quantified variables in *RHSs* of specification rules. In contrast to the (implicitly) $\forall$-quantified variables, that represent the set of *all* its instances, an ∃-quantified variable represents *one arbitrary* instance such that one has the possibility to leave parts of an output “unspecified”.

This can be useful for three reasons.

First, one possibly want the output of a specified input to just satisfy a particular form without worrying about the particular value. For example, assume a function f shall take lists and output lists of the same length where the last elements of input and output lists are equal but the concrete values for the other elements of the output lists are not important. One specification rule for f could then be:

```
∃ v w.  f ([x,y,z]) = [v,w,z] .
```

Second, recall that if χ_{call} introduces a call of a function f', then the specification rules $l \to r$ of f' must satisfy the condition $\mathit{Var}(l) = \mathit{Var}(r)$ (Definition 4.14, condition 1). The reason is that if $\mathit{Var}(l) \supset \mathit{Var}(r)$, no instantiation of the variables in l not occurring in r is determined and they can in principle be instantiated arbitrarily. The (instantiated) LHSs of f' become RHSs in the abduced specifications of the new subfunctions introduced by χ_{call}. Hence existentially quantified variables in specification RHSs would be a way for χ_{call} to deal with the general case that $\mathit{Var}(l) \supseteq \mathit{Var}(r)$.

Third, quantified variables in specification RHSs would allow to let a complete specified RHS be just a single existentially quantified variable, hence to let the output for the corresponding input unspecified. Hence this would be a way to effectively integrate non-complete (up to some complexity) specifications into the analytical approach to inductive program synthesis.

However, integrating the possibility to systematically deal with existentially quantified variables in specifications into IGOR2 is not trivial.

5. Experiments

We have implemented a prototype of the IGOR2 algorithm in the interpreted, term-rewriting based specification and programming language MAUDE [18].[1] MAUDE's so-called *functional modules* are constructor term rewriting systems that are assumed to be complete, i.e., confluent and terminating. As extensions to ordinary CSs, as defined in Section 2.2.4, MAUDE's functional modules are order-sorted and binary constructors can be specified to be associative, commutative, etc. and then rewriting is performed modulo these properties.

Since MAUDE's functional modules are essentially CSs, IGOR2 specifications as well as solution programs induced by IGOR2 are valid MAUDE modules. A further reason for choosing MAUDE as implementation and specification framework for IGOR2 was that MAUDE is *reflective*, i.e., that within a MAUDE function, each MAUDE module known to the interpreter can be taken as data object and can be inspected, manipulated, and even evaluated. Hence by using this feature, the implementation of data structures to represent, manipulate, and evaluate IGOR2 specifications and generated CSs incl. sorts, terms, rules/equations etc. can be omitted.

In addition to the synthesis operators as defined in Section 4.7, the implementation contains, in a more or less experimental form, the three extensions sketched in Section 4.9: Rapid rule-splitting, conditional rules, and specifications containing $\exists$-quantified variables (experimental and incomplete).

We have tested the implemented IGOR2 system with problems from two problem domains:

1. Functional programming problems and
2. artificial intelligence / problem-solving problems.

The complete specification files of all problems are provided in the appendix. All tests have been conducted under Ubuntu Linux on an Intel Dual Core 2.33 GHz with 4GB memory.

5.1. Functional Programming Problems

This block contains several recursive functions of natural numbers, lists, and lists of lists (matrices). Most of them are well-known and can be found in standard libraries, e.g., in the HASKELL standard libraries.

[1]MAUDE's homepage can be found at `http://maude.cs.uiuc.edu/`.

Table 5.1.: Tested functions for natural numbers

Function	Type	Description
$=$	INat INat $\to$ Bool	Equality
$\leq$	INat INat $\to$ Bool	Smaller or equal than
odd	INat $\to$ Bool	Is a number odd?
even	INat $\to$ Bool	Is a number even?
$+$	INat INat $\to$ INat	Addition
$*$	INat INat $\to$ INat	Multiplication
fac	INat $\to$ INat	Factorial
fib	INat $\to$ INat	Fibonacci numbers
ack	INat INat $\to$ INat	Ackermann function

The same function can be specified in different ways: One may only use I/O examples or also I/O patterns (i.e., variables); and one may provide more or less specification rules. For the functional programming problems, we aspired most concise specifications. Hence we used variables where it was possible, e.g., we specified + by I/O patterns like

$\mathsf{S\ S\ 0} + \mathsf{x} \to \mathsf{S\ S\ x}$

(where x is a variable) instead of I/O examples like

$\mathsf{S\ S\ 0} + \mathsf{S\ S\ S\ 0} \to \mathsf{S\ S\ S\ S\ S\ 0}$,

and we provided rather few than many rules.[2]

The only used extension in this block is rapid rule-splitting. Some tests for functions that require conditional rules, such as member or insert (inserting an element into a sorted list) or sort have previously been published in [48].

5.1.1. Functions of Natural Numbers

The first module includes functions and predicates for natural numbers. Table 5.1 gives an overview of the tested functions. For the boolean values we used the BOOL module included in MAUDE's standard prelude. It defines the sort Bool and the boolean values (constant constructors) true : $\to$ Bool and false : $\to$ Bool. We defined the type of natural numbers by the sort INat (abbreviating IGOR2-Nat, to circumvent conflicts with MAUDE's internal definition of natural numbers) and the constructors 0 : $\to$ INat (zero) and S_ : INat $\to$ INat (successor).

All functions are specified by some I/O examples or I/O patterns together in one MAUDE module. The concrete specification, i.e., the selection of target function(s) to be induced and functions that may be used as background knowledge is then provided

[2]However note that IGOR2, due to its analytical recurrence-detection approach, needs a certain number of I/O examples in order to induce recursive functions. In particular, IGOR2 cannot induce recursive functions from single examples, like some enumerative approaches, but needs at least one example per base-case and two examples per recursive case.

Table 5.2.: Results of tested functions for natural numbers. The maximum depth of rules (Definition 4.19) was set to 3 and possibly reduced in case of a timeout.

		Specification			
Test	Target funs	Size	Style	BK	Time (sec)
1	$=$	9	examples		7.052
2	$\leq$	6	patterns		0.032
3	odd, even	$4+4$	examples		0.020[a]
4	$+$	3	patterns		0.008
5	$*$	6	mixed		$0.088^{\perp}$[b]
6	$*$	6	mixed	$+$	$\oslash$
7	fac	4	examples		$0.060^{\perp}$[b]
8	fac	4	examples	$*$	$\oslash$, 11.192[c]
9	fib	6	examples		$1.212^{\perp}$[b]
10	fib	6	examples	$+$	$\oslash$
11	ack	17	examples		$\oslash$, 2.904[d]

[a]mutual recursive solution (Listing 4.1)
[b]intended function not included in program space; see Section 4.8.4 for a discussion
[c]with max depth 2
[d]with rapid rule-splitting (see Section 4.9.2)

by two parameters for each IGOR2 run, listing the names of functions to be induced and background functions, respectively.

Table 5.2 summarizes the test-setup and results. Each row in the table represents one tested problem. The columns, from left to right, show the number of the tested problem, the name(s) of the target function(s)[3], the number of provided specification rules, the style of the specification rules (I/O examples, i.e., ground rules, or I/O patterns, i.e., non-ground rules, or mixed, i.e., both ground and non-ground rules), provided background functions, and the induction time in seconds. A $\perp$ adhering to the induction time indicates a wrong solution in the sense that the induced function definition does *not* compute the *intended* function (though of course, the induced function correctly computes the specification). This only happened if the intended function was not included in the program space at all. For example, $*$ cannot be induced by the synthesis operators of IGOR2 if $+$ is not provided as background knowledge. See Section 4.8.4 for a consideration of classes of functions that can and cannot be synthesized by IGOR2.

Finally, a $\oslash$ instead of an induction time indicates a "timeout"—we aborted an IGOR2 execution if it did not come up with a solution within three minutes.

Maximum depth of rules (cp. Definition 4.19) was set to 3. In case of a timeout, we reduced the maximum depth to 2 to further restrict the number of applications of the function call operator χ_{call} and/or switched on rapid rule-splitting (see Section 4.9.2).

[3]Two names mean that these functions were induced in parallel.

Listing 5.1: I/O examples for the Ackermann function

```
ack(0, 0)             → S 0
ack(0, S 0)           → S S 0
ack(0, S S 0)         → S S S 0
ack(0, S S S 0)       → S S S S 0
ack(0, S S S S 0)     → S S S S S 0
ack(0, S S S S S 0)   → S S S S S S 0
ack(0, S S S S S S 0) → S S S S S S S 0
ack(S 0, 0)           → S S 0
ack(S 0, S 0)         → S S S 0
ack(S 0, S S 0)       → S S S S 0
ack(S 0, S S S 0)     → S S S S S 0
ack(S 0, S S S S 0)   → S S S S S S 0
ack(S 0, S S S S S 0) → S S S S S S S 0
ack(S S 0, 0)         → S S S 0
ack(S S 0, S 0)       → S S S S S 0
ack(S S 0, S S 0)     → S S S S S S S 0
ack(S S S 0, 0)       → S S S S S 0
```

Listing 5.2: Induced definition of the Ackermann function

```
ack (0, 0)            → S 0
ack (0, S x0)         → S S x0
ack (S x0, 0)         → ack (x0, S 0)
ack (S x0, S x1)      → ack (sub9 (S x0, S x1), sub10(S x0, S x1))
sub9 (S x0, S x1)     → x0
sub10 (S x0, S x1)    → ack (S x0, x1)
```

Listing 5.1 shows exemplarily the I/O examples that were used to specify the Ackermann function. Listing 5.2 shows the induced CS (prettyprinted by hand).[4] It is easy to verify that the induced definition for the Ackermann function is equivalent to the standard definition:

$$\begin{aligned} A(0, n) &= n + 1 \\ A(m + 1, 0) &= A(m, 1) \\ A(m + 1, n + 1) &= A(m, A(m + 1, n)) \end{aligned}$$

[4]In Section 4.8.4, the sequence of synthesis-operator applications to achieve the induced CS is shortly described. However note that we used rapid rule-splitting (Section 4.9.2) here such that all four patterns for ack of the induced solution were synthesized by only *one* application of χ_{split} instead of the two χ_{split} applications mentioned in Section 4.8.4.

5.1.2. List Functions

The second module includes general list functions, such as reverse, and functions on lists of natural numbers, such as sum. Table 5.3 gives an overview of the tested functions. Natural numbers are defined as described in the previous section. We defined the (general, parameterized) list type by the sorts List{X} and X$Elt, where X$Elt is the sort for the single elements of a list. The X is a formal parameter that may be instantiated by any concrete sort, e.g., by INat, to get lists of natural numbers. The list constructors are [] : → List{X} (empty list) and _:_ : X$Elt List{X} −> List{X} ("cons"ing a first element and a rest list) where we use mixfix notation for _:_. Furthermore we added a (mixfix-) constructor <_,_> : X$Elt X$Elt −> X$Elt in order to also have pairs of elements as list elements (e.g., for the zip function). Finally we have a further sort ListPair{X} that represents pairs of lists (e.g., for the functions unzip or split).

Table 5.4 summarizes the test-setup and results. The structure is the same as for the result table for the functions on natural numbers, except for that we added one column to list automatically invented subfunctions.[5] The columns from left to right denote the number of the tested problem, name(s) of target function(s), number(s) of provided specification rules, style of specification, provided background functions, invented functions, and induction time in seconds. A $\perp$ adhering to the induction time indicates a wrong solution, a $\oslash$ instead of an induction time indicates a "timeout" (if the run-time exceeded three minutes). Again we set the maximum depth of rules to 3 and in case of a timeout, we reduced the maximum depth to 2 and/or switched on rapid rule-splitting.

The only timeout for the general list functions appeared for swap (row 19). A second run with rapid rule-splitting enabled induced a correct function definition in about seven seconds.

Specification and induced solution for reverse (row 4) are those from Example 4.1, i.e., init and last are invented and induced as help functions. If init and/or last are provided as background functions (row 5), they are not re-invented but simply called. As one can see, this reduces the induction time. We get a similar result for the function splitAt, which splits a list at a given position into two lists. Induced *without* background knowledge, the help functions take and drop are automatically invented (row 9). If take and drop are provided, they are just called and the induction time gets reduced.

In the experiment shown in row 18, *four* recursive functions overall were induced in parallel. Two of them shiftL and shiftR were specified as target functions and the remaining two, init and last were automatically introduced as help functions for shiftR. The induced solution for this experiment is shown in Listing 5.3.

As functions for lists of natural numbers we tested sum and product in several ways. First, we tested sum without any background knowledge (row 20), resulting in a correct induced solution as shown in Listing 5.4. Second (row 21), we tested sum with + as background knowledge. Interestingly, this led to a timeout, though certainly, + is an

[5]Of course, the function *names* in the respective column are not introduced by IGOR2 but identified by the author to indicate which functions IGOR2 actually invented.

Table 5.3.: Tested list functions

General lists		
Function	Type	Description
init (l)	List{X} → List{X}	List l without the last element
last (l)	List{X} −> X$Elt	Last element of l
l1 ++l2	List{X} List{X} → List{X}	Append
length (l)	List{X} → lNat	Number of elements of l
reverse (l)	List{X} → List{X}	l in reverse order
l !! n	List{X} lNat −> X$Elt	Element of l at index n
take (l, n)	List{X} lNat → List{X}	Take first n elements of l
drop (l, n)	List{X} lNat → List{X}	Drop first n elements from l
splitAt (l, n)	List{X} lNat → ListPair{X}	Split l at index n into two lists
evenpos (l)	List{X} → List{X}	Keep elements at even indices
oddpos (l)	List{X} → List{X}	Keep elements at odd indices
split (l)	List{X} → ListPair{X}	Take odd and even positions into two seperate lists
replicate (n, x)	lNat X$Elt −> List{X}	List of length n with x at each position
intersperse (x, l)	X$Elt List{X} −> List{X}	Put x between each two elements of l
zip (l1, l2)	List{X} List{X} → List{X}	List of pairs from two lists
unzip (l)	List{X} → ListPair{X}	Pair of lists from list of pairs
shiftL (l)	List{X} → List{X}	Shift l to left, first element becomes last element
shiftR (l)	List{X} → List{X}	Shift l to right, last element becomes first element
swap (l, n, m)	List{X} lNat lNat → List{X}	Swapping the elements at indices n and m
Lists of natural numbers		
Function	Type	Description
sum (l)	List{lNat} → lNat	Sum of all numbers of l
prod (l)	List{lNat} → lNat	Product of all numbers of l

Table 5.4.: Results of tested list functions

		Specification				
Test	Target funs	Size	Style	BK	Invented	Time (sec)
1	init , last	$4+4$	patterns			0.056
2	++	3	patterns			0.032
3	length	3	patterns			0.020
4	reverse	5	patterns		init , last	0.280
5	reverse	5	patterns	init , last		0.088
6	!!	3	patterns			0.024
7	take	5	patterns			0.036
8	drop	5	patterns			0.152
9	splitAt	5	patterns		take, drop	0.156
10	splitAt	5	patterns	take, drop		0.040
11	evenpos	5	patterns			0.048
12	oddpos	5	patterns			0.144
13	split	5	patterns		oddpos, evenpos	0.212
14	replicate	3	patterns			0.020
15	intersperse	4	patterns			0.068
16	zip	6	patterns			0.132
17	unzip	3	patterns			0.072
18	shiftL , shiftR	$4+4$	patterns		last , init	0.336
19	swap	6	patterns			∅, 6.820[a]
20	sum	13	examples			0.152
21	sum	13	examples	+		∅
22	sum2	4	traces			0.036[b]
23	prod	13	examples			$0.568^{\perp}$
24	prod	13	examples	∗		∅

[a]with rapid rule-splitting enabled
[b]The same function as sum but we provided *traces* of the form sum ([x,y,z]) = x + y + z as specification, instead of I/O examples or I/O patterns

Listing 5.3: Induced CS for shiftL and shiftR, somewhat simplified by hand (non-recursive subfunctions eliminated by unfolding); the invented subfunctions sub3 and sub4 are the last and init functions, respectively

```
shiftL ([])           → []
shiftL (x : [])       → x : []
shiftL (x : y : xs) → y : shiftL (x : xs)
shiftR ([])         → []
shiftR (x : xs)     → sub3 (x : xs) : sub4 (x : xs)
sub3 (x : [])       → x
sub3 (x : y : xs)  → sub3 (y : xs)
sub4 (x : [])       → []
sub4 (x : y : xs)  → x : sub4 (y : xs)
```

Listing 5.4: Induced CS for sum

```
sum ([])         →  0
sum (0 : xs)     →  sum (xs)
sum (S x : xs)   →  S (sum (x : xs))
```

Listing 5.5: Induced CS for the swap function

```
swap (x0 : x1 : xs,       0, S 0)         →  x1 : x0 : xs
swap (x0 : x1 : x2 : xs, 0, S S x4)      →
        swap (x1 : swap (x0 : x2 : xs, 0, S x4), 0, S 0)
swap (x0 : x1 : x2 : xs, S x4, S S x5)  →  x0 : swap (x1 : x2 : xs, x4, S x5)
```

appropriate background function for sum. Yet the additional possibilities to apply χ_{call} when + was given led to a combinatorial explosion of the branching factor of the problem space. Hence inducing the *simpler* solution (compared to the solution when + is not given) in it would take much more time than the induction of the *more complex* solution *without* + but in a problem space with a much *smaller* branching factor. Third, we tested sum with *traces* as specification instead of I/O examples or I/O patterns. (We renamed the function to sum2 for this experiment.) This means that the specification rules had, e.g., the form

```
sum2 (x : y : z : []) → x + y + z
```

instead of, e.g.,

```
sum (S S 0 : 0 : S 0) → S S S 0 .
```

Finally we tested prod with and without $*$ as background knowledge. In the first case, we got an induced function, yet a wrong one (because prod without background knowledge is not included in the problem space). With prod as background knowledge we got a timeout.

The most complex function in this module is the swap function that swaps two elements of a list at two given indices. We restricted the provided examples to the case where the first given index is smaller than the second one and both indices actually occur in the list. Listing 5.5 shows the induced CS (after applying some simplification rules eliminating non-recursive subfunctions) correctly computing the swap function.

5.1.3. Functions of Lists of Lists (Matrices)

In the third and last module of the functional programming block, we tested functions for lists of lists, i.e., matrices. Actually, the inner lists are lists of natural numbers, represented by the sort lNat (see the previous sections). However, this has only the pragmatic reason that we could not instantiate the parameterized list type from the previous section by itself (without instantiating the type parameter for the inner lists). We have three sorts now: lNat for natural numbers, List{lNat} for lists of natural

Table 5.5.: Tested functions for lists of lists (matrices)

Function	Type	Description
concat (ls)	List{NatList} → List{lNat}	Concatenation of a list of lists to one list
map: (x, ls)	lNat List{NatList} → List{NatList}	apply x : l to each inner list l of ls
inits (l)	List{lNat} → List{NatList}	all prefixes of a list
tails (l)	List{lNat} → List{NatList}	all suffixes of a list
subsequences (l)	List{lNat} → List{NatList}	all subsequences of a list; (equivalent with powerset)
tails2 (ls)	List{NatList} → List{NatList}	the tail of all inner lists
transpose (ls)	List{NatList} → List{NatList}	transposing the matrix represented by ls
weave (ls)	List{NatList} → List{lNat}	collecting all elements of ls into one list; first all first elements, followed by all second elements etc.

Listing 5.6: Induced CS for weave; the automatically invented sub36 implements shiftL

```
weave ( nil )              →  []
weave ((x : xs) :: xss)  →  x : weave (sub36 ((x : xs) : xss))
sub36 ((x : []) :: nil )                  →  nil
sub36 ((x : []) :: (y : ys) :: xss)       →  (y : ys) :: xss
sub36 ((x : y : xs) :: nil )              →  (y : xs) :: nil
sub36 ((x : y : xs) :: (y : ys) :: xss)  →  (y : ys) :: sub36 ((x : y : xs) :: xss)
```

numbers, and List{NatList} for lists of lists of natural numbers. The constructors for List{lNat} are the same as in the previous section. The constructors for List{NatList} (list of lists) are nil (the empty list of lists) and _::_ (inserting a list at the front of a list of lists).

Table 5.5 gives an overview of the tested functions.

Table 5.6 summarizes the test-setup and results. The structure is the same as in the previous section. The columns, from left to right, denote the number of the tested problem, name(s) of target function(s), number(s) of provided specification rules, style of specification, provided background functions, invented functions, and induction time in seconds. A $\perp$ adhering to the induction time indicates a wrong solution, a $\oslash$ instead of an induction time indicates a “timeout” (if the run-time exceeded three minutes). Again we set the maximum depth of rules to 3 and in case of a timeout, we reduced the maximum depth to 2 and/or switched on rapid rule-splitting.

We exemplarily show the (simplified) induced CS for the weave function, that has been tested without background knowledge. IGOR2 invented the shiftL function for lists of lists (sub36 in the listing) to solve weave. Listing 5.6 shows the solution.

Table 5.6.: Results for tested functions for matrices

		Specification				
Test	Target funs	Size	Style	BK	Invented	Time (sec)
1	concat	13	patterns			0.240
2	map:	3	patterns			0.056
3	inits , tails	$3+3$	patterns	map:		0.316
4	tails2	4	patterns			0.056
5	subsequences	3	patterns	map:, ++		⊘
6	transpose	9	patterns			⊘
7	transpose	9	patterns	tails2		7.308
8	weave	11	patterns		shiftL	⊘, 33.350[a]

[a]with rapid rule-splitting enabled

5.2. Artificial Intelligence Problems

Learning from experience, with the result that future problems or tasks in the same or a similar domain can be solved more efficiently, is a core cognitive capability and studied in artificial intelligence as well as in cognitive psychology.

This section summarizes results that have already been published in [95]. The cited paper is joint work of the author of this thesis and Ute Schmid and Martin Hofmann. The author of this thesis was responsible for specifying the considered problems for Igor2 and for conducting the experiments, as summarized in the following. Besides these technical matters, the paper moreover contains an extensive motivation of the considered problems from a cognitive psychology perspective.

Table 5.7 gives an overview over the tested problems.

5.2.1. Learning to Solve Problems

Often, in cognitive psychology, speed-up effects in problem solving are modelled as composition of primitive rules as a result of their co-occurrence during problem solving, e.g., knowledge compilation in the cognitive architecture ACT [2] or operator chunking in the cognitive architecture SOAR [91]. Similarly, in AI planning *macro learning* was modelled as composition of primitive operators to more complex ones [57]. However, general problem-solving strategies are often *recursive*. A classical example is the famous *Towers-of-Hanoi* problem.

There is empirical evidence that humans are in general able to learn such generalized recursive knowledge from experience, e.g., after solving Tower-of-Hanoi problems, at least some people have acquired the recursive solution strategy [5].

Also in automated planning [32], extending macro-learning capabilities to learning of *recursive* macros could be of great practical relevance. A long-standing and still open problem in automated planning is the scalability regarding the number of objects involved in a particular problem. E.g., automated planners can effectively solve *Towers-*

Table 5.7.: Tested problems in artificial intelligence and cognitive psychology domains

Problem	**Description**
	Problem solving domain
clearBlock (x, t, s)	Unstack all blocks above block x in tower t in state s
tower (x, s)	Stack all blocks up to block x in the correct order to a tower in state s
rocket (objs, s)	Carry all objects in objs to the moon in state s by loading them, flying the rocket, and unloading them
hanoi (d, src, aux, dst, s)	*Towers-of-Hanoi:* Move all discs up to disc d from src peg to dst peg via aux peg
	Reasoning domain
ancestor (x, y, t)	Return path from node x to node y in binary tree t; models transitivity
	Language processing domain
sentence (d)	Learning a simple grammar in order to produce correct sentences

of-Hanoi only up to a certain number of discs ([97] mentions 15 discs as a limit for tractable problem sizes). The reason is the general exponential growth of the search tree with respect to the length of the solution plan. On the other side, if a planner would *know* the recursive strategy that generates the solution plan for an arbitrary number of discs, than search could be omitted and the solution plan can simply be generated by the (learned) recursive strategy.

The general idea to apply Igor2 in automated planning and problem solving is, that first a planner solves some small problems in a particular domain and then Igor2 is applied to extract the underlying recursive strategy in terms of a recursive functional program.

In classical AI-planning, states of the world are represented by sets of ground predicates, and operators to compute successor states are represented by *preconditions* and *effects*, which are both non-ground sets of literals. A concrete *action* results from instantiating all variables of an operator. An action can be applied to a state s, if all its positive precondition literals appear in s. The corresponding successor state results from adding the positive effects to s and deleting the negative effects from s. A *planning problem* is given by an *initial state* and a *goal description*, both represented as sets of ground predicates. The goal for the planner is then to find a (solution) *plan*—a sequence of actions—that achieves a state satisfying the goal description when applied to the initial state.

A very well-known (benchmark) domain for automated planning is the *blocksworld*, where several blocks lie on a table and can be stacked to several towers. It contains

Domain (the Puttable operator):

```
Puttable(x)
;; put clear block x lying on block y to the table
PRECOND: clear(x), on(x, y)
EFFECTS: ontable(x), clear(y), not(on(x,y))
```

Problem descriptions:

```
: init-1 clear(A), ontable(A)
: init-2 clear(A), on(A, B), ontable(B)
: init-3 on(B, A), clear(B), ontable(A)
: init-4 on(C, B), on(B, A), clear(C), ontable(A)
: goal clear(A)
```

Figure 5.1.: The Puttable operator and four example problems for the general clearBlock task

operators to stack one block on top of a tower and to put a topmost block from a tower to the table.

For a first example problem, we at first restrict ourselves to the Puttable operator that is shown in Figure 5.1. Furthermore, the figure shows four initial states and one goal description. The (recursive) task to be learned here is clearBlock, putting all blocks above a certain block x in a tower to the table such that x becomes cleared. In the given example problems, block A is to be cleared (as stated by the goal predicate). The four example problems result from initially positioning A at different positions in a tower:

1. A lies on the table and is already cleared.
2. A lies on another block and is already cleared.
3. A lies on the table and is covered by one other block.
4. A lies on the table and is covered by two other blocks.

A solution plan for initial state 4 in Figure 5.1, consisting of two actions, is:

```
Puttable(C), Puttable(B) .
```

Such a plan can be obtained by any PDDL planner [32] if it is applied to the domain and problem description shown in Figure 5.1.

The idea is now to learn a recursive program clearBlock that takes a block and a tower as input and returns the correct sequence of Puttable actions to clear the provided block, i.e., a solution plan, as output.

That means, the specification for Igor2 for this task consists of the initial problems listed in Figure 5.1 as example inputs and the corresponding plans as example outputs.

Listing 5.7: Examples of clearBlock for IGOR2

```
clearBlock(a, a Table, s)      → s
clearBlock(a, a b Table, s)    → s
clearBlock(a, b a Table, s)    → Puttable(b, s)
clearBlock(a, c b a Table, s)  → Puttable(b, Puttable(c, s))
```

Therefore, the plans must be rewritten as *terms*. This is done by introducing a further parameter s to the Puttable operator, a so-called *situation variable* [64]. The above plan then becomes the term

```
Puttable(C, Puttable(B, s)) .
```

For the IGOR2 specification, we additionally have replaced the constant blocks by variables. We have three types: State, Block, and Tower. Constructors for sort Tower are Table : → Tower (empty tower) and _ _ : Block Tower → Tower (the provided block is the new topmost block of the resulting tower with the given tower below it). The function to be induced is clearBlock : Block Tower State → State. Listing 5.7 shows the four examples for IGOR2 according to the four problems stated in Figure 5.1.

The described procedure to transform a number of planning problems and the corresponding plans into a specification for IGOR2 is not implemented yet, we have rather done this transformation by hand. For more complex problems, the transformation becomes less straight-forward.

The recursive program induced by IGOR2 for clearBlock, as well as the programs induced for the other problems in the *problem solving* domain, are shown in Listing 5.8. Note that the induced rules for clearBlock and tower are *conditional* rules. The conditional-rules extension (see Section 4.9.1) of IGOR2 was enabled for all artificial intelligence / cognitive psychology problems. In the case of clearBlock, no predicate was explicitly specified but the standard predicate, equality on terms, was used as one can see in the induced rules. In the case of tower, a predicate isTower? was provided in the specification.

Consider the induced recursive strategy for tower. It can be read as: “If the tower with topmost block o already exists in s, return just s. Otherwise, build the tower where the topmost block is the block precedent to o, then clear that block, then clear o, finally put o to the precedent block.” This strategy yields optimal (i.e., the shortest possible) solution plans.

Table 5.8 summarizes test-setup and induction times for the problem-solving problems.

5.2.2. Reasoning and Natural Language Processing

As a reasoning problem, we have considered the concept of *ancestor*. The competence underlying the correct application of the *ancestor* concept, that is, correctly classifying a person as ancestor of some other person, in our opinion is the correct application of

Listing 5.8: Induced programs in the problem solving domain

```
clearBlock (a, b t, s) → s                                 if a = b
clearBlock (a, b t, s) → clearBlock (a, t, Puttable (b, s))  if a ≠ b


tower (o, s) → s                            if isTower? (o, s)
tower (o, s) → put (o, sub1 (o, s),
                 clearBlock (o, clearBlock (sub1 (o, s),
                   tower (sub1 (o, s), s))))    if not (isTower? (o, s))
sub1 (S o, s) → o


rocket (Nil, s)   → Move (s)
rocket (o os, s)  → Unload (o, rocket (os, Load (o, s)))


hanoi (0, src, aux, dst, s)    → Move (0, src, dst, s)
hanoi (S d, src, aux, dst, s)  → hanoi (d, aux, src, dst,
                                  Move (S d, src, dst,
                                    hanoi (d, src, dst, aux, s)))
```

Table 5.8.: Results for tested problem-solving problems

Task	Num of Expls	BK	Invented	Time (sec)
clearBlock	4			0.036
tower	9	clearBlock, isTower?	blockBelow	1.2
rocket	3			0.012
hanoi	3			0.076

Listing 5.9: Induced rules for ancestor

```
ancestor (x, y, Nil)              → Nilp
ancestor (x, y, Node (z, l, r))  → isIn? (y, Node (z, l, r))                 if x = z
ancestor (x, y, Node (z, l, r))  → ancestor (x, y, l) or ancestor (x, y, r)   if x ≠ z
```

Original grammar:

```
S -> NP VP
NP -> d n
VP -> v NP | v S
```

Examples provided to IGOR2:

```
sentence(1) = (d n v d n !) .
sentence(S 1) = (d n v d n v d n !) .
sentence(S S 1) = (d n v d n v d n v d n !) .
```

Figure 5.2.: A phrase-structure grammar and according examples for IGOR2 (in the very original grammar, d n v are non-terminals D N V which go to concrete words)

the *transitivity* relation in some partial ordering. We believe that if a person has grasped the concept of transitivity in one domain, such as *ancestor*, this person will also be able to correctly apply it in other, previously unknown domains. For example, such a person should be able to correctly infer *is-a* relations in some ontology.

For simplicity of modeling, we used binary trees as domain model. For trees with arbitrary branching factor, the number of examples would have to be increased significantly. The transitivity rule learned by IGOR2 is given in Listing 5.9.

Finally, we presented IGOR2 examples to learn a phrase-structure grammar. This problem is also addressed in grammar inference research [93]. We avoided the problem of learning word-category associations and provided examples abstracted from concrete words. In particular, a function sentence should be learned that generates words (or sentences) of the target grammar of particular depths. Figure 5.2 shows the grammar to be learned and the corresponding examples that were provided as specification to IGOR2. The abstract example sentence structures correspond to sentences as [22]:

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.*

Listing 5.10 shows the induced rules. Note that IGOR2 has effectively *simplified* the original grammar without changing the represented language. Rewritten as grammar, the two learned rules would read:

Listing 5.10: Induced rules for the word-structure grammar

```
sentence (1)   → (d n v d n !)
sentence (S n) → (d n v sentence(n))
```

Table 5.9.: Empirical comparison of different inductive programming systems

Problems	Systems						
	ADATE	FFoil	Flip	Golem	Igor1	Igor2	MagH.
Lasts	365.62	$0.7^{\perp}$	×	1.062	0.051	5.695	19.43
Last	1.0	0.1	0.020	< 0.001	0.005	0.007	0.01
Member	2.11	$0.1^{\perp}$	17.868	0.033	—	0.152	1.07
odd/even	—	$< 0.1^{\perp}$	0.130	—	—	0.019	—
multlast	5.69	< 0.1	$448.900^{\perp}$	< 0.001	0.331	0.023	0.30
isort	83.41	×	×	0.714	—	0.105	0.01
reverse	30.24	—	—	—	0.324	0.103	0.08
weave	27.11	0.2	$134.240^{\perp}$	0.266	$0.001^{\perp}$	0.022	⊙
ShiftR	20.14	$< 0.1^{\perp}$	$448.550^{\perp}$	0.298	0.041	0.127	157.32
Mult	—	$8.1^{\perp}$	×	—	—	⊙	—
allodds	466.86	$0.1^{\perp}$	×	$0.016^{\perp}$	$0.015^{\perp}$	⊙	×

— not tested × stack overflow ⊙ time out ⊥ wrong

```
S -> d n v d n | d n v S
```

5.3. Comparison with Other Inductive Programming Systems

Generally, a substantiated comparison of the different existing inductive programming systems is difficult to achieve, mostly because, up to now, all systems have their own specification languages and need different types of specifications, e.g., I/O examples vs. I/O patterns vs. output evaluation functions vs. general predicates and complete (up to some complexity) specifications vs. single or random examples etc.

As a first attempt to systematize the relation of the different representations, in a joint work with Martin Hofmann [38], the author has proposed conditional term rewriting as a common representation framework for functional as well as logic-based inductive program synthesis systems. Furthermore, the cited paper contains a first empirical comparison of some inductive functional and inductive logic programming systems that are able to induce recursive functions. Table 5.9 summarizes the results.

Some tested problems were already mentioned in the previous sections. Further problems are *multlast*, which replaces all elements with the last element, *Lasts*, which applies *Last* on a list of lists, *isort*, which is insertion-sort, *allodds*, which checks for odd numbers, *weave* restricted to only two lists, and *Member*, which checks whether an element occurs in a list.

The tested systems include the functional generate-and-test systems ADATE and

MagicHaskeller (see Section 3.4), the sequential covering ILP systems Foil and Golem (see Section 3.3.3), the functional-logic system Flip, the functional analytical system Igor1 (see Section 3.2.3), and the combined analytical and search-based system Igor2, presented in this thesis (Chapter 4).

Because FFoil and Golem usually perform better with more examples, whereas Flip, MagicHaskeller and Igor2 do better with less, each system got as much examples as necessary up to certain complexity, but then exhaustively, so no specific cherry-picking was allowed.

For synthesizing *isort* all systems had a function to insert into a sorted list, and the predicate $<$ as background knowledge. Flip needed an additional function if to relate the *insert* function with the $<$. For all systems except for Flip and MagicHaskeller the definition of the background knowledge was extensional. Igor2 was allowed to use variables and for Golem additionally the accordant negative examples were provided. MagicHaskeller had paramorphic functions to iterate over a data type as background knowledge. Note that we did not test a system with a problem which it per se cannot solve due to its restriction bias. This is indicated with '—' instead of a runtime. A timeout after ten minutes is indicated with $\odot$. Table 5.9 shows the runtimes of the different systems on the example problems.

The results show that the functional generate-and-test methods (at least ADATE) need comparatively more time than the two extensional coverage ILP systems FFoil and Golem and the analytical functional systems Igor1 and Igor2. They further show that at least FFoil has problems to induce recursive programs (cp. Section 3.3.3) and that the purely analytical approach (Igor1) is seriously restricted regarding inducible functions. Finally it shows that integrating generalized analytical techniques into search, as done by Igor2, expands the class of inducible functions compared to the purely analytical approach (Igor1) and at the same time solves the test problems comparatively quickly.

6. Conclusions

In this thesis, we at first comprehensively surveyed and classified currently existing approaches to the inductive synthesis of functional and logic programs, second, presented an own algorithm, IGOR2, to the inductive synthesis of functional programs and showed certain properties of it, and third, evaluated IGOR2 by means of a number of recursive problems known from functional programming and artificial intelligence. In the following, we summarize the obtained results and provide directions for future research.

6.1. Main Results

Current methods to the induction of functional programs can be classified according to two general approaches—the analytical, recurrence detection approach (Section 3.2), where programs are derived from the provided I/O examples by detecting and inductively generalizing recurrent structures between them, and, on the other side, the enumerative, generate-and-test approach (Section 3.4), where candidate programs from some program class are repeatedly generated and tested until a program is found that satisfies the provided specification.

Classical analytical methods are fast but only at the prize of (i) restricted program schemas, (ii) that only basic datatype constructors and selection functions as primitives can be used—i.e., that no background knowledge can be provided—, and (iii) that the provided I/O examples must be complete up to some complexity regarding the domain—e.g., all lists up to a certain length must be provided as example inputs. On the other side, generate-and-test methods have no inherent restrictions regarding program schemas and primitives and are more flexible regarding their specifications. Provided sets of I/O examples need not to be complete and also extended forms of specifications, going beyond I/O examples or I/O patterns, can be used. But since the construction of candidate programs is not constrained by the provided specification, generate-and-test methods are in general much more time-consuming than analytical methods (cp. Table 5.9).

The presented algorithm IGOR2 generalizes the recurrence detection method and integrates it into search operators. The matching of (parts of) I/O examples of the target function in order to detect regularities and derive a recursive function call is extended to several target functions and background knowledge—the function call operators χ_{smplCall} and χ_{call} of IGOR2 not only find *recursive* calls but try to also match I/O examples belonging to different functions in order to introduce background functions or mutual recursive calls. Furthermore, the program schema of IGOR2 is much less restrictive than that of Summers (Section 3.2.1), for example. The result is that at each synthesis step several *candidate* refinements are possible, i.e., that the synthesis operators become *search* operators. IGOR2 uses its synthesis operators within a uniform cost search, where candidate programs with fewer cases are preferred, by assigning them lower costs, in order to achieve a maximal general solution program. However, since candidate programs are still constructed based on the provided specification, often large parts of the program space can be pruned. We have started to empirically verify this by means of a comparison of IGOR2 with other inductive programming systems (Section 5.3).

The possibility of pruning the problem space depends on the provided specification. In general, the more structure the I/O examples have, the fewer matchings between them are possible and the greater parts of the problem space can be pruned. This is the reason why structural problems on lists such as reverse are generally easier and faster to solve than, e.g., problems on natural numbers, as the conducted experiments (Chapter 5) show. In the worst case, if the outputs of specified functions are simply variables and hence provide no structure at all, combined with its breadth-first search approach, IGOR2 can, at the moment, degenerate to a purely enumerative algorithm (cp. Section 4.8.5).

The experiments in Chapter 5 show that IGOR2 is able to reliably and in reasonable time induce intended functions for a wide variety of problems ranging from problems of natural numbers over problems of lists and matrices to problems of domains known from artificial intelligence such as the blocksworld or the Towers-of-Hanoi. This shows the potential of (analytical functional) inductive programming as a tool to assist in functional program development as well as to model the capability of human cognition to learn general problem solving strategies from experience.

As one can also see, some relatively simple functions, such as multiplication of natural numbers, can *not* be induced *without* background knowledge, though IGOR2 is generally able to automatically find help functions, if needed. E.g., in order to be able to induce multiplication, IGOR2 needs addition as background knowledge. The reason why IGOR2 cannot find addition automatically as help function for multiplication is a restriction of the form of generatable function compositions, caused by the currently used synthesis operators. We have not substantially studied the class of inducible functions yet; however, a short introduction to the mentioned restrictions was given in Section 4.8.4.

We have shown (Section 4.8) that the search conducted by IGOR2 is terminating and complete under certain constraints and that programs induced by IGOR2 correctly compute the specified I/O examples.

6.2. Future Research

In general, IGOR2 shows that combining the analytical approach to the induction of functional programs with search in a program space is possible and promising. There are several possible starting points for further research in this general direction.

Depending on the provided specification, the analytical techniques more or less prune the problem space and in the worst case, IGOR2 also simply enumerates programs in an unconstrained way. In this case, explicit generate-and-test systems like ADATE or

MagicHaskeller are likely to show a better performance because of better heuristics for this case or other techniques, like preventing the enumeration of several semantically equivalent programs. One could try to integrate such techniques into Igor2 or alternatively, to combine a generate-and-test system like ADATE with Igor2. A first attempt, where the author of this thesis contributed, of such a combination is reported in [23].

Another direction could be to integrate general knowledge of (patterns of) algorithms into Igor2, possibly in the form of program schemas as proposed by several works concerned with inducing recursive *logic* programs from examples (cp. Section 3.3.4). Especially in *functional* programming, standard *higher-order* functions like map, reduce, filter etc. capture particular recursive schemes and are well-analyzed and frequently used in functional programming. Currently, Igor2 uses first-order term rewriting to express programs. Yet an extension to higher-order functions and in particular integrating the mentioned standard higher-order functions as general background knowledge into Igor2 could be sensible for several reasons.

First, programs become more compact by using predefined higher-order functions, because the recursive scheme including base- and recursive case is encapsulated in the definition of the higher-order function and need not to be re-stated. This potentially reduces the number of synthesis operator applications until a correct program is found, i.e., the depth of a solution in the search tree. Second, using these functions would help to clarify which classes of functions can be induced. Third, providing these functions as general background knowledge can help to overcome the current restriction regarding nested function calls as described in Section 4.8.4.

An extension of Igor2 by higher-order functions is described in [37].

Finally, extending the specification language by allowing for extensionally quantified variables in RHSs of I/O examples or I/O patterns (cp. Section 4.9.3) could be an approach to relaxing the current requirement of sets of I/O examples or I/O patterns that are complete up to a certain complexity regarding the domain of the target function.

Besides further developing general inductive programming techniques as described, we think that it would be useful to specifically try to tackle particular application areas like synthesizing prototypes of functions from test-cases in test-driven software development or learning domain-knowledge in automated planning. We think that identifying specific application fields and domains could help to sensibly identify strengths and weaknesses of existing methods, to extend them and to identify possibilities to integrate them in a useful way.

Bibliography

[1] David W. Aha, Stephane Lapointe, Charles X. Ling, and Stan Matwin. Inverting implication with small training sets. In *Proceedings of the European Conference on Machine Learning (ECML'94)*, volume 784 of *LNCS*, pages 29–48. Springer-Verlag, 1994.

[2] John R. Anderson and Christian Lebiere. *The Atomic Components of Thought.* Lawrence Erlbaum Associates, Inc., 1998.

[3] Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

[4] Dana Angluin. Equivalence queries and approximate fingerprints. In *Proceedings of the 2nd Annual Workshop on Computational Learning Theory (COLT'89)*, pages 134–145, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[5] Yuichiro Anzai and Herbert A. Simon. The theory of learning by doing. *Psychological Review*, 86(2):124–140, 1979.

[6] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1999.

[7] Kent Beck. *Test-Driven Development By Example.* Addison-Wesley, 2003.

[8] Henrik Berg, Roland Olsson, Per-Olav Rusås, and Morgan Jakobsen. Synthesis of control algorithms for autonomous vehicles through automatic programming. In Haiying Wang, Kay Soon Low, Kexin Wei, and Junqing Sun, editors, *Fifth International Conference on Natural Computation*, pages 445–453. IEEE Computer Society, 2009.

[9] Francesco Bergadano and Daniele Gunetti. An interactive system to learn functional logic programs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'93)*, 1993.

[10] Francesco Bergadano and Daniele Gunetti. *Inductive Logic Programming: From Machine Learning to Software Engineering.* MIT Press, Cambridge, MA, USA, 1995.

[11] Alan W. Biermann. The inference of regular LISP programs from examples. *IEEE Transactions on Systems, Man and Cybernetics*, 8(8):585–600, 1978.

[12] A.W. Biermann, R.I. Baum, and F.E. Petry. Speeding up the synthesis of programs from traces. *IEEE Transactions on Computers*, 24(2):122–136, 1975.

[13] A.W. Biermann and R. Krishnaswamy. Constructing programs from example computations. *IEEE Transactions on Software Engineering*, 2(3):141–153, 1976.

[14] Franck Binard and Amy Felty. Genetic programming with polymorphic types and higher-order functions. In *Proceedings of the 10th annual Conference on Genetic and Evolutionary Computation (GECCO'08)*, pages 1187–1194, New York, NY, USA, 2008. ACM.

[15] Robert S. Boyer and J. Strother Moore. Proving theorems about LISP functions. *Journal of the ACM*, 22(1):129–144, 1975.

[16] Ivan Bratko. *Prolog Programming for Artificial Intelligence.* Addison-Wesley, 1986.

[17] R. M. Cameron-Jones and J. R. Quinlan. Avoiding pitfalls when learning recursive theories. In R. Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1050–1057. Morgan Kaufmann, 1993.

[18] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The maude 2.0 system. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *LNCS*, pages 76–87. Springer-Verlag, 2003.

[19] William W. Cohen. Pac-learning recursive logic programs: Efficient algorithms. *Journal of Artificial Intelligence Research*, 2:501–539, 1995.

[20] William W. Cohen. Pac-learning recursive logic programs: Negative results. *Journal of Artificial Intelligence Research*, pages 541–573, 1995.

[21] Bruno Courcelle. Recursive applicative program schemes. In *Handbook of Theoretical Computer Science: Formal Models and Semantics*, volume B, chapter 9, pages 459–492. MIT Press, Cambridge, MA, USA, 1990.

[22] M. A. Covington. *Natural Language Processing for Prolog Programmers.* Prentice Hall, 1994.

[23] Neil Crossley, Emanuel Kitzelmann, Martin Hofmann, and Ute Schmid. Combining analytical and evolutionary inductive programming. In B. Goertzel, P. Hitzler, and M. Hutter, editors, *Second Conference on Artificial General Intelligence*, pages 19–24. Atlantis Press, 2009.

[24] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification I.* Springer-Verlag, 1985.

[25] C. Ferri-Ramírez, J. Hernández-Orallo, and M.J. Ramírez-Quintana. Incremental learning of functional logic programs. In *Proceedings of the 5th International Symposium on Functional and Logic Programming (FLOPS'01)*, volume 2024 of *LNCS*, pages 233–247. Springer-Verlag, 2001.

[26] Pierre Flener. *Logic Program Synthesis from Incomplete Information.* Kluwer Academic Publishers, 1995.

[27] Pierre Flener. Inductive logic program synthesis with DIALOGS. In S. Muggleton, editor, *Selected Papers of the 6th International Workshop on Inductive Logic Programming, (ILP'96)*, volume 1314 of *LNCS*, pages 175–198, 1997.

[28] Pierre Flener, Kung-Kiu Lau, and Mario Ornaghi. On correct program schemas. In Norbert E. Fuchs, editor, *Logic Program Synthesis and Transformation, 7th International Workshop, LOPSTR'97, Proceedings*, volume 1463 of *LNCS*, pages 128–147. Springer-Verlag, 1998.

[29] Pierre Flener and Ute Schmid. An introduction to inductive programming. *Artificial Intelligence Review*, 29(1):45–62, 2008.

[30] Pierre Flener and Serap Yilmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. *The Journal of Logic Programming*, 41(2-3):141–195, 1999.

[31] Mitsue Furusawa, Nobuhiro Inuzuka, Hirohisa Seki, and Hidenori Itoh. Induction of logic programs with more than one recursive clause by analyzing saturations. In *Proceedings of the 7th International Workshop on Inductive Logic Programming (ILP'97)*, volume 1297 of *LNCS*, pages 165–172. Springer-Verlag, 1997.

[32] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice.* Morgan Kaufmann Publishers, 2004.

[33] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[34] Lutz Hamel. Breeding algebraic structures - an evolutionary approach to inductive equational logic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'02)*, pages 748–755, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[35] Lutz Hamel and Chi Shen. An inductive programming approach to algebraic specification. In *Proceedings of the 2nd Workshop on Approaches and Applications of Inductive Programming (AAIP'07)*, pages 3–14, 2007.

[36] Martin Hofmann. *Automated Construction of XSL-Templates: An Inductive Programming Approach.* VDM Verlag, 2008.

[37] Martin Hofmann and Emanuel Kitzelmann. I/o guided detection of list catamorphisms. In John Gallagher and Janis Voigtländer, editors, *ACM SIGPLAN 2010 Workshop on Partial Evaluation and Program Manipulation, Proceedings*, pages 93–100. ACM Press, 2010.

[38] 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, *Second Conference on Artificial General Intelligence*, pages 55–60. Atlantis Press, 2009.

[39] Peter Idestam-Almquist. Efficient induction of recursive definitions by structural analysis of saturations. In Luc De Raedt, editor, *Advances in Inductive Logic Programming*. IOS Press, 1996.

[40] Alípio Jorge and Pavel Brazdil. Architecture for iterative learning of recursive definitions. In Luc De Raedt, editor, *Advances in Inductive Logic Programming*. IOS Press, 1996.

[41] Alípio M. G. Jorge. *Iterative Induction of Logic Programs*. PhD thesis, Departamento de Ciência de Computadores, Universidade do Porto, 1998.

[42] 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 *Sixth International Joint Conference on Artificial Intelligence (IJCAI'79)*, pages 440–447. Morgan Kaufmann, 1979.

[43] 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.

[44] Stefan Kahrs. Genetic programming with primitive recursion. In *Proceedings of the 8th annual Conference on Genetic and Evolutionary Computation (GECCO'06)*, pages 941–942, New York, NY, USA, 2006. ACM.

[45] 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.

[46] Susumu Katayama. Recent improvements of magichaskeller. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *Approaches and Applications of Inductive Programming, 3rd International Workshop, AAIP'09, Revised Papers*, volume 5812 of *LNCS*, pages 174–193. Springer-Verlag, 2010.

[47] Emanuel Kitzelmann. Inductive functional program synthesis – a term-construction and folding approach. Diplomarbeit, Technische Universität Berlin, 2003. Unpublished.

[48] Emanuel Kitzelmann. Analytical inductive functional programming. In Michael Hanus, editor, *18th International Symposium on Logic-Based Program Synthesis and Transformation, Revised Selected Papers*, volume 5438 of *LNCS*, pages 87–102. Springer-Verlag, 2009.

[49] 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.

[50] Emanuel Kitzelmann and Ute Schmid. An explanation based generalization approach to inductive synthesis of functional programs. In Emanuel Kitzelmann, Roland Olsson, and Ute Schmid, editors, *ICML-2005 Workshop on Approaches and Applications of Inductive Programming*, pages 15–27, 2005.

[51] Emanuel Kitzelmann and Ute Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research*, 7:429–454, 2006. Revised version of [50].

[52] 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.

[53] Yves Kodratoff. A class of functions synthesized from a finite number of examples and a LISP program scheme. *International Journal of Computer and Information Sciences*, 8(6):489–521, 1979.

[54] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. GAST: Generic automated software testing. In *Implementation of Functional Languages (IFL'02)*, volume 2670 of *LNCS*. Springer, 2003.

[55] Pieter Koopman and Rinus Plasmeijer. Synthesis of functions using generic programming. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *Approaches and Applications of Inductive Programming, 3rd International Workshop, AAIP'09, Revised Papers*, volume 5812 of *LNCS*, pages 25–49. Springer-Verlag, 2010.

[56] 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.

[57] Richard E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–77, 1985.

[58] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[59] John R. Koza, David Andre, Forrest H. Bennett, and Martin A. Keane. *Genetic Programming III: Darwinian Invention & Problem Solving.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[60] Stéphane Lapointe and Stan Matwin. Sub-unification: a tool for efficient induction of recursive programs. In *ML92: Proceedings of the Ninth International Workshop on Machine Learning*, pages 273–281, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[61] Leonid A. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3), 1973.

[62] Henry Lieberman, editor. *Your Wish is My Command: Programming by Example.* Morgan Kaufmann Publishers, 2001.

[63] Donato Malerba. Learning recursive theories in the normal ILP setting. *Fundamenta Informaticae*, 57(1):39–77, 2003.

[64] Z. Manna and R. Waldinger. How to clear a block: A theory of plans. *Journal of Automated Reasoning*, 3(4):343–378, 1987.

[65] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[66] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[67] José Meseguer and Joseph A. Goguen. Initiality, induction, and computability. In Maurice Nivat and John C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge University Press, 1985.

[68] Ryszard S. Michalski and Robert E. Stepp. Learning from observation: Conceptual clustering. In Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, chapter 11, pages 331–364. Tioga, 1983.

[69] Thomas M. Mitchell. *Machine Learning.* McGraw-Hill Higher Education, 1997.

[70] Chowdhury R. Mofizur and Masayuki Numao. Top-down induction of recursive programs from small number of sparse examples. In Luc De Raedt, editor, *Advances in Inductive Logic Programming.* IOS Press, 1996.

[71] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[72] S. H. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, pages 368–381, Tokyo, Japan, 1990. Ohmsha.

[73] Stephen Muggleton. Duce, an oracle-based approach to constructive induction. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, volume 1, pages 287–292. Morgan Kaufmann Publishers, 1987.

[74] Stephen Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[75] Stephen Muggleton and Wray L. Buntine. Machine invention of first-order predicates by inverting resolution. In J. Laird, editor, *Proceedings of the 5th International Conference on Machine Learning (ICML'88)*, pages 339–352. Morgan Kaufmann, 1988.

[76] Stephen H. Muggleton. Inductive logic programming: Derivations, successes and shortcomings. *SIGART Bulletin*, 5(1):5–11, 1994.

[77] Stephen H. Muggleton. Inverse entailment and progol. *New Generation Computing*, 13:245–286, 1995.

[78] Stephen H. Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[79] Martin Mühlpfordt. Syntaktische Inferenz Rekursiver Programmschemata. Diplomarbeit, Technische Universität Berlin, 2000.

[80] Shan-Hwei Nienhuys-Cheng and Roland de Wolf. Subsumption theorem and refutation completeness. In *Foundations of Inductive Logic Programming*, volume 1228 of *LNAI*, chapter 5. Springer-Verlag, 1997.

[81] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *LNAI*. Springer-Verlag, 1997.

[82] J. R. Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55 – 83, 1995.

[83] J. R. Olsson and D. M. W. Powers. Machine learning of human language through automatic programming. In *Proceedings of the International Conference on Cognitive Science*, pages 507–512, 2003.

[84] Dusko Pavlovic and Douglas R. Smith. Software development by refinement. In *Formal Methods at the Crossroads: From Panacea to Foundational Support*, volume 2757 of *LNCS*, pages 267–286, 2003.

[85] Gordon D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.

[86] J. R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In P. Brazdil, editor, *Proceedings of the 6th European Conference on Machine Learning*, LNCS, pages 3–20, London, UK, 1993. Springer-Verlag.

[87] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[88] Luc De Raedt and Luc Dehaspe. Clausal discovery. *Machine Learning*, 26(2):99–146, 1997.

[89] Riverson Rios and Stan Matwin. Efficient induction of recursive prolog definitions. In *Proceedings of the 11th Biennial Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence*, volume 1081 of *LNCS*, pages 240–248. Springer-Verlag, 1996.

[90] Raúl Rojas. *Neural Networks - A Systematic Introduction.* Springer-Verlag, 1996.

[91] Paul S. Rosenbloom and Alan Newell. The chunking of goal hierarchies: A generalized model of practice. In Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume 2, chapter 10. Morgan Kaufmann Publishers, 1986.

[92] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 3 edition, 2010.

[93] Yasubumi Sakakibara. Recent advances of grammatical inference. *Theoretical Computer Science*, 185(1):15–45, 1997.

[94] Ute Schmid. *Inductive Synthesis of Functional Programs: Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*, volume 2654 of *LNAI.* Springer, Berlin; New York, 2003.

[95] 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, *Second Conference on Artificial General Intelligence*, pages 162–167. Atlantis Press, 2009.

[96] Ute Schmid and Fritz Wysotzki. Applying inductive program synthesis to macro learning. In Steve Chien, Subbarao Kambhampati, and Craig A. Knoblock, editors, *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, pages 371–378. AAAI, 2000.

[97] Jürgen Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54(3):211–254, 2004.

[98] Uwe Schöning. *Logic for Computer Scientists.* Modern Birkhäuser Classics. Birkhäuser Boston, 2008.

[99] Ehud Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

[100] Douglas R. Smith. The synthesis of LISP programs from examples: A survey. In A.W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, pages 307–324. Macmillan, 1984.

[101] Irene Stahl. The appropriateness of predicate invention as bias shift operation in ilp. *Machine Learning*, 20(1):95–117, 1995.

[102] Irene Stahl and Irene Weber. The arguments of newly invented predicates in ILP. In *Proceedings of the Fourth Workshop on Inductive Logic Programming (ILP'94)*, 1994.

[103] P. D. Summers. *Program Construction from Examples.* PhD thesis, Dept. of Computer Science, Yale University, New Haven, US-CT, 1975.

[104] Phillip D. Summers. A methodology for LISP program construction from examples. *Journal of the ACM*, 24(1):161–175, 1977.

[105] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 2003.

[106] Paul E. Utgoff. Shift of bias for inductive concept learning. In Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume 2, chapter 5, pages 107–148. Morgan Kaufmann Publishers, 1983.

[107] L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[108] B. Wegbreit. Goal-directed program transformation. *IEEE Transactions on Software Engineering*, 2(2):69–80, 1976.

[109] Man Wong and Tuen Mun. Evolving recursive programs by using adaptive grammar based genetic programming. *Genetic Programming and Evolvable Machines*, 6(4):421–455, 2005.

[110] Tina Yu. Hierarchical processing for evolving recursive and modular programs using higher-order functions and lambda abstraction. *Genetic Programming and Evolvable Machines*, 2(4):345–380, 2001.

[111] 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-Verlag, 2006.

A. Specifications of the Experiments

In the following we list the complete specification files as provided to IGOR2 for the experiments conducted in Chapter 5.

A.1. Natural Numbers

```
fmod IGOR-NAT is
    sort INat .
    op 0 : -> INat [ctor] .
    op S_ : INat -> INat [ctor] .

    op _=_ : INat INat -> Bool .
    op _<=_ : INat INat -> Bool .
    op odd : INat -> Bool .
    op even : INat -> Bool .
    op _+_ : INat INat -> INat .
    op _-_ : INat INat -> INat .
    op _*_ : INat INat -> INat .
    op fac : INat -> INat .
    op fib : INat -> INat .
    op ack : INat INat -> INat .

    *** for internal encapsulation of
    *** function arguments into a tuple
    sort InVec .
    op in : INat INat -> InVec [ctor] .
    op in : INat -> InVec [ctor] .

    var x : INat .

    eq (0     = 0)     = true .
    eq (0     = S 0)   = false .
    eq (0     = S S 0) = false .
    eq (S 0   = 0)     = false .
    eq (S 0   = S 0)   = true .
    eq (S 0   = S S 0) = false .
    eq (S S 0 = 0)     = false .
    eq (S S 0 = S 0)   = false .
    eq (S S 0 = S S 0) = true .

    eq 0     <= x     = true .
    eq S 0   <= 0     = false .
    eq S 0   <= S x   = true .
    eq S S 0 <= 0     = false .
eq S S 0 <= S 0   = false .
eq S S 0 <= S S x = true .

eq odd (0)       = false .
eq odd (S 0)     = true .
eq odd (S S 0)   = false .
eq odd (S S S 0) = true .

eq even (0)       = true .
eq even (S 0)     = false .
eq even (S S 0)   = true .
eq even (S S S 0) = false .

eq 0       + x = x .
eq S 0     + x = S x .
eq S S 0   + x = S S x .
eq S S S 0 + x = S S S x .

eq 0     * x       = 0 .
eq S 0   * x       = x .
eq S S 0 * 0       = 0 .
eq S S 0 * S 0     = S S 0 .
eq S S 0 * S S 0   = S S S S 0 .
eq S S 0 * S S S 0 = S S S S S S 0 .

eq fac (0)       = S 0 .
eq fac (S 0)     = S 0 .
eq fac (S S 0)   = S S 0 .
eq fac (S S S 0) = S S S S S S 0 .

eq fib(0)           = 0 .
eq fib(S 0)         = S 0 .
eq fib(S S 0)       = S 0 .
eq fib(S S S 0)     = S S 0 .
eq fib(S S S S 0)   = S S S 0 .
eq fib(S S S S S 0) = S S S S S 0 .

eq ack(0, 0)               = S 0 .
eq ack(0, S 0)             = S S 0 .
eq ack(0, S S 0)           = S S S 0 .
eq ack(0, S S S 0)         = S S S S 0 .
eq ack(0, S S S S 0)       = S S S S S 0 .
eq ack(0, S S S S S 0)     = S S S S S S 0 .
eq ack(0, S S S S S S 0)   = S S S S S S S 0 .
eq ack(S 0, 0)             = S S 0 .
eq ack(S 0, S 0)           = S S S 0 .
eq ack(S 0, S S 0)         = S S S S 0 .
eq ack(S 0, S S S 0)       = S S S S S 0 .
eq ack(S 0, S S S S 0)     = S S S S S S 0 .
eq ack(S 0, S S S S S 0)   = S S S S S S S 0 .
eq ack(S S 0, 0)           = S S S 0 .
eq ack(S S 0, S 0)         = S S S S S 0 .
eq ack(S S 0, S S 0)       = S S S S S S S 0 .
```

```
    eq ack(S S S 0, 0)          = S S S S S 0 .
endfm
```

A.2. Lists

General Lists

```
fmod PLIST{X :: TRIV} is
    inc IGOR-NAT .

    sorts List{X} ListPair{X} .

    *** also pairs of X$Elt shall be X$Elt ...
    *** for zip
    op <_,_> : X$Elt X$Elt -> X$Elt [ctor] .

    *** list constructors
    op [] : -> List{X} [ctor] .
    op _:_ : X$Elt List{X} -> List{X} [ctor] .

    *** pair of lists constructor
    op <_;_> : List{X} List{X} -> ListPair{X} [ctor] .

    op head : List{X} -> X$Elt .
    op tail : List{X} -> List{X} .
    op init : List{X} -> List{X} .
    op last : List{X} -> X$Elt .
    op _++_ : List{X} List{X} -> List{X} .
    op length : List{X} -> INat .
    op reverse : List{X} -> List{X} .
    op _!!_ : List{X} INat -> X$Elt .
    op take : INat List{X} -> List{X} .
    op drop : INat List{X} -> List{X} .
    op splitAt : INat List{X} -> ListPair{X} .
    op split : List{X} -> ListPair{X} .
    op evenpos : List{X} -> List{X} .
    op oddpos : List{X} -> List{X} .
    op replicate : INat X$Elt -> List{X} .
    op intersperse : X$Elt List{X} -> List{X} .
    op zip : List{X} List{X} -> List{X} .
    op unzip : List{X} -> ListPair{X} .
    op shiftL : List{X} -> List{X} .
    op shiftR : List{X} -> List{X} .
    op swap : List{X} INat INat -> List{X} .
    op swap2 : List{X} INat INat -> List{X} .

    *** input arguments encapsulation
    op in : List{X} -> InVec [ctor] .
    op in : List{X} List{X} -> InVec [ctor] .
    op in : List{X} INat -> InVec [ctor] .
    op in : INat List{X} -> InVec [ctor] .
    op in : INat X$Elt -> InVec [ctor] .
    op in : X$Elt List{X} -> InVec [ctor] .
op in : List{X} INat INat -> InVec [ctor] .

var n : INat .
vars x y z v : X$Elt .
var xs ys : List{X} .

eq head (x : xs) = x .
eq tail (x : xs) = xs .

eq init (x : [])             = [] .
eq init (x : y : [])         = x : [] .
eq init (x : y : z : [])     = x : y : [] .
eq init (x : y : z : v : []) = x : y : z : [] .

eq last (x : [])             = x .
eq last (x : y : [])         = y .
eq last (x : y : z : [])     = z .
eq last (x : y : z : v : []) = v .

eq [] ++ xs                   = xs .
eq (x : []) ++ xs             = x : xs .
eq (x : y : []) ++ xs         = x : y : xs .

eq length ([])                = 0 .
eq length (x : [])            = S 0 .
eq length (x : y : [])        = S S 0 .

eq take (0, xs)             = [] .
eq take (S n, [])           = [] .
eq take (S 0, x : xs)       = x : [] .
eq take (S S 0, x : [])     = x : [] .
eq take (S S 0, x : y : xs) = x : y : [] .

eq drop (0, xs)             = xs .
eq drop (S n, [])           = [] .
eq drop (S 0, x : xs)       = xs .
eq drop (S S 0, x : [])     = [] .
eq drop (S S 0, x : y : xs) = xs .

eq reverse ([])                = [] .
eq reverse (x : [])            = x : [] .
eq reverse (x : y : [])        = y : x : [] .
eq reverse (x : y : z : [])    = z : y : x : [] .
eq reverse (x : y : z : v : []) = v : z : y : x : [] .

eq splitAt (0, xs)             = < []           ; xs > .
eq splitAt (S n, [])           = < []           ; [] > .
eq splitAt (S 0, x : xs)       = < (x : [])     ; xs > .
eq splitAt (S S 0, x : [])     = < (x : [])     ; [] > .
eq splitAt (S S 0, x : y : xs) = < (x : y : []) ; xs > .

eq split ([])                = < []           ; []           > .
eq split (x : [])            = < (x : [])     ; []           > .
eq split (x : y : [])          =< (x : [])     ; (y : [])     > .
eq split (x : y : z : [])      =< (x : z : []) ; (y : [])     > .
eq split (x : y : z : v : []) =< (x : z : []) ; (y : v : []) > .

eq evenpos ([])                  = [] .
eq evenpos (x : [])              = [] .
eq evenpos (x : y : [])          = y : [] .
eq evenpos (x : y : z : [])      = y : [] .
eq evenpos (x : y : z : v : []) = y : v : [] .

eq oddpos ([])                  = [] .
eq oddpos (x : [])              = x : [] .
eq oddpos (x : y : [])          = x : [] .
eq oddpos (x : y : z : [])      = x : z : [] .
eq oddpos (x : y : z : v : []) = x : z : [] .

eq replicate (0, x)         = [] .
eq replicate (S 0, x)       = x : [] .
eq replicate (S S 0, x)     = x : x : [] .

eq zip ([], ys)                 = [] .
eq zip (x : xs, [])             = [] .
eq zip (x : [], y : [])         =< x , y > : [] .
eq zip (x : [], y : z : [])     =< x , y > : [] .
eq zip (x : y : [], z : [])     =< x , z > : [] .
eq zip (x : y : [], z : v : []) =< x , z > : < y , v > : [] .

eq unzip ([])                              =< [] ; [] > .
eq unzip (< x , y > : [])                  =< (x : []) ; (y : []) > .
eq unzip (< x , y > : < z , v > : []) =< (x : z : []) ; (y : v : []) >
    .

eq (x : xs) !! 0 = x .
eq (x : y : xs) !! (S 0) = y .
eq (x : y : z : xs) !! (S S 0) = z .

eq intersperse (x, [])              = [] .
eq intersperse (x, y : [])          = y : [] .
eq intersperse (x, y : z : [])      = y : x : z : [] .
eq intersperse (x, y : z : v : []) = y : x : z : x : v : [] .

eq shiftL ([])                  = [] .
eq shiftL (x : [])              = x : [] .
eq shiftL (x : y : [])          = y : x : [] .
eq shiftL (x : y : z : [])      = y : z : x : [] .

eq shiftR ([])                  = [] .
eq shiftR (x : [])              = x : [] .
eq shiftR (x : y : [])          = y : x : [] .
eq shiftR (x : y : z : [])      = z : x : y : [] .

eq swap (x : y : xs,          0,      S 0)      = y : x : xs .
eq swap (x : y : z : xs,      0,      S S 0)    = z : y : x : xs .
```

```
    eq swap (x : y : z : xs,       S 0,    S S 0)   = x : z : y : xs .
    eq swap (x : y : z : v : xs, 0,      S S S 0) = v : y : z : x : xs .
    eq swap (x : y : z : v : xs, S 0,    S S S 0) = x : v : z : y : xs .
    eq swap (x : y : z : v : xs, S S 0, S S S 0) = x : y : v : z : xs .
endfm
```

Lists of Natural Numbers

```
fmod NATLIST is
    inc PLIST{INat} .

    ops sum sum2 prod : List{INat} -> INat .

    vars x y z : INat .

    eq sum([])                         = 0 .
    eq sum(0 : [])                     = 0 .
    eq sum((S 0) : [])                 = S 0 .
    eq sum((S S 0) : [])               = S S 0 .
    eq sum(0 : 0 : [])                 = 0 .
    eq sum(0 : (S 0) : [])             = S 0 .
    eq sum(0 : (S S 0) : [])           = S S 0 .
    eq sum((S 0) : 0 : [])             = S 0 .
    eq sum((S 0) : (S 0) : [])         = S S 0 .
    eq sum((S 0) : (S S 0) : [])       = S S S 0 .
    eq sum((S S 0) : 0 : [])           = S S 0 .
    eq sum((S S 0) : (S 0) : [])       = S S S 0 .
    eq sum((S S 0) : (S S 0) : [])     = S S S S 0 .

    eq sum2([])                        = 0 .
    eq sum2(x : [])                    = x .
    eq sum2(x : y : [])                = x + y .
    eq sum2(x : y : z : [])            = x + (y + z) .

    eq prod([])                        = 0 .
    eq prod(0 : [])                    = 0 .
    eq prod((S 0) : [])                = S 0 .
    eq prod((S S 0) : [])              = S S 0 .
    eq prod(0 : 0 : [])                = 0 .
    eq prod(0 : (S 0) : [])            = 0 .
    eq prod(0 : (S S 0) : [])          = 0 .
    eq prod((S 0) : 0 : [])            = 0 .
    eq prod((S 0) : (S 0) : [])        = S 0 .
    eq prod((S 0) : (S S 0) : [])      = S S 0 .
    eq prod((S S 0) : 0 : [])          = 0 .
    eq prod((S S 0) : (S 0) : [])      = S S 0 .
    eq prod((S S 0) : (S S 0) : [])    = S S S S 0 .
endfm
```

A.3. Lists of Lists

```
fmod LISTOFNATLISTS is
    inc PLIST{NatList} *
```

```
   (op _:_ : List{INat} List{NatList} -> List{NatList} to _::_,
op [] : -> List{NatList} to nil,
op _++_ : List{NatList} List{NatList} -> List{NatList} to _+++_,
op shiftL : List{NatList} -> List{NatList} to ShiftL) .

op concat : List{NatList} -> List{INat} .
op map: : INat List{NatList} -> List{NatList} .
op inits : List{INat} -> List{NatList} .
op tails : List{INat} -> List{NatList} .
op subsequences : List{INat} -> List{NatList} .
op tails2 : List{NatList} -> List{NatList} .
op transpose : List{NatList} -> List{NatList} .
op weave : List{NatList} -> List{INat} .

vars x y z v : INat .
vars a11 a12 a13 a21 a22 a23 a31 a32 a33
     b11 b12 b13 b21 b22 b31 : INat .
vars xs ys zs : List{INat} .


eq concat (nil)                                   = [] .

eq concat ([] :: nil)                             = [] .
eq concat ((x : []) :: nil)                       = x : [] .
eq concat ((x : y : []) :: nil)                   = x : y : [] .

eq concat ([] :: [] :: nil)                       = [] .
eq concat ([] :: (x : []) :: nil)                 = x : [] .
eq concat ([] :: (x : y : []) :: nil)             = x : y : [] .

eq concat ((x : []) :: [] :: nil)                 = x : [] .
eq concat ((x : []) :: (y : []) :: nil)           = x : y : [] .
eq concat ((x : []) :: (y : z : []) :: nil)       = x : y : z : [] .

eq concat ((x : y : []) :: [] :: nil)             = x : y : [] .
eq concat ((x : y : []) :: (z : []) :: nil)       = x : y : z : [] .
eq concat ((x : y : []) :: (z : v : []) :: nil) = x : y : z : v : [] .


eq map: (x, nil)                    = nil .
eq map: (x, xs :: nil)              = (x : xs) :: nil .
eq map: (x, xs :: ys :: nil)        = (x : xs) :: (x : ys) :: nil .


*** map: als bk
eq inits ([])         = [] :: nil .
eq inits (x : [])     = [] :: (x : []) :: nil .
eq inits (x : y : []) = [] :: (x : []) :: (x : y : []) :: nil .


eq tails ([])         = [] :: nil .
eq tails (x : [])     = (x : []) :: [] :: nil .
eq tails (x : y : []) = (x : y : []) :: (y : []) :: [] :: nil .
```

```
*** requires max depth of at least 3
*** append und map: als bk,
eq subsequences ([]) = nil .
eq subsequences (x : []) = [] :: x : [] :: nil .
eq subsequences (x : y : []) =
    [] :: y : [] :: x : [] :: x : y : [] :: nil .


eq tails2 (nil)                        = nil .
eq tails2 (x : xs :: nil)              = xs :: nil .
eq tails2 (x : xs :: y : ys :: nil) = xs :: ys :: nil .
eq tails2 (x : xs :: y : ys :: z : zs :: nil) = xs :: ys :: zs :: nil .


*** transpose funktioniert fuer volle matrizen (min. 1 element und
*** alle reihen gleiche anzahl elemente) mit tails2 als bk
*** tails2 selber generieren: time off
*** beispiele: bis zu 3 spalten und reihen
*** max (3 reihen, 3 spalten) kann auch weggelassen werden

*** one row
eq transpose (a11 : [] :: nil) = a11 : [] :: nil .
eq transpose (a11 : a12 : [] :: nil) =
    a11 : [] ::
    a12 : [] :: nil .
eq transpose (a11 : a12 : a13 : [] :: nil) =
    a11 : [] ::
    a12 : [] ::
    a13 : [] :: nil .

*** two rows
eq transpose (a11 : [] ::
              a21 : [] :: nil) =
    a11 : a21 : [] :: nil .
eq transpose (a11 : a12 : [] ::
              a21 : a22 : [] :: nil) =
    a11 : a21 : [] ::
    a12 : a22 : [] :: nil .
eq transpose (a11 : a12 : a13 : [] ::
              a21 : a22 : a23 : [] :: nil) =
    a11 : a21 : [] ::
    a12 : a22 : [] ::
    a13 : a23 : [] :: nil .

*** three rows
eq transpose (a11 : [] ::
              a21 : [] ::
              a31 : [] :: nil) =
    a11 : a21 : a31 : [] :: nil .
eq transpose (a11 : a12 : [] ::
              a21 : a22 : [] ::
                 a31 : a32 : [] :: nil) =
        a11 : a21 : a31 : [] ::
        a12 : a22 : a32 : [] :: nil .
    eq transpose (a11 : a12 : a13 : [] ::
                 a21 : a22 : a23 : [] ::
                 a31 : a32 : a33 : [] :: nil) =
        a11 : a21 : a31 : [] ::
        a12 : a22 : a32 : [] ::
        a13 : a23 : a33 : [] :: nil .


    eq weave (nil) = [] .

    eq weave (a11 : [] :: nil) = a11 : [] .
    eq weave (a11 : a12 : [] :: nil) = a11 : a12 : []  .
    eq weave (a11 : a12 : a13 : [] :: nil) = a11 : a12 : a13 : [] .

    eq weave (a11 : [] ::
              a21 : [] :: nil) = a11 : a21 : [] .
    eq weave (a11 : a12 : [] ::
              a21 : [] :: nil) = a11 : a21 : a12 : [] .
    eq weave (a11 : [] ::
              a21 : a22 : [] :: nil) = a11 : a21 : a22 : [] .
    eq weave (a11 : a12 : [] ::
              a21 : a22 : [] :: nil) = a11 : a21 : a12 : a22 : [] .
    eq weave (a11 : a12 : a13 : [] ::
              a21 : a22 : [] :: nil) = a11 : a21 : a12 : a22 : a13 : [] .

    eq weave (a11 : [] ::
              a21 : [] ::
              a31 : [] :: nil) = a11 : a21 : a31 : [] .
    eq weave (a11 : a12 : [] ::
              a21 : [] ::
              a31 : [] :: nil) = a11 : a21 : a31 : a12 : [] .
endfm
```

A.4. Artificial Intelligence Problems

```
*** REASONING ****************************************************

fmod ANCESTOR-BIN is
        *** ancestor relation in binary trees
        *** not as predicate but output is the (possibly empty) path

        sorts Val BinTree Path InVec .

        ops a b c d e : -> Val [ctor] .

        op nil : -> BinTree [ctor] .
        op node : Val BinTree BinTree -> BinTree [ctor] .

        op nilp : -> Path [ctor] .
op __ : Val Path -> Path [ctor] .

op Ancestor : Val Val BinTree -> Path [metadata "induce"] .
op in : Val Val BinTree -> InVec [ctor] .

*** background knowledge
***
*** IsIn takes a val and a bin tree and returns the path from root
*** to val if the latter appears in the bin tree, empty path
    otherwise
op IsIn : Val BinTree -> Path [metadata ""] .
op in : Val BinTree -> InVec [ctor] .

*** Or takes to pathes, at least one of them empty, and returns the
*** non-empty path if it exists, the empty path otherwise
op _Or_ : Path Path -> Path [metadata ""] .
op in : Path Path -> InVec [ctor] .


*** solution:
*** 1. Ancestor(X, Y, nil) = nilp .
*** 2. Ancestor(X, Y, node(Z, L, R)) = IsIn(Y, node(Z, L, R)) if X
   == Z .
*** 3. Ancestor(X, Y, node(Z, L, R)) =
***             Ancestor(X, Y, L) Or Ancestor(X, Y, R) if X =/= Z .


vars X Y : Val .

*** examples
*** seem to be more examples than necessary prima facie, but they
    are
*** needed to get variables for all parameters in all cases
***
*** base case
eq Ancestor(X, Y, nil) = nilp .

*** case X == Z
eq Ancestor(a, b,
                 node(a, nil, node(c, node(b, nil, nil),
                     node(e, nil, nil)))) =
        a c b nilp .
eq IsIn(b, node(a, nil, node(c, node(b, nil, nil), node(e, nil, nil
    )))) =
        a c b nilp .

eq Ancestor(a, b, node(a, node(b, nil, nil), nil)) = a b nilp .
eq IsIn(b, node(a, node(b, nil, nil), nil)) = a b nilp .

eq Ancestor(b, a, node(b, node(d, node(a, nil, nil), nil), nil)) =
        b d a nilp .
eq IsIn(a, node(b, node(d, node(a, nil, nil), nil), nil)) =
        b d a nilp .
        eq Ancestor(b, a, node(b, nil, node(a, nil, nil))) = b a nilp .
        eq IsIn(a, node(b, nil, node(a, nil, nil))) = b a nilp .

        *** case X =/= Z
        eq Ancestor(a, b,
                                node(d, node(a, nil,
                                                                node(c,
                                                                   node(b,
                                                                    nil,
                                                                   nil),
                                                                   node(e,
                                                                    nil,
                                                                   nil))),
                                                nil)) =
                a c b nilp .
        eq (a c b nilp) Or nilp = a c b nilp .

        eq Ancestor(a, b, node(c, nil, node(a, node(b, nil, nil), nil))) =
                a b nilp .
        eq nilp Or (a b nilp) = a b nilp .

        eq Ancestor(b, a,
                                node(c, nil,
                                                node(b, node(d, node(a, nil
                                                   , nil), nil), nil))) =
                b d a nilp .
        eq nilp Or (b d a nilp) = b d a nilp .

        eq Ancestor(b, a, node(c, node(b, nil, node(a, nil, nil)), nil)) =
                b a nilp .
        eq (b a nilp) Or nilp = b a nilp .
endfm



*** PROBLEM SOLVING ***********************************************

fmod ROCKET is
        sorts Object OList State InVec .

        op nil : -> OList [ctor] .
        op __ : Object OList -> OList [ctor] .

        ops load unload : Object State -> State [ctor] .
        op move : State -> State [ctor] .

        *** elements in OList shall be loaded, moved to the moon and
            unloaded;
        *** assumption: rocket at earth and objects unloaded in initial
            state
        op Rocket : OList State -> State [metadata "induce"] .
        op in : OList State -> InVec [ctor] .

        *** solution
        *** Rocket(nil, S) = move(S) .
        *** Rocket((O Os), S) = unload(O, Rocket(Os, load(O, S)))

        vars O1 O2 : Object .
        var S : State .

        eq Rocket(nil, S) = move(S) .
        eq Rocket((O1 nil), S) = unload(O1, move(load(O1, S))) .
        eq Rocket((O1 O2 nil), S) =
                unload(O1, unload(O2, move(load(O2, load(O1, S))))) .
endfm


fmod CLEARBLOCK is
        sorts Block Tower State InVec .

        op table : -> Tower [ctor] .
        op __ : Block Tower -> Tower [ctor] .

        op putt : Block State -> State [ctor] .          *** put to table

        op ClearBlock : Block Tower State -> State [metadata "induce"] .
        op in : Block Tower State -> InVec [ctor] .

        *** solution
        *** ClearBlock(A, B T, S) = S if A == B
        *** ClearBlock(A, B T, S) = ClearBlock(A, T, putt(B, S))

        vars A B C : Block .
        var S : State .

        *** examples
        *** second example indicates that the block to be cleared need not
        *** be the bottom block
        eq ClearBlock(A, A table, S) = S .
        eq ClearBlock(A, A B table, S) = S .
        eq ClearBlock(A, B A table, S) = putt(B, S) .
        eq ClearBlock(A, C B A table, S) = putt(B, putt(C, S)) .
endfm



fmod TOWER is
        sorts Object Tower State InVec .

        *** objects are blocks and the table
        *** they are ordered wrt s
        op table : -> Object [ctor] .
        op s_ : Object -> Object [ctor] .
```

```
*** a tower is a list of blocks
*** table should always be the last object because the table
*** is the bottom of each tower
op | : -> Tower [ctor] .
op __ : Object Tower -> Tower [ctor] .

*** a state is a list of towers
op nil : -> State [ctor] .
op _,_ : Tower State -> State [ctor] .

*** put is used to stack blocks on blocks as well as to put
*** a block to the table
op put : Object Object State -> State [ctor] .

*** the goal is to construct a tower ordered wrt s with the given
*** object as the top,
*** i.e., ((s^3 table) (s^2 table) (s table) table |)
*** if s^3 is given
op Tower : Object State -> State [metadata "induce"] .
op in : Object State -> InVec [ctor] .

*** background knowledge: ClearBlock
op CB : Object State -> State [metadata ""] .

*** predicate: is the tower finished?
op IsTower : Object State -> Bool [metadata "pred nomatch"] .


*** solution
*** Tower(O, S) = S if IsTower(O, S)
*** Tower(O, S) =
***             put(O, Sub1(O, S), CB(O, CB(Sub1(O, S), Tower(Sub1(
    O, S), S))))
***                     if not(IsTower(O, S))
*** Sub1(s(O), S) = O .


eq Tower(table, nil) = nil .
eq Tower(s table, nil) = put(s table, table, nil) .
eq CB(table, nil) = nil .
eq CB(s table, nil) = nil .

eq Tower(s table, ((s table) (s s table) table | , nil)) =
        put(s table, table, ((s table) (s s table) table | , nil))
            .

eq Tower(table, ((s table) (s s table) table | , nil)) =
        ((s table) (s s table) table | , nil) .

eq Tower(s s s s table,
                 ((s s s s table) (s table) table | ,
                  (s s s table) (s s table) table | , nil)) =
        put(s s s s table, s s s table,
                        put(s s s table , s s table ,
                                put(s s table , s table ,
                                        put(s s s table , table ,
                                                put(s s s s table , table ,
                                                        ((s s s s table) (s
                                                             table) table |
                                                             ,
                                                         (s s s table) (s s
                                                              table) table
                                                             | , nil))))))
                                                             .

        eq Tower(s s s table ,
                         ((s s s s table) (s table) table | ,
                          (s s s table) (s s table) table | , nil)) =
                put(s s s table , s s table ,
                        put(s s table , s table ,
                                put(s s s table , table ,
                                        put(s s s s table , table ,
                                                ((s s s s table) (s table)
                                                    table | ,
                                                 (s s s table) (s s table)
                                                     table | , nil))))) .

        eq Tower(s s table ,
                         ((s s s s table) (s table) table | ,
                          (s s s table) (s s table) table | , nil)) =
                put(s s table , s table ,
                        put(s s s table , table ,
                                put(s s s s table , table ,
                                        ((s s s s table) (s table) table |
                                            ,
                                         (s s s table) (s s table) table |
                                             , nil)))) .

        eq Tower(s table ,
                         ((s s s s table) (s table) table | ,
                          (s s s table) (s s table) table | , nil)) =
                ((s s s s table) (s table) table | ,
                 (s s s table) (s s table) table | , nil) .

        eq Tower(table ,
                         ((s s s s table) (s table) table | ,
                          (s s s table) (s s table) table | , nil)) =
                ((s s s s table) (s table) table | ,
                 (s s s table) (s s table) table | , nil) .


        eq CB(table , ((s table) (s s table) table | , nil)) =
                ((s table) (s s table) table | , nil) .
        eq CB(s table , ((s table) (s s table) table | , nil)) =
                ((s table) (s s table) table | , nil) .
```

```
eq CB(s s s s table ,
                 put(s s s table , s s table ,
                        put(s s table , s table ,
                                put(s s s table , table ,
                                        put(s s s s table , table ,
                                                ((s s s s table) (s
                                                     table) table |
                                                     ,
                                                 (s s s table) (s s
                                                      table) table
                                                     | , nil))))))
                                                     =
        put(s s s table , s s table ,
                put(s s table , s table ,
                        put(s s s table , table ,
                                put(s s s s table , table ,
                                        ((s s s s table) (s table)
                                            table | ,
                                         (s s s table) (s s table)
                                             table | , nil))))) .

eq CB(s s s table ,
                 put(s s s table , s s table ,
                        put(s s table , s table ,
                                put(s s s table , table ,
                                        put(s s s s table , table ,
                                                ((s s s s table) (s
                                                     table) table |
                                                     ,
                                                 (s s s table) (s s
                                                      table) table
                                                     | , nil))))))
                                                     =
        put(s s s table , s s table ,
                put(s s table , s table ,
                        put(s s s table , table ,
                                put(s s s s table , table ,
                                        ((s s s s table) (s table)
                                            table | ,
                                         (s s s table) (s s table)
                                             table | , nil))))) .

eq CB(s s s table ,
                 put(s s table , s table ,
                        put(s s s table , table ,
                                put(s s s s table , table ,
                                        ((s s s s table) (s table)
                                            table | ,
                                         (s s s table) (s s table)
                                             table | , nil))))) =
        put(s s table , s table ,
                put(s s s table , table ,
                    put(s s s s table, table,
                            ((s s s s table) (s table) table |
                                ,
                             (s s s table) (s s table) table |
                                , nil)))) .

eq CB(s s table,
                put(s s table, s table,
                        put(s s s table, table,
                                put(s s s s table, table,
                                        ((s s s s table) (s table)
                                            table | ,
                                         (s s s table) (s s table)
                                            table | , nil))))) =
        put(s s table, s table,
                put(s s s table, table,
                        put(s s s s table, table,
                                ((s s s s table) (s table) table |
                                    ,
                                 (s s s table) (s s table) table |
                                    , nil)))) .

eq CB(s s table,
                put(s s s table, table,
                        ((s s s s table) (s table) table | ,
                         (s s s table) (s s table) table | , nil)))
                                =
        put(s s s table, table,
                put(s s s s table, table,
                        ((s s s s table) (s table) table | ,
                         (s s s table) (s s table) table | , nil)))
                                .

eq CB(s table,
                ((s s s s table) (s table) table | ,
                 (s s s table) (s s table) table | , nil)) =
        put(s s s s table, table,
                ((s s s s table) (s table) table | ,
                 (s s s table) (s s table) table | , nil)) .


*** predicate implementation

op $IsTower : Object Tower -> Bool .

vars O O2 : Object .
var T : Tower .
var S : State .

eq IsTower(table, S) = true .
eq IsTower(s O, ((s O) T , S)) = $IsTower(O, T) .
eq IsTower(s O, (table | , S)) = IsTower(s O, S) .
eq IsTower(s O, nil) = false .
```

```
        eq IsTower(s O, ((s O2) T , S)) = IsTower(s O, (T , S)) [owise] .
        eq $IsTower(table, table |) = true .
        eq $IsTower(s O, (s O) T) = $IsTower(O, T) .
        eq $IsTower(O, T) = false [owise] .
endfm



fmod HANOI is
        sorts Disc Peg State InVec .

        op 0 : -> Disc [ctor] .                 *** smallest disc
        op s_ : Disc -> Disc [ctor] .   *** next bigger disc

        *** move disc from one peg to another in a current state
        op move : Disc Peg Peg State -> State [ctor] .

        *** move tower up to specified disc from start peg via aux peg to
        *** goal peg in a given state
        op Hanoi : Disc Peg Peg Peg State -> State [metadata "induce"] .
        op in : Disc Peg Peg Peg State -> InVec [ctor] .

        *** solution
        *** Hanoi(0, Src, Aux, Dst, S) = move(0, Src, Dst, S)
        *** Hanoi(s D, Src, Aux, Dst, S) =
        ***             Hanoi(D, Aux, Src, Dst,
        ***                     move(s D, Src, Dst,
        ***                             Hanoi(D, Src, Dst, Aux, S)))

        var S : State .
        vars Src Aux Dst : Peg .

        eq Hanoi(0, Src, Aux, Dst, S) = move(0, Src, Dst, S) .
        eq Hanoi(s 0, Src, Aux, Dst, S) =
                move(0, Aux, Dst, move(s 0, Src, Dst, move(0, Src, Aux, S))
                   ) .
        eq Hanoi(s s 0, Src, Aux, Dst, S) =
                move(0, Src, Dst,
                        move(s 0, Aux, Dst,
                                move(0, Aux, Src,
                                        move(s s 0, Src, Dst,
                                                move(0, Dst, Aux,
                                                        move(s 0, Src, Aux,
                                                                move(0, Src
                                                                   , Dst,
                                                                   S))))))
                                                                   ) .
endfm
```

```
*** NATURAL LANGUAGE PROCESSING ***********************************

fmod GENERATOR is
        *** generates sentences up to particular depth

        sorts Cat CList Depth InVec .

        ops d n v : -> Cat [ctor] .
        op ! : -> CList [ctor] .
        op __ : Cat CList -> CList [ctor] .

        op 1 : -> Depth [ctor] .
        op s_ : Depth -> Depth [ctor] .

        op Sentence : Depth -> CList [metadata "induce"] .
        op in : Depth -> InVec [ctor] .

        *** original grammar
        *** S -> NP VP
        *** NP -> D N
        *** VP -> V NP | V S

        *** solution
        *** Sentence(1) = (d n v d n !)
        *** Sentence(s N) = (d n v Sentence(N))

        eq Sentence(1) = (d n v d n !) .
        eq Sentence(s 1) = (d n v d n v d n !) .
        eq Sentence(s s 1) = (d n v d n v d n v d n !) .
endfm
```

Nomenclature

$(x_i)_{i\in I}$, (x_i)	Family of elements x_i with index set I
$[m]$	$\{n \in \mathbb{N} \mid 1 \leq n \leq m\}$
$[x]_\sim$	Equivalence class of x by $\sim$
$\chi_B(r, \Phi)$	Successor-rule sets and specifications; union of χ_{split}, χ_{sub}, χ_{smplCall}, χ_{call}
χ_{call}	Function call operator
χ_{smplCall}	Simple call operator
χ_{split}	Rule splitting operator
χ_{sub}	Subproblem operator
$\mathbb{N}$, $\mathbb{Z}$	Natural numbers and integers, respectively
$\mathcal{D}_R$, $\mathcal{C}_R$	Defined function symbols and constructors of a CS R, respectively
$\mathfrak{P}(X)$	Powerset of X
$\mathit{Defines}(r)$	Defined function symbol at the root of the LHS of a constructor system rule
id	Identity function
$\mathit{Lhss}(R)$	The set of all LHSs of a CS R
$\mathit{Lhs}(r)$, $\mathit{Rhs}(r)$	LHS and RHS of a rule r, respectively
$\mathit{Node}(t, p)$	Symbol of term t at position p
$\mathit{Pos}(t)$	Set of positions of the term t
$\mathit{Var}(t)$	Variables of term t
$\Phi(r)$, $\Phi(r)$	Specification subsets associated with a rule and a function, respectively
$\sim$, $\equiv$	Equivalence relation
$\xrightarrow{*}_R$	Rewrite relation of a TRS R
$\Xi_B(\langle P, \Phi\rangle)$	Set of successor CSs and corresponding specifications of candidate CS P and specification Φ with respect to background CS B

$\mid X \mid$	Cardinality of the set X
$e \succeq e'$	Expression e *subsumes* expression e' (where "expression" refers to terms, predicates, equations, or rules), i.e., there is a substitution σ such that $e\sigma = e'$
$t\|_p$	Subterm of term t at position p
$X/\sim$	Quotient set of X by $\sim$; the set of all equivalence classes of X by $\sim$
T_Σ	Ground Σ-Terms
$\mathcal{P}_{\Phi,B}$	IGOR2 problem space with respect to initial specification Φ and background CS B
$T_\Sigma(\mathcal{X})$	Σ-Terms over variables $\mathcal{X}$

Index