Hidden Logic

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science

by

Grigore Roșu

Committee in charge:

Professor Joseph A. Goguen, Chairperson
Professor Samuel R. Buss
Professor Kenneth Kreutz-Delgado
Professor Victor Vianu
Professor S. Gill Williamson

2000
Copyright
Grigore Roșu, 2000
All rights reserved.
# TABLE OF CONTENTS

Table of Contents .......................................................... iii

List of Figures ............................................................. v

Acknowledgments ........................................................... vi

Abstract ........................................................................... vii

I  Introduction .................................................................... 1  
   A. Purpose and Motivation .............................................. 2  
   B. Overview of the Thesis .............................................. 4  
   C. Some Literature and Historical Remarks ....................... 12

II Preliminaries ............................................................... 22  
   A. Sets, Functions, Relations .......................................... 23  
   B. Many-Sorted Signatures ............................................. 23  
   C. Many-Sorted Algebras .............................................. 24  
   D. Term Algebras ....................................................... 24  
   E. Many-Sorted Equational Logic ..................................... 26

III Basics of Hidden Logic .................................................. 28  
   A. Hidden Signatures .................................................... 28  
   B. Hidden Algebra ...................................................... 32  
   C. Experiments and Contexts .......................................... 35  
   D. Behavioral Equivalence and Congruent Operations .......... 38  
   E. Behavioral Satisfaction ............................................. 41  
   F. Behavioral Specification ............................................ 44

IV Behavioral Reasoning ...................................................... 49  
   A. Hidden Equational Deduction ..................................... 49  
   B. Coinduction ........................................................... 54  
      1. Coinduction Method ................................................ 54  
      2. Examples .................................................................. 54  
   C. Cobases and $\Delta$-Coinduction .................................... 66  
      1. Complete Sets of Observers ..................................... 66  
      2. Strong Cobases ..................................................... 68  
      3. Cobases in General ................................................ 70  
      4. $\Delta$-Coinduction .................................................. 72  
   D. Equivalent Behavioral Specifications ............................. 73  
      1. Reducing the Number of Behavioral Operations .......... 73  
   E. Circular $\Delta$-Coinduction ........................................... 76  
      1. Limitations of $\Delta$-Coinduction .............................. 77  
      2. Circular $\Delta$-Coinduction ...................................... 80  
   F. Proving Congruence of Operations ............................... 83
LIST OF FIGURES

I.1 Two classes of hidden logics. ............................................. 5
I.2 Fixed-data hidden logics. .................................................. 8
I.3 Loose-data hidden logics. .................................................. 9

III.1 ADJ-like diagram for cells. .............................................. 29
III.2 ADJ-like diagram for flags. ............................................ 30
III.3 ADJ-like diagram for sets. ............................................. 31
III.4 ADJ-like diagram for nondeterministic stacks. ......................... 31
III.5 A Γ-context. ............................................................... 35
III.6 Γ-behavioral equivalence of a, a′. ..................................... 39

IV.1 Hidden equational deduction system. .................................... 50
IV.2 General coinduction. ...................................................... 54
IV.3 A term in $T_{Σ′}(Γ′, Δ; • : h, Z)$. ..................................... 69
IV.4 $Δ$-coinduction inference rule. ......................................... 72
IV.5 $Δ^C$-coinduction inference rule. ....................................... 82
IV.6 Method for proving congruence of operations. ......................... 86

V.1 Implementing behavioral rewriting by standard term rewriting. ........ 109
V.2 Behavioral coinductive rewriting algorithm. .......................... 116
V.3 Extensionality rule. ....................................................... 119
V.4 Circular coinductive rewriting algorithm. ............................. 124

VI.1 Unhiding the signature of sets. ........................................ 173
VI.2 Unhiding the signature of streams. ................................... 174
VI.3 Proving behavioral properties by unhiding. .......................... 177
VI.4 Complexity hierarchy. .................................................. 189
I would like to thank to my advisor, Joseph Goguen, for his continuous encour-
agements, suggestions and friendship, without which, this thesis would not have been
existed. Special thanks to my colleagues and friends in the Meaning and Computation
Laboratory, Kai Lin and Bogdan Warinschi, who were always open to discuss difficult
problems or to play GO when we needed to relax and have fun.

My professors and advisors at the University of Bucharest, Virgil Căzănescu,
Sergiu Rudeanu, and Gheorghe Ștefănescu, introduced me to the wonderful world of
algebra and category theory, and showed me the door toward my current scientific belief.
Thank you!

The work in this thesis was greatly influenced and motivated by research done
within the CafeOBJ project at JAIST in Japan by Kokichi Futatsugi and Răzvan Dia-
conescu, to whom I warmly thank. Of great influence were also the pleasant meetings
with researchers in the coalgebra field, and I’d like to thank Bart Jacobs, Jan Rutten,
and Horst Reichel for organizing CMCS. Sporadic discussions with other researchers at
conferences, workshops, or personal meetings, directly or indirectly helped me in my
research; I wish to thank at least to the following: Michel Bidoit, Samuel Buss, Co-
rina Cîrstea, Andrea Corradini, Alexander Kurz, Rolf Hennicker, Ugo Montanari, Peter
Padawitz, and Andrei Tarlecki.

I would also like to warmly thank to all scientists who directly influenced my
research, publishing joint papers: Samuel Buss, Virgil Emil Căzănescu, Joseph Goguen,
Kai Lin, Akira Mori, Akiyoshi Sato, Nathan Segerlind, and Bogdan Warinschi. Some of
the results in our joint papers are included in this thesis and cited at the appropriate
places.

Special thanks to my wife, Luiza, for helping me find philosophical interpreta-
tions to some of the results in the thesis.
ABSTRACT OF THE DISSERTATION

Hidden Logic

by

Grigore Roşu

Doctor of Philosophy in Computer Science

University of California, San Diego, 2000

Professor Joseph A. Goguen, Chair

Cleverly designed software often fails to satisfy its requirements strictly, but instead satisfies them behaviorally, in the sense that they appear to be satisfied under every experiment that can be performed on the system. It is therefore becoming increasingly important to develop powerful techniques for behavioral specification and verification of systems, especially in the design stage, where most of the errors appear.

The general context of this thesis is formal methods for software and/or hardware development. We will present a promising new logic together with a language to support it, with applications to various branches of computer science, especially to the specification and automated verification of object-oriented and concurrent systems.

The thesis can be roughly divided into three major parts. The first is dedicated to presenting hidden logic in an easy going and intuitive way, with many examples, to developing sound inference rules, including a series of non-trivial coinduction rules, and to presenting a specification language in the OBJ family, called BOBJ, supporting hidden logic and reasoning. The second part is on automation of behavioral reasoning; more precisely, it introduces the techniques called behavioral rewriting, behavioral coinductive rewriting and circular coinductive rewriting, which are implemented in BOBJ. The third and the last part relates to more theoretical aspects of the logic, including relationships with other formalisms and theories, such as information hiding, coalgebra, institutions, Birkhoff-like axiomatizability, and incompleteness.

vii
Chapter I

Introduction

Software and hardware systems are rapidly growing in scale and functionality, leading to ever greater possibilities for subtle errors. Since automated systems are being increasingly used in many places in society, these errors may produce significant losses, including human lives. A major goal of software and hardware engineers is to build reliable systems despite their complexity. Unfortunately, this seems to be a very difficult task not only because of the speed with which technology changes, but especially because of the change in requirements and also because of the complexity of system requirements, the lack of language expressivity and of understanding of specifications.

One common way to approach this problem is by using formal methods, which are mathematically-based languages, techniques and tools for specifying and verifying systems. Their merit is to more or less automatically reveal inconsistencies, ambiguities and exceptions in specifications that may be very expensive or impossible to detect otherwise. Formal methods have already demonstrated success in specifying and verifying safety-critical software, protocol standards and hardware designs. However, it is worth mentioning that formal methods do not a priori guarantee correctness, and that sometimes it can be more difficult to specify than to implement a system, even though specifications can greatly increase our understanding of a system and thus our confidence in its correctness. We refer the interested reader to [28] for a survey and state of the art of formal methods.

The work in this thesis is part of a team effort in the Meaning and Computation Laboratory at the University of California at San Diego, to design, implement, evaluate
and popularize formal methods tools for \emph{behavioral} specification and verification. Hidden logic is the underlying logic of our system called Tatami \cite{61, 59, 58}, and in particular of its component called BOBJ\textsuperscript{1}, which is a behavioral specification and verification language in the OBJ family. Even though the examples in this thesis use the BOBJ notation, our goal is not to present BOBJ in detail, but rather its underlying logic; however, we will describe BOBJ’s features gradually, as needed in examples.

The latest information on hidden logic, including our most recent papers, links to related work by other researchers, and links to online tutorial material, can be found at \url{www.cs.ucsd.edu/users/goguen/projs/halg.html}, the hidden algebra home-page.

\section{Purpose and Motivation}

Algebra can be useful in many different ways in software and hardware engineering, including specification, validation, language design, and underlying theory. Specification and verification are important for the practical development of reliable systems, advances in language design can help improve the state of the art, and theory gives a better understanding of the whole enterprise, and moreover can help with building new tools to increase automation. The utility, vitality and growing links with other areas all suggest the existence of a significant emerging field, that might be called “algebraic engineering” \cite{66}. Although its mathematical roots lie within the framework of universal (also called “general”) algebra, as pioneered by Noether, Birkhoff, Tarski and others, it is part of computer science. A major goal of this research, following Goguen and Malcolm’s “Hidden Agenda” \cite{66}, is to extend universal algebra to account for the realities of modern software. This thesis is largely focused on foundational aspects, but it also takes seriously the task of providing real support for practical software engineering.

Today’s software systems are often concurrent and distributed, with interfaces that encapsulate the local states of objects and the operations that modify them. These features are the core of what has come to be called the “object paradigm,” which may be described as comprising:

\textsuperscript{1}BOBJ is a shorthand for “Behavioral OBJ,” and “OBJ” \cite{80} is a generic name for a broad spectrum of algebraic programming and specification languages, based on order sorted equational logic, possibly enriched with other logics.
1. objects, with local state, plus operations that modify or observe them;
2. classes, to classify objects through an inheritance hierarchy; and
3. concurrent distributed execution.

The object paradigm adds to the older notion of abstract machine support for code reuse through inheritance, and an affinity for concurrency. The hidden logic framework described in this thesis pays particular attention to the object paradigm, but it also considers ordinary programs and their components, since these can be regarded as abstract machines. An important motivation for pursuing this research is the observation that cleverly designed software and hardware often fails to satisfy its requirements strictly, but instead satisfies them behaviorally, in the sense that they appear to be satisfied under every experiment that can be performed on the system.

Hidden algebra was introduced by Goguen for the first time in [50] as a generalization of many-sorted algebra to give algebraic semantics for the object paradigm. It was developed further in [57, 62, 20, 65] among other places. Its distinctive feature is that sorts are split between visible and hidden, the visible sorts being for data, while the hidden sorts are for objects. A model, called a hidden algebra, which can be regarded as an abstraction of an implementation, consists of the universe of all possible states of an object, together with concrete interpretations of attributes and methods as functions from states to data or from states to states, respectively (hence, an attribute “observes” a state returning a visible value, while a method “modifies” a state). In other words, a hidden algebra is an algebra which protects a fixed data universe. The behavioral aspect makes hidden algebra more suitable for actual computing practice than standard algebra, especially because of the complexity and dynamic behavior of contemporary software systems.

Hidden logic is a generic name for various logics derived from or closely related to hidden algebra, giving sound rules for behavioral reasoning in all approaches, with a high potential for automation. We believe that hidden logic is the natural next step in the evolution of algebraic semantics and its first order proof technology.

The older literature on formal methods mainly addresses code verification, but this can be very difficult in practice. Moreover, empirical studies have shown that little of the cost of software comes from coding errors: most of the cost of correcting programs
comes from design and requirements errors [19]. Because many programs are written in languages with complex ugly semantics, are very poorly documented, and are very large, it is usually an enormous effort to verify them, and it is rarely worth the effort. For that reason, we concentrate on design and on verification of specifications, rather than verification of actual code.

1.B Overview of the Thesis

We assume the reader familiar with the basics of first order logics and set theory, as well as with various techniques used in theorem proving. Chapter II reviews the basic notions, notations, conventions and results which are important for understanding the thesis.

Many results presented in this thesis are original and have appeared in papers recently published or submitted for publication, which will be properly mentioned at the appropriate places. Some of the results are more practical, while others are more theoretical, for which reason we have structured the thesis as follows: Chapters III and IV introduce hidden logic, its syntax and semantics, as well as sound inference rules; Chapter V explores various techniques and algorithms for automation of behavioral reasoning; and Chapter VI, the last one, presents the most theoretical results. We will now briefly discuss the main results in these chapters.

Chapter III introduces the basic ingredients of hidden logic. The distinctive features of hidden logic are that the sorts are split in two disjoint sets, of visible and hidden sorts, and that the operations are also split in two disjoint sets, of behavioral and non-behavioral (or non-congruent) operations. The behavioral operations generate the experiments, which are visible terms over a special variable which only occurs once. The experiments generate a behavioral equivalence on each hidden algebra, by defining two states to be equivalent if and only if they cannot be distinguished by any experiment evaluated in that model. Obviously, the behavioral equivalence relation is preserved by the behavioral operations in any model, but it may not be preserved by the non-congruent operations in some models.

Following an idea in [22], we distinguish two large classes of hidden logics,
depending on whether a fixed data universe, of built-ins, is assumed to be fixed or not.

![Diagram showing two classes of hidden logics: Fixed-data Hidden Logic and Loose-data Hidden Logic](image)

**Figure I.1:** Two classes of hidden logics.

The first versions of hidden logic were slight generalizations of hidden algebra, so they were fixed-data, but we recently observed that all our current inference rules are in fact sound for the larger class of models which do not protect the data. Since there were already some loose-data versions of hidden logic, such as **coherent hidden algebra** in Japan [37, 40] and **observational logic** in France and Germany [11, 13, 87], we decided to analyze both classes in the thesis. Nevertheless, the fixed-data hidden logics seem desirable because real applications use standard booleans and integers rather than arbitrary models of some theory. For example, the alternating bit protocol cannot be shown correct unless implementations which do not distinguish 0 from 1 are forbidden.

The original contributions in Chapter III consist basically of uniform generalizations of other approaches in use. More precisely, we extend the original fixed-data hidden algebra to allow operations with multiple hidden arguments and to allow operations which are not behavioral. The multiple hidden arguments are needed, for example, to properly model the so called **binary methods** in modern programming languages, that is, methods taking two objects as arguments. Operations which are not behavioral seem necessary for applications like **length** for lists implementing sets. On the other hand, our loose-data hidden logic extends coherent hidden algebra by also allowing behavioral operations with many hidden arguments, and extends observational logic by allowing operations which are not behavioral. Interestingly, the behavioral equivalence is still the largest equivalence which is identity on visible sorts and is compatible with the behavioral operations (see Theorem 18) in all these hidden logics, a result which was initially thought to depend on a fixed data universe and monadic (in hidden arguments) oper-
lations\textsuperscript{2}, thus soundly extending the coinduction proof method to all the hidden logics currently in use. Of technical importance is a so called theorem of hidden constants (Theorem 25), which justifies the elimination of hidden universal quantifiers, which is important in proofs.

Chapter IV presents our current inference rules for behavioral reasoning which are sound in all the hidden logics. A first interesting observation is that equational and first-order reasoning are no longer sound, because some operations may not preserve the behavioral equality, thus making the inference rule of congruence unsound. We adapt a basic five rule equational deduction system accordingly to deal with non-congruent operations, and call it hidden equational deduction; if all the operations are congruent, then it becomes exactly the equational deduction system.

Induction is a well established technique to prove strict equalities of terms in the framework of abstract data types. Dually, coinduction is a technique for proving behavioral equalities in the framework of object types. We recommend [98] for a nice exposition of the duality between induction in algebra and coinduction in coalgebra. Unfortunately, coinduction in its pure form requires human intervention which is clearly an inconvenience, especially when automated proving is aimed. A major goal of this thesis and of our research in general, is to automate coinduction as much as possible. The basis of an induction, which is a set of operations or derived operations generating all the terms, plays a major role in inductive proofs, and a good choice of basis can enormously simplify a proof. One of our first contributions was the notion of cobasis (introduced in 1998 in [137]), which is a set of operations which (co)generate the behavioral equivalence on terms, together with a method to automatically obtain cobases using a syntactic criterion.

Given a cobasis $\Delta$, coinduction can be applied automatically; we capture that in a new inference rule called $\Delta$-coinduction, which was first presented in [137, 71, 72, 134]. Intuitively, $\Delta$-coinduction says that in order to show that two states $t, t'$ are behaviorally equivalent, it suffices to show that $\delta(t)$ and $\delta(t')$ are behaviorally equivalent\textsuperscript{3} for each $\delta \in \Delta$. This method can be easily automated by recurrently applying operations in

\textsuperscript{2}This restrictive framework is called “monadic fixed-data hidden logic” in Chapter VI.

\textsuperscript{3}We assumed that $\delta$ has only one argument here, for simplicity. See Subsection IV.C.4 for a complete definition.
the cobasis on top of states until, using hidden equational deduction on the axioms of the behavioral theory, two equivalent states are obtained. Unfortunately, this process may not terminate, and the reason is twofold: either bigger and bigger terms of the form $\delta_1(\delta_2(\ldots(\delta_n(t))\ldots))$ are being generated and the available rules cannot reduce them anymore, or it cycles because of circular definitions in the specification, that is, operations that are defined in terms on themselves (see Subsection IV.E.1 for some suggestive examples). Our experience so far is that the first situation does not occur in practical situations, or, if it appears, then something is missing or wrong in the specification. 

For this reason we focused on specifications with circularities. *Circular $\Delta$-coinduction*, first introduced in [139, 60], is a new technique for behavioral reasoning that extends $\Delta$-coinduction to specifications with circularities; we will also call it $\Delta^{C} -$coinduction. $\Delta$-coinduction and $\Delta^{C} -$coinduction are our main contribution to behavioral reasoning, and we hope that they are only the starting point toward stronger and more general proof methods in hidden logics. They are implemented and fully operational in BOBJ, a behavioral specification and verification system in the OBJ family. The interested reader can consult [58, 59, 61] and also the official BOBJ homepage on web at the URL address http://www-cse.ucsd.edu/groups/tatami/bobj/ for more on the evolution and Kai Lin’s implementation of BOBJ within the Tatami project and for interesting examples that push BOBJ to its limits.

In Chapter V, we further explore algorithms for automating behavioral reasoning. We present some technologies and algorithms which have been successfully implemented in BOBJ, including behavioral rewriting, behavioral coinductive rewriting, and circular coinductive rewriting. Our work on behavioral rewriting is inspired by and generalizes work done in the CafeOBJ project (see, for example, [37]), but behavioral coinductive rewriting [134] and circular coinductive rewriting [60] are original and we are not aware of them being implemented in any other system but BOBJ. Much of the work is done at the abstract level of $\Omega$-abstract rewriting systems, thus allowing us to apply our rewriting methods to other theories, such as the $\lambda$-calculus. *Behavioral rewriting* is for hidden equational deduction what term rewriting is for equational logic, and we show how behavioral rewriting can be non-trivially simulated by standard term rewriting, by considering a dynamic coloring of operations in terms. This has been
implemented in BOBJ. Behavioral coinductive rewriting combines behavioral rewriting with $\Delta$-coinduction, and significantly increases its power. Circular coinductive rewriting combines circular $\Delta$-coinduction with behavioral rewriting, thus giving birth to our strongest method for behavioral proving. We have successfully used it in various examples, including lazy functional programming examples, behavioral refinements, regular expressions, and the $\lambda$-calculus.

The last part of the thesis, Chapter VI, is the most technical, and consequently assumes the reader familiar with more mathematics, including more advanced concepts from algebra, and the basics of category theory and recursion theory. It contains many original results published in journal papers and conference proceedings. We start by showing that the models of hidden logic form a category, but that that category does not have nice properties in the general case of hidden logic. In particular, there are no final models, no initial models, no coproducts and no products. However, under some constraints restricting hidden logic to what we call monadic fixed-data hidden logic, we show the known fact (see, for example [107, 24]) that its models are isomorphic to a category of coalgebras, thus having many of the “coalgebraic” nice categorical properties, including final models and coproducts.

Following a plan started in [22], we distinguish various versions of fixed-data hidden logics in this chapter, each having its own interesting properties:

![Diagram of fixed-data hidden logics]

Figure I.2: Fixed-data hidden logics.
The meaning of arrows is that the target logic can be seen as a special case of the source logic, under some appropriate constraints. Therefore, any positive result for an upper logic (such as sound deduction, automated proving, etc.) propagates downward, while the negative results (such as incompleteness) propagate upward. The names are suggestive: monadic fixed-data hidden logic assumes that all behavioral operations have exactly one argument of hidden sort; finite fixed-data hidden logic assumes a finite fixed data algebra; flat fixed-data hidden logic assumes that the data algebra has no operations, that is, it is just a set; boolean fixed-data hidden logic is a special case of all the previous ones, in the sense that its data algebra is the two element set \{true, false\}.

Similarly, there can be distinguished at least three interesting loose-data hidden logics. Two of them, coherent hidden algebra and observational logic are much used by other scientists, e.g. [37, 40, 11, 13, 87]. The third one, semi-boolean loose-data hidden logic, is interesting, and is introduced as the simplest loose-data hidden logic known, having only one visible operation, which is the constant true:

![Diagram of loose-data hidden logics](image)

Figure I.3: Loose-data hidden logics.

Institutions were introduced by Goguen and Burstall [55] to formalize the informal notion of logical system. Having as a basic intuition Tarski’s classic semantic definition of truth [152], institutions also have the possibility of translating sentences and models along signature morphisms, with respect to an axiom called the satisfaction condition, which says that truth is invariant under change of notation. We show
in Subsection VI.A.4 (see also [71]) that hidden logic can be organized as institution in at least two interesting ways. The first follows the institution of the initial hidden algebra in [50], the coherent hidden algebra approach in [37, 40], and the observational logic institution in [87], while the second seems more promising for future research. The difference between the two institutions is that the first considers the declaration that a certain operation is behavioral as part of the signature, while the second considers it as a sentence.

A natural question when one sees a new logic is “how expressive is it?”, or in other words, “what’s the definitional power of its sentences?”. Garret Birkhoff [17] was the first who formalized this informal question, and proved in 1935 that a class of (universal) algebras is definable by equations if and only if it is closed under certain common operations, such as subalgebra, quotient algebra and product algebra. He called such a class a *variety*. Following Birkhoff’s idea, we introduce in Section VI.B the notion of *hidden variety* as a class of models closed under coproducts, quotients, sources of morphisms\(^4\) and representative inclusions\(^5\), and then show that a class of models is a hidden variety if and only if it is definable by behavioral equations. The work in Section VI.B is done at an abstract categorical level, including coalgebra and thus monadic hidden algebra as special cases, and it was previously published in [133, 136]. We are not aware of any similar characterization result for loose-data hidden logics.

Information hiding is an important technique in modern programming and specification. Hiding implementation details in programming allows an increased level of abstraction thus reducing the number of errors and giving a better understanding of a program, while hiding operations in specifications unexpectedly enlarges the declarational power of a specification language. In Section VI.C, we show that a hidden specification in any hidden logic can be “unhidden”, thus generating an ordinary algebraic specification over a larger signature, and then show that a hidden algebra behaviorally satisfies the hidden specification iff it ordinarily satisfies the unhidden specification where the auxiliary symbols are kept private. In the case of loose-data hidden logics, it is equivalent to saying that a hidden specification is semantically equivalent to information hiding into

\(^4\)Closure under sources of morphisms means that the source of a morphism is in the class whenever its target is in the class. This notion is stronger than the closure under submodels.

\(^5\)See Definition 140.
an ordinary specification, thus emphasizing the intuition that “behavior is information hiding”. An interesting observation is that the “unhiding” operation is algorithmical and can be easily implemented in an algebraic specification language, such as OBJ3 [80] or PVS [149]. It seems though (see Subsection VI.C.2), that proofs done by unhiding are at least as difficult and ugly as proofs done by context induction [86, 12, 44], so our result does seem to be of more theoretical relevance rather than practical importance. We find proofs by coinduction more elegant than proofs by context induction.

It is of great practical importance to have automated tools to check specifications for inconsistencies. Even though any loose-data hidden theory admits models, a trivial model being the one having one-element carriers, it is very easy to write inconsistent fixed-data hidden theories. Section VI.D discusses some of the types of inconsistency that can occur and then provides a consistency criterion. The material in this section is unpublished and was inspired by [67].

Another natural question is whether hidden logic is complete, that is, if there is any deduction system that can prove any (behavioral) truth. In spite of a completeness result by Corradini [30] (see also [25]) for a restrictive type of hidden logic⁶, it turns out that hidden logics are not complete in general. Section VI.E is dedicated to incompleteness results for hidden logics that we have recently published [22]. We show that both boolean fixed-data and semi-boolean loose-data hidden logics admit no complete deduction system, thus making all the hidden logics incomplete. Moreover, we give upper bounds for both the finite and the flat fixed-data hidden logics, as well as for the congruent loose-data hidden-logic, by showing that they are \( \Pi^0_2 \)-complete. If no restrictions are assumed for the data, then the behavioral satisfaction problem in fixed-data hidden logics can be even outside the arithmetic hierarchy.

Chapter VII draws some conclusions and sketches some future directions of research not mentioned in previous chapters.

---

⁶It is a special case of monadic hidden logic where either all the equations are of visible sort or else the satisfaction is strict.
I.C Some Literature and Historical Remarks

An adequate discussion of the complex historical and technical relations among the many approaches to algebraic and behavioral specification within which hidden logic evolved does not seem to be possible, but we will do our best to be sound, if not complete.

Universal Algebra

The roots of universal algebra seem to originate in some early lectures by Emmy Noether. However, most mathematicians agree that the famous Birkhoff Variety Theorem [17] in 1935, which states that a class of algebras is equationally definable if and only if it is closed under subalgebras, quotients and products, was the decisive starting point of universal algebra and one of the most beautiful algebraic results. Similar results were rapidly obtained for other classes of formulas, thus yielding a systematization of universal algebra. We find [154, 158] good starting references for readers interested in more on the evolution and history of universal algebra, and recommend [157] for a more computing oriented approach.

In computer science, an encapsulated data structure together with its accompanying operations is called an abstract data type. David Parnas [124] seems to have been the first software engineer to recognize (in 1972) that operations should be associated with data representations, “in the same way in which functions are associated with sets in universal algebra” as Goguen [47] emphasizes in 1975. Since then, computer science has made some significant contributions to universal algebra. We only mention a few well-known ones: completeness of many-sorted algebra [69, 122]; initial algebra semantics for abstract data types [47, 73, 76, 74, 79]; various approaches to order-sorted algebra [48, 70, 111, 151, 150, 125] (see [56] for a survey); and of course, hidden-sorted algebra or simply hidden algebra [50, 57, 65, 71], which will be amply discussed in this thesis.

Hidden Algebra

Hidden logic originates in a paper by Goguen [50] in 1989, under the name of hidden algebra. For that reason, we often use the name hidden algebra logic or simply hidden algebra instead of hidden logic in the thesis. Hidden algebra was built on earlier
algebraic work on abstract data types [47, 74, 79, 43], and is a natural extension of prior work by Goguen and Meseguer on “abstract machines” [68, 110].

Actually, some of the intuitions underlying hidden algebra seem to go back more than fifty years. Hidden algebra naturally generalizes automata theory, in the sense that the equivalence of states of an automaton can be viewed as a behavioral equivalence; there is a very early treatment of behavioral equivalence in work by Moore [114] in 1956. There is also a striking analogy between behavioral equivalence in hidden algebra and what is known as Leibniz congruence in the filled called abstract algebraic logic. The models of abstract algebraic logic, known as logical matrixes, are algebras together with a fixed set of elements in which predicates are interpreted (this designated set constitutes what we call the “data universe” in (fixed-data) hidden algebra, that is, the universe in which different elements can be distinguished). Then two formulas \( t, t' \) (or “hidden terms” or “states” in our terminology) are Leibniz equal if and only if they have exactly the same provable properties, that is, any theorem (which we would regard as an “experiment”) that contains \( t \) as a subformula remains a theorem when \( t \) is replaced by \( t' \). The fundamental concepts of matrix semantics were introduced by Łós [103] in 1949, and then extensively developed in Poland by many logicians. Although motivated by a kind of application that differs greatly from our focus applications to dynamical systems, there are some remarkable similarities in the mathematical developments that are involved.

The first systematic exposition of hidden algebra was presented by Goguen and Diaconescu in [57] in 1994, showing how equational specifications can describe objects, inheritance and modules, and how equational reasoning can be efficiently used to prove properties about concurrent systems; a universal characterization of parallel connection is also provided in [57]. The two sister papers [65, 64] by Goguen and Malcolm are probably the best references for what we now call monadic fixed data hidden algebra, uniformly presenting the main known results and proof techniques of hidden algebra, including a systematic list of objectives and goals for the entire formalism, the proof method called hidden coinduction initiated in 1994 in [108, 62], consistency, initiality

---

7We thank Cristian Calude for pointing out this early reference.

8We thank Zinovy Diskin for pointing out this analogy and suggesting us a paper by Blok and Pigozzi [18] where we found the reference to Łós’s work on logical matrixes [103].
and finality results, as well as plenty of examples and interesting references. It is worth mentioning the paper [67] by Goguen, Malcolm and Kemp, which introduces the hidden Horn clause logic and provides the mathematical foundations for solving equational queries in a hidden equational setting, via a modified Herbrand theorem; a consistency result is also presented together with a generalization of a result by Diaconescu [36] allowing to lift results in hidden algebra to hidden Horn clause logic with equality.

To better handle the continuously increasing number of practical situations we and other scientists encountered, hidden algebra has undergone a series of generalizations and notational simplifications. Roșu and Goguen [137] show that many results previously thought to depend on the monadic nature of operations in hidden sorts hold in a more general setting, with polyadic operations, non-behavioral operations, and even with no fixed data universe. These results include the maximality of behavioral equivalence as a binary relation compatible with the behavioral operations and the extension of coinduction method to a non-coalgebraic framework. Perhaps the most important contributions of [137] are the notion of cobasis, as dual notion to induction’s basis, which is a first significant step toward the automation of coinduction, and a syntactic cobasis criterion which is now implemented in BOBJ. In [71], Goguen and Roșu show that behavioral abstraction semantically is a special case of information hiding and provide two organizations of hidden logic as institutions and a natural relationship between them. Various algorithms to automate hidden coinduction in the general setting are investigated by Roșu, Goguen and Lin in [134, 139, 60], and finally a surprisingly powerful one, called circular coinductive rewriting has been implemented in BOBJ. Despite the success of these algorithms in proving things about concrete behavioral specifications, the set of equations which are behaviorally true is not recursively enumerable in general, as shown by Buss and Roșu [22], so there is no algorithm that can prove every behavioral truth, and, in particular, hidden logic does not admit any complete set of rules of deduction. All of these results are further developed and presented in a uniform simplified notation in this thesis.

A closely related generalization of hidden algebra which was a continuous source of inspiration for us is called coherent hidden algebra; it was, and is still being, developed at JAIST in Japan, by Diaconescu, Futatsugi and others within the CafeOBJ project [37,
Coherent hidden algebra differs from our current generalization of hidden algebra in that only the operations used for experiments are required to have at most one hidden argument. This technical limitation is based on the belief that the observations that can be performed on a system are monadic in nature; however, it doesn’t seem to be a necessary technical condition in proving any of the theoretical results and there are some natural examples that involve non-monadic operations. The need for polyadic operations in hidden sorts is also emphasized by the recent advance of so called binary methods in modern object-oriented programming languages such as Java and C++. Like BOBJ, CafeOBJ [37] is a specification language in the OBJ family providing strong support for automated reasoning. It is the first system that implemented behavioral rewriting, as early as 1996, an adaptation of standard term rewriting by Diaconescu, Futatsugi and others [37, 40] which deals with the non-standard feature that some operations do not preserve the behavioral equality relation. CafeOBJ implements a very special case of circular coinductive rewriting, which we call attribute coinduction. A paper by Matsumoto and Futatsugi introducing test set coinduction [109] may be of interest to readers exploring techniques to automate coinduction.

We are not aware of any fully satisfactory exposition of order-sorted hidden algebra yet\(^9\), treating all the current advances in hidden algebra in a uniform way, including many hidden argument and non-behavioral operations. Burstall and Diaconescu [21] provide a nice general solution that adds “behavior” to any logic regarded as an institution [55], in particular to order-sorted algebra thus obtaining order-sorted hidden algebra. Unfortunately, the construction in [21] is based on the fact that behavioral equivalence can be regarded as a congruence, and thus used to factor models, and that operations are monadic in hidden sorts. However, recent approaches to behavioral specification show that behavioral equivalence may be a congruence for only some of the operations and that polyadic operations are not only useful but also indispensable in some situations. A useful research subject would be to explore in what extent the results in [21] can be extended to include these features and how the entire enterprise can be used in practical situations. Malcolm and Goguen [108, 62] provide a more concrete construction of order-sorted hidden algebra and show it at work on refinement proofs of

\(^9\)In fact, there isn’t a full agreement among scientists on order-sorted algebra in general yet.
objects. The construction in [108, 62] seems to be easy to modify to include the current generalizations of hidden algebra presented in this thesis. Cirstea, Malcolm and Worrell [27] use coalgebra to integrate hidden sorted algebra and order-sorted algebra, but their construction is based on the monadicity of operations and also on a fixed data universe which give the coalgebraic flavor of hidden algebra. The relationships between all these different formulations of order sorted hidden algebra certainly deserves further research. Even though BOBJ provides support for (an ad-hoc) order sorted hidden reasoning, we do not approach this delicate subject in this thesis.

Among other contributions to hidden algebra, we particularly suggest the work by Malcolm [107] on coalgebraic aspects of hidden algebra, especially on relationships between behavioral equivalence and bisimulation, and between behavioral equivalence and minimal realization of machines in a categorical setting. The relationship between behavioral equivalence and bisimulation was also explored by Lucanu, Gheorghieș and Apetrei in [104]. The works of Cirstea [23, 24, 25, 26] are also of big interest; she further investigated the coalgebraic nature of hidden algebra, showing among other interesting results that the forgetful functor from hidden algebras to sets admits a right adjoint, so that hidden algebra forms a comonad; also work of James Worrell, showing that hidden algebra is a topos [159]. The doctorate thesis of Vegliò [155] contains a good 1998 survey of fixed data hidden algebra. The work by Roșu [133, 136] on adaptations of Birkhoff’s Variety Theorem to the framework of hidden algebra can be a starting point toward what we’d like to call universal hidden algebra.

**Behavioral Satisfaction and Equivalence**

Behavioral satisfaction of equations is a core concept of hidden logic, used to get an algebraic treatment of state that abstracts away from implementation details. More precisely, states are represented by terms of hidden sort, and an equation on states is behaviorally satisfied by a model if and only if the states are equal under any *experiment* or *observation* or *visible context*. This elegant idea was introduced by Reichel [126] (see also [127]) in 1981 in the context of partial algebras, and since then has been used and adapted by many researchers in various settings, including us. In this thesis, we present at appropriate places three different equivalent ways to build contexts: one uses only
behavioral operations and as many variables as one wants among which one is called “special”, another uses only the special variable the rest being replaced by “special” constants standing for the data, and a third one builds contexts within a model, that is, it uses the elements of the model in the contexts; these are equivalent in the sense that they all generate the same behavioral equivalence relation.

Among other early references to behavioral satisfaction and equivalence, we’d mention the 1982 work by Goguen and Meseguer [68] on “abstract modules” that also presents a final algebra theorem for behavioral equivalence, which is a starting point toward coalgebraic treatments of monadic fixed-data hidden algebra and relationships between behavioral equivalence and bisimulation, as in [107, 104]. Sannella and Wirsing’s early work [147] from 1983 can be also of interest.

Observational Logics

There is a large class of behavioral logics based on observability of states, having the same intuition of “indistinguishability under observations” that we have for behavioral equivalence in hidden logic. In fact, hidden logic is also a kind of observational logic, but we distinguish it from the others because it is unique with all the following desired properties: it allows both a fixed-data and a loose-data semantics; behavioral equivalence may not be a congruence, so that one cannot factor through it; any operation can have many hidden arguments, and as a consequence, it may not admit final or initial models. However, the results in the thesis which do not explicitly require fixed-data are applicable to all the observational logics; we mention that the inference rules and the algorithms for automated proving do not require fixed-data.

Besides the already mentioned seminal works by Reichel [126, 127], Goguen and Meseguer [68, 110, 50], and Sannella and Wirsing [147], there are a few other papers before 1990 that are important milestones in the development of behavioral specification and implicitly of hidden logic, such as Reichel [128] in 1987 and Orejas, Nivela and Ehrig [119, 120] in 1988 and 1989, respectively. Of special interest are the works by Sannella and Tarlecki [146, 148] in 1985 and 1988, respectively, within their general project to structure specifications in any institution.

There is a series of papers by Bernot, Bidoit, Hennicker, Knapik and Wirsing
that lead to the evolution of observational logic [87], a behavioral logic that together with coherent hidden algebra had a major influence on the development of hidden logic. One major lesson we learned from observational logic is that polyadicity of behavioral operations in hidden sorts does not bring any practical, theoretical or methodological complication, and that it should be considered natural and even necessary. Additionally, to our knowledge, [11] is the first place where behavioral equivalence is defined using only a subset of behavioral operations; this is a crucial observation, and is in close relationship with the notion of cobasis on which the automation of coinduction is built in hidden logic. These papers are mentioned later at appropriate places in the thesis. However, our main criticism with respect to observational logic is that its models are built in such a way that all the operations preserve the behavioral equivalence, or are congruent in our terminology; our current belief, motivated by a series of concrete situations including the behavioral specification of a nondeterministic stack presented in Example 9, spread out through Chapter III, is that non-congruent operations will play an important rôle in behavioral specifications, especially in the presence of nondeterminism and concurrency. The practical importance of non-congruent operations was first noticed by Diaconescu and Futatsugi [40] in 1998, who also introduce the notion of coherent operations for operations that preserve the behavioral equivalence; we prefer the term “behaviorally congruent” or just “congruent” instead of “coherent” because the congruence rule of equational deduction is sound in hidden logic for an operation iff that operation is behaviorally congruent.

Context induction was introduced by Hennicker [86] in 1991 and further explored by Gaudel and Privara [44] and by Berregeb, Bouhoula and Rusinowitch [12], which represents the first serious step towards the automation of behavioral proving. In Subsection VI.C.2, we show that context induction occurs naturally in the process of “unhiding” a behavioral specification; unfortunately, proofs by context induction tend to be rather complicated, so we prefer circular coinductive rewriting.

Coalgebra

There is a distinguished tradition of research in coalgebra, and a continuously increasing number of scientists who approach this very elegant subject from various per-
spectives. Some of hidden algebra’s ingredients are closely related to their counterparts in coalgebra, such as, for example, behavioral equivalence relates to bisimulation. The full relationship between hidden logic and coalgebra is still to be explored, but our understanding at this moment is that the models of a special segment of hidden logic, called monadic fixed-data hidden logic, are isomorphic to the coalgebras over certain “algebraic” functors. Since monadic fixed-data hidden logic is the original setting of hidden algebra in 1989 [50], and so the starting point of hidden logic, we have payed special attention to coalgebra and its current advances.

A good introduction to coalgebra and coalgebraic coinduction, as well as to the duality between algebra and coalgebra, is the tutorial by Jacobs and Rutten [98]; it also provides many early references on the evolution of coalgebra. The main use of coalgebra that relates to our approach is its link to state-based dynamic systems. According to [98], there is no single reference in which this link is made explicitly, but important insights can be found in Giarrantana, Gimona and Montanari [45], in Wand [156], in Arbib and Manes [4], in Kamin [99], in Aczel and Mendler [1, 2], and in Milner and Tofte [113], among many others.

One direction of research in coalgebra is to show existence of final systems, thus giving rise to an abstract notion of bisimulation that can be used to give a semantics for process algebras, as Aczel and Mendler [1, 2] and Barr [6]; Rutten provides a nice synthesis on the existence of final systems in [145]. A peculiar feature of hidden logic in contrast with coalgebra is that it admits neither final nor initial models in its general setting, as shown in Buss and Roșu [22] and also in Chapter VI.

Another direction is to regard coalgebra dually to universal algebra, thus giving birth to what Rutten called universal coalgebra [145]. Following a challenge launched by Jacobs in [95], Rutten [145] introduces the notion of covariety as a class of coalgebras closed under coproducts, quotients and subcoalgebras, and then gives a Birkhoff-like covariety result. Taking over Rutten’s result, Corradini [30] showed that the class of certain coalgebras (actually, certain hidden algebras) which satisfy a special coalgebraic equational specification, is closed under subalgebras, quotients and coproducts (or sums), but presented a counter-example showing that the other implication is not necessarily true. Roșu [133, 136] then showed that a new closure operation is needed in order to
get a complete characterization of equationally definable classes of coalgebras. Gumm
and Schröder also investigated characterization properties of covarieties [83, 82]. They
introduced the notion of complete covariety [83] (covariety closed under bisimulation)
and gave a Birkhoff-like characterization making use of formulas over an appropriate
language. Furthermore, Gumm introduced the coequations [82] (elements of certain
cofree coalgebras) and implications of coequations and provided characterization results
for covarieties and quasicovarieties (classes of coalgebras closed only under quotients and
coproducts) in a dual manner to those of Birkhoff. However, coequations as elements
of cofree coalgebras are infinite structures, and therefore difficult to use in practical
situations to specify systems. Our main concern is not to perfectly dualise the Birkhoff
axiomatizability results, but to investigate the definitional power of the intuitive, well-
understood and practical equations within a coalgebraic segment of hidden logic. Another
interesting covariety result appears in Kurz [100] in the context of modal logic.

Hagino [84], Malcolm [106] and Gordon [81] apply coalgebra to functional pro-
gramming, and Rutten and Turi [144, 145, 143] apply it to automata theory. Reichel
[129] was the first to apply coalgebra explicitly to the object paradigm, and then Jacobs
further investigated objects’ coalgebraic flavor [95, 96, 97]; since hidden algebra’s main
purpose from its very beginning in 1989 was to give semantics to the object paradigm
[50], Reichel and Jacobs’ works suggest that one should look even closer at the relation-
ship between hidden logic and coalgebra. A first observation related to seminal works
by Malcolm [107] and Cîrstea [24, 23] on coalgebraic views of hidden algebra, is that
even the very early formulation of hidden algebra allowed a typical non-coalgebraic fea-
ture, namely the existence of hidden constants; an immediate consequence is the lack
of final models. Nevertheless, hidden constants are indispensable in the treatment of
nondeterminism advocated by hidden algebra and a special machinery to handle them
is not needed in hidden logic. Later on, the need for operations with many hidden ar-
guments [37, 40, 137, 71] further emphasized the non-coalgebraic nature of hidden logic
in its general form, as first noticed by Buss and Roșu [22] and explained in Chapter
VI in more detail. Fortunately, the validity of coinduction as a proof method in hidden
logic does not depend upon the existence of final models, but only upon the existence
of a largest behavioral equivalence, as shown in [40, 137] (see also Chapter IV). Our
experience with BOBJ so far is that these non-coalgebraic features of hidden logic are very useful in practical situations; the examples in this thesis show some of these.

There is an increasing recent interest in combining coalgebra with algebra. For example, Turi and Plotkin [153] use their combination to describe denotational and operational semantics, Hensel and Reichel [90] use it to define equations on terminal coalgebras, and Hennicker and Kurz [88, 101] to extend current coalgebraic techniques with observational logic principles. However, both algebraic and coalgebraic settings inherently live together in hidden logic, without any special treatment for either of them: if one wants “coalgebraic” features then one should just add hidden sorts, think of terms of hidden sort as “states,” and then use coinduction and behavioral equational deduction to prove behavioral properties about them, or induction and standard equational reasoning to prove properties about the visible world, the so called “data”; hidden constants and non-monadic operations in hidden sorts do not bring any trouble, so one has no such limitations. This is our simple methodological approach to behavioral specification and verification, and is faithfully supported by BOBJ.
Chapter II

Preliminaries

Grigore Moisil\textsuperscript{1}, a mathematician, logician and philosopher, asked his colleagues revise a math book he was just writing. They almost all complained that he insisted too much on details and let important notions undefined. When he came with the next version, his colleagues complained again, that he removed some necessary explanations and introduced other trivial explanations, still letting important concepts unexplained. Then he exploded:

- How can I know what’s important and what’s trivial?

This chapter introduces all the notions that are important for understanding the thesis. Its rôle is to let the reader know our terminology, conventions, and mathematical language rather than to dwell on what we consider rather simple notions. The reader already familiar with basic concepts of algebra, especially universal and many-sorted algebra, and equational logic and its relationship with algebra will find this thesis easier to read. We refer the interested reader to [69, 122, 63] for more on these subjects and for proofs of the results presented in this chapter. We use the notation and conventions originally introduced by Goguen and Meseguer in [69]

Even if we sometimes mention categorical concepts, category theory is not needed until Subsection VI.A.1; we recommend the reader who is not familiar with category theory skip those parts, or to read “collection” instead of category. However, we find [102, 91, 49] good references for category theory and refer the interested reader to them.

\textsuperscript{1}Grigore Moisil was a professor in the Departments of Mathematics and Philosophy and the director of the Center of Computing at the University of Bucharest, Romania. He was my MS advisor’s advisor.
II.A Sets, Functions, Relations

Given a set $A$, we let $A^*$ be the set of finite lists, or strings, with elements in $A$; we take the liberty to use both the concatenation notation, for example $l = a_1a_2 \cdots a_n$, and the [head,tail] notation, for example $l = [a,l']$, depending on which is more convenient in a certain context. Given two sets, $A$ and $B$, we let $[A \to B]$ denote the set of functions of source $A$ and target $B$. If $f : A \to B$ is a function and $w \in A^*$ is the list $a_1a_2 \cdots a_n$, then $f(w)$ is the list $f(a_1)f(a_2) \cdots f(a_n)$ in $B^*$; if $C \subseteq A$ then $f \upharpoonright C : C \to B$ is the restriction of $f$ to $C$. If $f : A \to B$ and $g : B \to C$ are two functions, then we let $f;g : A \to C$ denote their composition. For those familiar with category theory, we let Set denote the category of sets and functions.

If $S$ is a set, $A = \{A_s \mid s \in S\}$ is an $S$-sorted, or $S$-indexed, set, and $w \in S^*$ is the list $s_1s_2 \cdots s_n$, then $A_w$ denotes the product $A_{s_1} \times A_{s_2} \times \cdots \times A_{s_n}$. If $A$ and $B$ are $S$-sorted sets, then an $S$-sorted relation $R$ between $A$ and $B$ is an $S$-sorted set of relations $\{R_s \subseteq A_s \times B_s \mid s \in S\}$; we simply write $R \subseteq A \times B$. $R$ is an $S$-sorted equivalence iff each $R_s$ is an equivalence. We prefer the infix notation for binary relations in general, that is, if $\to \subseteq A \times B$ then we write $a \to b$ rather then $(a,b) \in \to$.

II.B Many-Sorted Signatures

Given a set $S$ of elements called sorts, an $S$-sorted signature $\Sigma$ is an $(S^* \times S)$-indexed set $\{\Sigma_{w,s} \mid w \in S^*, s \in S\}$ of operations. When $S$ is not important, we just call it a many-sorted signature. When $S$ is important, we often write $(S,\Sigma)$ instead of just $\Sigma$. If an operation $\sigma$ is in $\Sigma_{w,s}$, then $(w,s)$ is the type of $\sigma$, and we write $\sigma : w \to s$. We let $\Sigma_s$ denote the set $\Sigma_{\lambda,s}$ where $\lambda$ is the empty string; the operations in $\Sigma_s$ are called constants of sort $s$ and we write $\sigma : \to s$ instead of $\sigma \in \Sigma_s$. In practice, most of the sets $\Sigma_{w,s}$ are empty. $\Sigma$ is finite iff there is only a finite number of non-empty sets $\Sigma_{w,s}$, and those are finite sets. Given an $S$-signature $\Sigma$ and an $S'$-signature $\Sigma'$, a morphism of many-sorted signatures, say $\varphi : \Sigma \to \Sigma'$, is a pair of functions $\varphi = (f,g)$ such that $f : S \to S'$ and $g = \{g_{w,s} : \Sigma_{w,s} \to \Sigma'_{f(w),f(s)} \mid (w,s) \in S^* \times S\}$. The morphism $\varphi = (f,g)$ is called an inclusion iff both $f$ and $g$ are inclusions, and in this case we write $\Sigma \hookrightarrow \Sigma'$. The many-sorted signatures and the morphisms of many-sorted signatures
form a category, which we denote \textbf{Sign}. If \( V \subseteq S \) and \( \Sigma \) is an \( S \)-signature, then \( \Sigma|_V \), the \textit{\( V \)-reduct of} \( \Sigma \), is the \( V \)-signature \( \{ \Sigma_{w,v} \mid (w,v) \in V^* \times V \} \).

II.C Many-Sorted Algebras

Given an \( S \)-sorted signature \( \Sigma \), a \textbf{\( \Sigma \)-algebra} \( A \) is an \( S \)-sorted set \( \{ A_s \mid s \in S \} \) together with functions \( A_\sigma : A_w \to A_s \) for all operations \( \sigma \in \Sigma_{w,s} \). The set \( A_s \) is called the \textbf{carrier of sort} \( s \) of \( A \). Given two \( \Sigma \)-algebras \( A \) and \( B \), a \textbf{morphism of \( \Sigma \)-algebras}, say \( f : A \to B \), is an \( S \)-sorted function \( f = \{ f_s : A_s \to B_s \} \) such that for any \( \sigma : s_1s_2\cdots s_n \to s \) in \( \Sigma \) and any \( a_1 \in A_{s_1}, a_2 \in A_{s_2}, \ldots, a_n \in A_{s_n} \), it is the case that \( f_s(A_\sigma(a_1,a_2,\ldots,a_n)) = B_\sigma(f_{s_1}(a_1),f_{s_2}(a_2),\ldots,f_{s_n}(a_n)) \). \( \Sigma \)-algebras together with morphisms of \( \Sigma \)-algebras form a category which we denote \( \text{Alg}_\Sigma \). A \( \Sigma \)-algebra \( I \) is called \textbf{initial} iff for any \( \Sigma \)-algebra \( A \) there exists a unique morphism of \( \Sigma \)-algebras \( \alpha_\Sigma : I \to A \). Dually, a \( \Sigma \)-algebra \( F \) is \textbf{final} iff for any \( \Sigma \)-algebra \( A \) there exists a unique morphism of \( \Sigma \)-algebras \( \omega_\Sigma : A \to F \). Notice that any two initial or final \( \Sigma \)-algebras are isomorphic, and that any \( \Sigma \)-algebra having one-element carriers is final and these are the only final \( \Sigma \)-algebras. Given a morphism of signatures \( \varphi = (f,g) : (S,\Sigma) \to (S',\Sigma') \) and a \( \Sigma' \)-algebra \( A \), then the \textbf{\( \varphi \)-reduct of} \( A \), denoted \( A|_\varphi \), is the \( \Sigma \)-algebra defined by \( (A|_\varphi)_s = A_{f(s)} \) for each \( s \in S \) and \( (A|_\varphi)_\sigma = A_{g(\sigma)} \) for each \( \sigma \in \Sigma \). If \( \varphi \) is an inclusion, then we just write \( A|_\Sigma \) and call it the \textbf{\( \Sigma \)-reduct of} \( A \). An inclusion \( \Sigma \hookrightarrow \Sigma' \) is called a \textbf{conservative extension} iff for any \( \Sigma \)-algebra \( A \) there is some \( \Sigma' \)-algebra \( A' \) such that \( A'|_\Sigma = A \).

A \textbf{\( \Sigma \)-congruence} \( \sim \) on a \( \Sigma \)-algebra \( A \) is an \( S \)-sorted equivalence which is compatible with the operations in \( \Sigma \), that is, \( A_\sigma(a_1,a_2,\ldots,a_n) \sim_s A_\sigma(a'_1,a'_2,\ldots,a'_n) \) for each \( \sigma : s_1s_2\cdots s_n \to s \) and \( a_1,a'_1 \in A_{s_1}, a_2,a'_2 \in A_{s_2}, \ldots, a_n,a'_n \in A_{s_n} \) with \( a_1 \sim_{s_1} a'_1, a_2 \sim_{s_2} a'_2, \ldots, a_n \sim_{s_n} a'_n \).

II.D Term Algebras

Given \( (S,\Sigma) \), let \( T_\Sigma \) be the smallest \( S \)-sorted set having the property that \( \sigma(t_1,t_2,\ldots,t_n) \in T_{\Sigma,s} \) whenever \( \sigma : s_1s_2\cdots s_n \to s \) and \( t_1 \in T_{\Sigma,s_1}, t_2 \in T_{\Sigma,s_2}, \ldots, t_n \in T_{\Sigma,s_n} \). Notice that \( T_\Sigma \) can be organized as a \( \Sigma \)-algebra and that its carriers are empty.
if $\Sigma$ contains no constants. $T_\Sigma$ is called the term $\Sigma$-algebra and its elements of sort $s$ are called (ground) terms of sort $s$; it has the following important property:

**Proposition 1** $T_\Sigma$ is an initial $\Sigma$-algebra.

If $X$ is an $S$-sorted set, then $T_\Sigma(X)$ denotes the $\Sigma$-algebra $T_{\Sigma \cup X}|_\Sigma$, where $\Sigma \cup X$ is the signature extending $\Sigma$ by adding the elements in $X$ as constants. $T_\Sigma(X)$ is called the term $\Sigma$-algebra over variables in $X$. Given a term $t \in T_\Sigma(X)$, we let $\text{var}(t)$ denote the $S$-sorted subset of $X$ of variables occurring in $t$. $T_\Sigma(X)$ has the following property of universality, which generalizes the previous proposition:

**Proposition 2** Given a $\Sigma$-algebra $A$, then any function $\theta : X \rightarrow A$ can be uniquely extended to a morphism of $\Sigma$-algebras $\theta^* : T_\Sigma(X) \rightarrow A$.

To simplify writing, we ambiguously write $\theta$ instead of $\theta^*$. Intuitively, $\theta^*$ “evaluates” terms into the “domain” $A$.

Given a signature $(S, \Sigma)$, a derived operation $\delta : s_1...s_n \rightarrow s$ of $\Sigma$ is a term in $T_{\Sigma,s}(\{z_1,...,z_n\})$, where $z_1,...,z_n$ are special variables of sorts $s_1,...,s_n$. For any $\Sigma$-algebra $A$, the interpretation of $\delta$ in $A$ is the map $A_\delta : A_{s_1} \times \cdots \times A_{s_n} \rightarrow A_s$ defined as $A_\delta(a_1,...,a_n) = \theta(\delta)$, where $\theta : \{z_1,...,z_n\} \rightarrow A$ takes $z_i$ to $a_i$ for all $i = 1...n$. We let $(S, \text{Der}(\Sigma))$, or simply $\text{Der}(\Sigma)$, denote the signature of all derived operations of $\Sigma$. Notice that the inclusion of signatures $\Sigma \hookrightarrow \text{Der}(\Sigma)$ is a conservative extension.

The following are some more unusual notational conventions which will be tacitly used in the thesis. Given an operation $\sigma : s_1...s_n \rightarrow s$ and a set of variables $W = \{w_1 : s_1,...,w_n : s_n\}$, we may write $\sigma(W)$ instead of $\sigma(w_1,...,w_n)$. When we are only interested in one argument of $\sigma$, say the $j$th, we may write it as the last argument and let $\sigma_{\overline{s}_j}$ denote the sequence $s_1...s_{j-1}s_{j+1}...s_n$, that is, we may write $\sigma : \overline{s}_j \rightarrow s$. Moreover, if $W = \{w_1 : s_1,...,w_{j-1} : s_{j-1},w_{j+1} : s_{j+1},...,w_n : s_n\}$ is a set of variables and if $x : s_j$ is another variable, then $\sigma(W,x)$ denotes the term $\sigma(w_1,...,w_{j-1},x,w_{j+1},...,w_n)$. Similarly, given a $\Sigma$ algebra $A$, $A_\sigma : A_{\overline{s}_j} \times A_{s_j} \rightarrow A_s$ is another notation for $A_\sigma : A_{s_1} \times \cdots \times A_{s_n} \rightarrow A_s$, $\overline{s}_j \in A_{\overline{s}_j}$ is another notation for $(a_1,...,a_{j-1},a_{j+1},...,a_n) \in A_{s_1} \times \cdots \times A_{s_{j-1}} \times A_{s_{j+1}} \cdots A_{s_n}$, and $A_\sigma(\overline{a}_j,a_j)$ is the same as $A_\sigma(a_1,...,a_n)$. Given a sort $h$ (or a term $t$ of sort $h$, or an element $a$ of sort $h$ in a $\Sigma$-algebra $A$), an operation
σ ∈ Σ is **appropriate for** h (or t, or a) iff σ has at least one argument of sort h.

When we say that a property depending on a sort h holds for all appropriate σ, we also mean for all appropriate arguments of σ. For example, given two terms t, t′ ∈ TΣ,h(X), the assertion “A ⊨ Σ (∀W,X) σ(W,t) = σ(W,t′) for all appropriate σ ∈ Σ” means that A ⊨ Σ (∀W,X) σ(w1,...,wj−1,t,wj+1,...,wn) = σ(w1,...,wj−1,t′,wj+1,...,wn) for all operations σ ∈ Σs1...sn appropriate for h, all j ∈ 1,...,n such that sj = h, and all W = {w1: s1,...,wj−1: sj−1,wj+1: sj+1,...,wn: sn}.

II.E Many-Sorted Equational Logic

A Σ-equation is a triple (X,t,t′), written (∀X) t = t′, where X is an S-sorted set of variables and t,t′ are terms in TΣ,s(X) of the same sort. If the sort s of t and t′ is important, then we can write =s instead of =. A conditional Σ-equation has the form (∀X) t = t′ if C, where t, t′ ∈ TΣ,s(X) and C is a finite set of pairs t1 = t′1,t2 = t′2,...,tn = t′n, where t1,t′1 ∈ TΣ,s1, t2,t′2 ∈ TΣ,s2, ..., and tn,t′n ∈ TΣ,sn, respectively. If C is empty then the conditional Σ-equation is just an ordinary Σ-equation.

Given a conditional Σ-equation (∀X) t = t′ if t1 = t′1,t2 = t′2,...,tn = t′n, say e, a Σ-algebra A satisfies e, written A ⊨ Σ e, iff for each θ: X → A, if θ(ti) = θ(t′i) for each i ∈ {1,2,...,n} then θ(t) = θ(t′). Notice that A ⊨ Σ (∀X) t = t′ iff θ(t) = θ(t′) for each θ: X → A. The satisfaction relation extends naturally to sets of equations, in the sense that A satisfies a set E of conditional Σ-equations iff it satisfies each e ∈ E, and in this case we write A ⊨ Σ E. Moreover, if E is as above and e is any conditional Σ-equation, then we write E ⊨ Σ e whenever A ⊨ Σ E implies A ⊨ Σ e. The following are well known immediate properties of equational satisfaction:

**Proposition 3** If A is a Σ-algebra such that A ⊨ Σ (∀X′) t = t′ if C, then:

1. A ⊨ Σ (∀X′) t = t′ if C′ for each X ⊆ X′ and C ⊆ C′;
2. A ⊨ Σ (∀X′) t′ = t if C;
3. A ⊨ Σ (∀X′) t = t′ if C whenever A ⊨ Σ (∀X′) t′ = t′;
4. A ⊨ Σ (∀Y) ρ(t) = ρ(t′) if ρ(C) for any substitution ρ: X → TΣ(Y), where ρ(C) is the set {ρ(ti) = ρ(t′i) | ti = t′i ∈ C}.
The interested reader can consult, for example, [69] for a proof.

An **algebraic specification**, sometimes called “ordinary specification” in this thesis, is a pair \((\Sigma, E)\), where \(E\) is a set of conditional \(\Sigma\)-equations. If \(S = (\Sigma, E)\) is a specification then \(A\) satisfies \(S\), written \(A \models S\) iff \(A \models_\Sigma E\). Similarly, \(S \models e\) iff \(E \models_\Sigma e\). A **theory** is a specification \((\Sigma, E)\) such that \(E = E^*\), where \(E^*\) is the set of all the conditional \(\Sigma\)-equations \(e\) with \(E \models_\Sigma e\). A specification \((\Sigma', E')\) is an **extension** of \((\Sigma, E)\) iff \(\Sigma \hookrightarrow \Sigma'\) and \(E \subseteq E'^*\). \((\Sigma', E')\) is a **conservative extension** of \((\Sigma, E)\) if it is an extension and for each \(\Sigma\)-algebra \(A\) that satisfies \(E\) there is some \(\Sigma'\)-algebra \(A'\) that satisfies \(E'\) and \(A'|_\Sigma = A\).

Now let’s fix a set of conditional \(\Sigma\)-equations \(E\). We can define a derivability relation \(\models_\Sigma\) between \(E\) and other \(\Sigma\)-equations iteratively as follows:

- **Reflexivity**: \(E \models_\Sigma (\forall X) t = t\)
- **Symmetry**: \(\frac{E \models_\Sigma (\forall X) t = t'}{E \models_\Sigma (\forall X) t' = t}\)
- **Transitivity**: \(\frac{E \models_\Sigma (\forall X) t = t', E \models_\Sigma (\forall X) t' = t''}{E \models_\Sigma (\forall X) t = t''}\)
- **Substitution**: \(\frac{(\forall Y) t = t' \text{ if } C \in E, \theta : Y \rightarrow T_\Sigma(X), E \models_\Sigma \theta(C)}{E \models_\Sigma (\forall X) \theta(t) = \theta(t')}\)
- **Congruence**: \(\frac{E \models_\Sigma (\forall X) t = t'}{E \models_\Sigma (\forall W, X) \sigma(W, t) = \sigma(W, t'), \text{ for each } \sigma \in \Sigma}\)

where \(\theta(C)\) is the set of equations \(\{(\forall X) \theta(t_i) = \theta(t'_i) \mid t_i = t'_i \in C\}\).

**Theorem 4 Completeness**: If \(e\) is a \(\Sigma\)-equation, then \(E \models_\Sigma e\) iff \(E \models_\Sigma e\).

The interested reader is referred to [69, 122] for proof.
Chapter III

Basics of Hidden Logic

The basic ingredients of hidden algebra together with simple and natural examples are introduced in this chapter. It is worth mentioning that the concepts, notations and definitions suffered many changes during the last decade, and that various generalizations with different terminologies are currently in use. One of the goals of this chapter is to present the most general approaches of which we are aware, and to establish a uniform terminology at least for the rest of the thesis.

The first section presents hidden signatures, emphasizing the two slightly different directions in use, one with a loose data universe and the other with a fixed data universe. The second and the third sections introduce the models of hidden logic, formalizing the informal notions of experiment and behavioral equivalence of states. Behavioral satisfaction is described in the fourth section, and the last section is dedicated to the so called theorem of constants which justifies the replacement of hidden variables by constants in proofs.

III.A  Hidden Signatures

A common feature to all variations of hidden logic is that the sorts, or types, are split into visible and hidden. The visible sorts stay for data, while the hidden sorts stay for states.

Definition 5  Given two disjoint sets $V$ and $H$ of visible sorts and of hidden sorts, respectively, a loose data hidden $(V,H)$-signature is a many-sorted $(V \cup H)$-signature.
A fixed data hidden \((V,H)\)-signature is a pair \((\Sigma,D)\), where \(\Sigma\) is a loose data hidden \((V,H)\)-signature and \(D\), called data algebra, is a many-sorted \(\Sigma|_V\)-algebra. The operations in \(\Sigma\) with one hidden argument and visible result are called attributes, those with one hidden argument and hidden result are called methods, those with two hidden arguments and hidden result are called binary methods, and those with only visible arguments (zero or more) and hidden result are called hidden constants.

We write “hidden signature” instead of “loose data hidden \((V,H)\)-signature” or “fixed data hidden \((V,H)\)-signature” whenever there is no confusion. Since many definitions and results in the thesis hold for both approaches, we will explicitly mention “loose data” or “fixed data” only when that assumption is needed; if just “hidden signature” is used then it means that the definition or the result holds for both loose data and fixed data versions. A fixed data hidden signature \((\Sigma,D)\) is often written just \(\Sigma\).

**Example 6 Cell.** Perhaps the simplest hidden signature is the one associated to a memory cell, that is, a location that can keep a value of a certain type, say \(\text{Elt}\); therefore, \(V = \{\text{Elt}\}\) in this example. A hidden sort, say \(\text{Cell}\), is needed for cells, and of course, two basic operations, a method and an attribute, respectively,

\[
\text{put} : \text{Elt Cell} \rightarrow \text{Cell}, \quad \text{and} \\
\text{get} : \text{Cell} \rightarrow \text{Elt},
\]

where \(\text{put}(E,C)\) puts the element \(E\) in the cell \(C\), while \(\text{get}(C)\) returns the current element in \(C\). The following ADJ-like [75, 78, 77] diagram shows the signature in an easily understandable form:

![ADJ-like diagram for cells](image)

Notice that the visible sorts are represented as white ellipses, while the hidden sorts are shadowed boxes, to emphasize the idea of “black box”. The cell can be regarded as an
object, or as a blackbox, or as a state, whose internal structure or implementation is intended to be hidden. On the other hand, the elements, or the data, are visible, in the sense that we can “see and touch” them; formally, that means that one can say when two elements are strictly equal or not. When a particular implementation of elements is not of interest, the above hidden signature is “loose data”. When the elements come as a specific datatype $D$, e.g., the natural numbers, then the signature is “fixed data”, and the cells are restricted to only data of that type, in this case natural numbers.

Example 7 Flag. The use of semaphores \([42]\) for scheduling and protecting resources is well known. A flag is associated with each non-preemptive resource. When the resource is allocated to a process, its semaphore is put up, and access is prohibited. When the process releases the resource, its semaphore is put down and it can be allocated to another process. Semaphores are implemented in various ways, ranging from full software to full hardware implementations. Many modern processors support semaphores to speed up operating systems, and often include a function to reverse a flag. The next picture represents the fixed data hidden signature of a slightly simplified version of semaphores (see also Example 34); we call them “flags”:

![ADJ-like diagram for flags](image)

The attribute $\mathit{up?}$ returns $\mathit{true}$ iff the flag is up, and the methods $\mathit{up}$, $\mathit{down}$ and $\mathit{rev}$ put the flag up, down, and reverse it, respectively. Notice that the sort $\mathit{Flag}$ is hidden because one should not be interested in how it is implemented. The hidden signature is “fixed data” since one wants to rely on a built-in fixed implementation of booleans.

Example 8 Set. A more complex example of hidden signature is that of sets. The following picture presents two visible sorts, $\mathit{Elt}$ and $\mathit{Bool}$, a hidden sort $\mathit{Set}$, and the
usual operations on sets ($U$ is union and $\&$ is intersection of sets, respectively).

Figure III.3: ADJ-like diagram for sets.

Notice that $\text{in}$ is an attribute, $\text{add}$ is a method, union and intersection of sets are binary methods, while $\text{empty}$ is a hidden constant. $\Sigma |_V$ contains exactly the operations on $\text{Bool}$ in this example, because no operations were assumed on $\text{Elt}$. If the fixed data version is intended, then the hidden signature also contains a concrete implementation ($\Sigma |_V$-algebra) of booleans and elements.

Example 9 Nondeterministic Stack. This example describes the fixed data hidden signature of a nondeterministic stack, that is, a stack in which natural numbers are pushed nondeterministically.

Figure III.4: ADJ-like diagram for nondeterministic stacks.
The method \texttt{push} generates and puts a new random number on the stack, the attribute \texttt{top} returns the value of the last generated number, and \texttt{pop} removes that number. The hidden signature $\Sigma$ contains the two sorts in the picture above together with all the operations drawn; $\Sigma|_V$ is the signature of natural numbers, that is $\{0, s, +, \ast\}$. The natural numbers are supposed builtin in this example, so the data algebra $D$ is the\footnote{The one provided by the system.} algebra of natural numbers.

III.B Hidden Algebra

A hidden algebra can be regarded as (an implementation of) a system, or part of it. Another view is that of a “blackbox”, in the sense that one shouldn’t bother what’s inside, but rather how it behaves, that is, how it \textit{appears} under experiments. A hidden algebra consists of a finite or infinite $(V \cup H)$-sorted set representing the state and data universes of a system, or object, together with implementations of all the operations in the signature. Again, we distinguish between the loose data and fixed data versions:

\textbf{Definition 10} A loose data hidden $\Sigma$-algebra $A$ is any $\Sigma$-algebra. A fixed data hidden $(\Sigma, D)$-algebra $A$ is a $\Sigma$-algebra $A$ such that $A|_{\Sigma|_V} = D$.\footnote{The one provided by the system.}

$A$ is simply called a “hidden algebra” whenever the loose data or the fixed data environment is clear from the context or is not important. Notice that in the fixed data case, the data is protected as expected; that means, for example, that an implementation of a cell of natural numbers, whose hidden signature was presented in Example 6, cannot affect the builtin natural numbers provided by the system.

\textbf{Example 6 Cell} (continued). If we consider cells that keep natural numbers (let $\mathbb{N}$ be the set of natural numbers) only, an intuitive fixed data hidden algebra $A$ would be the following:

\[
\begin{align*}
A_{\text{Elt}} &= \mathbb{N} \\
A_{\text{Cell}} &= \{[n] \mid n \in \mathbb{N}\} \\
A_{\text{get}}(\langle n \rangle) &= n \\
A_{\text{put}}(m, \langle n \rangle) &= [m],
\end{align*}
\]
where only the last value put in a cell is remembered. However, there may be good reasons to prefer implementations keeping a history of actions, so that one can “undo” or “unroll” them. Using [head,tail] list notation with [] for the empty list, such an implementation is equivalent to the following model:

\[\begin{align*}
A_{\text{Elt}} &= N, & A_{\text{Cell}} &= N^* \\
A_{\text{get}}([]) &= 0, & A_{\text{get}}([n,l]) &= n \\
A_{\text{put}}(m,l) &= [m,l].
\end{align*}\]

At this stage, we cannot avoid “bad” implementations, such as

\[\begin{align*}
A_{\text{Elt}} &= N, & A_{\text{Cell}} &= \{\star\} \\
A_{\text{get}}(\star) &= 0, & A_{\text{put}}(n,\star) &= \star,
\end{align*}\]

where \(\star\) is a special element. We’ll see later how (behavioral) equations can restrict the class of models.

If loose data implementations are intended, then \(N\) may be replaced by any set. Therefore, the loose data semantics allows more models than the fixed data one. ■

**Example 7 Flag (Continued).** The expected implementation of a flag is the following:

\[\begin{align*}
A_{\text{Bool}} &= \mathbb{B}, & A_{\text{Flag}} &= \{u,d\} \\
A_{\text{up}}(u) &= \text{true}, & A_{\text{up}}(d) &= \text{false} \\
A_{\text{up}}(u) &= u, & A_{\text{up}}(d) &= u \\
A_{\text{down}}(u) &= d, & A_{\text{down}}(d) &= d \\
A_{\text{rev}}(u) &= d, & A_{\text{rev}}(d) &= u
\end{align*}\]

where \(\mathbb{B}\) is a data algebra of booleans. As in Example 6, there may be hidden algebras keeping histories of transitions as states, for example:

\[\begin{align*}
A_{\text{Bool}} &= \mathbb{B}, & A_{\text{Flag}} &= \{u,d,r\}^* \\
A_{\text{up}}([u,l]) &= \text{true}, & A_{\text{up}}([d,l]) &= \text{false}, & A_{\text{up}}([r,l]) &= \text{not}(A_{\text{up}}(l)) \\
A_{\text{up}}(l) &= [u,l], & A_{\text{down}}(l) &= [d,l] & A_{\text{rev}}(l) &= [r,l],
\end{align*}\]

for any finite list \(l\) of symbols in \(\{u,d,r\}\), where \text{not} is the standard negation on booleans, provided by \(\mathbb{B}\). ■
Example 8 Set (continued). Sets can be implemented in at least two interesting ways, depending on whether multiple occurrences of elements are acceptable or not. To keep the presentation simple, suppose that we work in a fixed data universe framework, that of natural numbers, so we have sets of natural numbers. The first hidden algebra is the (expected) one in which the states consists of sets of numbers and the operations on sets are the usual ones:

\[
\begin{align*}
A_{\text{Bool}} & = \mathbb{B}, & A_{\text{Elt}} & = \mathbb{N}, & A_{\text{Set}} & = \mathcal{P}(\mathbb{N}) \\
A_{\text{empty}} & = \emptyset, & A_{\text{add}}(n,S) & = \{n\} \cup S, & A_{\text{in}}(n,S) & = n \in S \\
A_{0}(S,S') & = S \cup S', & A_{\&}(S,S') & = S \cap S'
\end{align*}
\]

The second hidden algebra of sets, inspired from LISP where sets are viewed as lists, allows multiple occurrences of elements. Sets are implemented as finite lists, with \textit{in} as membership, \textit{empty} the empty list, \textit{add} placing a number at the front of a list, and \textit{U} appending two lists (we use the same [head,tail] notation for lists as in Example 6):

\[
\begin{align*}
A_{\text{Bool}} & = \mathbb{B}, & A_{\text{Elt}} & = \mathbb{N} \\
A_{\text{Set}} & = \mathbb{N}^*, & A_{\text{empty}} & = [] \\
A_{\text{add}}(n,l) & = [n,l], & A_{\text{in}}(n,l) & = (\text{member } n \text{ } l) \\
A_{0}(l,l') & = (\text{append } l \text{ } l'), & A_{\&}(l,l') & = (\text{inters } l \text{ } l')
\end{align*}
\]

where \((\text{inters } l \text{ } l')\) gives a list containing each element in \(l\) that also appears in \(l'\), with multiplicity properly respected.

Example 9 Nondeterministic Stack (continued). An implementation might use a function \(f : \text{Nat} \rightarrow \text{Nat}\) where \(f(n)\) is the \(n\)th randomly generated number. To ensure that \(n\) changes with each new call, we can keep it as a variable with the stack, incremented whenever a new number is pushed. Such an implementation is equivalent to the following model (we again use the [head,tail] notation for lists):

\[
\begin{align*}
A_{\text{Nat}} & = \mathbb{N}, & A_{\text{NdStack}} & = \mathbb{N} \times \mathbb{N}^* \\
A_{\text{empty}} & = (0,[]) \\
A_{\text{top}}(n,[]) & = 0, & A_{\text{top}}(n,[m,l]) & = m \\
A_{\text{pop}}(n,[]) & = (n,[]), & A_{\text{pop}}(n,[m,l]) & = (n,l) \\
A_{\text{push}}(n,l) & = (n+1,[f(n),l]), \quad \text{for all natural numbers } n,m \text{ and all lists } l \text{ of natural numbers.}
\end{align*}
\]
Therefore, there may be many possible implementations of a hidden signature, each with its associated hidden algebra. A natural question is whether there is any "special" model, such as the initial model (the term algebra) for algebraic signatures [74, 43]. It turns out that for certain hidden signatures there exists a final model (see Subsection VI.A.3), but this is not an important issue for this work because we do not advocate a final (co)algebra semantics.

### III.C Experiments and Contexts

Before we introduce the main ingredient of hidden logic, which is behavioral equality, we need to formalize the informal notion of "experiment":

**Definition 11** Given a hidden signature $\Gamma$, an (appropriate) $\Gamma$-context for sort $s$ is a term in $T_{\Gamma}(\{\bullet : s\} \cup Z)$ having exactly one occurrence of a special variable$^2$ $\bullet$ of sort $s$, where $Z$ is an infinite set of special variables. We let $C_{\Gamma}[\bullet : s]$ denote the set of all $\Gamma$-contexts for sort $s$, and $\text{var}(c)$ the finite set of variables of a context $c$, except $\bullet$.

![Figure III.5: A $\Gamma$-context.](image)

If $c \in C_{\Gamma}[\bullet : s]$ and $t \in T_{\Sigma,s}(X)$, then $c[t]$ denotes the term in $T_{\Sigma}(Z \cup X)$ obtained from $c$ by substituting $t$ for $\bullet$. A $\Gamma$-context which is a term of visible result is called a $\Gamma$-experiment. We let $E_{\Gamma}[\bullet : s]$ denote the set of all $\Gamma$-experiments for sort $s$. When the sort of contexts or experiments is important then we use the same notation as for terms: $C_{\Gamma,s'}[\bullet : s]$ is the set of $\Gamma$-contexts of sort $s'$ for sort $s$, while $E_{\Gamma,v}[\bullet : s]$ is the set of $\Gamma$-experiments of sort $v$ for sort $s$.

---

$^2$"Special variables" are assumed to be different from any other variable in a given situation.
The interesting contexts and experiments are those for hidden sort, that is, those with \( s \in H \). Those of visible sort are allowed more for esthetical reasons, to make the presentation smoother. The role of experiments is to access the information of interest encapsulated in a state (or object, or “blackbox”).

**Example 6 Cell** (continued). If \( \Gamma \) is the signature of cells then the \( \Gamma \)-experiments for cells have the form \( \text{get}(\text{put}(z_1, \text{put}(z_2, ..., \text{put}(z_k, \bullet) ...))) \) for arbitrary \( k \geq 0 \) and variables \( z_1, z_2, ..., z_k \) of sort \( \text{Elt} \). The set of \( \Gamma \)-contexts \( C_\Gamma[\bullet : \text{Cell}] \) consists of all the \( \Gamma \)-experiments plus all the terms of the form \( \text{put}(z_1, \text{put}(z_2, ..., \text{put}(z_k, \bullet) ...)) \).

**Example 7 Flag** (Continued). Let \( \Gamma \) be the hidden signature of flags. Then (by abuse of notation) the \( \Gamma \)-experiments for sort \( \text{Flag} \) are all those terms of the form \( \{ \text{not}\}^*(\text{up}?(\{\text{up, down, rev}\}^*(\bullet))) \), that is, terms starting with a series of \( \text{not} \) operations from \( \text{Bool} \), followed by the attribute \( \text{up}? \), followed by any number of methods \( \text{up, down, rev} \), and then by the special variable \( \bullet : \text{Flag} \). The \( \Gamma \)-contexts extend the set of \( \Gamma \)-experiments with all the terms \( \{ \text{up, down, rev}\}^*(\bullet) \).

**Example 8 Set** (continued). The \( \Gamma \)-contexts and \( \Gamma \)-experiments are more complex if \( \Gamma \) is the signature of sets. Instead of listing all the patterns, which would be tedious, we’d rather list just a few of them:

\[
\begin{align*}
n \in & \bullet \quad \text{(\( \Gamma \)-experiment)} \\
n \in & (\text{add}(n', \bullet)) \quad \text{(\( \Gamma \)-experiment)} \\
\bullet & \cup s \\
\bullet & \cup (s' \& \bullet) \\
n \in & (s \cup (s' \& \bullet)) \\
n \in & (s \cup (\text{add}(n', s' \& \bullet) \& \text{add}(n', \text{empty}))) \quad \text{(\( \Gamma \)-experiment)}
\end{align*}
\]

Notice that we used in-fix notation for \( \in, \cup, \) and \( \& \).

Therefore, contexts and experiments give a formal way to represent the tests that can be performed on the system. The state to be tested will be placed in the position of \( \bullet \). It is often the case that not all the operations of a hidden signature are intended to be used in experiments, but only part of them. To do that, we first need the following:
Definition 12 A loose data hidden subsignature of $\Sigma$ is a loose data hidden $(V, H)$-signature $\Gamma$ with $\Gamma \subseteq \Sigma$ and $\Gamma|_V = \Sigma|_V$. A fixed data hidden subsignature of $(\Sigma, D)$ is a fixed data hidden $(V, H)$-signature $(\Gamma, D)$ over the same data, with $\Gamma \subseteq \Sigma$ and $\Gamma|_V = \Sigma|_V$.

Example 9 Nondeterministic Stack (continued). Having in mind the random number generator implementation described in Section III.B, it is clear that since $\text{top}$ and $\text{pop}$ suffice to reach all the numbers stacked, they can completely “observe” the state. Moreover, $\text{push}$ can not be used in experiments because its behavior is intended to be unknown. Taking $\Gamma$ to be the hidden subsignature obtained by removing $\text{push}$ from the signature presented in Example 9, the only $\Gamma$-experiments are those of the form $\text{top}(\text{pop}^*)(\bullet)$; the $\Gamma$-contexts additionally include the terms $\text{pop}^*(\bullet)$.

Once a context or experiment is given together with a certain state of a system, the next step is to evaluate it.

Definition 13 Let $\Sigma$ be a hidden signature, $\Gamma$ a hidden subsignature of $\Sigma$, and $c \in C_\Gamma[\bullet : s]$ a $\Gamma$-context. Given a term $t \in T_\Sigma(X)$ of sort $s$, we let $c[t] \in T_\Sigma(\text{var}(c) \cup X)$ denote the $\Sigma$-term obtained by replacing $\bullet$ with $t$. Formally, $c[t] = (\bullet \rightarrow t)^*(c)$, where $(\bullet \rightarrow t)^*: T_\Sigma(\text{var}(c) \cup \{\bullet : s\}) \rightarrow T_\Sigma(\text{var}(c) \cup X)$ is the unique extension of the map $(\bullet \rightarrow t): \text{var}(c) \cup \{\bullet : s\} \rightarrow T_\Sigma(\text{var}(c) \cup X)$ which is identity on $\text{var}(c)$ and takes $\bullet : s$ to $t$. Furthermore, $c$ generates a map $A_c: A_h \rightarrow [A^{\text{var}(c)} \rightarrow A_s]$ on each $\Sigma$-algebra $A$, defined by $A_c(a)(\theta) = a^*_\theta(c)$, where $a^*_\theta$ is the unique extension of the map (denoted $a_\theta$) that takes $\bullet$ to $a$ and each $z \in \text{var}(c)$ to $\theta(z)$.

We take the liberty of replacing $A^{\text{var}(c)}$ by a product of sets whenever this eases reading.

Example 6 Cell (continued). Consider the experiment $\gamma = \text{get}(\text{put}(z, \bullet))$ and the context $c = \text{put}(z, \text{put}(z', \bullet))$. Three fixed data hidden algebras for cells were presented in Section III.B. For the first, $A_\gamma: \{[n] : n \in \mathbb{N}\} \rightarrow [\mathbb{N} \rightarrow \mathbb{N}]$ is the function defined by $A_\gamma([n])(m) = m$, while $A_c: \{[n] : n \in \mathbb{N}\} \rightarrow [\mathbb{N} \times \mathbb{N} \rightarrow \{[n] : n \in \mathbb{N}\}]$ is the function defined by $A_c([n])(m, m') = [m]$. For the second, $A_\gamma: \mathbb{N}^* \rightarrow [\mathbb{N} \rightarrow \mathbb{N}]$ is the function defined by $A_\gamma(l)(m) = m$, while $A_c: \mathbb{N}^* \rightarrow [\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}^*]$ is the function defined by $A_c(l)(m, m') = [m, m', l]$. For the third and least interesting algebra, $A_\gamma: \{\bullet\} \rightarrow [\mathbb{N} \rightarrow \mathbb{N}]$
is the function defined by $A_\gamma(\star)(m) = 0$, while $A_c : \{\star\} \to [\mathbb{N} \times \mathbb{N} \to \{\star\}]$ is the function defined by $A_c(\star)(m) = \star$. For the loose data situation, $\mathbb{N}$ may be replaced by any set.

**Example 7** Flag (continued). Let $\gamma$ be the experiment $\text{up}?(\text{rev}(%(\text{down}(\bullet))))$ and $c$ be the context $\text{rev}(\text{rev}(\text{up}(\text{down}(\bullet))))$. Notice that neither $\gamma$ nor $c$ have variables. For the first implementation presented in Example 7 in Section III.B, $A_\gamma : \{u, d\} \to \mathbb{B}$ is the constant function $\text{false}$, while $A_c : \{u, d\} \to \{u, d\}$ is the constant function $u$.

For the second hidden algebra, $A_\gamma : \{u, d, r\}^* \to \mathbb{B}$ is also the constant function $\text{false}$ (because $\text{not}(\text{not}(\text{false})) = \text{false}$ in $\mathbb{B}$), while $A_c : \{u, d, r\}^* \to \{u, d, r\}^*$ is defined by $A_c(l) = [r, r, u, d, l]$.

**Example 8** Set (continued). If $\gamma$ is the experiment $n$ in $S \cup (S' \& \bullet)$ and $c$ is the context $S \cup (\bullet \& \text{add}(n, \text{empty}))$ then $A_\gamma : \mathcal{P}(\mathbb{N}) \to [\mathbb{N} \times \mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N}) \to \mathbb{B}]$ is defined by $A_\gamma(X)(m, S, S') = \text{the truth value of the expression } m \in S \text{ or } m \in S' \text{ and } m \in X$ and $A_c : \mathcal{P}(\mathbb{N}) \to [\mathbb{N} \times \mathcal{P}(\mathbb{N}) \to \mathcal{P}(\mathbb{N})]$ is defined by $A_c(X)(m, S) = S \cup (X \cap \{n\})$ for the first hidden algebra of sets in Example 8 in Section III.B, and $A_\gamma : \mathbb{N}^* \to [\mathbb{N} \times \mathbb{N}^* \times \mathbb{N}^* \to \mathbb{B}]$ is defined by $A_\gamma(x)(m, l, l') = (\text{member } m (\text{append } l (\text{inters } l' x)))$ and $A_c : \mathbb{N}^* \to [\mathbb{N} \times \mathbb{N}^* \to \mathbb{N}^*]$ is defined by $A_c(x)(m, l) = (\text{append } l (\text{inters } x [m]))$ for the implementation of sets as lists.

**Example 9** Nondeterministic Stack (continued). We consider a $\Gamma$-experiment and a $\Gamma$-context, say $\gamma = \text{top}(\text{pop}(\bullet))$ and $c = \text{pop}(\text{pop}(\bullet))$. Then $A_\gamma : \mathbb{N} \times \mathbb{N}^* \to \mathbb{N}$ is defined by $A_\gamma(n, []) = 0$, $A_\gamma(n, [m]) = 0$, and $A_\gamma(n, [m, m', l]) = m'$, while $A_c : \mathbb{N} \times \mathbb{N}^* \to \mathbb{N} \times \mathbb{N}^*$ is defined by $A_c(n, []) = (n, [])$, $A_c(n, [m]) = (n, []), A_c(n, [m, m', l]) = (n, l)$, for all $n, m, m' \in \mathbb{N}$ and $l \in \mathbb{N}^*$.

### III.D Behavioral Equivalence and Congruent Operations

We are now ready to define the most distinctive feature of hidden logic, behavioral equivalence. The intuition is that two states are behaviorally equivalent iff they cannot be distinguished by any experiment that can be performed on the system.

**Definition 14** Given a hidden $\Sigma$-algebra $A$ and a hidden subsignature $\Gamma$ of $\Sigma$, the
equivalence given by \( a \equiv_\Gamma a' \text{ iff } A_\gamma(a)(\theta) = A_\gamma(a')(\theta) \) for all \( \Gamma \)-experiments \( \gamma \) and all maps \( \theta: \text{var}(\gamma) \rightarrow A \) is called \( \Gamma \)-behavioral equivalence on \( A \).

![Diagram](image)

Figure III.6: \( \Gamma \)-behavioral equivalence of \( a, a' \).

We write \( \equiv \) instead of \( \equiv_\Gamma \) whenever \( \Sigma \) and \( \Gamma \) can be deduced from the context, and write \( \equiv_\Sigma \) whenever \( \Sigma = \Gamma \).

**Example 6 Cell** (continued). Behavioral equivalence is identity for the first model of cells (see Example 6 in Section III.B). Indeed, if \([m] \equiv [n]\) then by testing the experiment \( \gamma = \text{get}(\bullet) \), one gets \( A_\gamma([m]) = A_\gamma([n]) \), so that \( m = n \). On the other hand, for the second model of cells, \([n,l] \equiv [n,l']\) for any natural number \( n \) and any lists of natural numbers \( l,l' \), because \( A_{\text{get}(\bullet)} \) evaluates to \( n \) when applied to the two states, \( A_{\text{get(put}(z,\bullet))} \) evaluates to the value of \( z \), namely \( \theta(z) \), \( A_{\text{get(put}(z,\text{put}(z',\bullet)))} \) also evaluates to \( \theta(z) \), etc.

In fact, it can be proved (Section IV.C) that only the experiment \( \text{get}(\bullet) \) is needed.

**Example 7 Flag** (continued). Behavioral equality is also the identity for the first implementation of flags in Example 7 of Section III.B because the experiment \( \text{up?}(\bullet) \) gives complete information about the state (if its evaluation on a state \( s \) is \text{true} then \( s \) is \( u \) and otherwise \( s \) is \( d \)). But for the second implementation, for example, \([r,u,d,r,d,l] \equiv [r,r,r,d,l',l']\) for any lists \( l,l' \in \{u,d,r\}^* \). Indeed, the experiment \( \text{up?}(\bullet) \) evaluates to \text{false} on both states, and this is the only experiment of interest. That can be easily justified by an inductive argument as follows: for any appropriate context \( c \), \( A_{\text{up?}(\text{up}(c(\bullet)))} \) evaluates to \text{true}, \( A_{\text{up?}(\text{down}(c(\bullet)))} \) evaluates to \text{false}, and for any two states \( f, f' \in \{u,d,r\}^* \), \( A_{\text{up?}(\text{rev}(c(\bullet)))}(f) = A_{\text{up?}(\text{rev}(c(\bullet)))}(f') \text{ iff } A_{\text{up?}(c(\bullet))}(f) = A_{\text{up?}(c(\bullet))}(f') \).

**Example 8 Set** (continued). For the two hidden algebras for sets of natural numbers presented in Example 8 in Section III.B, it can be easily seen by an inductive argument...
as above that membership is the only experiment of interest. Thus, in the first hidden algebra, two sets are behaviorally equivalent iff they are equal sets. On the other hand, in the second hidden algebra which implements sets as lists, two sets are behaviorally equivalent iff they have the same elements as lists; the order and the number of occurrences of elements do not matter. For example, \([1, 2] \equiv [2, 1, 2, 2, 1]\).

**Example 9 Nondeterministic Stack** (continued). Two states are behaviorally equivalent iff they contain the same generated numbers in the same order, when the experiments are those given in Section III.C. That follows easily because the only experiments allowed are \(\text{top}(\text{pop}^*(\ast))\). Thus, \((n, l) \equiv (n', l)\) for any numbers \(n, n'\) and any list \(l\).

**Definition 15** Given any equivalence \(\sim\) on \(A\), an operation \(\sigma\) in \(\Sigma_{s_1...s_n,s}\) is congruent for \(\sim\) iff \(A_\sigma(a_1, ..., a_n) \sim A_\sigma(a'_1, ..., a'_n)\) whenever \(a_i \sim a'_i\) for \(i = 1...n\). An operation \(\sigma\) is \(\Gamma\)-behaviorally congruent for \(A\) iff it is congruent for \(\equiv^\Gamma_\Sigma\).

We will often say just “congruent” instead of “behaviorally congruent”\(^3\).

**Example 16** All the operations in Examples 6, 7, 8, 9 except \(\text{push}\) of Example 9 are congruent for all the corresponding hidden algebras presented. For example, \(\emptyset\) is congruent for the hidden algebra implementing sets as lists because \((\text{append} \ l_1 \ l_2)\) and \((\text{append} \ l'_1 \ l'_2)\) have the same elements when the lists \(l_1, l'_1\) and \(l_2, l'_2\) have the same elements (the order and the number of occurrences do not matter), respectively. However, \(\text{push}\) is not congruent for the implementation presented; indeed, given two behaviorally equivalent states \((n, l)\) and \((n', l)\), then \(A_{\text{push}}(n, l) = (n + 1, [f(n), l])\) and \(A_{\text{push}}(n', l) = (n' + 1, [f(n'), l])\) may not be behaviorally equivalent because \(f(n) \neq f(n')\) in general.

**Definition 17** A hidden \(\Gamma\)-congruence on \(A\) is an equivalence on \(A\) which is the identity on visible sorts and is such that each operation in \(\Gamma\) is congruent for it.

The following is the basis for several of our results, especially coinduction; it generalizes a similar result in [65] to operations that can have more than one hidden argument or are not behavioral.

\(^3\)A similar notion was given by Padawitz in [121].
Theorem 18 Given a hidden subsignature $\Gamma$ of $\Sigma$ and a hidden $\Sigma$-algebra $A$, then $\Gamma$-behavioral equivalence is the largest hidden $\Gamma$-congruence on $A$.

Proof: We first show that $\equiv_{\Sigma}^{\Gamma}$ is a hidden $\Gamma$-congruence. It is straightforward that it is the identity on visible sorts because we can take the trivial experiment $\gamma = \bullet$ for any visible sort. Now let $\sigma : s_1...s_n \rightarrow s$ be any operation in $\Gamma$, let $a_1 \equiv_{\Sigma,s_1}^{\Gamma} a'_1$, ..., $a_n \equiv_{\Sigma,s_n}^{\Gamma} a'_n$, let $\gamma$ be any $\Gamma$-experiment for sort $s$, and let $\theta : \text{var}(\gamma) \rightarrow A$ be any map. Let $z_1 : s_1,...,z_n : s_n$ be variables in $Z$ distinct from those in $\text{var}(\gamma)$, and take the $\Gamma$-experiments $\gamma_j = \gamma[\sigma(Z_j, \bullet)]$ for sorts $s_j$, where $Z_j$ is the set of variables $\{z_1,...,z_{j-1},z_{j+1},...,z_n\}$, for all $1 \leq j \leq n$. Notice that $\text{var}(\gamma_j) = Z_j \cup \text{var}(\gamma)$. Now, let us consider the maps $\theta_j : \text{var}(\gamma_j) \rightarrow A$ to be defined by $\theta_j(z_i) = a'_i$ for $1 \leq i < j$, $\theta_j(z_i) = a_i$ for $j < i \leq n$, and $\theta_j(z') = \theta(z')$ for $z' \in \text{var}(\gamma)$, for all $1 \leq j \leq n$. Notice that $A_\gamma(A_\sigma(a_1,...,a_n))(\theta) = A_{\epsilon_1}(a_1)(\theta_1)$, $A_{\gamma_j}(a_j)(\theta_j) = A_{\gamma_j}(a'_j)(\theta_j)$ for all $1 \leq j \leq n$ because $a_j \equiv_{\Sigma,s_j}^{\Gamma} a'_j$ and $\gamma_j$ and $\theta_j$ are appropriate $\Gamma$-experiments and maps, that $A_{\gamma_j}(a'_j)(\theta_j) = A_{\gamma_{j+1}}(a_{j+1})(\theta_{j+1})$ for all $1 \leq j < n$, and that $A_{\gamma_n}(a'_n)(\theta_n) = A_{\gamma}(A_\sigma(a'_1,...,a'_n))(\theta)$. Then $A_{\gamma}(A_\sigma(a_1,...,a_n))(\theta) = A_{\gamma}(A_\sigma(a'_1,...,a'_n))(\theta)$, that is, $A_\sigma(a_1,...,a_n) \equiv_{\Sigma,s}^{\Gamma} A_\sigma(a'_1,...,a'_n)$. Therefore, the operation $\sigma$ is $\Gamma$-behaviorally congruent for $A$, and so $\equiv_{\Sigma}^{\Gamma}$ is a hidden $\Gamma$-congruence.

Now let $\sim$ be another hidden $\Gamma$-congruence on $A$ and let $a \sim a'$. Because each operation in $\Gamma$ is congruent for $\sim$, $A_\gamma(a)(\theta) \sim A_\gamma(a')(\theta)$ for any $\Gamma$-experiment $\gamma$ for sort $s$ and any map $\theta : \text{var}(\gamma) \rightarrow A$, and because $\sim$ is the identity on visible sorts, $A_\gamma(a)(\theta) = A_\gamma(a')(\theta)$. Therefore $a \equiv_{\Sigma,s}^{\Gamma} a'$, that is, $\sim \subseteq \equiv_{\Sigma}^{\Gamma}$. $\Box$

Notice that the existence of the largest hidden $\Gamma$-congruence does not depend on the existence of a final model, like in the case of coalgebra [145, 98, 95] (see also Section VI.A.3).

III.E Behavioral Satisfaction

Any logic provides a binary relation between its models and its formulas, called “satisfaction”. The models of hidden logic are hidden algebras, while the formulas are equations:

Definition 19 A hidden $\Sigma$-algebra $A$ $\Gamma$-behaviorally satisfies the conditional $\Sigma$-
equation \((\forall X) t = t'\) if \(t_1 = t'_1, \ldots, t_n = t'_n\), say \(e\), iff for each \(\theta : X \to A\), if \(\theta(t_i) \equiv^\Sigma \theta(t'_i)\) for \(i = 1, \ldots, n\), then \(\theta(t) \equiv^\Sigma \theta(t')\); in this case we write \(A \models^\Sigma e\). If \(E\) is a set of conditional \(\Sigma\)-equations, we write \(A \models^\Sigma E\) if \(A\) \(\Gamma\)-behaviorally satisfies each conditional \(\Sigma\)-equation in \(E\).

When \(\Sigma\) and \(\Gamma\) are clear from context, we may write \(\equiv\) and \(\models\) instead of \(\equiv^\Sigma\) and \(\models^\Sigma\), respectively. Notice that the behavioral satisfaction relation depends on \(\Gamma\), the set of operations used for experiments. If the equation is unconditional, then the definition above becomes: \(A \models^\Gamma (\forall X) t = t'\) iff \(\theta(t) \equiv^\Gamma \theta(t')\) for each \(\theta : X \to A\).

**Example 6 Cell** (continued). Let \(e\) be the equation \((\forall E,C) \text{get} \circ \text{put}(E,C) = E\), where \(E\) and \(C\) are variables of sorts \(\text{Elt}\) and \(\text{Cell}\), respectively. Then the first two models in Example 6 in Section III.B satisfy \(e\), while the third does not. Given a natural number \(n\) and a cell \(c\) (i.e., a map \(\theta : \{N,C\} \to A\)), both terms of \(e\) evaluate to \(n\) in the first two models, while the first term evaluates to 0 and the second to \(n\) in the third model. On the other hand, \((\forall E,C,C') \text{put}(E,C) = \text{put}(E,C')\) is strictly satisfied by the first and the third models, but only behaviorally satisfied by the second.

Notice that only standard satisfaction was needed for the first equation in the example above. That’s because the equation had a visible sort and that the behavioral equality relation is the identity on visible sorts.

**Example 7 Flag** (continued). Perhaps one of the most interesting equations for flags is \((\forall F) \text{rev} \circ \text{rev}(F) = F\). This equation is strictly satisfied by the first model of flags in Example 7 in Section III.B, but it is only behaviorally satisfied by the second. Indeed, a flag in the first model is either \(u\) or \(d\), and \(A_{\text{rev}}(A_{\text{rev}}(u)) = A_{\text{rev}}(d) = u\) and \(A_{\text{rev}}(A_{\text{rev}}(d)) = A_{\text{rev}}(u) = d\). On the other hand, a flag in the second model can be any list \(l \in \{u,d,r\}^*\) and \(A_{\text{rev}}(A_{\text{rev}}(l)) = [r,r,l]\); using a technique similar to the one in Example 7 in Section III.D, it can be easily proved that \([r,r,l] \equiv l\).

**Example 8 Set** (continued). We encourage the interested reader to show that usual properties of sets, such as commutativity of union and intersection, distributivity, etc., are behaviorally satisfied by the two models of sets presented in Section III.B. However, notice that the first model satisfies them strictly, while the second does not.
Example 9 Nondeterministic Stack (continued). It can be readily seen that the equation \((\forall S) \text{pop} (\text{push}(S)) = S\) is behaviorally satisfied by the random number generator implementation described in Section III.B, but it is not strictly satisfied. Indeed,

\[ A_{\text{pop}} (A_{\text{push}}(n, l)) = A_{\text{pop}} (n + 1, [f(n), l]) = (n + 1, l) \]

and \((n, l)\) is behaviorally equivalent to \((n + 1, l)\) (they have the same generated numbers stacked; see Example 9 in Section III.D).

Notice that all equations used so far in examples were unconditional. In fact, most of the equations involved in concrete examples in this thesis and the related work of which we are aware are unconditional; moreover, the few conditional equations occurring in practical situations almost always have visible conditions, in the following sense:

Definition 20 A conditional \(\Sigma\)-equation, say \(e, (\forall X) t = t' \text{ if } t_1 = t'_1, ..., t_n = t'_n\) with \(n \geq 0\) is of visible condition if the sorts of \(t_1, t'_1, ..., t_n, t'_n\) are visible, respectively. We let \(\mathcal{E}_\Gamma[e]\) denote either the set of conditional \(\Sigma\)-equations

\[ \{(\forall X, \text{var}(\gamma)) \gamma[t] = \gamma[t'] \text{ if } t_1 = t'_1, ..., t_n = t'_n \} \mid \gamma \in \mathcal{E}_\Gamma[\bullet : h] \]

when the sort \(h\) of \(t\) and \(t'\) is hidden, or the set \{\(e\)\} when the sort of \(t\) and \(t'\) is visible. If \(E\) is a set of conditional \(\Sigma\)-equations of visible conditions, then let \(\mathcal{E}_\Gamma[E]\) be the set of conditional \(\Sigma\)-equations \(\bigcup_{e \in E} \mathcal{E}_\Gamma[e]\).

An (unconditional) \(\Sigma\)-equation is of course a special case of conditional \(\Sigma\)-equation of visible condition. It turns out that the equations of visible condition play an important rôle in hidden logic. This is not surprising, because, unlike for conditional equations of hidden condition where in order to test the condition one may need to test an infinite number of experiments, only a finite number of evaluations is needed. In some sense, the difference between conditional equations of visible condition and those of hidden condition is the same as the difference between conditional equations with finite conditions and with infinite conditions in equational logic. The following is a key result, reducing behavioral satisfaction to strict equational satisfaction of an infinite number of sentences:
Proposition 21 Let \( E \) be a set of conditional \( \Sigma \)-equations of visible condition. Then

1. \( A \models^\Gamma_\Sigma E \) iff \( A \models_\Sigma \mathcal{E}_\Gamma[E] \), for any hidden \( \Sigma \)-algebra \( A \);

2. Under the loose data framework, if \( e \) is a conditional \( \Sigma \)-equation of visible condition, then \( E \models^\Gamma_\Sigma e \) iff \( \mathcal{E}_\Gamma[E] \models_\Sigma \mathcal{E}_\Gamma[e] \).

Therefore, behavioral satisfaction is reduced to strict satisfaction.

Proof: It suffices to show that \( A \models^\Gamma_\Sigma e \) iff \( A \models_\Sigma \mathcal{E}_\Gamma[e] \), for any hidden \( \Sigma \)-algebra \( A \) and any conditional equation \( e \) of visible condition, \( (\forall X) \ t = t' \) if \( t_1 = t'_1, \ldots, t_n = t'_n \). Since \( e \) is of visible condition, \( \theta(t_i) \equiv \theta(t'_i) \) iff \( \theta(t_i) = \theta(t'_i) \), for any \( \theta : \ X \to A \) and any \( 1 \leq i \leq n \). First, suppose that \( A \models^\Gamma_\Sigma e \) and let us consider any equation \( (\forall X, var(\gamma)) \ \gamma[t] = \gamma'[t'] \) if \( t_1 = t'_1, \ldots, t_n = t'_n \) in \( \mathcal{E}_\Gamma[e] \). Let \( \theta' : X \cup var(\gamma) \to A \) be any assignment such that \( \theta'(t_i) = \theta'(t'_i) \) for all \( 1 \leq i \leq n \). Since \( t, t', t_1, t'_1, \ldots, t_n, t'_n \) contain only variables in \( X \), one gets that \( \theta'|_X(t_i) = \theta'|_X(t'_i) \) for all \( 1 \leq i \leq n \), so \( \theta'|_X(t) = \theta'|_X(t') \). Further, by the definition of behavioral equivalence, \( A_\gamma(\theta'|_X(t))(\theta'|_\text{var}(\gamma)) = A_\gamma(\theta'|_X(t'))(\theta'|_\text{var}(\gamma)) \). But notice that \( A_\gamma(\theta'|_X(t))(\theta'|_\text{var}(\gamma)) = \theta(\gamma[t]) \) and similarly for \( t' \), so \( \theta(\gamma[t]) = \theta(\gamma[t']) \).

Conversely, suppose that \( A \models_\Sigma \mathcal{E}_\Gamma[e] \) and let \( \theta : X \to A \) be a map such that \( \theta(t_i) = \theta(t'_i) \) for all \( 1 \leq i \leq n \). Let \( \gamma \) be any \( \Gamma \)-experiment appropriate for \( t, t' \) and let \( \theta_\gamma : \text{var}(\gamma) \to A \) be any map. Let \( \theta' : X \cup \text{var}(\gamma) \to A \) be the map such that \( \theta'|_X = \theta \) and \( \theta'|_{\text{var}(\gamma)} = \theta_\gamma \). Then \( \theta'(\gamma[t]) = \theta'(\gamma[t']) \), and since \( \theta'(\gamma[t]) = A_\gamma(\theta(t))(\theta_\gamma) \) and similarly for \( t' \), one gets that \( \theta(t) = \theta(t') \). \( \square \)

III.F Behavioral Specification

Example 6 spread over the previous sections showed that there may be implementations of hidden signatures which do not satisfy desired properties. In order to restrict the class of models of a hidden signature, expected behavioral properties are added to the signature:

Definition 22 A behavioral (or hidden) \( \Sigma \)-specification (or -theory) is a triple \( (\Sigma, \Gamma, E) \), where \( \Sigma \) is a hidden signature, \( \Gamma \) is a hidden subsignature of \( \Sigma \), and \( E \) is a set of \( \Sigma \)-equations. The operations in \( \Gamma \) are called behavioral. We generally let \( \mathcal{B}, \mathcal{B}', \mathcal{B}_1 \), etc., denote behavioral specifications. A hidden \( \Sigma \)-algebra \( A \) behaves like
satisfies (or is a model of) a behavioral specification $B = (\Sigma, \Gamma, E)$ iff $A \models^\Gamma E$, and in this case we write $A \models B$; we write $B \models e$ whenever $A \models B$ implies $A \models^\Gamma e$. ■

Example 6 Cell (continued). The following is a behavioral specification of cells, in BOBJ notation:

```boj
bth CELL[X :: TRIV] is sort Cell .
   op put : Elt Cell -> Cell .
   op get : Cell -> Elt .
   var E : Elt . var C : Cell .
   eq get(put(E, C)) = E .
end
```

The module TRIV is builtin, and it does nothing but declares a visible sort Elt. The modules CELL is parameterized by [X :: TRIV], meaning that a loose data framework is desired. The declaration “sort Cell .” says that Cell is a hidden sort; hidden because the specification is behavioral (because of the keyword “bth”). There is only one equation, of visible sort, saying that “one gets what one puts”. In fact, all the equations used in this thesis and BOBJ are behavioral, in the sense that they are intended to be strictly satisfied when their sort is visible, while behaviorally satisfied when their sort if hidden.

The first two hidden algebras presented in Example 6 in Section III.B are models of CELL, while the third one isn’t. ■

Example 7 Flag (continued). The BOBJ specification of flags also contains only visible equations:

```boj
bth FLAG is sort Flag .
   ops (up_) (down_) (rev_) : Flag -> Flag .
   op up?_ : Flag -> Bool .
   var F : Flag .
   eq up? up F = true .
   eq up? down F = false .
   eq up? rev F = not up? F .
end
```

Notice the special syntax used to declare multiple operations of the same type and the mix-fix notation [80] to declare their arguments. The mix-fix notation is optional: by default, the prefix notation is considered. ■
Example 8 Set (continued). The following is a behavioral specification of sets:

```
bth SET[X :: TRIV] is sort Set .
  op empty : -> Set .
  op _in_ : Elt Set -> Bool .
  op add : Elt Set -> Set .
  ops (_U_) (_&_) : Set Set -> Set .
  vars E E' : Elt . vars S S' : Set .
  eq E in empty = false .
  eq E in add(E' , S) = (E == E') or (E in S).
  eq E in S U S' = (E in S) or (E in S') .
  eq E in S & S' = (E in S) and (E in S') .
end
```

Notice that the constants, e.g. `empty` above, are treated as operations without arguments in BOBJ. The operation `=_: s s -> Bool` occurring in the second equation above is special and built-in for any sort `s`; operationally, the two arguments are reduced to normal forms and then it returns true when the two are equal and returns false when they are different.

Example 9 Nondeterministic Stack (continued). The behavioral specification of nondeterministic stack may seem very simple, but it is tricky:

```
bth NDSTACK is protecting NAT .
  sort Stack .
  op top _ : Stack -> Nat .
  op push _ : Stack -> Stack [ncong] .
  op pop _ : Stack -> Stack .
  var S : Stack .
  eq pop push S = S .
end
```

The statement “`protecting NAT`” says that the module `NAT` of natural numbers is imported and semantically protected, that is, the models (hidden algebras) of `NDSTACK` are not in conflict with the models of natural numbers; for example, 0 must be different from 1 in all these models. `NDSTACK` contains only one equation of hidden sort, saying the basic consistency property, namely that a stack is not behaviorally changed by generating and then removing a new number. Notice that `top push S` is left undefined, as one doesn’t know (and doesn’t want to know) at this stage what number will be generated next.
**Definition 23** An operation \( \sigma \in \Sigma \) is **behaviorally congruent for** \( B \) iff \( \sigma \) is behaviorally congruent for every \( A \models B \).

The following proposition provides some operations which are congruent:

**Proposition 24** If \( B = (\Sigma, \Gamma, E) \) is a behavioral specification, then all operations in \( \Gamma \) and all hidden constants are behaviorally congruent for \( B \).

**Proof:** Since \( \equiv^\Gamma_\Sigma \) is a hidden \( \Gamma \)-congruence, all the operations in \( \Gamma \) are behaviorally congruent. On the other, the hidden constants are also behaviorally congruent because they have only visible arguments and the behavioral equivalence is the identity on those sorts. \( \square \)

We will see later that, depending on \( E \), other operations can be also congruent. In fact, our experience so far is that in most practical situations, all the operations are congruent. The next result supports the elimination of hidden universal quantifiers in proofs. Many logicians also call it the “Generalization Theorem”:

**Theorem 25 Theorem of Hidden Constants:** If \( B = (\Sigma, \Gamma, E) \) is a behavioral specification, \( e \) is the \( \Sigma \)-equation \( (\forall Y, X) \ t = t' \) if \( t_1 = t'_1, \ldots, t_n = t'_n \) where \( X \) contains only hidden variables, and \( e_X \) is the \( (\Sigma \cup X) \)-equation \( (\forall Y) \ t = t' \) if \( t_1 = t'_1, \ldots, t_n = t'_n \), then \( B \models e \) iff \( B_X \models e_X \), where \( B_X = (\Sigma \cup X, \Gamma, E) \).

**Proof:** Suppose \( B \models e \), let \( A' \) be a \( (\Sigma \cup X) \)-algebra such that \( A' \models B_X \), and let \( A \) be the \( \Sigma \)-algebra \( A'|_\Sigma \). Notice that the \( \Gamma \)-behavioral equivalences on \( A \) and \( A' \) coincide, and that \( A \models^\Gamma_\Sigma E \). We will write \( \equiv \) for both behavioral equivalences. Let \( \theta : Y \to A' \) be such that \( \theta(t_i) \equiv \theta'(t'_i) \) for \( i = 1 \ldots n \), and let \( \tau : Y \cup X \to A \) be defined by \( \tau(y) = \theta(y) \) for all \( y \in Y \), and \( \tau(x) = A'_x \) for all \( x \in X \). Notice that \( \tau(t_i) = \theta(t_i) \equiv \theta'(t'_i) = \tau'(t'_i) \) for \( i = 1 \ldots n \), and that \( A \models^\Gamma_\Sigma e \) since \( A \models^\Gamma_\Sigma E \). Therefore, \( \tau(t) \equiv \tau'(t) \), so that \( \theta(t) \equiv \theta'(t) \). Consequently, \( A' \models^\Gamma_{\Sigma \cup X} e_X \), so that \( B_X \models e_X \).

Conversely, suppose \( B_X \models e_X \), let \( A \) be a \( \Sigma \)-algebra with \( A \models B \), and let \( \tau : Y \cup X \to A \) be such that \( \tau(t_i) \equiv^\Gamma_\Sigma \tau(t'_i) \) for \( i = 1 \ldots n \). Let \( A' \) be the \( (\Sigma \cup X) \)-algebra with the same carriers as \( A \), and the same interpretations of operations in \( \Sigma \), but with \( A'_x = \tau(x) \) for each \( x \) in \( X \). Notice that the \( \Gamma \)-behavioral equivalences on \( A \) and \( A' \)
coincide, both noted \( \equiv \) from now on. Also notice that \( A' \models_{\Sigma,X}^F E \), so that \( A' \models_{\Sigma,X} \epsilon X \).

Let \( \theta : Y \to A' \) be the map defined by \( \theta(y) = \tau(y) \) for each \( y \in Y \). It is straightforward that \( \theta(t_i) = \tau(t_i) \equiv \tau(t'_i) = \theta(t'_i) \) for \( i = 1 \ldots n \), so that \( \theta(t) \equiv \theta(t') \), that is, \( \tau(t) \equiv \tau(t') \).

Therefore, \( A \models_{\Sigma} \epsilon \), so that \( B \models \epsilon \). \( \Box \)

The following justifies implication elimination for conditional hidden equations:

**Proposition 26 Deduction Theorem:** Given behavioral specification \( B = (\Sigma, \Gamma, E) \) and \( t_1, t'_1, \ldots, t_n, t'_n \) ground hidden terms, let \( E' \) be \( E \cup \{ (\forall \emptyset) t_1 = t'_1, \ldots, (\forall \emptyset) t_n = t'_n \} \), and let \( B' \) be the behavioral specification \( (\Sigma, \Gamma, E') \). Then

\[ B' \models (\forall X) t = t' \iff B \models (\forall X) t = t' \text{ if } t_1 = t'_1, \ldots, t_n = t'_n. \]

**Proof:** Given any hidden \( \Sigma \)-algebra \( A \), let \( A_{t_1}, A_{t'_1}, \ldots, A_{t_n}, A_{t'_n} \) be the evaluations of the ground terms \( t_1, t'_1, \ldots, t_n, t'_n \) in \( A \), that is, their images in \( A \) by the unique morphism \( \alpha_A : T_{\Sigma} \to A \). Suppose that \( B' \models (\forall X) t = t' \); let \( A \models B \) and let \( \theta : X \to A \) be a map such that \( \theta(t_i) \equiv \theta(t'_i) \) for all \( 1 \leq i \leq n \). Since \( t_i, t'_i \) are ground, one gets that \( \theta(t_i) = A_{t_i}, \theta(t'_i) = A_{t'_i} \) for all \( 1 \leq i \leq n \). Therefore \( A \models B' \), so \( A \models_{\Sigma} (\forall X) t = t' \), and that implies that \( \theta(t) \equiv \theta(t') \).

Conversely, suppose that \( B \models (\forall X) t = t' \text{ if } t_1 = t'_1, \ldots, t_n = t'_n \); let \( A \) be a hidden \( \Sigma \)-algebra such \( A \models B' \), and let \( \theta : X \to A \) be a map. Then \( A \) behaviorally satisfies \( B \) and all the ground equations \( (\forall \emptyset) t_i = t'_i \), that is, \( A_{t_i} = A_{t'_i} \), or \( \theta(t_i) = \theta(t'_i) \), for all \( 1 \leq i \leq n \). Therefore, \( \theta(t) \equiv \theta(t') \). \( \Box \)

We will see many situations where the last two results apply in Section IV.B.
Chapter IV

Behavioral Reasoning

This section introduces and justifies our rules for behavioral deduction. Suppose that $B = (\Sigma, \Gamma, E)$ is a fixed hidden specification throughout this chapter.

IV.A Hidden Equational Deduction

Ordinary equational deduction is not sound for behavioral satisfaction. This is because the congruence rule of deduction is not sound for operations that are not behaviorally congruent. One such example was the operation $\text{push}$ on the nondeterministic stack in Example 9. However,

**Proposition 27** The following hold:

1. $B \models (\forall X) t = t$.
2. $B \models (\forall X) t = t'$ implies $B \models (\forall X) t' = t$.
3. $B \models (\forall X) t = t'$ and $B \models (\forall X) t' = t''$ imply $B \models (\forall X) t = t''$.
4. If $B \models (\forall Y) t = t'$ if $t_1 = t'_1, ..., t_n = t'_n$ and $\theta : Y \rightarrow T_\Sigma(X)$ is a substitution such that $B \models (\forall X) \theta(t_i) = \theta(t'_i)$ for $i = 1..n$, then $B \models (\forall X) \theta(t) = \theta(t')$.
5. If $B \models (\forall X) t = t'$ then
   a. If $\text{sort}(t, t') \in V$ then $B \models (\forall X, W) \sigma(W, t) = \sigma(W, t')$ for all $\sigma \in \text{Der}(\Sigma)$,
   b. If $\text{sort}(t, t') \in H$ then $B \models (\forall X, W) \delta(W, t) = \delta(W, t')$ for all congruent $\delta \in \Sigma$.

**Proof:** The first three items are easy and we let them as exercises.
For the fourth, let $A$ be a hidden $\Sigma$-algebra such that $A \models B$ and let $\tau : X \to A$ be any map. Since $B \models (\forall X) \theta(t_i) = \theta(t'_i)$ for $i = 1 \ldots n$, one gets that $\tau(\theta(t_i)) \equiv \tau(\theta(t'_i))$.

Further, since $B \models (\forall Y) t = t'$ if $t_1 = t'_1, \ldots, t_n = t'_n$ and since $\theta; \tau : Y \to A$ is a map, it follows that $\tau(\theta(t)) \equiv \tau(\theta(t'))$.

For 5.a, let $A \models B$ and let $\theta : W \cup X \to A$. Notice that $\theta(\sigma(W,t)) = \sigma(W,\theta \downarrow W, \theta \downarrow X(t))$ and $\theta(\sigma(W,t')) = \sigma(W,\theta \downarrow W, \theta \downarrow X(t'))$. Since the sort of $t, t'$ is visible and $A \models \Gamma (\forall X) t = t'$, it follows that $\theta_X(t) = \theta_X(t')$. Therefore, $\theta(\sigma(W,t)) = \theta(\sigma(W,t'))$.

For 5.b, let $A$ and $\theta$ be as above; then $\theta_X(t) \equiv \Gamma \theta_X(t')$. Since $\delta$ is congruent for $A$, it follows that $A_{\delta}(\theta \downarrow W, W, \theta \downarrow X(t)) \equiv A_{\delta}(\theta \downarrow W, W, \theta \downarrow X(t'))$, that is, that $\theta(\delta(W,t)) \equiv \Gamma \theta(\delta(W,t'))$.

The rules below accordingly modify the usual equational deduction. Let us define $\equiv_{Eq}$ on terms by (1)–(5) below.

1. Reflexivity: $t \equiv_{Eq} t$

2. Symmetry: $t \equiv_{Eq} t' \Rightarrow t' \equiv_{Eq} t$

3. Transitivity: $t \equiv_{Eq} t', t' \equiv_{Eq} t'' \Rightarrow t \equiv_{Eq} t''$

4. Substitution: $\forall Y. t = t' \text{ if } t_1 = t'_1, \ldots, t_n = t'_n \in E, \theta : Y \to T_{\Sigma}(X), \theta(t_i) \equiv_{Eq} \theta(t'_i)\Rightarrow \theta(t) \equiv_{Eq} \theta(t')$

5. Congruence: $\Gamma$:
   
   \begin{align*}
   a) & \quad t \equiv_{Eq} t', \text{ sort}(t, t') \in V \Rightarrow \sigma(W,t) \equiv_{Eq} \sigma(W,t'), \text{ for each } \sigma \in Der(\Sigma) \\
   b) & \quad t \equiv_{Eq} t', \text{ sort}(t, t') \in H \Rightarrow \delta(W,t) \equiv_{Eq} \delta(W,t'), \text{ for each congruent } \delta \in \Sigma
   \end{align*}

Figure IV.1: Hidden equational deduction system.
In some situations $\mathcal{B}$ needs to be mentioned within derivations. For that reason, we let $\models \mathcal{B}$ be defined by $\mathcal{B} \models (\forall X) t = t'$ iff $t \equiv_{\text{Eq}} t'$.

If all operations are congruent then these rules become the usual rules of equational deduction. Notice that (5b) only applies to congruent operations. Therefore, the more congruent operations there are, the more powerful the inference system becomes, and the more behavioral equalities can be proved. Because all behavioral operations are behaviorally congruent (Proposition 24), we already have cases where (5b) applies. Use of this method is facilitated by techniques for proving that operations are congruent, including easy to check syntactic criteria, given in Section IV.E.

The following result expresses soundness of these rules with respect to both equational and behavioral satisfaction, thus generalizing the observation in [37, 40] that equational deduction is sound whenever all operations are congruent.

**Proposition 28** If $\mathcal{B} \models (\forall X) t = t'$ then $E \vdash_{\Sigma} (\forall X) t = t'$ and $\mathcal{B} \equiv (\forall X) t = t'$. If all operations are congruent, then equational deduction is sound for behavioral satisfaction.

**Proof:** The soundness for equational satisfaction follows from the observation that the rules (1)–(5) generate a subrelation of equational derivability, because the rule 5.b applies only for a reduced number of operations. Therefore, $\mathcal{B} \models (\forall X) t = t'$ implies $E \vdash_{\Sigma} (\forall X) t = t'$, and now it follows by the soundness of equational deduction.

The soundness for behavioral satisfaction follows easily by Proposition 27, because each rule is sound. □

The rules (1)–(5) above differ from those in [40] in allowing both congruent and non-congruent operations; moreover, behavioral rewriting, which will be presented in detail in Section V.C, is a special case in the same way that standard term rewriting is a special case of equational deduction.

**Example 6 Cell** (continued). It can be relatively easily shown that the equation $(\forall E, C) \text{get}(\text{put}(\text{get}(C), C)) = \text{get}(\text{put}(\text{get}(C), \text{put}(E, C)))$ can be derived using the rules (1)–(5) above from the behavioral specification CELL in Example 6 in Section III.F. It can be also proved automatically by BOBJ:

```
open CELL .
```
\texttt{red \textit{get(put(get(C), C)) == get(put(get(C), put(E, C))) \; close}}

which returns the following result:

\begin{verbatim}
reduce in CELL : get(put(get(C), C)) == get(put(get(C), put(E, C)))
result Bool: true
rewrite time: 87ms parse time: 18ms
\end{verbatim}

The rewrite and parse times obviously depend on machine speed; we ran it on a Sun Sparc 10 with many users. The block \texttt{\textit{open CELL \ldots close}} recreates the system environment when the module \texttt{CELL} was last defined, and then allows the user to add new statements if needed or fires the behavioral reduction engine of BOBJ with commands like \texttt{\textit{red}}. The command \texttt{\textit{red}} is a shorthand for \texttt{\textit{reduce}}; it takes a term and reduces it to its behavioral normal form by iteratively applying the five inference rules of hidden equational deduction. It will be explained in more detail in Chapter V.

On the other hand, there is no way to show that \texttt{\textit{put(E,C) \equiv Eq \; put(E,C')}} even if the equation \((\forall E,C,C') \texttt{put(E,C) = put(E,C')}\) is behaviorally satisfied by \texttt{CELL}, as we’ll shortly prove by coinduction. ■

**Example 7 Flag** (continued). It is not difficult to show that \texttt{\textit{up?(rev(rev(F))) \equiv Eq \; up?(F)}}. The following BOBJ code proves it:

\begin{verbatim}
open FLAG .
red up?(rev(rev(F))) == up?(F) . *** should be true
close
\end{verbatim}

But, as in the previous example, there are equations which are behaviorally satisfied by \texttt{FLAG} but cannot be proved by just equational deduction. One such example is \((\forall F) \texttt{rev(rev(F)) = F} \). ■

**Example 8 Set** (continued). As before, there are equations in \texttt{SET} which can be proved by equational deduction, for example \((\forall E,S,S') E \in (S \& (S' \cup S)) = E \in S\) with the following associated BOBJ proof score:

\begin{verbatim}
open SET .
red E in S \& (S' \cup S) == E in S . *** should be true
close
\end{verbatim}
but there are others, such as \((\forall S, S')\) \(S \& (S' \cup S) = S\), which can not.

\[\]

**Example 9 Nondeterministic Stack** (continued). Despite of the existence of the equation \((\forall S)\) pop(push(S)) = S in NDSTACK, the equation \((\forall S)\) push(pop(push(S))) = push(S) cannot be proved. This is because the operation push is not congruent, so the rule (5) b) cannot be used. Obviously, BOBJ cannot prove it:

\[
\begin{align*}
\text{open NDSTACK .} \\
\text{red (push (pop (push S))) == (push S) .} \text{***> should be false}
\end{align*}
\]

with the output:

\[
\begin{align*}
\text{reduce in NDSTACK : (push (pop (push S))) == (push S)} \\
\text{result Bool: false} \\
\text{LHS sort Stack: push (pop (push S))} \\
\text{RHS sort Stack: push S} \\
\text{rewrite time: 79ms} \quad \text{parse time: 20ms}
\end{align*}
\]

Actually, this equation must not be satisfied by the “good” models, because push is supposed to generate different numbers in different situations. In particular, considering the model given in Section III.B, the left hand side term evaluates to \((n+2, [f(n+1), l])\) while the right hand side term evaluates to \((n+1, [f(n), l])\) in the same state \((n, l)\). On the other hand, it can be easily shown that pop (push (pop (push S))) \(\equiv_{Eq} S:\)

\[
\begin{align*}
\text{open NDSTACK .} \\
\text{red (pop (push (pop (push S)))) == S .} \text{***> should be true}
\end{align*}
\]

That reduction works because pop (push (pop (push S))) is first reduced to the term pop (push S) using the equation \((\forall S)\) pop push S = S in NDSTACK where S is substituted by pop (push S).

\[\]

Unlike equational deduction, rules (1)–(5) are not complete for behavioral satisfaction; but they do seem to provide proofs for some cases of interest. In fact, the behavioral satisfaction is incomplete (see Section VI.E).
IV.B Coinduction

This section introduces a new proof method, called coinduction, which combined with hidden equational reasoning becomes very powerful, at least powerful enough to prove many of the behavioral properties we were interested in so far. BOBJ system already supports various versions of coinduction, which gradually evolved as we found new situations where the previous versions didn’t work.

The first subsection presents the general method and justifies its correctness, while the second one shows coinduction at work; we selected various examples. Finally, a brief subsection of conclusions will discuss some of coinduction’s problems.

IV.B.1 Coinduction Method

Since behavioral equivalence is the largest hidden congruence in any model (see Theorem 18), the following method to prove that a $\Sigma$-equation $(\forall X) \ t = t'$ is behaviorally satisfied by a specification $B$ is sound:

<table>
<thead>
<tr>
<th>COINDUCTION METHOD: (INPUT: $B = (\Sigma, \Gamma, E)$ and a pair of $\Sigma$-terms $(t, t')$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1: Pick an “appropriate” binary relation $R$ on terms</td>
</tr>
<tr>
<td>Step 2: Prove that $R$ is a hidden $\Gamma$-congruence</td>
</tr>
<tr>
<td>Step 3: Show that $t R t'$</td>
</tr>
</tbody>
</table>

Figure IV.2: General coinduction.

The relation $R$ is called the candidate relation and even though it may depend on the particular equation $(\forall X) \ t = t'$, in many situations it is the same for a large class of equations.

IV.B.2 Examples

Example 6 Cell (continued). Let us prove by coinduction some properties of cells which cannot be proved by just equational deduction. They can be all proved using the same relation $R$, namely:

```
bth COINDUCTION is protecting CELL .
  op _R_ : Cell Cell -> Bool [ncong] .
```
Therefore, two cells are in $R$ iff they cannot be distinguished by the attribute $\text{get}$. Step 2 requires to prove that $R$ is a hidden $\Gamma$-congruence; it is of course preserved by $\text{get}$ by construction, and we can prove more than that it is preserved by $\text{put}$, namely:

$$
\text{red } \text{put}(E, C) R \text{put}(E, C') . \quad \text{*** should be true}
$$

Now we are ready to go to Step 3 and prove some behavioral equalities:

$$
\text{red } \text{put}(E,\text{put}(E,C)) R \text{put}(E,C) . \quad \text{*** should be true}
$$

$$
\text{red } \text{put}(\text{get}(C),\text{put}(E,C)) R C . \quad \text{*** should be true}
$$

Thus the three equations above are behaviorally satisfied by $\text{CELL}$.

Why did we declare $R$ to be non-congruent? Let us first analyze the following result saying that if $R$ is congruent then it includes the behavioral equivalence:

**Proposition 29** Let $\Sigma$ be a hidden signature extending the signature of $\text{BOOL}$ and containing an operation $R : s s \rightarrow \text{Bool}$ for each sort $s$. Then $\equiv_\Sigma \subseteq A_R$ in any hidden $\Sigma$-algebra $A$ for which $R$ is congruent and reflexive.

**Proof:** Let $a \equiv_\Sigma a'$. Since $A_R$ is reflexive, it follows that $a' A_R a'$, and since $R$ is congruent, it follows that $a A_R a'$. Therefore, $\equiv_\Sigma \subseteq A_R$. $\Box$

Thus, if one assumes $R$ to be congruent, like in [37], then the most one can hope for $R$ is to be exactly the behavioral equivalence, which is of course a limitation of coinduction. To simplify writing, we’ll often let $R$ ambiguously denote the relation $A_R$.

**Example 7 Flag** (continued). Following the method described above, in Step 1 we first define the relation:

```haskell
bth COINDUCTION is protecting FLAG .
op _R_ : Flag Flag -> Bool [ncong] .
vars F F' : Flag .
eq F R F' = (up?(F) == up?(F')) .
end
```
then in Step 2 we show that it is a hidden congruence:

\begin{verbatim}
open COINDUCTION .
ops f f' : -> Flag .
eq up?(f) = up?(f') . *** assume f R f'
red up(f) R up(f') .
red down(f) R down(f') .
red rev(f) R rev(f') .
close
\end{verbatim}

and then in Step 3 we prove some properties:

\begin{verbatim}
red rev(rev(F)) R F . *** should be true
red rev(up(down(rev(down(F)))))) R rev(rev(rev(rev(down(F'))))) .
\end{verbatim}

Notice that the two properties could be proved using the same relation \( R \), like in the previous example.

The application of Step 2 in the previous example needs some justification: it implicitly uses Theorem 25 (the theorem of constants) and Proposition 26. First, let us notice that the following holds:

**Proposition 30** In the same hypothesis as Proposition 29, \( R \) is a hidden \( \Gamma \)-congruence on \( A \) iff \( A \equiv (\forall W,x,x') \delta(W,x) R \delta(W,x') = true \) if \( x R x' = true \) for each \( \delta \in \Gamma \) and \( R \) is the identity on visible sorts.

**Proof:** Suppose that \( R \) is a hidden \( \Sigma \)-congruence on \( A \) and let \( \theta : W \cup \{x,x'\} \rightarrow A \) be a map such that \( \theta(x) R \theta(x') \). Since \( R \) is a hidden \( \Gamma \)-congruence and \( \delta \in \Gamma \), it follows that \( A_\delta(\theta(W),\theta(x)) R A_\delta(\theta(W),\theta(x')) \), that is, \( \theta(\delta(W,x)) R \theta(\delta(W,x')) \).

Conversely, suppose that \( R \) is the identity on visible and that \( A \) behaviorally satisfies the equation above. Let \( \delta : s_1...s_n \rightarrow s \) be an operation in \( \Gamma \), let \( \overline{a_j} \in A_{\overline{\delta}} \), and let \( a_j, a'_j \in A_{s_j} \) such that \( a_j R a'_j \), for some \( 1 \leq j \leq n \). It suffices to show that \( A_\delta(\overline{a_j},a_j) R A_\delta(\overline{a_j},a'_j) \). Indeed, let \( \theta : W \cup \{x,x'\} \rightarrow A \) be the map defined by \( \theta(W) = \overline{\theta}_W \), \( \theta(x) = a_j \), and \( \theta(x') = a'_j \). Since \( \theta(x) R \theta(x') \), then \( \theta(\delta(W,x)) R \theta(\delta(W,x')) \). Therefore, \( A_\delta(\overline{a_j},a_j) R A_\delta(\overline{a_j},a'_j) \).

Since we’d like \( R \) to be a hidden \( \Gamma \)-congruence for any model of \( B \), we have to prove that \( B \equiv (\forall W,x,x') \delta(W,x) R \delta(W,x') = true \) if \( x R x' = true \), which by the
Theorem of hidden constants (Theorem 25) is equivalent to

\[ B_{\{x,x\}} \equiv (\forall W) \delta(W,x) R \delta(W,x') = \text{true} \text{ if } x R x' = \text{true}, \]

where \( B_{\{x,x\}} \) is the behavioral specification which extends \( B \) with the two hidden constants \( x, x' \) (the spec obtained after we added the constants \( f \) and \( f' \) in the previous example); since the condition \( x R x' = \text{true} \) is ground, by Proposition 26, it is further equivalent to \( B' \equiv (\forall W) \delta(W,x) R \delta(W,x') = \text{true} \), where \( B' \) is the spec \( B_{\{x,x\}} \) extended with the ground equation \( (\forall \emptyset) x R x' = \text{true} \) (the spec obtained after we added the equation \( \text{up?}(f) = \text{up?}(f') \), which is equivalent to \( f R f' \), in the previous example).

We will apply the same strategy in many proofs by coinduction, without mentioning Theorem 25 and Proposition 26 anymore.

**Example 8 Set** (continued). Let us now prove some properties of sets. First, let the candidate relation be the expected one, that is, two sets are in \( R \) iff they have the same elements:

```plaintext
bth COINDUCTION is protecting SET .
   op _R_ : Set Set -> Bool [ncong] .
   op e : -> Elt .
   vars S S' : Set .
   eq S R S' = (e in S) == (e in S') . end
```

Notice that we fixed an element \( e \) as a visible constant. This is a technical issue, its reason being an operational limitation of BOBJ that equations cannot have variables in the right hand side term which do not occur in the left hand side. However, a case analysis is needed whenever a new element is involved, as below in the Step 2 of coinduction showing that \( R \) is a hidden congruence:

```plaintext
open COINDUCTION .
   ops a1 a1' a2 a2' : -> Set .
   eq e in a1 = e in a1' . *** assume that a1 R a1'
   eq e in a2 = e in a2' . *** assume that a2 R a2'
   red (a1 U a2) R (a1' U a2') . *** should be true
   red (a1 & a2) R (a1' & a2') . *** should be true
   op e' : -> Elt .
   red add(e', a1) R add(e', a1') . *** should be true
   eq e' = e . *** case analysis
   red add(e', a1) R add(e', a1') . *** should be true
close
```
Once the candidate relation is a hidden congruence, usual properties of sets can be proved
(Step 3):

```
open COINDUCTION .
ops a a' a'' : -> Set .
red (a U a') R (a' U a) .   **** should be true
red (a & a') R (a' & a) .   **** should be true
red a & (a' U a) R a .     **** should be true
red a U (a' & a) R a .     **** should be true
red a U (a' & a'') R (a U a') & (a U a'') .  **** should be true
red a & (a' U a'') R (a U a') & (a U a'') .  **** should be true
red a U (a' & (a U a'')) R (a' & a'') U a .  **** should be true
red (a U a') & a'' R a U (a' & a'') .  **** should be false
close
```

Notice that a result false of a reduction does not imply that that property is not true; it just means that BOBJ could not prove it\(^1\).

\[1\] Probably false? would be a better notation than just false.

In all the examples above, the candidate relation was defined only by means of attributes, that is, two states were in relation if and only if they couldn’t be distinguished by attributes. We sometimes call this method attribute coinduction. Notice that it is very easy to automate and it has been implemented in both CafeOBJ \[37\], where it is denoted by =**, and in previous\(^2\) versions of Kumo \[58, 59\]. The next examples show more sophisticated proofs by coinduction where attribute coinduction doesn’t work anymore.

**Example 31 Stream.** Streams are already a common, if not the most common, data type of infinite structures, intensively used in lazy functional programming. A stream is basically an infinite list of elements, having one end available from which elements can be incrementally observed:

```
bth STREAM[X :: TRIV] is sort Stream .
op head : Stream -> Elt .
op tail : Stream -> Stream .
op _&_ : Elt Stream -> Stream .
var E : Elt . var S : Stream .
eq head(E & S) = E . eq tail(E & S) = S .
end
```

\[2\] Attribute coinduction has been recently generalized to what we call Δ-coinduction in Section IV.C.
There are two operations, head and tail, with which any of the elements in the stream can be reached, and a constructor, \&, which adds a new element to a stream. Intuitively, two streams are behaviorally equivalent iff they contain the same elements in the same order.

Now we’d like to prove a basic and natural property of streams, namely that 
\((\forall S) \text{head}(S) \& \text{tail}(S) = S\) is behaviorally satisfied by any implementation of STREAM. It is immediate that it cannot be proved by just equational deduction, because there is no way to get rid of the operation \& in the left hand side term. We will prove it by coinduction, but before proceeding to the definition of \(R\), let’s first formalize some experiments on streams, i.e., contexts of the form head(tail(...(tail(•)...))), as a BOBJ theory:

\[
\begin{align*}
\text{bth EXPERIMENT}[X :: TRIV] \text{ is protecting STREAM}[X] + NAT . \\
op \text{exp} : \text{Nat Stream} \rightarrow \text{Elt} . \\
\text{vars } S, S' : \text{Stream} . \ \text{var } N : \text{Nat} . \\
eq \text{exp}(0,S) = \text{head}(S) . \\
eq \text{exp}(s(N),S) = \text{exp}(N,\text{tail}(S)) . \\
end
\]

Notice that the definition above is inductive, and thus the equation \((\forall N, S) \text{exp}(N,S) = \text{head}(\text{tail}^N(S))\) holds in all the models of EXPERIMENT for which induction on naturals is valid, including the expected extensions of models of STREAM. Now we can prove by induction that for every natural number \(N\), \(\text{exp}(N, E \& S)\) is \(E\) when \(N = 0\) and \(\text{exp}(N - 1, S)\) otherwise:

\[
\begin{align*}
\text{open EXPERIMENT} . \ \text{var } E : \text{Elt} . \\
\text{red } \text{exp}(0, E \& S) == E . \quad \text{*** should be true} \\
\text{red } \text{exp}(s(N), E \& S) == \text{exp}(N,S) . \quad \text{*** should be true} \\
\text{close}
\end{align*}
\]

Therefore, the specification EXPERIMENT and the following one are equivalent\(^3\):

\[
\begin{align*}
\text{bth LEMMA-\&}[X :: TRIV] \text{ is pr EXPERIMENT}[X] . \\
\text{var } E : \text{Elt} . \ \text{var } S : \text{Stream} . \ \text{var } N : \text{Nat} . \\
eq \text{exp}(0, E \& S) = E . \\
eq \text{exp}(s(N), E \& S) = \text{exp}(N,S) . \\
end
\]

\(^3\)For a formal definition of equivalence of behavioral specifications see Section IV.D.
We are now ready to define the candidate relation $R$, as the union of an infinite number of relations denoted $R[N]$ for each natural number $N$:

$$\text{bth COINDUCTION}[X :: \text{TRIV}] \text{ is protecting EXPERIMENT}[X] .$$
vars $S$ $S'$ : Stream . var $N$ : Nat .
op $\cdot R[\cdot] \cdot$ : Stream Nat Stream $\rightarrow$ Bool [ncong] .
eq $S$ $R[N]$ $S'$ = (exp($N$, $S$) == exp($N$, $S'$)) .
end

Thus Step 1 is finished. Let us show now that $R$ is a hidden congruence:

open COINDUCTION $+$ LEMMA-$\&$ .
ops $s$ $s'$ : $\rightarrow$ Stream . var $E$ : Elt . var $N$ : Nat .
eq head($s$) = head($s'$) . *** * these two eqns say
eq exp($N$, tail($s$)) = exp($N$, tail($s'$)) . *** * ($s$ $R$ $s'$)
eq exp($N$, $s$) = exp($N$, $s'$) . *** * also ($s$ $R$ $s'$)
red head($s$) == head($s'$) . ***> head preserves $R$
red tail($s$) $R[N]$ tail($s'$) . ***> tail preserves $R$
red ($E$ & $s$) $R[0]$ ($E$ & $s'$) . ***> these two reds say
red ($E$ & $s$) $R[s(N)]$ ($E$ & $s'$) . ***> _&_ preserves $R$
close

The first two equations above are equivalent (by equational deduction) to the following two:

$$\text{eq exp( 0 ,s) = exp( 0 ,s') .}$$
$$\text{eq exp(s(N),s) = exp(s(N),s') .}$$

that is, by induction they are equivalent to the assumption $s$ $R$ $s'$. It is also needed (for the last reduction) to be expressed in a more compact form, thus the third equation.

Now we can go to Step 3 and prove by induction the desired property:

$$\text{red head(S) & tail(S) R[0] S . ***> should be true}$$
$$\text{red head(S) & tail(S) R[s(N)] S . ***> should be true}$$

Notice that the relation $R$ chosen in this example is exactly the behavioral equivalence, as in most of the examples in fact.

It is therefore the case that sometimes, in order to define the candidate relation, the specification needs to be first extended. To be rigorous, one has to prove that the extension is conservative, that is, that any model of the initial specification can be extended to a model over the larger specification, thus providing the environment to
define the candidate relation. This can be relatively easily seen in the previous example, because $\text{exp}$ acts as a derived operation.

**Example 32 Reversing a Stream.** In this example, let us consider streams of booleans together with a method $\text{rev}$ which reverses all the elements in a stream:

```
| bth REV is pr STREAM[BOOL] .
| op rev : Stream -> Stream .
| var S : Stream .
| eq head(rev(S)) = not head(S) .
| eq tail(rev(S)) = rev(tail(S)) .
| end
```

Our goal is to show that, similarly to the Example 7 of flags, double reversing of a stream $S$ produces a stream which is behaviorally equivalent to $S$. But first, let us show that the equation $(\forall N,S) \text{exp}(N,\text{rev}(S)) = \text{not exp}(N,S)$ holds:

```
| open REV + EXPERIMENT[BOOL] .
| *** exp(N,rev(S)) = not exp(N,S) .
| var S : Stream .
| red exp(0,rev(S)) == not exp(0,S) .           *** should be true
| op n : -> Nat .
| eq exp(n,rev(S)) = not exp(n,S) .            *** ind hypothesis
| red exp(s(n),rev(S)) == not exp(s(n),S) .     *** should be true
| close
```

Therefore, we can soundly introduce the following:

```
| bth LEMMA-REV is pr REV + EXPERIMENT[BOOL] .
| var N : Nat . var S : Stream .
| eq exp(N,rev(S)) = not exp(N,S) .
| end
```

We consider the same candidate relation as in the previous example. Let us proceed now to Step 2, showing that $R$ is also a hidden congruence for $\text{REV}$, that is, showing that it is preserved by $\text{rev}$:

```
| open LEMMA-REV + COINDUCTION[BOOL] .
| var N : Nat .
| ops s s' : -> Stream .
| eq exp(N,s) = exp(N,s') . *** i.e., (s R s')
| red rev(s) R[N] rev(s') . *** should be true
| close
```

\[4\text{Suppose that all the specifications in the previous example are available.}\]
We are now ready to proceed to Step 3:

\[
\text{open LEMMA-REV + COINDUCTION[BOOL]}.
\]

\[
\begin{align*}
\text{var } & S : \text{Stream}. \quad \text{var } N : \text{Nat}. \\
\text{red } & \text{rev}(\text{rev}(S)) \ R[N] S. \quad \text{\textbullet\textbullet\textbullet should be true}
\end{align*}
\]

\[
\text{close}
\]

Therefore \( \text{REV} \equiv (\forall S) \ \text{rev}(\text{rev}(S)) = S. \) \[\blacksquare\]

**Example 33 Zipping Streams.** Let us now consider a binary operation on streams, \( \text{zip} \), which “zips” two streams; for example, \( \text{zip}(0 2 4 \ldots, 1 3 5 \ldots) \) is the stream \( 0 1 2 3 4 5 \ldots \). The operation \( \text{zip} \) is called \text{merge} in [98]; in fact, this example is inspired from [98] and [139]. The following two equations are natural:

\[
\text{bth ZIP}[X :: \text{TRIV}] \text{ is pr STREAM}[X].
\]

\[
\begin{align*}
\text{op zip} & : \text{Stream Stream} \rightarrow \text{Stream}. \\
\text{vars } & S S' : \text{Stream}. \\
\text{eq head(zip(S,S'))} & = \text{head(S)}. \\
\text{eq tail(zip(S,S'))} & = \text{zip(S',tail(S))}.
\end{align*}
\]

Let us now consider two other operations, \( \text{odd} \) and \( \text{even} \), this time unary, which take a stream and return the stream of elements on odd positions and on even positions, respectively:

\[
\text{bth ODD-EVEN}[X :: \text{TRIV}] \text{ is pr STREAM}[X].
\]

\[
\begin{align*}
\text{ops odd even} & : \text{Stream} \rightarrow \text{Stream}. \\
\text{var } & S : \text{Stream}. \\
\text{eq head(odd(S))} & = \text{head(S)}. \\
\text{eq tail(odd(S))} & = \text{even(tail(S))}. \\
\text{eq even(S)} & = \text{odd(tail(S))}.
\end{align*}
\]

An intuitive property of \( \text{zip} \), \( \text{odd} \) and \( \text{even} \), is that \( \text{zip(odd(S),even(S))} = S \) for any stream \( S \), that is, the equation \((\forall S) \ \text{zip(odd(S),even(S))} = S \) is behaviorally satisfied by any model of \( \text{ZIP + ODD-EVEN} \). We’ll prove it by coinduction, but first let us prove some lemmas about experiments in the new context, that is, that \( \text{ZIP + ODD-EVEN + EXPERIMENT} \) behaviorally satisfies the equations \((\forall N, S) \ \exp(N,\text{odd(S)}) = \exp(2 \ast N, S) \), \((\forall N, S) \ \exp(N,\text{even(S)}) = \exp(2 \ast N + 1, S) \), and \((\forall N, S, S') \ \exp(2 \ast N,\text{zip(S,S')}) = \exp(N, S): \)
open ZIP + ODD-EVEN + EXPERIMENT. vars S S' : Stream.
red exp(0, odd(S)) == exp(2*0, S). *** should be true
red exp(0, even(S)) == exp(s(2*0), S). *** should be true
red exp(2*0, zip(S,S')) == exp(0,S). *** should be true

op n : -> Nat.
eq exp(n, odd(S)) = exp(2*n, S). *** * induction hypothesis
eq exp(n, even(S)) = exp(s(2*n), S). *** * idem
eq exp(2*n, zip(S,S')) = exp(n,S). *** * idem

red exp(s(n), odd(S)) == exp(s(s(2*n)), S). ***> true
red exp(s(n), even(S)) == exp(s(s(2 * n))), S). ***> true
red exp(s(s(2*n)), zip(S,S')) == exp(s(n),S). ***> true

close

The proof was by induction on natural numbers, as in the previous two examples.
Hence, the following behavioral specification has the same models as ZIP + ODD-EVEN + EXPERIMENT:

bth LEMMA-ZOE is pr ZIP + ODD-EVEN + EXPERIMENT.
vars S S' : Stream. var N : Nat.
eq exp(N, odd(S)) = exp(2*N, S).
eq exp(N, even(S)) = exp(2*N + 1, S).
eq exp(2*N, zip(S,S')) = exp(N,S).

end

One can now proceed to Step 2 of coinduction and show that $R$ is a hidden congruence for ZIP + ODD-EVEN:

open LEMMA-ZOE + COINDUCTION. var S : Stream.
ops s s' : -> Stream. var N : Nat.
eq exp(N,s) = exp(N,s'). *** (s R s')
red odd(s) R[N] odd(s'). *** should be true
red even(s) R[N] even(s'). *** should be true
red zip(S,s) R[2*N] zip(S,s'). *** should be true
red zip(S,s) R[s(2*N)] zip(S,s'). *** should be true
red zip(s,S) R[2*N] zip(s',S). *** should be true
red zip(s,S) R[s(2*N)] zip(s',S). *** should be true

close

The proof was by case analysis, since a natural number can be either even or odd, that is, either of the form $2 \times N$ or of the form $s(2 \times N)$. Using a similar strategy, we can prove the desired property:
open LEMMA-ZOE + COINDUCTION. var S : Stream. var N : Nat.
red zip(odd(S),even(S)) R[2*N] S.
red zip(odd(S),even(S)) R[s(2*N)] S.
close

Therefore, \( ZIP + ODD-EVEN \equiv (\forall S) \text{zip(odd(S),even(S))} = S \). ■

The following is an example showing a proof method which looks like coinduction, but it is not coinduction.

**Example 34 Lists of Flags.** Here is a BOBJ specification of flags extending the one in Example 7 in Section III.F where the resource to which a flag is associated is also considered:

```
bth FLAG' is protecting FLAG + QID.
  op resource_ : Flag -> Id.
  var F : Flag.
  eq resource up(F) = resource F.
  eq resource down(F) = resource F.
  eq resource rev(F) = resource F.
end
```

The module QID is built in BOBJ and provides “quoted identifiers”, that is, identifiers of the form ‘abc.

When many resources of the same type are available (e.g., printers), their flags are kept in a list (an array may not be desirable since the number of resources varies dynamically) from which the scheduler chooses the first unallocated resource when a request is received. Here is a BOBJ specification of lists of flags (note that cons has two hidden arguments):

```
bth FLAG-LIST is protecting FLAG’.
  sort List. subsort Flag < List.
  op car_ : List -> Flag.
  op cdr_ : List -> List.
  op cons : Flag List -> List.
  var F : Flag. var L : List.
  eq car cons(F, L) = F.
  eq cdr cons(F, L) = L.
end
```
The statement “subsort Flag < List” declares List as a supersort of Flag; BOBJ provides support for order-sorted hidden reasoning even though the theory is not completely done yet. For now, let’s just consider that, in any model, the elements of sort Flag are also regarded as elements of sort List.

The behavioral equations here allow flexible implementations. For example, an operating system can allocate at its discretion software or hardware implementations for flags, so that \texttt{car cons(F, L)} is only behaviorally equivalent to \texttt{F} (we have left some details unspecified, such as \texttt{car} and \texttt{cdr} of the empty list, to make the spec easier to understand).

Now consider a new spec where lists of flags can be put up and down. This is useful for operating systems to put resources in a safe state for system shutdown, or when hardware or software anomalies are detected.

\begin{verbatim}
 bth FLAG-LIST’ is protecting FLAG-LIST .
   var F : Flag . var L : List .
   ops (up_) (down_) : List -> List .
   eq car up L = up car L . eq cdr up L = up cdr L .
   eq car down L = down car L . eq cdr down L = down cdr L .
\end{verbatim}

Now we can prove that \texttt{up(cons(F, L))} is equivalent to \texttt{cons(up(F), up(L))} and that \texttt{down(cons(F, L))} is behaviorally equivalent to \texttt{cons(down(F), down(L))} for all flags F and lists of flags L. Let us first choose a candidate relation:

\begin{verbatim}
 bth COINDUCTION? is protecting FLAG-LIST’ .
   op _R_ : Flag Flag -> Bool .
   op _R_ : List List -> Bool .
   vars F F’ : Flag . vars L L’ : List .
   eq F R F = true . eq L R L = true .
   eq F R F’ = (up? F == up? F’) and (resource F == resource F’) .
   eq L R L’ = ((car L) R (car L’)) and ((cdr L) R (cdr L’)) .
\end{verbatim}

Notice that \texttt{R} is congruent this time. Our intention was not to completely define the candidate relation, but rather to give axioms it should satisfy; therefore, the code above should be read “let \texttt{R} be a binary congruent relation such that the three equations hold”. To be rigorous, we should prove that such relations exist; we let it as an exercise to the reader to show that behavioral equivalence is such a relation. The code above is a bit
dangerous, because of its (co)recurrent definition of the candidate relation, which can lead to non-terminating rewriting; but it works in this case, because the equation \( \text{eq } L R L = \text{true} \) is applied before the last equation. We now demonstrate the two interesting properties:

\[
\begin{align*}
\text{red up}(\text{cons}(F, L)) & \text{ R cons}(\text{up}(F), \text{up}(L)). & \text{should be true} \\
\text{red down}(\text{cons}(F, L)) & \text{ R cons}(\text{down}(F), \text{down}(L)). & \text{should be true}
\end{align*}
\]

It is not necessary to prove that \( R \) is a hidden congruence in this example. That’s because the two properties above hold for any binary relation \( R \) satisfying the properties in COINDUCTION\( \text{?} \), including the behavioral equivalence.

The lesson we should learn from these examples is that pure coinduction can be very difficult and cumbersome to apply in practice. This motivates the work in the next sections on automation of coinduction.

**IV.C Cobases and \( \Delta \)-Coinduction**

The major problem with coinduction is that it requires human intervention to define a “good” candidate relation \( R \). As it has been many times observed in practice, the definition of \( R \) often follows some patterns, like in the case of attribute coinduction. The notions and results in this section can be viewed as a first step in our work toward the mechanization of coinduction.

**IV.C.1 Complete Sets of Observers**

A first interesting related notion is that of complete set of observers by Bidoit and Hennicker [15], as a subset of contexts which can “generate” all the experiments that can be performed on the system. The following definition is adapted from [15] to our notation and terminology:

**Definition 35** Given a hidden signature \( \Gamma \), a complete set of observers for \( \Gamma \) is a set of \( \Gamma \)-contexts, say \( \Delta \), such that for each \( \Gamma \)-experiment \( \gamma \in \mathcal{E}_\Gamma[\bullet] \) there is some \( \Gamma \)-context \( \delta \in \Delta \) which is a subcontext\(^5\) of \( \gamma \).

\(^5\)That is, a subterm; notice that \( \delta \) contains the variable \( \bullet \) occurring in \( \gamma \).
Therefore, the experiment $\gamma$ has the form $\gamma_\delta[\delta]$ for some other “smaller” experiment $\gamma_\delta$. It is worth noticing that the notion of complete set of observers already has a dual flavor to that of basis for structural induction, where for each element $t$ of an abstract data type, there is some other element $t_\delta$ and an operation $\delta$ in the basis such that $t = \delta[t_\delta]$.

**Example 36** Let us consider the hidden subsignature $\Gamma$ of the signature of streams in Example 31 containing only the operations $\text{head}$ and $\text{tail}$. Obviously, $E_\Gamma[\bullet]$ consists of all the terms of the form $\text{head}(\text{tail}(\ldots(\text{tail}(\bullet))))$, for an arbitrary number of occurrences of $\text{tail}$. Then it is easy to observe that

\[
\Delta_1 = \Gamma = \{\text{head}(\bullet), \text{tail}(\bullet)\},
\]
\[
\Delta_2 = \{\text{head}(\bullet), \text{head}(\text{tail}(\bullet)), \text{tail}(\text{tail}(\bullet))\},
\]
\[
\Delta_3 = \{\text{head}(\bullet), \text{head}(\text{tail}(\bullet)), \text{head}(\text{tail}(\text{tail}(\bullet))), \text{tail}(\text{tail}(\text{tail}(\bullet)))\},
\]
\[
\ldots
\]
\[
\Delta_\infty = E_\Gamma[\bullet],
\]

are all complete sets of observers for $\Gamma$. ■

To simplify writing, we ambiguously let $\Gamma$ also denote the subset of $\Gamma$-contexts obtained directly, without composition, from the operations in $\Gamma$, such as the $\Delta_1$ above. If the operations in $\Gamma$ have more than one argument, then we consider all possible combinations\(^6\); for example, if $\Gamma$ is the hidden subsignature containing only the union of sets in Example 8, then we also let $\Gamma$ denote the set $\{z \cup \bullet, \bullet \cup z\}$. The following is immediate provides two common complete sets of observers:

**Proposition 37** For any $\Gamma$, both $\Gamma$ and $E_\Gamma[\bullet]$ are complete sets of observers.

As with induction, where some bases can be better than others for particular proofs, it may be possible that some complete sets of observers are better than others in practical situations. For example, if one defines a stream $\text{blink} = 0 \ 1 \ 0 \ 1 \ldots$ by

\[
\begin{align*}
\text{eq head}(\text{blink}) &= 0 . \\
\text{eq head}(\text{tail}(\text{blink})) &= 1 . \\
\text{eq tail}(\text{tail}(\text{blink})) &= \text{blink} .
\end{align*}
\]

\(^6\)Modulo renaming of variables, of course.
it is almost certain that the complete set of observers $\Delta_2$ in Example 36 is better than the others.

We do not insist more on properties of complete sets of observers here because we’ll do it for more general notions in the next subsections; however, we refer the interested reader to [15] for more on complete sets of observers.

IV.C.2  Strong Cobases

It is often the case that it is not clear what basis is best for certain proofs by induction; moreover, sometimes it is not even clear whether a certain subset of (derived) operations is a basis at all. For example, there are “consecrated” bases of induction for natural numbers, such as (zero and successor) or (zero, one and double successor) which can be used in most situations, but of course there are some others which may be very useful, if not crucial, in tricky examples. As a case study, we refer the reader to [51] for a clever and nice proof by induction of the Euler formula

$$\varphi(n) = n \cdot \prod_{p|n} \left(1 - \frac{1}{p}\right),$$

where $\varphi(n)$ is the number of positive integers less than $n$ that are relatively prime to $n$, and $p$ ranges over prime numbers; the induction’s basis consists there of the infinite set of all prime numbers as constants and multiplications with prime numbers as generators. The fact that this basis is correct follows by properties of natural numbers which can be, of course, proved from the axioms that characterize the natural numbers.

For similar but dual reasons, we semantically generalize the notion of complete sets of observers to what we call cobasis (actually, the more general notion of (strong) cobasis [137] appeared before that of complete set of observers [15]). The notion of cobasis was introduced in [137], but we soon realized it could be generalized (a common phenomenon in new areas of science) [71, 72, 134]:

**Definition 38** Given a behavioral specification $\mathcal{B}' = (\Sigma', \Gamma', E')$, a hidden subsignature $\Delta$ of $\Sigma'$ and a hidden sort $h$, let $T_{\Sigma'}(\Gamma', \Delta; \bullet : h, Z)$ be the set of all $\Sigma'$-terms $\gamma'$ with variables in $\{\bullet : h\} \cup Z$, such that for each occurrence of $\bullet$ in $\gamma'$, if $(\sigma_1, ..., \sigma_n)$ is the sequence of operations in $\Sigma'$ from the root of $\gamma'$ to that $\bullet$ and $j = \max\{1, \sup\{k \mid k \leq \}$
\( n \) and \( \sigma_k \) has a visible result} \} then \( \sigma_n \in \Delta \) and \( \sigma_j, ..., \sigma_{n-1} \in \Gamma' \).

![Diagram](GrigoreRosuThesisFigures.png)

Figure IV.3: A term in \( T_{\Sigma'}(\Gamma', \Delta; \bullet : h, Z) \).

Then \( \Delta \) is a strong cobasis for \( B = (\Sigma, \Gamma, E) \) iff \( B' \) is a conservative extension of \( B \) and for any \( \Gamma \)-experiment \( \gamma \) in \( E_{\Gamma'}[\bullet : h] \) there is some \( \gamma' \) in \( T_{\Sigma'}(\Gamma', \Delta; \bullet : h, var(\gamma)) \) such that \( B' \equiv (\forall \bullet, var(\gamma)) \gamma = \gamma' \).

This seemingly complex formulation includes all practical situations we are aware of. In particular:

**Proposition 39** Every complete set of observers is a strong cobasis.

**Proof:** It is straightforward, if one takes \( B' \) the conservative extension \( (Der(\Sigma), \Gamma, E) \) of \( (\Sigma, \Gamma, E) \). Indeed, if \( \gamma \) is a \( \Gamma \)-experiment, then there is some other \( \Gamma \)-experiment \( \gamma_\delta \) and some \( \Gamma \)-context \( \delta \), such that \( \gamma = \gamma_\delta[\delta] \). It is now easy to observe that \( \gamma_\delta[\delta] \) is an element in \( T_{Der(\Sigma)}(\Gamma, \Delta; \bullet, Z) \).

The main conceptual difference between strong cobasis and complete set of observers is that our notion relates to a behavioral specification rather than just a signature, which means that the equations can be also used in deducing that \(^7\)for each

\(^7\) Notice that if there is no \( k < n \) such that \( \sigma_k \) has visible result then the supremum is 0 and \( j = 1 \), that is, the supremum is taken over natural numbers.
Γ-experiment γ there is some Γ-context δ ∈ Δ which is a subcontext of γ”; in addition, as we’ll see later, the form of equations in a specification might suggest getting rid of some contexts from the complete set of observers which can greatly simplify the proofs.

IV.C.3 Cobases in General

Our more general notion of cobasis (see also [71, 72, 134]) is as follows:

Definition 40 If \( B' = (\Sigma', \Gamma', E') \) is a conservative extension of \( B = (\Sigma, \Gamma, E) \) and if \( \Delta \subseteq \Sigma' \), then \( \Delta \) is a cobasis for \( B \) iff for all hidden sorted terms \( t, t' \in T_{\Sigma'h}(X) \), if \( B' \models (\forall X) \delta(W, t) = \delta(W, t') \) for all appropriate \( \delta \in \Delta \) then \( B \models (\forall X) t = t' \). □

The following is a key result showing that all the special cases of strong cobasis are cobases, being a first step toward automation of coinduction; it was first proved in [137] (Theorem 12):

Theorem 41 Every strong cobasis is a cobasis.

Proof: Let \( \Delta \) be a cobasis for \( B = (\Sigma, \Gamma, E) \) in a conservative extension \( B' = (\Sigma', \Gamma', E) \).

We first show that \( B' \models (\forall Z, X) \gamma'[t] = \gamma'[t'] \) for all \( \gamma' \in T_{\Sigma'}(\Gamma', \Delta; \bullet, Z) \). Let \( A' \) be a hidden \( \Sigma' \)-algebra such that \( A' \models B' \), let \( \varphi' : X \rightarrow A' \) and \( \theta' : Z \rightarrow A' \) be any two assignments, and let \( a = \varphi'(t) \) and \( a' = \varphi'(t') \). Let \( Q \subseteq T_{\Sigma'}(\Gamma'; \Delta; \bullet, Z) \) be the set of all \( \Sigma' \)-terms \( \gamma' \) with variables in \( \{\bullet\} \cup Z \), such that \( \sigma_n \in \Delta \) and \( \sigma_1, ..., \sigma_{n-1} \in \Gamma' \) for each occurrence of \( \bullet \) in \( \gamma \), where \( (\sigma_1, ..., \sigma_n) \) is the sequence of operations in \( \Sigma' \) from the root of \( \gamma \) to that \( \bullet \) and \( j = \sup\{k \mid k \leq n \text{ and } \sigma_k \text{ is visible}\} \geq 1 \). We prove by structural induction on \( \gamma \in T_{\Sigma'}(\Gamma'; \Delta; \bullet, Z) \) the following assertion:

“If \( \gamma \in Q \) then \( A'_\gamma(a)(\theta') = A'_\gamma(a')(\theta') \) else \( A'_\gamma(a)(\theta') \equiv T_{\Sigma'}^{h} A'_\gamma(a')(\theta') \)."

If \( \gamma = z' \in Z \) then \( A'_\gamma(a)(\theta') = \theta'(z') = A'_\gamma(a')(\theta') \), so the assertion holds. Now suppose that \( \gamma = \sigma(\gamma_1, ..., \gamma_k, \bullet, ..., \bullet) \), where \( \sigma : s_1 ... s_n \rightarrow s, k \leq n, \text{ and } \gamma_1, ..., \gamma_k \in T_{\Sigma'}(\Gamma'; \Delta; \bullet, Z) \) (without restricting the generality, we suppose that the possible occurrences of \( \bullet \) as arguments of \( \sigma \) are the last \( n - k \) arguments). Let \( a_i = A'_{\gamma_i}(a)(\theta') \) and \( a'_i = A'_{\gamma_i}(a')(\theta') \) for all \( i = 1, ..., k \), and notice that \( A'_\gamma(a)(\theta') = A'_\sigma(a_1, ..., a_k, a, a, ..., a) \) and \( A'_\gamma(a')(\theta') = A'_\sigma(a'_1, ..., a'_k, a', a', ..., a') \). By the induction hypothesis, \( a_i \equiv T_{\Sigma'}^{h} a'_i \) for all \( i = 1, ..., k \). One distinguishes the following cases:
Case 1: $\sigma \in \Delta$. Since $A' \models_{\Sigma'} (\forall Z_j, X) \delta(Z_j, t) = \delta(Z_j, t')$ for all appropriate $\delta \in \Delta$ (in particular for $\sigma$) and $j$, one can easily show:

$$A'_\sigma(a_1, ..., a_k, a, ..., a) \equiv_{\Sigma'} A'_\sigma(a_1, ..., a_k, a', a, ..., a) \equiv_{\Sigma'} A'_\sigma(a_1, ..., a_k, a', a', ..., a) \equiv_{\Sigma'} A'_\sigma(a_1, ..., a_k, a', a', ..., a').$$

Now one distinguishes two subcases:

Case 1.1: $\sigma \in \Delta - \Gamma'$. Then $\gamma_1, ..., \gamma_k \in Q$, so by the induction hypothesis, $a_i = a'_i$ for all $i = 1, ..., k$. Therefore $A'_\sigma(a)(\theta') \equiv_{\Sigma'} A'_\sigma(a')(\theta')$. If $\gamma \in Q$ then the sort of $\sigma$ is visible, that is, $A'_\gamma(a)(\theta') = A'_\gamma(a')(\theta')$.

Case 1.2: $\sigma \in \Delta \cap \Gamma'$. Because $\sigma$ is behaviorally congruent, it immediately follows that $A'_\sigma(a_1, ..., a_k, a', a', ..., a') \equiv_{\Sigma'} A'_\sigma(a_1, ..., a_k, a', a', ..., a')$. Therefore, one gets that $A'_\sigma(a)(\theta') \equiv_{\Sigma'} A'_\sigma(a')(\theta')$. If the sort of $\sigma$ is visible then $A'_\gamma(a)(\theta') = A'_\gamma(a')(\theta')$. If $\gamma \in Q$ and the sort of $\sigma$ is hidden then $k = n$ and $\gamma_1, ..., \gamma_n \in Q$, in which case $a_i = a'_i$ for all $i = 1, ..., n$, and hence $A'_\gamma(a)(\theta') = A'_\gamma(a')(\theta')$.

Case 2: $\sigma \in \Gamma' - \Delta$. Then $\sigma$ has no arguments $\bullet$, that is $k = n$, and since $\sigma$ is behaviorally congruent, $A'_\sigma(a_1, ..., a_k, a', a', ..., a') \equiv_{\Sigma'} A'_\sigma(a_1, ..., a_k, a', a', ..., a')$, that is, $A'_\gamma(a)(\theta') \equiv_{\Sigma'} A'_\gamma(a')(\theta')$. If the sort of $\sigma$ is visible then $A'_\gamma(a)(\theta') = A'_\gamma(a')(\theta')$. If $\gamma \in Q$ and the sort of $\sigma$ is hidden then $\gamma_1, ..., \gamma_n \in Q$, in which case $a_i = a'_i$ for all $i = 1, ..., n$, and hence $A'_\gamma(a)(\theta') = A'_\gamma(a')(\theta')$.

Case 3: $\sigma \in \Sigma' - \Gamma' - \Delta$. Then $k = n$ and $\gamma_1, ..., \gamma_n \in Q$. So by the induction hypothesis, $a_i = a'_i$ for all $i = 1, ..., n$, so $A'_\gamma(a)(\theta') = A'_\gamma(a')(\theta')$.

Thus $B' \models (\forall Z, X) \gamma'[t] = \gamma'[t']$ for all $\gamma'$ in $T(\Gamma', \Delta; \bullet, Z)$. Now, let $\gamma$ be a $\Gamma$-context appropriate for $t$ and $t'$, and let $\gamma'$ be a term in $T_{\Sigma'}(\Gamma', \Delta; \bullet, Z)$ with $B' \models (\forall \bullet, Z) \gamma = \gamma'$, where $Z = \text{var}(\gamma)$. Let $A$ be any hidden $\Sigma$-algebra such that $A \models B$, let $\varphi : X \rightarrow A$ be any assignment, let $a = \varphi(t)$ and $a' = \varphi(t')$, and let $A'$ be a hidden $\Sigma'$-algebra such that $A' \models B'$ and $A'[\Sigma] = A$. Then $A' \models_{\Sigma'} (\forall \bullet, Z) \gamma = \gamma'$ and $A' \models_{\Sigma'} (\forall Z, X) \gamma'[t] = \gamma'[t']$. Let $\theta : Z \rightarrow A$ be any assignment and let $\psi : Z \cup X \rightarrow A'$
be defined as \( \theta \) on \( Z \) and as \( \varphi \) on \( X \). It can be readily seen that \( \varphi(\gamma'|t) = A'_\gamma(a)(\theta) \) and that \( \varphi(\gamma'|t') = A'_\gamma(a')(\theta) \). Then \( A'_\gamma(a)(\theta) = A'_\gamma(a')(\theta) \), \( A'_\gamma(a')(\theta) = A'_\gamma(a')(\theta) \), and \( A'_\gamma(a)(\theta) = A'_\gamma(a')(\theta) \). Since \( \theta \) was chosen arbitrarily, \( A'_\gamma(a) = A'_\gamma(a') \), that is, \( A_\gamma(a) = A_\gamma(a') \). Hence \( a \equiv_\Sigma a' \). □

IV.C.4 \( \Delta \)-Coinduction

Once a cobasis is available, coinduction can be applied automatically. To ease the presentation, in this subsection we work under the following

**Assumption:** \( \Delta \) is a cobasis of \( B \) with \( B' = (\text{Der}(\Sigma), \Gamma, E) \) and \( \Delta \subseteq \text{Der}(\Gamma) \).

Let \( \equiv_{Eq,\Delta} \) be the relation generated by rules (1)–(5) in Section IV.A \footnote{Strictly speaking, \( \equiv_{Eq} \) should be replaced by \( \equiv_{Eq,\Delta} \) in rules (1)–(5).} and

\[
\begin{align*}
(6) \quad \text{\( \Delta \)-Coinduction:} & \quad \delta(W,t) \equiv_{Eq,\Delta} \delta(W,t') \quad \text{for all appropriate } \delta \in \Delta \\
& \quad \Rightarrow t \equiv_{Eq,\Delta} t'
\end{align*}
\]

Figure IV.4: \( \Delta \)-coinduction inference rule.

The following proposition is immediate, by the definition of cobasis:

**Proposition 42** \( \equiv_{Eq} \subseteq \equiv_{Eq,\Delta} \subseteq \equiv \).

Therefore, in order to prove that two terms \( t, t' \) are behaviorally equivalent, it suffices to show that \( t \equiv_{Eq,\Delta} t' \).

**Example 43** Since \{\texttt{head}, \texttt{tail}\} is a complete set of observers (see Examples 36 and 31) for streams, it is also a cobasis. Then it follows immediately by \( \Delta \)-coinduction that \texttt{head}(S) & \texttt{tail}(S) \equiv S \) for any stream \( S \). This proof is by far simpler than the cumbersome one in Example 31. ■

We encourage the reader prove all the previously proved by pure coinduction behavioral equalities by \( \Delta \)-coinduction and notice how elegant \( \Delta \)-coinduction is.
IV.D Equivalent Behavioral Specifications

The common interest in proving that operations are congruent [40, 15, 137] has a variety of motivations, depending on the methodological approach. Our approach emphasizes removing congruent operations from cobases, so that coinduction proofs can be as efficient as possible. We now remind the reader of the notion of equivalent behavioral specifications and some related results (for more detail, see [71, 72, 137]):

Definition 44 Behavioral specifications $B_1 = (\Sigma, \Gamma_1, E_2)$ and $B_2 = (\Sigma, \Gamma_2, E_2)$ are behaviorally equivalent iff they have the same models, each with the same behavioral equivalence; in this case, we write $B_1 \equiv B_2$. ■

Notice that $\equiv$ is an equivalence relation on behavioral specifications.

IV.D.1 Reducing the Number of Behavioral Operations

The following definition is also related to the notion of cobasis; in fact, our experience so far is that they coincide on practical examples:

Definition 45 A set of operations $\Delta \subseteq \Sigma$ is context complete for a behavioral specification $B = (\Sigma, \Gamma, E)$ iff for any $\Gamma$-experiment $\gamma$ there is some $\Delta$-term $\gamma_\Delta$ such that $B \models (\forall \bullet, \text{var}(\gamma)) \gamma = \gamma_\Delta$. ■

The following result underlies the intuition that congruent operations can be removed from the cobasis.

Theorem 46 Let $B_1 = (\Sigma, \Gamma_1, E)$ and $B_2 = (\Sigma, \Gamma_2, E)$ be two behavioral specifications such that $\Gamma_1 \subseteq \Gamma_2$, and $E$ contains only equations of visible condition. Then

1. $B_1$ is a conservative extension of $B_2$;
2. $B_1 \equiv B_2$ iff $A \models B_1$ implies $\models\Sigma_1 \subseteq \models\Sigma_2$;
3. $B_1 \equiv B_2$ iff all operations in $\Gamma_2 - \Gamma_1$ are congruent for $B_1$;
4. If $B_1 \equiv B_2$ then $\Gamma_1$ is a cobasis for $B_2$;
5. If $\Gamma_1$ is context complete for $B_2$ then $B_1 \equiv B_2$. 
Proof: 1. Since every $\Gamma_1$-experiment is a $\Gamma_2$-experiment, $\equiv_{\Sigma}^{\Gamma_2} \subseteq \equiv_{\Sigma}^{\Gamma_1}$ in any hidden \(\Sigma\)-algebra \(A\). Let \(A\) be a hidden \(\Sigma\)-algebra such that \(A \models B_2\), let \(e\) be any \(\Sigma\)-equation in \(E\), say \((\forall X) t = t'\) if \(t_1 = t'_1, ..., t_n = t'_n\), and let \(\theta: X \rightarrow A\) be any assignment such that \(\theta(t_1) \equiv_{\Sigma}^{\Gamma_1} \theta(t'_1), ..., \theta(t_n) \equiv_{\Sigma}^{\Gamma_1} \theta(t'_n)\). Because \(t_1, t'_1, ..., t_n, t'_n\) have visible sorts, one gets that \(\theta(t_1) = \theta(t'_1), ..., \theta(t_n) = \theta(t'_n)\). Since \(A \models B_2\), \(\theta(t) \equiv_{\Sigma}^{\Gamma_2} \theta(t')\), so \(\theta(t) \equiv_{\Sigma}^{\Gamma_1} \theta(t')\). Therefore \(A \equiv_{\Sigma}^{\Gamma_1} e\). Consequently \(A \models B_1\), that is, \(B_1\) is a conservative extension of \(B_2\).

2. If \(B_1\) and \(B_2\) are equivalent then \(A \models B_1\) implies \(\equiv_{\Sigma}^{\Gamma_1} \subseteq \equiv_{\Sigma}^{\Gamma_2}\) since \(\equiv_{\Sigma}^{\Gamma_1} = \equiv_{\Sigma}^{\Gamma_2}\). Conversely, suppose that \(A \models B_1\). Then \(\equiv_{\Sigma}^{\Gamma_1} \subseteq \equiv_{\Sigma}^{\Gamma_2}\), and since \(\equiv_{\Sigma}^{\Gamma_2} \subseteq \equiv_{\Sigma}^{\Gamma_1}\), one gets that \(\equiv_{\Sigma}^{\Gamma_1} = \equiv_{\Sigma}^{\Gamma_2}\) and implicitly \(A \models B_2\). On the other hand, if \(A \models B_2\) then \(A \models B_1\) as proved above, and implicitly \(\equiv_{\Sigma}^{\Gamma_1} = \equiv_{\Sigma}^{\Gamma_2}\).

3. If \(B_1\) and \(B_2\) are equivalent then \(\equiv_{\Sigma}^{\Gamma_1} = \equiv_{\Sigma}^{\Gamma_2}\) for any hidden \(\Sigma\)-algebra \(A\) with \(A \models B_1\). Since the operations in \(\Gamma_2\) are congruent for \(\equiv_{\Sigma}^{\Gamma_2}\) (see Theorem 18), they are also congruent for \(\equiv_{\Sigma}^{\Gamma_1}\), so they are behaviorally congruent for \(B_1\). Conversely, suppose that all operations in \(\Gamma_2\) are behaviorally congruent for \(B_1\) and let \(A\) be a hidden \(\Sigma\)-algebra such that \(A \models B_1\). Then for every \(a, a' \in A_h\) such that \(a \equiv_{\Sigma, h}^{\Gamma_1} a'\), for every \(\Gamma_2\)-experiment \(\gamma\) and for every \(\theta: var(\gamma) \rightarrow A\), we get \(A_\gamma(a)(\theta) = A_\gamma(a')(\theta)\), that is, \(a \equiv_{\Sigma, h}^{\Gamma_2} a'\). Therefore \(\equiv_{\Sigma}^{\Gamma_1} \subseteq \equiv_{\Sigma}^{\Gamma_2}\), so by 2., \(B_1\) and \(B_2\) are equivalent.

4. By Propositions 37 and 39 and by Theorem 41, \(\Gamma_1\) is a cobasis for \(B_1\). If \(B_1 \equiv B_2\) then \(B_1 \models (\forall X) t = t'\) if \(B_2 \models (\forall X) t = t'\) for any \(\Sigma\)-equation \((\forall X) t = t'\), so \(\Gamma_1\) is also a cobasis for \(B_2\).

5. Let \(A\) be any hidden \(\Sigma\)-algebra such that \(A \models B_1\), and let \(a \equiv_{\Sigma, h}^{\Gamma_1} a'\). Since for every \(\Gamma_2\)-experiment \(\gamma\) there is some \(\gamma_1\) in \(T_{\Gamma_1}(\{\bullet\} \cup var(\gamma))\) such that \(B_1 \models (\forall \bullet, var(\gamma)) \gamma = \gamma_1\), we get that \(A_\gamma = A_{\gamma_1}\) as functions \(A_h \rightarrow (A^{\text{var}(\gamma)} \rightarrow D)\), where \(A_{\gamma_1}\) is defined similarly to \(A_\gamma\), that is, \(A_{\gamma_1}(a)(\theta) = a^*_h(\gamma_1)\). Because \(\gamma_1\) has visible sort and contains only operations in \(\Gamma_1\) (which are congruent for \(\equiv_{\Sigma}^{\Gamma_1}\)), we get \(A_{\gamma_1}(a)(\theta) = A_{\gamma_1}(a')(\theta)\) for any \(\theta: var(\gamma) \rightarrow A\). Therefore \(A_\gamma(a)(\theta) = A_\gamma(a')(\theta)\) for any \(\theta\), that is, \(a \equiv_{\Sigma, h}^{\Gamma_2} a'\). Therefore, \(\equiv_{\Sigma}^{\Gamma_1} \subseteq \equiv_{\Sigma}^{\Gamma_2}\), and so by 2., \(B_1\) and \(B_2\) are equivalent. \(\square\)

Therefore, if one suspects that a certain subset of behavioral operations \(\Delta \subseteq \Gamma\) is a cobasis for \(B = (\Sigma, \Gamma, E)\), the following can be done:
1. Let $B_\Delta = (\Sigma, \Delta, E)$.

2. Show that all operations in $\Gamma - \Delta$ are congruent for $B_\Delta$.

3. Conclude that $\Delta$ is a cobasis for $B$.

We will apply these steps implicitly in proofs, without citing Theorem 46.

**Example 47** We show that the restriction on conditional equations cannot be removed with the following behavioral theory:

```plaintext
bth B1 is sort S .
   op f : S -> Bool . ** attribute
   op g : S -> Bool [ncong] . ** attribute
   vars X X' : S .
   ceq g(X) = g(X') if X == X' .
end
```

where `ceq` is the BOBJ keyword for behavioral conditional equations. Notice that $g$ is congruent for $B_1$ anyway, because of the conditional equation. Now consider another behavioral theory where $g$ is also behavioral:

```plaintext
bth B2 is sort S .
   op f : S -> Bool . ** attribute
   op g : S -> Bool . ** attribute
   vars X X' : S .
   ceq g(X) = g(X') if X == X' .
end
```

Let $\Sigma$ be the (common) signature of $B_1$ and $B_2$, containing the operations on the booleans plus $f : S \rightarrow \text{Bool}$ and $g : S \rightarrow \text{Bool}$. Then for any hidden $\Sigma$-algebra $A$, $A \models B_1$ iff $A_f(a) = A_f(a')$ implies $A_g(a) = A_g(a')$ for any $a, a' \in A_s$, and $A \models B_2$ under no restrictions. Therefore $B_1$ and $B_2$ are not equivalent because there exist hidden $\Sigma$-algebras satisfying $B_2$ which do not satisfy $B_1$. Because $B_1$ and $B_2$ satisfy all the hypotheses in Theorem 46 except the one regarding the conditional equations, it follows that this restriction is needed.

**Example 48** SET of Example 8 in Section III.F is equivalent to a behavioral specification in which membership is the only behavioral operation, because all the other operations are congruent.
Example 49 Let LIST be the following behavioral specification:

\begin{verbatim}
  bth LIST is sort List .
  pr NAT .
  op car : List -> Nat .
  op cdr : List -> List .
  op cons : Nat List -> List .
  op _in_ : Nat List -> Bool .
  vars N N' : Nat . var L : List .
  eq car(cons(N,L)) = N .
  eq cdr(cons(N,L)) = L .
  eq N' in cons(N,L) = (N == N') or (N in L) .
end
\end{verbatim}

If $\Psi$ is its data signature (natural numbers and booleans) $\Sigma|_\Psi$, and $\Sigma$ and $E$ are its hidden signature and equations, then the spec is $(\Sigma, \Psi \cup \{\text{car, cdr, cons, in}\}, E)$. It can be easily seen (also following by a congruence criterion presented later (Corollary 54)), that $\text{cons}$ is congruent for $\text{LIST1} = (\Sigma, \Psi \cup \{\text{car, cdr, in}\}, E)$, and so Theorem 46 implies that LIST and LIST1 are equivalent. They have many models, including the standard finite lists and infinite lists. Note that $\text{car}$ and $\text{in}$ can behave unexpectedly on the unreachable states, and all the states are unreachable here.

Another interesting behavioral specification is $\text{LIST2} = (\Sigma, \Psi \cup \{\text{car, cdr}\}, E)$, for which $\text{cons}$ is also behaviorally congruent, but $\text{in}$ is not necessarily congruent, because it can be defined in almost any way on states which are not a $\text{cons}$ of other states.

Finally let $\text{LIST3}$ be the behavioral specification $(\Sigma, \Psi \cup \{\text{in}\}, E)$. Again by the congruence criterion, $\text{cons}$ is behaviorally congruent for $\text{LIST3}$. One model for $\text{LIST3}$ is the $\Sigma$-algebra of finite lists (with any choice for $\text{car(nil)}$ and $\text{cdr(nil)}$, such as 0 and nil), in which two lists are behaviorally equivalent iff they contain the same natural numbers (without regard to their order and number of occurrences). Therefore $\text{car}$ and $\text{cdr}$ are not behaviorally congruent for $\text{LIST3}$.

### IV.E Circular $\Delta$-Coinduction

This section gives a new inference rule for behavioral reasoning. We call this rule circular $\Delta$-coinduction because it handles some examples with circularities (i.e., infinite recursions) that could not be handled by our previous rules in this chapter (and also in [137, 71, 72, 134]); we will also call it $\Delta^\triangleright$-coinduction.
After exploring how to prove the congruence of operations in [137], we became convinced that this does not differ essentially from proving other behavioral properties, except perhaps that it is usually easier. Also certain “coinductive patterns” that appeared in specifying operations inspired a congruence criterion that could automatically decide whether an operation is congruent [137]; moreover, this criterion followed from the $\Delta$-coinduction rule that was strong enough for all proofs we knew at that time. But the fact that congruence of the \texttt{zip} operation of the \texttt{STREAM} example (in Section IV.E.1) does not follow from this criterion, suggests that more powerful deduction rules are needed.

Results in this section arose in part through discussions with Michel Bidoit, Răzvan Diaconescu, Kokichi Futatsugi, and Rolf Hennicker at WDS’99 in Iaşi, WADT’99 in Bonas, and FM’99 in Toulouse. Bidoit and Hennicker [15] gave a general congruence criterion from which the congruence of \texttt{zip} followed easily, and so influenced by the relation between coinduction rules and congruence criteria found in [137], we sought a general inference rule from which the criterion in [15] would follow as naturally as our criterion in [137] followed from $\Delta$-coinduction, and which could prove behavioral properties not provable by $\Delta$-coinduction. The result of this search was circular $\Delta$-coinduction\textsuperscript{9}.

IV.E.1 Limitations of $\Delta$-Coinduction

We give some specifications where the six rules generating the relation $\equiv_{Eq,\Delta}$ are not enough to prove certain simple properties which can be easily proved by circular $\Delta$-coinduction. The first example specifies infinite bit streams, with an operation that reverses each bit. The data theory for bits is given first:

\begin{verbatim}
dth BIT is
  sort Bit
  ops 0 1 : -> Bool
  op not : Bool -> Bool
  eq not(1) = 0
  eq not(0) = 1
end
\end{verbatim}

\textsuperscript{9}We hope readers will find this an inspiring example of how science evolves, and of how scientific interaction can improve a subject.
where dth is the BOBJ keyword for data theories, in which the sorts, the operations and the equations are considered all visible.

```plaintext
bth REV is protecting BIT
   sort Stream
   op head : Stream -> Bool
   op tail : Stream -> Stream
   op _&_ : Bool Stream -> Stream
   op rev : Stream -> Stream
   var B : Bool   var S : Stream
   eq head(B & S) = B
   eq tail(B & S) = S
   eq head(rev(S)) = not(head(S))
   eq tail(rev(S)) = rev(tail(S))
end
```

We can apply a congruence criterion from [137] (see also Corollary 55) to see that \&_ is congruent for the behavioral specification with only head and tail considered behavioral. Similarly, a more general congruence criterion from [15] (see also Corollary 54) implies that rev is congruent too. Thus Theorem 46 implies that \( \Delta = \{ \text{head}, \text{tail} \} \) is a cobasis for REV. Now it is easy to use \( \Delta \)-coinduction to prove properties like \( \text{rev}(B \& S) = \text{not}(B) \) & rev(S), since it is immediate that head(rev(B & S)) and head(not(B) & rev(S)) are both equal to not(B), and that tail(rev(B & S)) and tail(not(B) & rev(S)) are both equal to rev(S).

On the other hand, \( \Delta \)-coinduction plus the other five rules cannot prove the behavioral equality \( \text{rev}(\text{rev}(S)) = S \), because tail(rev(rev(S))) = tail(S), and tail(tail(rev(rev(S)))) = tail(tail(S)), and so on through an infinite recurrence. However, circular \( \Delta \)-coinduction can prove this property, and much more, including operation congruence properties that do not follow from the congruence criterion of [15]. Moreover, all operations that are congruent by the criteria of [15, 137] can be proved congruent by \( \Delta^{\circ} \)-coinduction (see Corollaries 54 and 55).

The following behavioral specification of infinite streams of natural numbers is interesting not only because natural properties of it cannot be proved by coinduction, but also because it admits two unrelated cobases:

```plaintext
bth STREAM is protecting NAT
   sort Stream
   op head : Stream -> Bool
```
op tail : Stream -> Stream
op _&_ : Nat Stream -> Stream
op odd : Stream -> Stream
op even : Stream -> Stream
op zip : Stream Stream -> Stream

var N : Nat var S S' : Stream

eq head(N & S) = B
eq tail(N & S) = S

eq head(odd(S)) = head(S)

eq tail(odd(S)) = even(tail(S))

eq head(even(S)) = head(tail(S))

eq tail(even(S)) = even(tail(tail(S)))

eq head(zip(S,S')) = head(S)

eq tail(zip(S,S')) = zip(S',tail(S))

end

As usual, head, tail and _&_ give the first element, the rest but the first element, and add an element to the front of a stream, respectively, while odd and even give the streams formed by the elements in the odd and even positions, respectively, and zip interleaves two streams. For example, odd(1 2 3 4 5 6 7 8 9 ...) is 1 3 5 7 9 ..., while even(1 2 3 4 5 6 7 8 9 ...) is 2 4 6 8 ..., and zip(1 3 5 7 9 ..., 2 4 6 8 ...) is 1 2 3 4 5 6 7 8 9 ....

Notice that all operations are behavioral, because they preserve the intended behavioral equivalence, which is “two streams are equivalent iff they have the same elements in the same order.” However, given a model, there are at least two interesting ways to generate this behavioral equivalence on that model. One uses observations (or contexts) built with head and tail, and the other uses observations built with head, odd and even. For example, both

head(tail(tail(tail(tail(S))))))

head(even(odd(odd(S))))

“observe” the fifth element of S, while the term

head(even(even(odd(odd(odd(S))))))

“observes” the 27th element:
There are reasons to consider the second cobasis better than the first. For example, a stream’s elements can be reached more quickly (e.g., the 27th element can be observed in 6 steps instead of 27), so that less computation is needed for testing. Also properties like \( \text{zip}(\text{odd}(S), \text{even}(S)) = S \) have much easier proofs with \( \{ \text{head}, \text{odd}, \text{even} \} \)-coinduction, but seem impossible using \( \{ \text{head}, \text{tail} \} \) as observers without circular \( \Delta \)-coinduction. We encourage the reader use Theorem 46 and show that indeed \( \Delta = \{ \text{head}, \text{tail} \} \) and \( \Delta' = \{ \text{head}, \text{odd}, \text{even} \} \) are both valid cobases. However, we will show in Example 101 how this can be formally proved using BOBJ.

Adding circular \( \Delta \)-coinduction to \( \Delta \)-coinduction and the five rules adapted from equational reasoning yields an inference system for behavioral properties which allows us to prove everything we know so far, but of course there is no guarantee that new inference rules will not be needed for more exotic examples. In fact, there is no complete inference system for behavioral satisfaction (see Section VI.E).

### IV.E.2 Circular \( \Delta \)-Coinduction

Let \( B = (\Sigma, \Gamma, E) \) be a behavioral specification that is fixed within this subsection, and let \( \Delta \) be a complete set of observers (see Definition 35), i.e., a special cobasis (see Proposition 39 and Theorem 41). To simplify notation, we consider all equations to be quantified by exactly the variables that occur in their two terms, and omit them whenever possible; also write \( t \equiv t' \) instead of \( B \models (\forall X) \ t = t' \).

**Definition 50** Substitutions \( \theta_1, \theta_2 : X \to T_1(Y) \) are behaviorally equivalent, written \( \theta_1 \equiv \theta_2 \), iff \( \theta_1(x) \equiv \theta_2(x) \) for every \( x \in X \). Terms \( t_1 \) and \( t_2 \) are strongly behaviorally equivalent, written \( t_1 \tilde{=} t_2 \), iff for any \( B \)-algebra \( A \) and any \( \tau_1, \tau_2 : X \to A \) with \( \tau_1(x) \overset{\theta_1}{=} \tau_2(x) \) for each \( x \in X \), \( \tau_1(t_1) \overset{\theta_1}{=} \tau_2(t_2) \).

Notice that \( \tilde{=} \) is symmetric and transitive but may not be reflexive, since, for example, terms of the form \( \sigma(x_1, ..., x_n) \) are not strongly equivalent to any term if \( \sigma \) is not
congruent (see also 5 of Proposition 51).

**Proposition 51** The following assertions hold:

1. \( t_1 \sim t_2 \) implies \( t_1 \equiv t_2 \);
2. \( t \equiv u \) iff \( t \equiv u \), whenever \( u \) is a \( \Gamma \)-term\(^{10} \);
3. \( t_1 \sim t_2 \) iff \( \gamma[t_1] \sim \gamma[t_2] \) for all appropriate \( \Gamma \)-experiments \( \gamma \);
4. \( t_1 \sim t_2 \) and \( \theta_1 \equiv \theta_2 \) imply \( \theta_1(t_1) \sim \theta_2(t_2) \);
5. \( \sigma \) is congruent iff \( \sigma(x_1, \ldots, x_n) \sim \sigma(x_1, \ldots, x_n) \).

**Proof:**

1. It is straightforward, because one can take \( \tau_1 = \tau_2 \) in Definition 50.

2. If \( t \sim u \) then \( t \equiv u \) by 1. Now suppose that \( t \equiv u \) and let \( \tau_1, \tau_2 \) be like in Definition 50. Since \( u \) contains only congruent operations, then one can easily show by structural induction that \( \tau_1(u) \equiv \tau_2(u) \). On the other hand, since \( \tau_1(t) \equiv \tau_2(t) \equiv \tau_2(u) \), it follows that \( t \equiv u \).

3. Suppose that \( t_1 \sim t_2 \), that \( \gamma \) is a \( \Gamma \)-experiment and that \( \tau_1, \tau_2: \text{var}(t_1, t_2) \cup \text{var}(\gamma) \rightarrow A \) are maps as in Definition 50. It is immediate that \( \tau_1(t_1) \equiv \tau_2(t_2) \). Since \( \gamma \) contains only congruent operations, it can be easily seen that \( \tau_1(\gamma[t_1]) = \tau_2(\gamma[t_2]) \). Conversely, suppose that \( \gamma[t_1] \equiv \gamma[t_2] \) for all appropriate \( \Gamma \)-experiments \( \gamma \), and let \( \tau_1, \tau_2: \text{var}(t_1, t_2) \rightarrow A \) be two maps as in Definition 50. It suffices to show that for any \( \Gamma \)-experiment \( \gamma \), \( \tau_1(\tau_1(t_1)) = \tau_2(\tau_2(t_2)) \) as functions in \( \{[\text{var}(\gamma) \rightarrow A] \rightarrow A \} \). Notice that giving a function in \( [\text{var}(\gamma) \rightarrow A] \) implies extending \( \tau_1, \tau_2 \) to functions \( \text{var}(t_1, t_2) \cup \text{var}(\gamma) \rightarrow A \), in which case, \( \tau_1(\gamma[t_1]) = \tau_1(\tau_2(t_2)) = \tau_2(\gamma[t_2]) \).

4. It follows easily, noticing that for any \( \tau_1, \tau_2: Y \rightarrow A \) with \( \tau_1(y) \equiv \tau_2(y) \), and any \( \theta_1, \theta_2: X \rightarrow T_\Gamma(Y) \) with \( \theta_1 \equiv \theta_2 \), it is the case that the maps \( \theta_1; \tau_1, \theta_2; \tau_2: X \rightarrow A \) also verify the property that \( (\theta_1; \tau_1)(x) \equiv (\theta_2; \tau_2)(x) \) for each \( x \in X \).

5. \( \sigma \) is congruent iff \( A_\sigma(a_1, \ldots, a_n) \equiv A_\sigma(a'_1, \ldots, a'_n) \) for any \( a_1, a'_1, \ldots, a_n, a'_n \) with \( a_1 \equiv a'_1, \ldots, a_n \equiv a'_n \) iff \( \tau_1(\sigma(x_1, \ldots, x_n)) \equiv \tau_2(\sigma(x_1, \ldots, x_n)) \) for \( \tau_1(x_i) = a_i \) and \( \tau_2(x_i) = a'_i \).

\(^{10}\)We write "\( \Gamma \)-terms" for simplicity, but all results hold for terms built with congruent operations.
for all $1 \leq i \leq n$ iff $\sigma(x_1, ..., x_n) \equiv \tilde{\sigma}(x_1, ..., x_n)$.

For the rest of the section, we assume some well-founded partial order $<$ on $\Gamma$-contexts which is preserved by the operations in $\Gamma$. For example, one such order is the depth of contexts.

**Definition 52** Terms $t_1$ and $t_2$ are $\Delta \triangledown$-coinductively equivalent, written $t_1 \equiv_{\Delta}^{\triangledown} t_2$, iff for each appropriate $\delta \in \Delta$, either $\delta[t_1] \equiv \delta[t_2] \equiv u$ for some $\Gamma$-term $u$, or $\delta[t_1] \equiv \theta_1(c[t_1])$ and $\delta[t_2] \equiv \theta_2(c[t_2])$ for some $\theta_1 \equiv \theta_2$ and $c < \delta$.

**Theorem 53** $t_1 \equiv_{\Delta}^{\triangledown} t_2$ implies $t_1 \equiv t_2$.

**Proof:** We first show by well-founded induction that for every appropriate experiment $\gamma$, $\gamma[t_1] \equiv \gamma[t_2]$. Let $\gamma$ be any experiment and assume that $\gamma'[t_1] \equiv \gamma'[t_2]$ for all experiments $\gamma' < \gamma$. Since $\Delta$ is a complete set of observers, there is some experiment $\gamma''$ such that $\gamma = \gamma''[\delta]$ for some $\delta \in \Delta$. If there is some $\Gamma$-term $u$ such that $\delta[t_1] \equiv \delta[t_2] \equiv u$ then $\gamma[t_1] \equiv \gamma[t_2] \equiv \gamma''[u]$ and $\gamma''[u]$ is a $\Gamma$-term, so by 2 of Proposition 51, $\gamma[t_1] \equiv \gamma[t_2]$. On the other hand, if $\delta[t_1] \equiv \theta_1(c[t_1])$ and $\delta[t_2] \equiv \theta_2(c[t_2])$ for some $\theta_1 \equiv \theta_2$ and $c < \delta$, then since the variables appearing in contexts are assumed to be always different from the other variables, one gets that $\gamma[t_1] = \theta_1(\gamma''[c[t_1]])$ and $\gamma[t_2] = \theta_2(\gamma''[c[t_2]])$, and so by the induction hypothesis for $\gamma' = \gamma''[c] < \gamma''[\delta] = \gamma$ and 4 of Proposition 51, $\gamma[t_1] \equiv \gamma[t_2]$. The rest follows by 3 of Proposition 51.

Therefore, the following inference rule is sound for behavioral satisfaction:

\[
\begin{align*}
(7) \quad \Delta \triangledown \text{-Coinduction} : & \quad t_1 \equiv_{\Delta}^{\triangledown} t_2 \\
& \quad (\forall X) \ t_1 = t_2
\end{align*}
\]

Figure IV.5: $\Delta \triangledown$-coinduction inference rule.

The following congruence criterion, which we will call the **BH criterion**, is the essence of that in [15]:

**Corollary 54** Given a complete set $\Delta$ of observers and given $\sigma \in \Sigma$ such that for each $\delta \in \Delta$, either $\delta[\sigma(x_1, ..., x_n)] \equiv u$ for some $\Gamma$-term $u$, or else $\delta[\sigma(x_1, ..., x_n)] = c[\sigma(t_1, ..., t_n)]$ for some $\Gamma$-terms $t_1, ..., t_n$ and $c < \delta$, then $\sigma$ is congruent.
Proof: Theorem 53 with \( t_1 = t_2 = \sigma(x_1, ..., x_n) \) and \( \theta_1 = \theta_2 = \theta \) with \( \theta(x_i) = t_i \) for all \( 1 \leq i \leq n \), gives \( \sigma(x_1, ..., x_n) \cong \sigma(x_1, ..., x_n) \). Then 5 of Proposition 51 gives congruence of \( \sigma \). □

The following simpler but common congruence criterion, which we here call the RG criterion, was presented in [137] together with the suggestion that it could be easily implemented in a system like CafeOBJ:

**Corollary 55** Given an operation \( \sigma \in \Sigma \) such that for each \( \delta \in \Gamma \), if the equation \( \delta[\sigma(x_1, ..., x_n)] = u \) for some \( \Gamma \)-term \( u \) is in \( E \), then \( \sigma \) is congruent.

**Proof:** This is the special case of the BH criterion where \( \Delta = \Gamma \) and there is no circularity (i.e., recurrence) in the definition of \( \sigma \). □

### IV.F Proving Congruence of Operations

This section discusses alternative techniques for proving that operations are behaviorally congruent, besides the congruence criteria we have already presented in the previous section. We see the work in this section “potentially useful”, because the congruence criteria are still strong enough to work on all our examples so far. The following reduces behavioral congruence to behavioral satisfaction of a certain equation, thus underlining the assertional character of this property which is further explored in Section VI.A.4:

**Proposition 56** Given \( B = (\Sigma, \Gamma, E) \) and \( \sigma \in \Sigma_{v_1...v_n}h_1...h_k,s \), let \( e_\sigma \) be the \( \Sigma \)-equation

\[(\forall Y, x_1, x'_1, ..., x_k, x'_k) \, \sigma(Y, x_1, ..., x_k) = \sigma(Y, x'_1, ..., x'_k) \text{ if } x_1 = x'_1, ..., x_k = x'_k, \text{ where } Y = \{y_1 : v_1, ..., y_m : v_m\}. \]

Then

1. \( \sigma \) is \( \Gamma \)-behaviorally congruent for a hidden \( \Sigma \)-algebra \( A \) iff \( A \models e_\sigma \), and
2. \( \sigma \) is behaviorally congruent for \( B \) iff \( B \models e_\sigma \).

**Proof:** It suffices to show 1. Suppose that \( \sigma \) is behaviorally congruent and let \( \theta : Y \cup \{x_1, x'_1, ..., x_k, x'_k\} \rightarrow A \) be any map such that \( \theta(x_i) \equiv \theta(x'_i) \) for all \( 1 \leq i \leq n \). Since \( \sigma \) is behaviorally congruent, one gets \( A_\sigma(\theta(Y), \theta(x_1), ..., \theta(x_k)) \equiv A_\sigma(\theta(Y), \theta(x'_1), ..., \theta(x'_k)) \),
that is, $\theta(\sigma(Y, x_1, \ldots, x_k)) \equiv \theta(\sigma(Y, x_1', \ldots, x_k'))$. Conversely, suppose that $A \models e_\sigma$ and let $a_Y \in A_{v_1 \ldots v_m}$ and $a_1, a'_1 \in A_{h_1}, \ldots, a_k, a'_k \in A_{h_k}$, such that $a_1 \equiv a'_1, \ldots, a_k \equiv a'_k$. Then let $\theta: Y \cup \{x_1, x_1', \ldots, x_k, x_k'\} \to A$ be defined by $\theta(Y) = a_Y$ and $\theta(x_i) = a_i, \theta(x_i') = a'_i$ for all $1 \leq i \leq n$. Since $A \models e_\sigma$, one gets that $\theta(\sigma(Y, x_1, \ldots, x_k)) \equiv \theta(\sigma(Y, x_1', \ldots, x_k'))$, that is, that $A_\sigma(a_Y, a_1, \ldots, a_k) \equiv A_\sigma(a_Y, a'_1, \ldots, a'_k)$.

**Example 57** The following behavioral theory of sets differs from that in Example 8 in Section III.F by having just one behavioral operation, in:

```
bth SET[X :: TRIV] is sort Set .
  op _in_ : Elt Set -> Bool .
  op empty : -> Set [ncong] .
  op add : Elt Set -> Set [ncong] .
  op _U_ : Set Set -> Set [ncong] .
  op _&_ : Set Set -> Set [ncong] .
vars E E' : Elt . vars S S' : Set .
  eq E in empty = false .
  eq E in add(E',S) = (E == E') or (E in S) .
  eq E in (S U S') = (E in S) or (E in S') .
  eq E in (S & S') = (E in S) and (E in S') .
end
```

By declaring the four operations non-congruent, in fact we allow more models. However, in the sequel we prove that all operations are congruent anyway, which means that actually there are not more models. By Proposition 24, both in and empty are congruent. Let $\Delta$ be the signature of TRIV together with in and notice that $\Delta$ is a cobasis for SET (because $\Delta$ contains exactly the signature of TRIV and the behavioral operations), so the six inference rules are sound for the behavioral satisfaction.

**Congruence of add:** By Proposition 56, we have to prove that

$$SET \models (\forall E : Elt, S, S' : Set) add(E, S) = add(E, S') \text{ if } S = S'.$$

By the theorem of hidden constants (Theorem 25), this is equivalent to proving

$$SET_S \models (\forall E : Elt) add(E, x) = add(E, x') \text{ if } x = x',$$

where $SET_S$ adds to $SET$ two hidden constants, $x$ and $x'$. By Proposition 26, it is equivalent to

$$SET' \models (\forall E : Elt) add(E, x) = add(E, x'),$$
where \( \text{SET}' \) adds to \( \text{SET} \) the equation \((\forall \emptyset) \ x = x'\). Now we use the six inference rules to prove that \( \text{SET}' \vdash_{\text{in}} (\forall E : \text{Elt}) \text{add}(E, x) = \text{add}(E, x') \). The following inferences give the proof:

1. \( \text{SET}' \vdash_{\text{in}} (\forall E', E : \text{Elt}) E' = E' \) \hspace{1cm} (1)
2. \( \text{SET}' \vdash_{\text{in}} (\forall E', E : \text{Elt}) x = x' \) \hspace{1cm} (4)
3. \( \text{SET}' \vdash_{\text{in}} (\forall E', E : \text{Elt}) E' \text{ in } x = E' \text{ in } x' \) \hspace{1cm} (5)
4. \( \text{SET}' \vdash_{\text{in}} (\forall E', E : \text{Elt}) (E' == E) = (E' == E) \) \hspace{1cm} (1)
5. \( \text{SET}' \vdash_{\text{in}} (\forall E', E : \text{Elt}) (E' == E) \text{ or } (E' \text{ in } x) = (E' == E) \text{ or } (E' \text{ in } x') \) \hspace{1cm} (5)
6. \( \text{SET}' \vdash_{\text{in}} (\forall E', E : \text{Elt}) E' \text{ in } \text{add}(E, x) = (E' == E) \text{ or } (E' \text{ in } x) \) \hspace{1cm} (4)
7. \( \text{SET}' \vdash_{\text{in}} (\forall E', E : \text{Elt}) E' \text{ in } \text{add}(E, x') = (E' == E) \text{ or } (E' \text{ in } x') \) \hspace{1cm} (4)
8. \( \text{SET}' \vdash_{\text{in}} (\forall E', E : \text{Elt}) E' \text{ in } \text{add}(E, x) = E' \text{ in } \text{add}(E, x') \) \hspace{1cm} (2, 3)
9. \( \text{SET}' \vdash_{\text{in}} (\forall E : \text{Elt}) \text{add}(E, x) = \text{add}(E, x') \) \hspace{1cm} (6)

The rest follows by the soundness of the six rule inference system.

**Congruence of \( \text{\_U}\):** By Proposition 56, Theorem 25 and Proposition 26, this is equivalent to \( \text{SET}' \equiv (\forall \emptyset) \ x_1 \ U \ x_2 = x_1' \ U \ x_2' \), where \( \text{SET}' \) adds to \( \text{SET} \) the hidden constants \( x_1, x_1', x_2, x_2' \) and the equations \((\forall \emptyset) \ x_1 = x_1', (\forall \emptyset) \ x_2 = x_2'\). One can infer the following:

1. \( \text{SET}' \vdash_{\text{in}} (\forall E : \text{Elt}) E \text{ in } x_1 = E \text{ in } x_1' \) \hspace{1cm} (1, 4, 5)
2. \( \text{SET}' \vdash_{\text{in}} (\forall E : \text{Elt}) E \text{ in } x_2 = E \text{ in } x_2' \) \hspace{1cm} (1, 4, 5)
3. \( \text{SET}' \vdash_{\text{in}} (\forall E : \text{Elt}) (E \text{ in } x_1) \text{ or } (E \text{ in } x_2) = (E \text{ in } x_1') \text{ or } (E \text{ in } x_2') \) \hspace{1cm} (5)
4. \( \text{SET}' \vdash_{\text{in}} (\forall E : \text{Elt}) E \text{ in } (x_1 \ U \ x_2) = (E \text{ in } x_1) \text{ or } (E \text{ in } x_2) \) \hspace{1cm} (4)
5. \( \text{SET}' \vdash_{\text{in}} (\forall E : \text{Elt}) E \text{ in } (x_1' \ U \ x_2') = (E \text{ in } x_1') \text{ or } (E \text{ in } x_2') \) \hspace{1cm} (4)
6. \( \text{SET}' \vdash_{\text{in}} (\forall E : \text{Elt}) E \text{ in } (x_1 \ U \ x_2) = E \text{ in } (x_1' \ U \ x_2') \) \hspace{1cm} (2, 3)
7. \( \text{SET}' \vdash_{\text{in}} (\forall \emptyset) \ x_1 \ U \ x_2 = x_1' \ U \ x_2' \) \hspace{1cm} (6)

The rest follows by the soundness of the six rule inference system. The congruence property for \( \text{\_U} \) and \( \text{neg} \) follow similarly.

A similar strategy was used in proving congruence of all the operations in the previous example. We capture it in a general method for proving congruence of an operation \( \sigma : \text{wh}_1...\text{wh}_k \rightarrow s \) for a hidden specification \( \mathcal{B} = (\Sigma, \Gamma, E) \), illustrated in Figure IV.6. The correctness of this method follows from Theorem 25 and Proposition 26. Let’s see now how it works on another example:

**Example 58** The following is a behavioral specification of stacks:
**Method for Proving Congruence:**

*Step 1:* Choose a suitable $\Delta \subseteq \Sigma$ and prove it is a cobasis. Often $\Delta$ is just $\Gamma$.

*Step 2:* Add appropriate new hidden constants $x_1, x'_1, \ldots, x_k, x'_k$ and equations 

$$(\forall \emptyset) x_1 = x'_1, \ldots, (\forall \emptyset) x_k = x'_k.$$ 

Let $B'$ denote the new specification.

*Step 3:* Show $B' \models_{\Delta} (\forall \emptyset) \sigma(Y, x_1, \ldots, x_k) = \sigma(Y, x'_1, \ldots, x'_k)$.

Figure IV.6: Method for proving congruence of operations.

```plaintext
bth STACK[X :: TRIV] is sort Stack .
op top : Stack -> Elt .
op pop : Stack -> Stack .
op push : Elt Stack -> Stack [ncong] .
var E : Elt . var S : Stack .
eq top(push(E,S)) = E .
eq pop(push(E,S)) = S .
end
```

Let us prove the congruence of $\text{push}$ using the method described above:

- **Step 1:** Let $\Delta$ be the signature of TRIV together with $\text{top}$ and $\text{pop}$. Then $\Delta$ is a cobasis for $\text{STACK}$ because it contains exactly the data signature and the behavioral operations.
- **Step 2:** Introduce two hidden constants $x$ and $x'$ and the equation $(\forall \emptyset) x = x'$.

Let $\text{STACK}'$ be the new hidden specification.

- **Step 3:** Prove $(\forall E : \text{Elt}) \text{push}(E, x) = \text{push}(E, x')$. One natural proof is:

1. $\text{STACK}' \models_{\text{top}, \text{pop}} (\forall E : \text{Elt}) \text{top}(\text{push}(E, x)) = E$  \hspace{1cm} (4)
2. $\text{STACK}' \models_{\text{top}, \text{pop}} (\forall E : \text{Elt}) \text{top}(\text{push}(E, x')) = E$  \hspace{1cm} (4)
3. $\text{STACK}' \models_{\text{top}, \text{pop}} (\forall E : \text{Elt}) \text{top}(\text{push}(E, x)) = \text{top}(\text{push}(E, x'))$  \hspace{0.5cm} (2), (3)
4. $\text{STACK}' \models_{\text{top}, \text{pop}} (\forall E : \text{Elt}) \text{pop}(\text{push}(E, x)) = x$  \hspace{1cm} (4)
5. $\text{STACK}' \models_{\text{top}, \text{pop}} (\forall E : \text{Elt}) \text{pop}(\text{push}(E, x')) = x'$  \hspace{1cm} (4)
6. $\text{STACK}' \models_{\text{top}, \text{pop}} (\forall E : \text{Elt}) x = x'$  \hspace{1cm} (4)
7. $\text{STACK}' \models_{\text{top}, \text{pop}} (\forall E : \text{Elt}) \text{pop}(\text{push}(E, x)) = \text{pop}(\text{push}(E, x'))$  \hspace{0.5cm} (2), (3)
8. $\text{STACK}' \models_{\text{top}, \text{pop}} (\forall E : \text{Elt}) \text{push}(E, x) = \text{push}(E, x')$  \hspace{1cm} (6)

■
Chapter V

Automation of Behavioral Reasoning

Rewriting is an efficient method in automated theorem proving. The usual technique to prove that two terms are equal, whatever “equal” might mean in a certain context, is to reduce them to their normal forms, compare their normal forms, and if they are the same then deduce that the two terms are equal. It is well known that this procedure is complete for equational reasoning whenever the rewriting system is confluent and terminates.

However, pure equational reasoning is not satisfactory enough in many practical situations. For example, the commutativity of addition cannot be proved without a form of induction over the Peano specification of natural numbers. Similarly but dually, most interesting properties of behavioral theories cannot be proved only by behavioral rewriting; coinduction [62, 145, 98, 66, 64, 85] and context induction [86, 44, 12] are among the most frequent higher level techniques used to prove behavioral equivalences of states, but unfortunately, they both require human intervention. Proving automatically behavioral equivalences is a useful feature of specification languages. CafeOBJ [37, 39] has implemented behavioral rewriting\(^1\) to make behaviorally sound reductions of terms. In this chapter, we propose new techniques which combine behavioral rewriting and coinduction.

\(^1\)Actually a subrelation of it; see Section V.C.
After a short section of preliminaries where we slightly modify some of the notions presented in Chapter II to take into account the exact set of variables over which a term was intended to be defined, we introduce in Section V.B the notion of \(\Omega\)-abstract rewriting system, where \(\Omega\) is a signature, as a generalization of abstract rewriting systems, in the sense that it is an \(\Omega\)-algebra together with a relation, the rewriting relation, which is required to be compatible with all the operations in \(\Omega\). The presence of a signature allows us to introduce another novel notion, that of extensional relation, and also the technique called extensional rewriting. Confluence properties are explored, in particular showing that the extensional rewriting is the smallest extensional congruence including the rewriting relation whenever the rewriting relation is confluent.

Behavioral rewriting is presented in Section V.C and it is for the five inference rules in Section IV.A what standard rewriting is for equational logic and (many sorted) algebra. In particular, we show that if the behavioral rewriting relation is confluent, then an equation \((\forall X) t = t'\) is derivable by the five rule inference system iff \(t\) and \(t'\) behaviorally rewrite to a common term. Interestingly, behavioral rewriting can be non-trivially implemented by standard term rewriting. This new result is very important because it provides an elegant way to add support for behavioral verification to languages like PVS [149], Maude [29], and many others. We present this result in Subsection V.C.1, together with some suggestions for further research.

Section V.D introduces behavioral coinductive rewriting, a method built on behavioral rewriting, its purpose being to prove automatically more behavioral equalities than behavioral rewriting alone can do. Unlike behavioral rewriting which rewrites well formed terms, behavioral coinductive rewriting rewrites sets of pairs of terms, called goals. Each high-level step of the method is either a many step behavioral rewriting or a \(\Delta\)-coinduction step applied to a goal, that is, a replacement of that goal with a finite set of goals generated by putting the operations in the cobasis on top of both terms in that goal. Two terms \(t\) and \(t'\) are behaviorally equivalent if the goal \(\{(t, t')\}\) can be reduced to a set of pairs of equal terms (called trivial goals). We obtain behavioral coinductive rewriting by instantiating abstract work on extensional rewriting done at the level of \(\Omega\)-abstract rewriting systems in Section V.B. Confluence and completeness (with regards to the first six inference rules) are investigated, and syntactic criteria for
termination and completeness of an algorithm based on behavioral coinductive rewriting are given in subsection V.D.1. Subsection V.D.2 shows that extensional rewriting can be instantiated into another direction, $\lambda$-calculus, suggesting that much of the work in some concrete theories like extensional $\lambda$-calculus could be done at an abstract level, increasing the chance to be reused. Our main motivation for the research in Section V.C is to capture common features of theories providing both a form of rewriting and one of extensionality, like extensional $\lambda$-calculus, and then to apply them to behavioral logics.

Circular coinductive rewriting, our strongest method for automated behavioral proving, is presented in Section V.E. It was introduced in [60] and implemented by Kai Lin in BOBJ. It is an algorithm for proving behavioral equalities that integrates behavioral rewriting with circular $\Delta$-coinduction. We give examples showing that this algorithm is surprisingly powerful in practice, even though a recent result [22] (see Section VI.E) shows that no such algorithm can be complete. Of course, incompleteness is more the rule than the exception for non-trivial theorem proving problem classes.

V.A Preliminaries

The standard definition of terms over a set of variables, as presented in Chapter II, is as “the smallest many-sorted set which contains all the variables and the expressions $\sigma(t_1, ..., t_n)$ for all $\sigma \in \Sigma_{s_1...s_n,s}$, whenever it already contains $t_i$ for all $1 \leq i \leq n$”. The terms form a $\Sigma$-algebra which has the universality property (Proposition 2).

Therefore, a term $t$ over some variables is also a term over more variables. It turns out later in this chapter that the set of variables over which a term has been intended to be considered is an important issue in extensional rewriting. For this reason, we redefine terms slightly differently:

**Definition 59** Given a set $\mathcal{X}$ of variables, the many-sorted set $T_{\Sigma}(\mathcal{X})$ of terms over $\mathcal{X}$ is defined recurrently as follows:

- $x.\{x\}$ is in $T_{\Sigma,s}(\mathcal{X})$ for each $x \in \mathcal{X}$,
- $\sigma(t_1.X_1, ..., t_n.X_n).X_1 \cup \cdots \cup X_n$ is in $T_{\Sigma,s}(\mathcal{X})$ if $\sigma \in \Sigma_{s_1...s_n,s}$ and $t_i.X_i \in T_{\Sigma,s_i}(\mathcal{X})$ for all $1 \leq i \leq n$,
- $t.X'$ is in $T_{\Sigma}(\mathcal{X})$ whenever $t.X$ is in $T_{\Sigma}(\mathcal{X})$, $X \subseteq X' \subseteq \mathcal{X}$, and $X'$ is finite.
Thus, many occurrences of the same term in the standard sense are allowed, indexed by finite sets of variables including the variables occurring in the term. So, $T_{\Sigma}(\mathcal{X})$ is much larger than the usual term algebra. The unicity in the universality property does not hold anymore, but an assignment of variables to values into an algebra can still be extended to terms:

**Proposition 60** $T_{\Sigma}(\mathcal{X})$ is a $\Sigma$-algebra. If $A$ is another $\Sigma$-algebra and $\theta : \mathcal{X} \to A$ is a many-sorted function, then $\theta^* : T_{\Sigma}(\mathcal{X}) \to A$ is a morphism of $\Sigma$-algebras, where $\theta^*$ is defined recurrently\(^2\) as follows:

- $\theta^*(x.\{x\}) = \theta(x),$
- $\theta^*(\sigma(t_1.X_1, ..., t_n.X_n).X_1 \cup \cdots \cup X_n) = A_\sigma(\theta^*(t_1.X_1), ..., \theta^*(t_n.X_n))$, and
- $\theta^*(t.X') = \theta^*(t.X)$.

**Proof:** It is straightforward that $T_{\Sigma}(\mathcal{X})$ is a $\Sigma$-algebra, because $T_{\Sigma,\sigma}(t_1.X_1, ..., t_n, X_n)$ can be defined as the term $\sigma(t_1.X_1, ..., t_n, X_n).X_1 \cup \cdots \cup X_n$, for any $\sigma \in \Sigma_{s_1...s_n,s}$ and $t_i.X_i \in T_{\Sigma,s_i}(\mathcal{X})$. It is also straightforward that $\theta^*$ is a morphism of $\Sigma$-algebras, because it was defined as a morphism. However, notice that $\theta^*$ does not uniquely extend\(^3\) $\theta$ to a morphism anymore; this is because there can be other morphisms, say, $\varphi : T_{\Sigma}(\mathcal{X}) \to A$, which are defined as $\theta^*$ on the first two types of terms, but which do not verify $\varphi(t.X') = \varphi(t.X)$ when $X \subseteq X'$.

To simplify writing, we write just $\theta$ instead of $\theta^*$. The notion of $\Sigma$-equation also needs to be redefined more precisely:

**Definition 61** A $\Sigma$-equation of sort $s$ is a pair $(t.X, t'.X)$ of terms in $T_{\Sigma,s}(\mathcal{X})$; we write it as usual, $(\forall X) t = t'$.

For the rest of the paper, we assume that $\mathcal{X}$ is an infinite set of variables. It is easy to see that $t.X \in T_{\Sigma}(X)$ whenever $t.X \in T_{\Sigma}(\mathcal{X})$; to simplify writing, we sometimes write “$t \in T_{\Sigma}(X)$” instead of “$t.X \in T_{\Sigma}(X)$”, and just $t$ instead of $t.X$ whenever $X$

\(^2\)We use the same notations as in Definition 59.

\(^3\)In the sense that they take $x.\{x\}$ to $\theta(x)$.
is understood from the context or not important. By convention, terms \( t \) and \( t' \) are assumed to be over the same variables, i.e., there is some \( X \) s.t. \( t, t' \in T_X(X) \), whenever the pair \( (t, t') \) is mentioned to be an element in any binary relation on terms.

### V.B Rewriting in \( \Omega \)-Abstract Rewriting Systems

Many important results about term rewriting, including termination and confluence, are actually special cases of more general results about a binary relation on a set. The classical notion of abstract rewriting system, as a pair \( (A, \rightarrow) \), where \( A \) is a set and \( \rightarrow \) is a binary relation on \( A \), was extended to the many-sorted case in [51], but it still seems to be too restrictive for our framework.

Our novel notion of \( \Omega \)-abstract rewriting system is introduced in this section, together with some abstract rewriting techniques and results. Though it would significantly ease the reading, knowledge about abstract rewriting systems is not assumed in this section.

**Definition 62** Let \( \Omega \) be a many-sorted signature containing only unary operations. An \( \Omega \)-abstract rewriting system, abbreviated \( \Omega \)-ARS, is a pair \( (A, \rightarrow) \), where \( A \) is a \( \Omega \)-algebra and \( \rightarrow \) is a relation on \( A \) compatible with the operations.

If \( \Omega \) is empty then \( \Omega \)-ARSs are nothing but abstract rewriting systems, abbreviated ARSs. In this section, we consider an arbitrary but fixed \( \Omega \)-ARS, \( (A, \rightarrow) \). Let \( \xrightarrow{*} \) denote the reflexive and transitive closure of \( \rightarrow \). We sometimes write \( "(a, b) \xrightarrow{*} (a', b')" \) instead of \( "a \xrightarrow{*} a' \text{ and } b \xrightarrow{*} b'" \).

**Definition 63** Let \( \mathcal{G}_A \) be the set of finite strings with elements in \( A \times A \). The elements of \( \mathcal{G}_A \) are called goals and will be noted \( G, G', G_1, \) etc., or \( [(a_1, b_1), \ldots, (a_n, b_n)] \), or\(^4\) \( [X, (a_i, b_i), Y] \). An operation \( \omega \in \Omega_{s,s'} \) is appropriate for an element \( a \in A_{s'} \) iff \( s = s'' \).

Given a goal \( G = [X, (a_i, b_i), Y] \) such that there is some appropriate \( \omega \in \Omega \) for \( a_i \) and \( b_i \), let \( G_{\Omega(i)} \) be the goal \( [X, (A_{\omega_1}(a_i), A_{\omega_1}(b_i)), \ldots, (A_{\omega_k}(a_i), A_{\omega_k}(b_i)), Y] \), where \( \omega_1, \ldots, \omega_k \) are all the appropriate operations in \( \Omega \).

\(^4\)X and \( Y \) stand for the sequences of pairs \( (a_1, b_1), \ldots, (a_{i-1}, b_{i-1}) \) and \( (a_{i+1}, b_{i+1}), \ldots, (a_n, b_n) \).
We use \( G_{\Omega(i_1,i_2,...,i_n)} \) as a shorthand for \(((G_{\Omega(i_1)})_{\Omega(i_2)})...{\Omega(i_n)}\) and sometimes write it \( G_{\Omega(\alpha)} \), where \( \alpha \) is the sequence \((i_1,...,i_n)\). Notice that if \( \alpha \) is a valid sequence for \( G \), that is, if \( G_{\Omega(\alpha)} \) exists, then \( \alpha \) is a valid sequence for any \( G' \) which replaces each pair \((a,b)\) of \( G \) by some \((a',b')\) having the same sort as \((a,b)\); in other words, the validity of \( \alpha \) does not depend on the pairs of a goal, but only on their sort and their order.

**Definition 64** Let \( \sqsupseteq_r, \sqsupseteq_e \) be relations on \( G_A \), called rewriting and extensionality, respectively, defined as \( G \sqsupseteq_r G' \) iff \( G = [(a_1,b_1),...,(a_n,b_n)], G' = [(a'_1,b'_1),...,(a'_n,b'_n)] \), and \((a_i,b_i) \rightarrow_r (a'_i,b'_i)\) for all \( 1 \leq i \leq n \), and \( G \sqsupseteq_e G' \) iff \( G' = G_{\Omega(i)} \) for some \( 1 \leq i \leq n \).

Let \( \models \) be the relation \((\sqsupseteq_r \cup \sqsupseteq_e)^*\). If \( G \models G' \) then we say that \( G \) extensionally rewrites to \( G' \).

Let \( \sqrt{\ } \) denote the diagonal relation on \( G_A \), i.e., \( \sqrt{\ } = \{ (G,G) \mid G \in G_A \} \). Given two relations \( R \subseteq A \times B \) and \( S \subseteq B \times C \), their composition is the relation \( R;S \subseteq A \times C \) defined as \( a \ (R;S) \ c \) iff there is some \( b \in B \) such that \( a \ R \ b \) and \( b \ S \ c \). Since functions are special relations, one can have compositions between functions and relations as well. \( R^{-1} \) is the inverse of a relation \( R \), that is, \( a \ R^{-1} \ b \) iff \( b \ R \ a \). We remind the reader that a relation \( R \) is confluent if and only if \( (R^{-1})^*;R^* \subseteq R^*;(R^{-1})^* \).

**Proposition 65** The following hold:

1. \( \sqsupseteq_r;\sqsupseteq_r = \sqsupseteq_r \)
2. \( G \sqsupseteq_r G' \) yields \( G_{\Omega(i)} \sqsupseteq_r G'_{\Omega(i)} \) for all \( 1 \leq i \leq |G| \),
3. \( \sqsupseteq_r;\sqsupseteq_e \subseteq \sqsupseteq_e;\sqsupseteq_r \)
4. \( \sqsupseteq_r;\sqsupseteq_e \subseteq \sqsupseteq_e;\sqsupseteq_r \)
5. \( \sqsupseteq_e;\sqsupseteq_r \subseteq \sqsupseteq_r \)
6. \( \sqsupseteq_e;\sqsupseteq_r \subseteq \sqsupseteq_r;\sqsupseteq_e \)
7. \( \sqsupseteq_e;\sqsupseteq_r \subseteq \sqsupseteq_r;\sqsupseteq_e \)
8. \( \sqsupseteq_e;\sqsupseteq_e \subseteq \sqrt{\ } \cup \ (\sqsupseteq_e;\sqsupseteq_e) \)
9. \( \sqsupseteq_e;\sqsupseteq_e \subseteq \sqrt{\ } \cup \ (\sqsupseteq_e;\sqsupseteq_e) \)
10. \( \sqsupseteq_e \) is confluent\(^5\)
11. \( \sqsupseteq_e \subseteq \sqrt{\ } \cup \ (\sqsupseteq_e;\sqsupseteq_e) \)

\(^5\)The confluence of \( \sqsupseteq_e \) says that given a goal \( G \) and sequences \( \alpha,\beta \) of valid indexes, that is, the goals \( G_{\Omega(\alpha)} \) and \( G_{\Omega(\beta)} \) exist, then there are some sequences of indexes \( \alpha',\beta' \) such that \( G_{\Omega(\alpha,\beta')} = G_{\Omega(\beta',\alpha')} \). Notice that \( \alpha' \) and \( \beta' \) do not depend on the pairs \((a,b)\) in \( G \), but only on their sorts and their order.
12. \(\vdash_e \subseteq \vdash_r;\vdash_e\).

13. If \(\rightarrow\) is confluent then \(\Rightarrow_r\) is strongly confluent.

14. If \(\rightarrow\) is confluent then \(\vdash_r;\Rightarrow_e;\vdash_r\subseteq\Rightarrow_r;\vdash_r\).

15. If \(\rightarrow\) is confluent then \(\vdash\) is strongly confluent.

Proof:

1. It follows by the transitivity of \(\ast_r\).

2. It is true because \(\rightarrow\) is a congruence.

3. Suppose that \(G \Rightarrow_r G' \Rightarrow_e G''\), where \(1 \leq i \leq |G'|\). Then \(G \Rightarrow_e G_{\Omega(i)} \Rightarrow_r G''\).

4. It follows immediately from 3, by induction.

5. It suffices to show that \(\vdash_e;\Rightarrow_r\) is reflexive, transitive and includes \(\Rightarrow_r\) and \(\Rightarrow_e\).

These properties are easy to check; for example, transitivity follows from 3:

\[(\vdash_e;\Rightarrow_r); (\vdash_e;\Rightarrow_r) = \vdash_e; (\Rightarrow_r;\vdash_e); \Rightarrow_r \subseteq \vdash_e; (\vdash_e;\Rightarrow_r); \Rightarrow_r = \vdash_e;\Rightarrow_r\]

6. Suppose that \(G_{\Omega(i)} \vdash_e G \Rightarrow_r G'\). Then by 2, \(G_{\Omega(i)} \Rightarrow_r G_{\Omega(i)}' \vdash_e G'\).

7. It follows easily from 6, by induction.

8. Suppose that \(G_{\Omega(i)} \vdash_e G \Rightarrow_e G_{\Omega(j)}\). If \(i = j\) then \((G_{\Omega(i)}, G_{\Omega(j)}) \in \sqrt{\cdot}\). If \(i \neq j\), say \(i < j\), then it can be easily seen that \(G_{\Omega(i,j-1+k)} = G_{\Omega(i,i)}\), where \(k\) is given in Definition 63. Therefore \(G_{\Omega(i)} \Rightarrow_e G_{\Omega(i,j)} \vdash_e G_{\Omega(j)}\).

9. It is relatively easy to prove by induction that \(\vdash_e^n;\Rightarrow_e \subseteq (\sqrt{\cup} \Rightarrow_e;\vdash_e);\vdash_e^{n-1}\) for all \(n > 1\), so \(\vdash_e;\Rightarrow_e \subseteq (\sqrt{\cup} \Rightarrow_e;\vdash_e);\vdash_e = (\sqrt{\cup} \Rightarrow_e);\vdash_e\).

10. It can be easily proved by induction that \(\vdash_e;\Rightarrow_e^n \subseteq (\sqrt{\cup} \Rightarrow_e)^n;\vdash_e\), so \(\vdash_e;\Rightarrow_e \subseteq (\sqrt{\cup} \Rightarrow_e)^n;\vdash_e = \vdash_e;\vdash_e^n\).

11. It can be proved as follows:

\[
\vdash_e;\Rightarrow_e = \vdash_r;\ast_e;\vdash_e \quad \text{(by 5)}
\]

\[
\subseteq \vdash_r; (\sqrt{\cup} \Rightarrow_e);\ast_e \quad \text{(by 9)}
\]

\[
= (\vdash_r; (\vdash_r;\Rightarrow_e));\ast_e
\]

\[
\subseteq (\vdash_r; (\vdash_r;\Rightarrow_e));\ast_e \quad \text{(by 6)}
\]

\[
= (\sqrt{\cup} \Rightarrow_e);\vdash_e \quad \text{(by 5)}
\]
12. From 11, it can be easily proved by induction that \( \vdash_e \subseteq (\vee \Rightarrow_e)^* \vdash_e \). But \((\vee \Rightarrow_e)^* \) is equal to \( \Rightarrow_e \).

13. It is immediate, because \( \rightarrow \) confluent implies \( \Rightarrow \) strongly confluent.

14. One proof can be:

\[
\vdash; \Rightarrow_r = \vdash_r \vdash; \Rightarrow \Rightarrow_e \Rightarrow_r \quad \text{(by 5)} \\
\subseteq \vdash_r \vdash; \Rightarrow \Rightarrow_e \quad \text{(by 14)} \\
= \Rightarrow_r \vdash 
\]

15. It can be proved as follows:

\[
\vdash; \Rightarrow = \vdash; \Rightarrow_e \Rightarrow_r \quad \text{(by 5)} \\
\subseteq \Rightarrow_e \vdash; \Rightarrow_r \quad \text{(by 14)} \\
= \Rightarrow \vdash 
\]

\[\Box\]

**Definition 66** A goal is **trivial** iff it contains only pairs of equal elements. Let \( \downarrow_n \) be the relation defined as \( a \downarrow_n b \) iff \( [(a, b)] ((\vee \Rightarrow_e)^n \Rightarrow_r) G_{\perp} \) for some trivial goal \( G_{\perp} \), and let \( \downarrow = \bigcup_{n \geq 0} \downarrow_n \).

Therefore, \( a \downarrow_n b \) iff the goal \( [(a, b)] \) can be extensionally reduced to a trivial goal by at most \( n \) extensional steps. Obviously, \( \downarrow_n \subset \downarrow_{n+1} \) for all \( n \geq 0 \), and \( a \downarrow b \) iff \( [(a, b)] \Rightarrow G_{\perp} \) for some trivial goal \( G_{\perp} \).

**Definition 67** A relation \( R \) on \( A \) is **extensional** iff for any elements \( a, b \in A \) for which there are some appropriate \( \omega \in \Omega \), \( A_\omega(a) \ R A_\omega(b) \) for all appropriate \( \omega \in \Omega \) implies \( a \ R b \).

**Lemma 68** The following hold:

1. \( \downarrow \) is reflexive,
2. \( \downarrow \) is symmetric,
3. If \( \rightarrow \) is confluent then \( \downarrow \) is transitive,
4. $\downarrow$ is extensional.

Proof:

1. Goals $[(a, a)]$ are already trivial.

2. It is obvious because $\Rightarrow_r$ and $\Rightarrow_e$ do not make any distinction between the left and right elements of pairs.

3. Suppose that $a, b, c$ have the same sort and that $a \downarrow b$ and $b \downarrow c$. Then there are some sequences of indexes $\alpha$ and $\beta$ such that $[(a, b)]_{\Omega(\alpha \beta)} \Rightarrow_r G^{(a,b)}$ and $[(b, c)]_{\Omega(\beta \alpha)} \Rightarrow_r G^{(b,c)}$, for some trivial goals $G^{(a,b)}$ and $G^{(b,c)}$. Since $\alpha$ and $\beta$ are valid sequences for both $[(a, b)]$ and $[(b, c)]$, and since $\Rightarrow_e$ is confluent (10 in Proposition 65), there are two sequences $\alpha'$ and $\beta'$ such that $[(a, b)]_{\Omega(\alpha \beta)} \Rightarrow_r G^{(a,b)}$ and $[(b, c)]_{\Omega(\beta \alpha)} \Rightarrow_r G^{(b,c)}$. We claim that there is some trivial goal $G^{(a,c)}$ such that $[(a, c)]_{\Omega(\alpha \beta)} \Rightarrow_r G^{(a,c)}$. By 7 in Proposition 65, there are some trivial goals $G^{(a,b)}_{\perp}$ and $G^{(b,c)}_{\perp}$ such that $[(a, b)]_{\Omega(\alpha \beta)} \Rightarrow_r G^{(a,b)}_{\perp}$ and $[(b, c)]_{\Omega(\beta \alpha)} \Rightarrow_r G^{(b,c)}_{\perp}$. Notice that the pairs on the $i$-th position in the goals $[(a, b)]_{\Omega(\alpha \beta)}$, $[(b, c)]_{\Omega(\beta \alpha)}$, and $[(a, c)]_{\Omega(\alpha \beta)}$ have the form $(a', b')$, $(b', c')$, and $(a', c')$, respectively, where $a' = A_{\omega_1; \ldots; \omega_n}(a)$, $b' = A_{\omega_1; \ldots; \omega_n}(b)$, and $c' = A_{\omega_1; \ldots; \omega_n}(c)$, for some sequence $\omega_1; \ldots; \omega_n$ of appropriate operations in $\Omega$, and that the following also holds: $a' (\Rightarrow^*; \Rightarrow^*) b' (\Rightarrow^*; \Rightarrow^*) c'$. Then $a' (\Rightarrow^*; \Rightarrow^*) c'$ by the confluence of $\Rightarrow$.

4. Let $a, b$ be elements for which there are some appropriate $\omega \in \Omega$, and suppose that $A_{\omega}(a) \downarrow A_{\omega}(b)$ for any appropriate $\omega \in \Omega$. Then $[(a, b)]_{\Omega(1)} \Rightarrow G_{\perp}$ for some trivial goal $G_{\perp}$; since $[(a, b)] \Rightarrow_e [(a, b)]_{\Omega(1)}$, one gets that $[(a, b)] \Rightarrow G_{\perp}$, that is, $a \downarrow b$.

\[ \square \]

**Theorem 69** $\downarrow$ is included in any extensional congruence including $\Rightarrow$. Moreover, if $\Rightarrow$ is confluent then $\downarrow$ is the smallest extensional congruence that includes $\Rightarrow$.

**Proof:** Let us first prove

\[ \text{If the sequence is empty, then suppose that } a' = a, b' = b, \text{ and } c' = c. \]
Lemma 70 If $a \downarrow_n b$ for some $n \geq 1$ then there is some $m < n$ such that $A_\omega(a) \downarrow_m A_\omega(b)$.

Proof: Indeed, let $n_0 \leq n$ be the smallest number such that $a \downarrow_{n_0} b$. If $n_0 = 0$ then $a \xrightarrow{\omega \omega} b$; by the compatibility of $\rightarrow$, $A_\omega(a) \xrightarrow{\omega \omega} A_\omega(b)$, therefore $A_\omega(a) \downarrow_0 A_\omega(b)$. If $n_0 > 0$ then $[(a, b)] \Rightarrow_e [(a, b)]_{\Omega(1)} \downarrow_{n_0-1} G_\perp$ for some trivial goal $G_\perp$. Hence $A_\omega(a) \downarrow_{n_0-1} A_\omega(b)$. Now take $m = n_0 - 1$. □

Suppose that $R$ is another extensional congruence including $\rightarrow$, and let us prove by induction on $n$ that $a \downarrow_n b$ implies $a R b$. If $n = 0$ then $a \xrightarrow{\omega \omega} b$, so $a R b$ since $R$ includes $\rightarrow$ and is an equivalence. If $n \geq 1$ then by the observation above, there is $m < n$ such that $A_\omega(a) \downarrow_m A_\omega(b)$ for all appropriate $\omega \in \Omega$. By the induction hypothesis, $A_\omega(a) R A_\omega(b)$, so by the extensionality of $R$, $a R b$.

It can be easily seen that $\downarrow$ includes $\rightarrow$, and, by Lemma 68, that $\downarrow$ is an extensional equivalence if $\rightarrow$ is confluent. To show its compatibility with operations, let $a, b$ be two elements such that $a \downarrow b$ and $\omega \in \Omega$ be an appropriate operation for them. Then there is some $n \geq 1$ such that $a \downarrow_n b$. Hence, there is some $m < n$ such that $A_\omega(a) \downarrow_m A_\omega(b)$ for each appropriate $\omega \in \Omega$. Therefore, $A_\omega(a) \downarrow A_\omega(b)$, so $\downarrow$ is a congruence. □

V.C Behavioral Rewriting

Behavioral rewriting is for the first five inference rules presented in Section IV.A what standard term rewriting is for equational logic. Since we are not aware of any reference introducing the concept of behavioral rewriting in its full generality as we see it, we do it in this section.

Definition 71 A $\Sigma$-rewriting rule is a pair of terms $(l.Y, r.Y)$, written $(\forall Y) l \rightarrow r$. A behavioral (or hidden) $\Sigma$-rewriting system is a triple $(\Sigma, \Gamma, R)$, where $\Sigma$ is a hidden signature, $\Gamma$ is a hidden subsignature of $\Sigma$, and $R$ is a set of $\Sigma$-rewriting rules. ■

Ordinary term rewriting is not sound for the behavioral satisfaction anymore. This is because non-behavioral operations might not preserve the behavioral equivalence. Consequently, it needs to be modified, but first, let us establish the following:

\footnote{Notice that $\downarrow_0 \subseteq \downarrow_1$, so we can consider $n \geq 1$.}
**Framework:** $R = (\Sigma, \Gamma, \mathcal{R})$ is a behavioral $\Sigma$-rewriting system and $\mathcal{B} = (\Sigma, \Gamma, \mathcal{E})$ is the associated bspec, that is, $\mathcal{E} = \{ (\forall Y) \ l = r \mid (\forall X) \ l \rightarrow r \in \mathcal{R} \}$.

Now, we modify the standard term rewriting to take into account the fact that some operations may not be behaviorally congruent:

**Definition 72** The behavioral (term) rewriting relation associated to the behavioral rewriting system $\mathcal{R}$ is the smallest relation $\Rightarrow$ such that:

- for each $(\forall Y) \ l \rightarrow r$ in $\mathcal{R}$ and each $\theta : Y \rightarrow T_{\Sigma}(X)$, $\theta(l) \Rightarrow \theta(r)$,
- if $t \Rightarrow t'$ and $\text{sort}(t, t') \in V$ then $\sigma(W, t) \Rightarrow \sigma(W, t')$ for all $\sigma \in \text{Der}(\Sigma)$, and
- if $t \Rightarrow t'$ and $\text{sort}(t, t') \in H$ then $\delta(W, t) \Rightarrow \delta(W, t')$ for all $\delta \in \Gamma$.

When $\mathcal{R}$ is important, we write $\Rightarrow_{\mathcal{R}}$ instead if $\Rightarrow$.

Behavioral rewriting modifies the standard term rewriting as follows: each time a hidden redex is found, apply the rewriting rule when there are only behavioral operations on the path from that redex going toward the root until a visible sort is found. If a visible sort is not found, the rewriting is still applied if all operations on the path from the redex to the root are behavioral.

**Definition 73** A $\Sigma$-context $c$ is behavioral iff all operations on the path to $\bullet$ in $c$ are behavioral, and $c$ is safe iff either it is behavioral or there is some visible behavioral context $c'$ such that $c = c''[c']$ for some appropriate $c''$. A $\Sigma$-term $t$ is weakly linear iff for any variable $x$ occurring in $t$ such that $t = c_1[x] = c_2[x]$, $c_1$ is safe if and only if $c_2$ is safe. $\mathcal{R}$ is weakly left (right) linear iff $l (r)$ is weakly linear for any rewriting rule $(\forall Y) \ l \rightarrow r$ in $\mathcal{R}$.

The following can be seen as an equivalent definition of $\Rightarrow$:

**Proposition 74** $t \Rightarrow t'$ iff there is a rewriting rule $(\forall Y) \ l \rightarrow r$ in $\mathcal{R}$, a safe context $c$, and a substitution $\theta$ such that $t = c[\theta(l)]$ and $t' = c[\theta(r)]$.

---

8 According to our methodology, as many operations as possible should be declared behavioral; in particular, all congruent operations should be declared behavioral [137]. If one does not find our methodology appropriate, than one can read “behavioral or congruent” instead of just “behavioral” for the rest of the paper.
**Proof:** It follows directly from the construction of \( \xrightarrow{} \) in Definition 72 and that of a safe context in Definition 73. \( \square \)

CafeOBJ successfully implements only a subrelation of behavioral rewriting (used within the command `reduce`). More exactly, the rewriting rule is applied when there are only behavioral operations on the path from the redex going toward the root until a visible sort is found. If no visible sort is found on that path, the rewriting does not take place. Therefore, CafeOBJ’s behavioral rewriting does not consider the case when all operations on the path to the redex are behavioral and there is no operation of visible result on that path. We believe that the behavioral rewriting might be easily implemented in its full generality within the CafeOBJ’s currently unsound `behavioral-reduce` command [37]. However, we implemented it under the command `reduce` in BOBJ.

Before we proceed further, we need to establish a few other notations and conventions for the rest of the section. In all examples of cobases we met in practical examples, \( B' \) can be \( (\text{Der}(\Sigma), \Gamma, E) \) and \( \Delta \) contains only \( \Gamma \)-terms. To ease the presentation, the following

**Assumption:** \( \Delta \) is a cobasis of \( B \) with \( B' = (\text{Der}(\Sigma), \Gamma, E) \) and \( \Delta \subseteq \text{Der}(\Gamma) \).

Let \( \Omega \) be the signature built as follows: it has the same sorts as \( \Delta \), and for each \( \delta \in \Delta_{s_1...s_i...s_n} \) having the \( i \)-th argument of hidden sort, add an operation \( \delta_i \) in \( \Omega_{s_i,s} \).

**Proposition 75** \( T_\Sigma(X) \) can be organized as an \( \Omega \)-algebra, denoted by \( T \), and \( \equiv \) is an extensional congruence on \( T \).

**Proof:** Indeed, given \( \omega = \delta_i \in \Omega_{s_i,s} \) and a term \( t.X \) of hidden sort \( s_i \), let \( \omega[t] \) be the term \( \delta_i(x_1, ..., x_{i-1}, t, x_{i+1}, ..., x_n) \), where \( x_1, ..., x_{i-1}, x_{i+1}, ..., x_n \) are some arbitrary but fixed variables in \( X - X \). The choice of these variables is a function only of \( X \). Thus, if \( t'.X \) is another term of sort \( s_i \), then \( \omega[t'] \) is \( \delta_i(x_1, ..., x_{i-1}, t', x_{i+1}, ..., x_n) \), for the same variables \( x_1, ..., x_{i-1}, x_{i+1}, ..., x_n \). Now, let \( (T_\Sigma(X))_\omega(t) \) be defined as \( \omega[t] \). It is easy to see that \( \equiv \) is an equivalence; by Definition 40, \( \equiv \) is extensional. \( \square \)

Then the \( \Delta \)-coinduction rule introduced in Subsection IV.C.4 can also be formulated as:

\[
\begin{align*}
\text{(6) \; \Delta-Coinduction:} & \quad \frac{\omega[t] \equiv_{Eq,\Delta} \omega[t'] \text{ for all appropriate } \omega \in \Omega}{t \equiv_{Eq,\Delta} t'}
\end{align*}
\]
The following proposition follows immediately from the definitions:

**Proposition 76** $\equiv_{Eq}$ is a congruence and $\equiv_{Eq,\Delta}$ is an extensional congruence on $T$, and $\equiv_{Eq} \subseteq \equiv_{Eq,\Delta} \subseteq \equiv$.

The next proposition says that the whole machinery developed for $\Omega$-abstract rewriting systems can be used in the framework of behavioral rewriting. It also suggests that behavioral rewriting alone is not strong enough for automated behavioral proving, in the same way in which hidden equational deduction was not strong enough without coinduction:

**Proposition 77** $(T, \Rightarrow)$ is an $\Omega$-abstract rewriting system and $\Rightarrow \subseteq \equiv_{Eq}$.

**Proof:** First, notice that by Proposition 75, $T$ is an $\Omega$-algebra. The fact that $\Rightarrow$ is compatible with the operations in $\Omega$ follows by Definition 62 and the definition of $\Omega$ above. □

Confluence and/or termination criteria for behavioral rewriting may be an interesting area of research, but we are not interested in them in this paper. However, our experience is that in many practical situations all operations are congruent, so $\Gamma$ might be taken to be $\Sigma [137]$, in which case behavioral rewriting is exactly ordinary rewriting. Also, notice that termination of ordinary rewriting produces termination of behavioral rewriting, because $\Rightarrow$ is a subrelation of ordinary rewriting; thus, any termination criterion for ordinary rewriting is still applicable for behavioral rewriting.

Classical results in term rewriting can be generalized to behavioral rewriting:

**Proposition 78** If $\Rightarrow$ is confluent then $\equiv_{Eq} = \Rightarrow; \Leftarrow$.

**Proof:** It follows immediately by Propositions 77 and 76 that $\Rightarrow; \Leftarrow \subseteq \equiv_{Eq}$. Also, it can be easily seen that $\Rightarrow; \Leftarrow$ is closed under the rules (1)–(5); confluence of $\Rightarrow$ is needed only for closure under transitivity. Since $\equiv_{Eq}$ is the smallest relation closed under the five rules, $\equiv_{Eq} \subseteq \Rightarrow; \Leftarrow$. □

If $\Rightarrow$ is canonical then let $bnf(t)$ denote the normal form of $t$.

**Corollary 79** If $\Rightarrow$ is canonical then $t \equiv_{Eq} t'$ iff $bnf(t) = bnf(t')$. 
V.C.1 Implementing Behavioral Rewriting by Standard Rewriting

In this subsection we present a possible implementation of behavioral rewriting by standard term rewriting. Since there already exist very fast term rewriting engines, such as that of Maude [29], it may be more efficient to just follow a procedure similar to the following to simulate behavioral rewriting than to implement a behavioral rewriting engine.

We suggest the reader to imagine the operations in terms painted with one of two colors, either green or red. A rewriting step can be applied to a position in a term if and only if the operation on that position is painted green. To make the writing simpler, we will use the same names for the green operations and an index \( r \) for the red operations. Therefore, we need to introduce copies of some operations. More exactly, given a hidden signature \( \Sigma \), let \( \Sigma_r \) be the standard signature having the same sorts as \( \Sigma \), and an operation \( \sigma_r : s_1...s_n \rightarrow h \) for each operation of hidden result \( \sigma : s_1...s_n \rightarrow h \) in \( \Sigma \). Also, let \( \Sigma' \) be the signature \( \Sigma \cup \Sigma_r \cup g : s \rightarrow s \mid \) for all sorts \( s \). Now, let \( \Gamma \) be a hidden subsignature of \( \Sigma \) and let \( \text{PAINTER}_{\Sigma, \Gamma} \) be the \( \Sigma' \)-term rewriting system consisting of the following rules:

1) \( (\forall X) g(\sigma_r(X)) \rightarrow g(\sigma(X)) \), for all operations \( \sigma : s_1...s_n \rightarrow h \) in \( \Sigma \) of hidden result \( h \), where \( X \) is a set of appropriate distinct variables, \( \{x_1 : s_1, ..., x_n : s_n\} \), and \( \sigma(X) \) is a shorthand for \( \sigma(x_1, ..., x_n) \);

2) \( (\forall Z_j, X) \tau(Z_j, \sigma(X)) \rightarrow \tau(Z_j, \sigma_r(X)) \), for all \( \tau : w_1...w_{j-1}hw_{j+1}...w_k \rightarrow s \) in \( \Sigma - \Gamma \) and \( \sigma : s_1...s_n \rightarrow h \) in \( \Sigma \) such that \( h \) is a hidden sort, where \( X \) is a set of variables as above and \( Z_j = \{z_1 : w_1, ..., z_{j-1} : w_{j-1}, z_{j+1} : w_{j+1}, ..., z_k : w_k\} \), and \( \tau(Z_j, \sigma(X)) \) is a short hand for \( \tau(z_1, ..., z_{j-1}, \sigma(x_1, ..., x_n), z_{j+1}, ..., z_k) \);

3) \( (\forall Z_j, X) \sigma'_r(Z_j, \sigma(X)) \rightarrow \sigma'_r(Z_j, \sigma_r(X)) \), for all \( \sigma' : w_1...w_{j-1}hw_{j+1}...w_k \rightarrow s \) and \( \sigma : s_1...s_n \rightarrow h \) in \( \Sigma \) such that \( h \) is a hidden sort;

4) \( (\forall Z_j, X) \delta(Z_j, \sigma_r(X)) \rightarrow \delta(Z_j, \sigma(X)) \), for all \( \delta : w_1...w_{j-1}hw_{j+1}...w_k \rightarrow s \) in \( \Gamma \) and \( \sigma' : s_1...s_n \rightarrow h \) in \( \Sigma \) such that \( h \) is a hidden sort.

The role of these rewriting rules is to dynamically paint any term such that the behav-
ioral rewriting steps can be soundly applied only on green positions. According to the description of the behavioral rewriting technique at the beginning of the section, the color of an operation $\sigma$ on a certain position $p$ in a term $u$ should be green if and only if one of the following two situations appear: there are only behavioral operations on the path from $p$ (excluding $\sigma$) to the root of $u$ (including the operation on the top of $u$), or, there are only behavioral operations on the path starting with $p$ (excluding $\sigma$) and going upward in $u$ until the first visible sort is found (including the operation which has the visible result).

An immediate fact is that the operation on the top of any term must be green. This is the role of the first rewriting rule. The operations $g$ are introduced only to have control over the operation on the top of the terms. The second rewriting rule says that an operation of hidden sort which is immediately below a non-behavioral operation must be red, that is, the behavioral rewriting cannot be soundly applied at that position. The third and the fourth rewriting rules show how the colors are propagated top-down.

**Proposition 80** \(\text{PAINTER}_{\Sigma, \Gamma}\) is canonical.

**Proof:** It can be easily seen that \(\text{PAINTER}_{\Sigma, \Gamma}\) is an orthogonal $\Sigma'$-term rewriting system, so it is confluent. To show its termination, we show by structural induction that any operation on any position in a fixed term can change its color at most a finite number of times by doing rewritings in \(\text{PAINTER}_{\Sigma, \Gamma}\). Let us analyze all the situations under which an operation can change its color. Let $u$ be any $\Sigma'$-term and let $\alpha : s_1 ... s_n \rightarrow h$ be an operation of hidden result in $\Sigma \cup \Sigma_r$ on a position $p$ in $u$. Notice that $\alpha$'s color can be changed by a rule of type 1) only if its color is green and the operation just above it in $u$ is a $g$ operation, and that its color can never be changed back to red. The color of $\alpha$ can be changed by a rule of type 2) only if its color is green, and that its color is never going to be changed back to green, because there is no way to apply the rule 4) at that position. The color of $\alpha$ can be changed by a rule of type 3) only if its color is green and the color of the operation above it in $u$, say $\sigma_r$, is red. The only possibility to change its color back to green is to firstly change the color of $\sigma_r$ to green, and then to apply one of the rules 2) or 4). By the induction hypothesis, the color of $\sigma_r$ can be changed only for a finite number of times, so the color of $\alpha$ can be changed only for a finite number of
times, too. The color of $\alpha$ can be changed by a rule of type 4) only if its color is red and the color of the operation above it, $\delta$, is green. The only possibility to change its color back to red is to firstly change the color of $\delta$ to red, and then to apply a rule of type 3). By the induction hypothesis, the color of $\delta$ can be changed only for a finite number of times, so the color $\alpha$ can be changed only for a finite number of times, too. Therefore, the color of operations in any $\Sigma'$-term can be changed only for a finite number of times.

Since the rewriting rules of PAINTER$_{\Sigma,\Gamma}$ do not change the structure of terms, but only the color of their operations, and since the color of operations can change for only a finite number of times, we deduce that PAINTER$_{\Sigma,\Gamma}$ terminates. □

Therefore, any $\Sigma'$-term has a unique normal form.

**Definition 81** Let $\Sigma^{\Gamma}$ be the $\Sigma'$-term rewriting relation associated with PAINTER$_{\Sigma,\Gamma}$. Given a $\Sigma'$-term $u$, let $\varphi(u)$ be its unique normal form in PAINTER$_{\Sigma,\Gamma}$; if $t$ is a $\Sigma$-term, $\varphi(t)$ is often called the coloring of $t$. Given a $\Sigma \cup \Sigma'$-term $u$, let $\psi(u)$ be the $\Sigma$-term forgetting all the colors of $u$.

Let $G : \Sigma \cup \Sigma_r \rightarrow \Sigma$ and $R : \Sigma \cup \Sigma_r \rightarrow \Sigma \cup \Sigma_r$ be two signature morphisms which are identities on sorts and for any $\sigma \in \Sigma$,

- $\sigma^G = \sigma$,
- $\sigma^G_r = \sigma$,
- $\sigma^R = \sigma$ if the result of $\sigma$ is visible,
- $\sigma^R = \sigma_r$ if the result of $\sigma$ is hidden, and
- $\sigma^R_r = \sigma_r$.

Given a $(\Sigma \cup \Sigma_r)$-term $u$, let $u^G$ and $u^R$ denote the terms obtained by changing the root operation of $u$, say $\rho$, by $\rho^G$ and $\rho^R$, respectively. More precisely, let

- $x^G = x^R = x$ for any variable $x$,
- $(\rho(u_1, ..., u_n))^G = \rho^G(u_1, ..., u_n)$ for any $\rho \in \Sigma \cup \Sigma_r$, and
- $(\rho(u_1, ..., u_n))^R = \rho^R(u_1, ..., u_n)$ for any $\rho \in \Sigma \cup \Sigma_r$.

Notice that $u^G$ and $u^R$ are not the renamed versions of $u$ by the morphisms $G$ and $R$, respectively.
Proposition 82  Given any two \( (\Sigma \cup \Sigma_r) \)-terms \( u \) and \( u' \), any \( \Sigma \)-term \( t \) and any \( \Sigma \)-context \( c \) appropriate for \( t \), the following hold:

1) \( g(u) \xrightarrow{\Sigma,t} g(u') \) implies \( \psi(u) = \psi(u') \);

2) \( \psi(\varphi(t)) = t \);

3) If \( u = \rho(u_1,\ldots,u_n) \) then \( \varphi(u) = \rho(\varphi(u_1^G),\ldots,\varphi(u_n^G)) \) when \( \rho \in \Gamma \), and \( \varphi(u) = \rho(\varphi(u_1^R),\ldots,\varphi(u_n^R)) \) when \( \rho \in (\Sigma - \Gamma) \cup \Sigma_r \);

4) \( \varphi(u^G) = \varphi(\psi(u)) \) and \( \varphi(u^R) = \varphi(\psi(u)^R) \);

5) \( \varphi(c[t]) = \varphi(c)[\varphi(t)] \) if \( c \) is safe, and \( \varphi(c[t]) = \varphi(c)[\varphi(t)^R] \) otherwise;

6) If \( t \) is weakly linear\(^9\) then for each \( \Sigma \)-substitution \( \theta \), there is a \( (\Sigma \cup \Sigma_r) \)-substitution \( \theta' \) such that \( \varphi(\theta(t)) = \theta'(\varphi(t)) \).

Proof:  1) Since each of the four types of rules in \( \text{PAINTER}_{\Sigma,\Gamma} \) modifies only the colors of operations, and since \( \psi \) forgets their colors, one can immediately see that this assertion is true.

2) Since \( t \xrightarrow{\Sigma,\Gamma} \varphi(t) \), we get \( g(t) \xrightarrow{\Sigma,\Gamma} \varphi(g(t)) \), so by 1), \( \psi(t) = \psi(\varphi(t)) \). But \( \psi(t) = t \).

3) It follows easily from the fact that \( \text{PAINTER}_{\Sigma,\Gamma} \) is canonical.

4) It can be proved by structural induction. There are three cases:
   (i) If \( u \) is a variable then it is obvious;
   (ii) If \( u = \delta(u_1,\ldots,u_n) \) or \( u = \delta_r(u_1,\ldots,u_n) \) for some \( \delta \in \Gamma \), then
   \[ \varphi(u^G) = \varphi(\delta(u_1,\ldots,u_n)) \]
   \[ = \delta(\varphi(u_1^G),\ldots,\varphi(u_n^G)) \quad \text{(by 3)} \]
   \[ = \delta(\psi(u_1),\ldots,\psi(u_n)) \quad \text{(by hyp)} \]
   \[ = \varphi(\psi(u_1),\ldots,\psi(u_n)) \quad \text{(by 3)} \]
   \[ = \varphi(\psi(u)) \]
   and
   \[ \varphi(u^R) = \varphi(\delta_r(u_1,\ldots,u_n)) \]
   \[ = \delta_r(\varphi(u_1^R),\ldots,\varphi(u_n^R)) \quad \text{(by 3)} \]
   \[ = \delta_r(\psi(u_1),\ldots,\psi(u_n)) \quad \text{(by hyp)} \]
   \[ = \varphi(\delta_r(\psi(u_1),\ldots,\psi(u_n))) \quad \text{(by 3)} \]
   \[ = \varphi(\psi(u)^R) \]

\(^9\)Suppose that \( t \) contains no variables \( \bullet \) (used in contexts).
(iii) If $u = \tau(u_1, \ldots, u_n)$ or $w = \tau_r(u_1, \ldots, u_n)$ for some $\tau \in \Sigma - \Gamma$, then

\[
\begin{align*}
\varphi(u^G) &= \varphi(\tau(u_1, \ldots, u_n)) \\
&= \tau(\varphi(u_1^R), \ldots, \varphi(u_n^R)) \quad \text{(by 3)} \\
&= \tau(\varphi(\psi(u_1)^R), \ldots, \varphi(\psi(u_n)^R)) \quad \text{(by hyp)} \\
&= \varphi(\tau(\psi(u_1), \ldots, \psi(u_n))) \quad \text{(by 3)} \\
&= \varphi(\psi(u))
\end{align*}
\]

and

\[
\begin{align*}
\varphi(u^R) &= \varphi(\tau_r(u_1, \ldots, u_n)) \\
&= \tau_r(\varphi(u_1^R), \ldots, \varphi(u_n^R)) \quad \text{(by 3)} \\
&= \tau_r(\varphi(\psi(u_1)^R), \ldots, \varphi(\psi(u_n)^R)) \quad \text{(by hyp)} \\
&= \varphi(\tau_r(\psi(u_1), \ldots, \psi(u_n))) \quad \text{(by 3)} \\
&= \varphi(\psi(\psi(w)^R))
\end{align*}
\]

5) We show by induction on the length of the path from the root of $c$ to $\star$ that

\[
\varphi(c[t]) = \begin{cases} 
\varphi(c)[\varphi(t)] & \text{if } c \text{ is safe} \\
\varphi(c)[\varphi(t^R)] & \text{if } c \text{ is not safe}
\end{cases}
\]

for all appropriate $\Sigma$-terms $t$. If $c = \star$ then $\varphi(c) = \star$ and $c$ is safe, and $\varphi(c[t]) = \varphi(t) = \varphi(c)[\varphi(t)]$. Now suppose that $c$ has the form $c'[\sigma(t_1, \ldots, t_{i-1}, \star, t_{i+1}, \ldots, t_n)]$ for some context $c'$ which verifies the property above, some operation $\sigma \in \Sigma$, and some appropriate $\Sigma$-terms $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n$. Then

(i) If $c$ is safe then either the sort of $\star$ is visible, or the sort of $\star$ is hidden but then $\sigma \in \Gamma$ and $c'$ is safe. If the sort of $\star$ is visible, then obviously $\varphi(c[t]) = \varphi(c)[\varphi(t)]$ since there is no rewriting rule in $\text{PAINTER}_{\Sigma, \Gamma}$ that can color a visible operation in red. If the sort of $\star$ is hidden, then

\[
\begin{align*}
\varphi(c[t]) &= \varphi(c'[\sigma(t_1, \ldots, t_{i-1}, t, t_{i+1}, \ldots, t_n)]) \\
&= \varphi(c'[\sigma(\varphi(t_1), \ldots, \varphi(t_{i-1}), t, \varphi(t_{i+1}), \ldots, \varphi(t_n))]) \\
&= \varphi(c'[\sigma(\varphi(t_1), \ldots, \varphi(t_{i-1}), \star, \varphi(t_{i+1}), \ldots, \varphi(t_n))]) \\
&= (\varphi(c'[\sigma(\varphi(t_1), \ldots, \varphi(t_{i-1}), \star, \varphi(t_{i+1}), \ldots, \varphi(t_n))]) \varphi(t)) \\
&= (\varphi(c'[\sigma(\varphi(t_1, \ldots, t_{i-1}, \star, t_{i+1}, \ldots, t_n))]) \varphi(t)) \\
&= \varphi(c)[\varphi(t)].
\end{align*}
\]
(ii) If \( c \) is not safe, then the sort of \( \star \) is hidden and either \( \sigma \in \Sigma - \Gamma \), or \( \sigma \in \Gamma \) and \( c' \) is not safe. If \( \sigma \in \Sigma - \Gamma \) then

\[
\varphi(c[t]) = \varphi(c'[\sigma(t_1, \ldots, t_{i-1}, t, t_{i+1}, \ldots, t_n)])
\]

\[
= \varphi(c')[\varphi(\sigma^C(t_1, \ldots, t_{i-1}, t, t_{i+1}, \ldots, t_n))] \quad \text{(by hyp)}
\]

\[
= \varphi(c')[\sigma^C(\varphi(t^R_1), \ldots, \varphi(t^R_{i-1}), \varphi(t^R_i), \varphi(t^R_{i+1}), \ldots, \varphi(t^R_n))]) \quad \text{(by 3)}
\]

\[
= \varphi(c')[\sigma^C(\varphi(t^R_1), \ldots, \varphi(t^R_{i-1}), \varphi(t^R_i), \varphi(t^R_{i+1}), \ldots, \varphi(t^R_n)))](\varphi(t^R))
\]

\[
= (\varphi(c')[\varphi(\sigma^C(t_1, \ldots, t_{i-1}, t, t_{i+1}, \ldots, t_n)])(\varphi(t^R)) \quad \text{(by 3)}
\]

\[
= (\varphi(c')[\varphi(\sigma^C(t_1, \ldots, t_{i-1}, t, t_{i+1}, \ldots, t_n)])(\varphi(t^R)) \quad \text{(by hyp)}
\]

\[
= \varphi(c)[\varphi(t^R)],
\]

where \( C \) is either \( G \) or \( R \), depending on whether \( c' \) is safe or not. If \( \sigma \in \Gamma \) and \( c' \) is not safe, then the proof follows exactly as above, replacing \( C \) by \( R \).

6) Let \( \theta \) be any \( \Sigma \)-substitution which is the identity on variables \( \star \) used in contexts. We firstly show by induction on the number of variables different from \( \star \) occurring in \( \overline{t} \), that for any \( \Sigma \)-term \( \overline{t} \) with at most one occurrence of any variable different from \( \star \), there is a \((\Sigma \cup \Sigma_r)\)-substitution \( \theta_{\overline{t}} \) such that for each variable \( x \) in \( \overline{t} \) different from \( \star \),

\[
\theta_{\overline{t}}(x) = \begin{cases} 
\varphi(\theta(x)) & \text{if } \overline{t}_x \text{ is safe,} \\
\varphi(\theta(x)^R) & \text{if } \overline{t}_x \text{ is not safe,}
\end{cases}
\]

where \( \overline{t}_x \) is the context with \( \overline{t} = \overline{t}_x[x] \), and \( \varphi(\theta(\overline{t})) = \theta_{\overline{t}}(\varphi(\overline{t})) \).

Let \( \overline{t} \) be a term as above. If \( \overline{t} \) has no variables different from \( \star \) then \( \varphi(\theta(\overline{t})) = \varphi(\overline{t}) = \theta_{\overline{t}}(\varphi(\overline{t})) \) for any substitutions \( \theta \) and \( \theta_{\overline{t}} \). Now, let \( x \neq \star \) be a variable in \( \overline{t} \) such that \( \overline{t} = \overline{t}_x[x] \). Then \( \theta(\overline{t}) = \theta(\overline{t}_x)[\theta(x)] \), and since \( \overline{t}_x \) is safe iff \( \theta(\overline{t}_x) \) is safe, one gets that

\[
\varphi(\theta(\overline{t})) = \varphi(\theta(\overline{t}_x)[\theta(x)])
\]

\[
= (\varphi(\theta(\overline{t}_x)))[\varphi(\theta(x)^C)] \quad \text{(by 5)}
\]

\[
= (\theta_{\overline{t}_x}(\varphi(\overline{t}_x)))[\varphi(\theta(x)^C)] \quad \text{(by hyp)}
\]

\[
= \theta_{\overline{t}}(\varphi(\overline{t}_x)[x])
\]

\[
= \theta_{\overline{t}}(\varphi(\overline{t})) \quad \text{(by 5)},
\]

where \( C \) is either \( G \) or \( R \), depending on whether \( \overline{t}_x \) is safe or not, and where \( \theta_{\overline{t}}(x) = \varphi(\theta(x)^C) \) and \( \theta_{\overline{t}}(y) = \theta_{\overline{t}_x}(y) \) for all \( y \neq x \).
In order to complete the proof, let us now consider that \( t \) is a weakly linear \( \Sigma \)-term which contains no variables \( \star \). Let \( \overline{t} \) be the “linearization” of \( t \), that is, the term obtained from \( t \) replacing the occurrences of the same variable \( x \) by distinct new variables, say \( x^1, \ldots, x^{k_x} \). Let us ambiguously consider that \( \theta(x^1) = \cdots = \theta(x^{k_x}) \), meaning that \( \theta(t) = \theta(\overline{t}) \). Then there is a \( (\Sigma \cup \Sigma_r) \)-substitution \( \theta_t \) such that \( \theta_t(t) = \theta_t(\overline{t}) \). Since \( t \) is weakly linear, one can easily observe that \( \theta_t(x^i) = \theta_t(x^j) \) for all \( 1 \leq i, j \leq k_x \). Therefore, there is a \( (\Sigma \cup \Sigma_r) \)-substitution \( \theta' \) such that \( \theta'(\varphi(t)) = \theta(\varphi(\overline{t})) \). □

**Framework:** \( \varphi(R) \) is the set \( \{(\forall X)(\varphi(l) \to \varphi(r)) \mid (\forall X)l \to r \in R\} \), \( \varphi(\mathcal{R}) \) is the associated \( (\Sigma \cup \Sigma_r) \)-rewriting system, and \( \xrightarrow{\varphi(R)} \) is the associated rewriting relation. \( \mathcal{R}' \) is the \( \Sigma' \)-rewriting system \( \varphi(\mathcal{R}) \cup \text{PAINTER}_{\Sigma, \Gamma} \) and \( \Rightarrow \) is the relation \( \xrightarrow{\varphi(R)} \cup \Sigma \Gamma \).

Definition 73 can be easily generalized to \( (\Sigma \cup \Sigma_r) \)-contexts:

**Definition 83** A \( (\Sigma \cup \Sigma_r) \)-context \( c \) is **behavioral** iff all operations on the path to \( \bullet \) in \( c \) are in \( \Gamma \), and \( c \) is **safe** iff either it is behavioral or there is some visible behavioral context \( c' \) such that \( c = c'[c'] \) for some appropriate \( c'' \). ■

**Proposition 84** If \( c \) is safe, then \( \psi(c[\theta(\varphi(l))]) \Rightarrow \psi(c[\theta(\varphi(r))]) \) for any appropriate rewriting rule \( (\forall X)(\varphi(l) \to \varphi(r)) \) in \( \varphi(\mathcal{R}) \) and any substitution \( \theta \).

**Proof:** Let \( c \) be any safe \( (\Sigma \cup \Sigma_r) \)-context, \( (\forall Y)(\varphi(l) \to \varphi(r)) \) be any appropriate rewriting rule in \( \varphi(\mathcal{R}) \), and \( \theta \) be any substitution. Notice that \( \psi(c) \) is a safe \( \Sigma \)-context in the sense of Definition 73. Then by Proposition 74, \( \psi(c)[(\theta; \psi)(l)] \Rightarrow \psi(c)[(\theta; \psi)(r)] \).

The rest follows by 1 in Proposition 82. □

**Theorem 85** Suppose that \( g(\varphi(t)) \Rightarrow^* g(u) \) is an innermost rewriting sequence, where \( t \) is a \( \Sigma \)-term and \( u \) is a \( (\Sigma \cup \Sigma_r) \)-term. Then

1) \( t \Rightarrow^* \psi(u) \);

2) If \( \mathcal{R} \) is weakly left linear and \( g(u) \) is a n.f. in \( \mathcal{R}' \) then \( \psi(u) \) is a bnf in \( \mathcal{R} \);

3) If \( \mathcal{R} \) terminates on \( t \) then \( \mathcal{R}' \) terminates on \( g(\varphi(t)) \).

**Proof:** Let us firstly prove the following
Lemma 86 The following hold:

(a) if \( u \) contains a subterm which is an instance by \( \theta \) of the left-hand side of a rule of type (2) or (3) in \( \text{PAINTER}_{\Sigma, \Gamma} \), then \( \theta(\sigma(X)) \) is in normal form;

(b) if \( u = c[u'] \) for some \( (\Sigma \cup \Sigma_r)\)-context \( c \) such that \( u' \) is not in normal form, then \( c \) is safe;

(c) if we let \( g(u_1), g(u_2), \ldots, g(u_k) \) denote all the terms occurring after applications of rewriting rules in \( \varphi(R) \), then \( t \Rightarrow \psi(u_1) \Rightarrow \psi(u_2) \Rightarrow \cdots \Rightarrow \psi(u_k) = \psi(u) \).

Proof: a) Let us first consider that \( u \) contains a subterm\(^{10} \theta(\tau(Z_j, \sigma(X))) \), for some substitution \( \theta \). There are three possibilities by which the operation \( \tau \in \Sigma - \Gamma \) can appear over the operation of hidden result \( \sigma \): by using a rule of type 1), a rule of type 4), or a rule in \( \varphi(R) \). If a rule of type 1) or 4) was used to change the color of \( \tau \) from red to green at a certain step in the rewriting sequence using a substitution \( \theta' \), then because of the innermost rewriting strategy assumed, \( \theta'(\sigma(X)) \) was already in normal form at that moment. Again, because of the innermost strategy used and because the subterm \( \theta'(\tau(Z_j, \sigma(X))) \) is not in normal form for \( \text{PAINTER}_{\Sigma, \Gamma} \) but all the left-hand sides of \( \varphi(R) \) are in normal form for \( \text{PAINTER}_{\Sigma, \Gamma} \), we deduce that no subsequent reduction toward \( g(u) \) modifies the subterm \( \theta'(\tau(Z_j, \sigma(X))) \), so it is equal to \( \theta(\tau(Z_j, \sigma(X))) \). Hence \( \theta(\sigma(X)) = \theta'(\sigma(X)) \), so \( \theta(\sigma(X)) \) is in normal form. If a rule \( (\forall \gamma) \ l \rightarrow r \) is used at a certain step to produce the operation \( \tau \) over the operation \( \sigma \) under a substitution \( \theta' \), then because \( \varphi(r) \) is in normal form for \( \text{PAINTER}_{\Sigma, \Gamma} \), the only possibility is that \( \tau \) occurs in \( \varphi(r) \) and has a variable of sort \( h \), say \( x \), as its \( j \)th argument, such that \( \theta'(x) = \theta'(\sigma(X)) \). Again, because of the innermost strategy used, \( \theta'(\sigma(X)) \) must be in normal form and left unchanged until the end of the reduction. Therefore, \( \theta(\sigma(X)) \) is in normal form. If \( u \) contains a subterm which is an instance by \( \theta \) of the left-hand side of a rewriting rule of type 3) in \( \text{PAINTER}_{\Sigma, \Gamma} \), then it can be similarly shown that \( \theta(\sigma(X)) \) is in normal form. The main difference from the above is that the operation \( \sigma' \) can appear over the operation \( \sigma \) by using rules of type 2) or 3) instead of 1) or 4).

b) Suppose that \( c \) is not safe, that is, there is at least an operation, say \( \rho \), in \( (\Sigma - \Gamma) \cup \Sigma_r \) on the path to \( \ast \) in \( c \), such that its argument term containing \( \ast \) has only operations of hidden result in \( \Gamma \) (zero or more) on the path to \( \ast \). More rigorously, there is a subterm \( \rho(\tau(Z_j, \sigma(X))) \) of \( u \) such that \( u' \) is a subterm (not necessary proper) of \( \theta(\sigma(X)) \), so \( \theta(\sigma(X)) \) is not in normal form. But this is in contradiction with a). Hence \( c \) is safe.

c) Therefore, the derivation \( g(\varphi(t)) \Rightarrow^\ast g(u) \) is of the form:

\[
g(\varphi(t)) \xrightarrow{\varphi(R)} g(u_1) \xrightarrow{\Sigma, \Gamma}^* g(u_1') \xrightarrow{\varphi(R)} g(u_2) \xrightarrow{\Sigma, \Gamma}^* g(u_2') \xrightarrow{\varphi(R)} \cdots \xrightarrow{\varphi(R)} g(u_k) \xrightarrow{\Sigma, \Gamma}^* g(u)
\]

\(^{10}\)See the rules of type 2) in \( \text{PAINTER}_{\Sigma, \Gamma} \) for the notational conventions.
Because of 2 in Proposition 82, \( \psi(u_i) = \psi(u'_i) \) for all \( 1 \leq i \leq k \) (suppose that \( u_k' \) is \( u \)). Since \( \psi(\varphi(t)) = t \) by 1 in Proposition 82, it suffices to show that \( \psi(u'_i) \Rightarrow \psi(u_{i+1}) \) for all \( 0 \leq i < k \) (suppose that \( u'_0 \) is \( \varphi(t) \)). Let us consider such an \( i \). Since \( g(u'_i) \varphi(R) \Rightarrow g(u_{i+1}) \), there is some rewriting rule in \( \varphi(R) \), say \( (\forall Y) l \rightarrow r \), and there is some \((\Sigma \cup \Sigma_r)\)-context \( c \) and a substitution \( \theta \) such that \( u'_i = c[\theta(\varphi(l))] \) and \( u_{i+1} = c[\theta(\varphi(r))] \). By b) we obtain that \( c \) is safe, and by Proposition 84 that \( \psi(u'_i) \Rightarrow \psi(u_{i+1}) \). □

1) It follows easily from c) in Lemma 86.

2) Let us firstly prove the following

**Lemma 87** If \( \mathcal{R} \) is weakly left linear, \( t \) is a b.n.f. in \( \mathcal{R} \) iff \( \varphi(t) \) is a n.f. in \( \mathcal{R}' \).

**Proof:** First, suppose that \( t \) is b.n.f. such that \( \varphi(t) \) is not a n.f.. Then there is some \((\Sigma \cup \Sigma_r)\)-term \( u \) such that \( g(\varphi(t)) \varphi(R) \Rightarrow g(u) \), so by c) in Lemma 86, \( t \Rightarrow \psi(u) \), contradiction. Conversely, if \( \varphi(t) \) is a n.f. and \( t \) is not a b.n.f., then by Proposition 74, there is a rewriting rule \( (\forall Y) l \rightarrow r \), a safe context \( c \), and a \( \Sigma \)-substitution \( \theta \) such that \( t = c[\theta(l)] \). Since \( c \) is safe, by 5) in Proposition 82, \( \varphi(t) \varphi(R) \Rightarrow \varphi(c)[\theta(l)] \). Since \( l \) is weakly linear, by 6) in Proposition 82, there is a \((\Sigma \cup \Sigma_r)\)-substitution \( \theta' \) such that \( \varphi(t) = \varphi(c)(\theta'(\varphi(l))) \). Therefore, \( \varphi(t) \varphi(R) \Rightarrow \varphi(c)[\theta'(\varphi(r))] \), contradiction. □

If \( g(u) \) is a normal form then \( u \) is also a normal form and is rooted in an operation in \( \Sigma \), so \( u = \varphi(u) \); moreover, by 4) in Proposition 82, \( u = \varphi(\psi(u)) \). Then by Lemma 87, \( \psi(u) \) is a behavioral normal form.

3) If \( \mathcal{R}' \) does not terminate on \( g(\varphi(t)) \), since \( \text{PAINTER}_{\Sigma, \Gamma} \) terminates, by c) one can easily see that there exists an infinite behavioral rewriting sequence of \( t \) in \( \mathcal{R} \). □

Now, we’d like to devise a behavioral rewriting engine, from now on abbreviated \( \text{BRE} \), taking as input a behavioral rewriting system \( \mathcal{R} \) and a term \( t \) over the signature of \( \mathcal{R} \), and returning a behavioral normal form of \( t \) in \( \mathcal{R} \):

\[
(\mathcal{R}, t) \Rightarrow \boxed{\text{BRE}} \Rightarrow \text{bnf}(t).
\]

Suppose that we have a term rewriting engine which can be set to use an innermost rewriting strategy, such as OBJ3 [80], CafeOBJ [37], Maude [29], or any other, which we abbreviate \( \text{RE} \) from now on. Inspired by the theory above, we can implement \( \text{BRE} \) on
**Algorithm** $BRE(\mathcal{R}, t)$

1. generate $\varphi(\mathcal{R})$ and $\text{PAINTER}_{\Sigma, \Gamma}$
2. get $\varphi(t)$ using $\text{RE}(\text{PAINTER}_{\Sigma, \Gamma}, t)$
3. get $nf(g(\varphi(t)))$, say $g(u)$, using $\text{RE}(\varphi(\mathcal{R}) \cup \text{PAINTER}_{\Sigma, \Gamma}, g(\varphi(t)))$
4. return $\psi(u)$.

Figure V.1: Implementing behavioral rewriting by standard term rewriting.

The top of $RE$ like in Figure V.1. Notice that step 1 can be easily implemented, in $O(n^2)$, where $n$ is the size of $\mathcal{R}$. Proposition 80 implies that Step 2 terminates and $\varphi(t)$, or the painted $t$, is unique. By 3) in Theorem 85, step 3 terminates whenever $\mathcal{R}$ terminates as a behavioral rewriting system, and by 2) in Theorem 85, if $\mathcal{R}$ is weakly linear then a behavioral normal form for $\mathcal{R}$ can be extracted from any normal form returned at step 3. If more sophisticated algorithms are needed, note that by 1) in Theorem 85, step 3 can be stopped at any time on an intermediate term, say $g(u)$, which then can produce an intermediate term, $\psi(u)$, in a possible behavioral rewriting sequence.

**Example 88** We remind the reader the behavioral specification of non-deterministic stack presented in Example 9 in Section III.F, but this time without mix-fix-notation:

```plaintext
bth NDSTACK is protecting NAT.
  sort Stack.
  op top : Stack -> Nat.
  op pop : Stack -> Stack.
  op push : Stack -> Stack [ncong].
  var S : Stack.
  eq pop(push(S)) = S.
end
```

Then $\text{PAINTER}_{\Sigma, \Gamma}$ becomes (written as a BOBJ standard theory):

```plaintext
th PAINTER is pr NAT.
  sort Stack.
  op top : Stack -> Nat [strat (1 0)].
  op pop : Stack -> Stack [strat (1 0)].
  op popr : Stack -> Stack [strat (1 0)].
  op push : Stack -> Stack [strat (1 0)].
  op pushr : Stack -> Stack [strat (1 0)].
  op g : Stack -> Stack [strat (1 0)].
```
\[
\begin{align*}
\text{op } g : \text{Nat} \rightarrow \text{Nat} & \quad [\text{strat (1 0)}]. \\
\text{var } S : \text{Stack}. \\
\text{eq } g(\text{popr}(S)) = g(\text{pop}(S)). & \quad *** \text{ type 1) } \\
\text{eq } g(\text{pushr}(S)) = g(\text{push}(S)). & \quad *** \text{ type 1) } \\
\text{eq } \text{push}(\text{pop}(S)) = \text{push}(\text{popr}(S)). & \quad *** \text{ type 2) } \\
\text{eq } \text{push}(\text{push}(S)) = \text{push}(\text{pushr}(S)). & \quad *** \text{ type 2) } \\
\text{eq } \text{popr}(\text{pop}(S)) = \text{popr}(\text{popr}(S)). & \quad *** \text{ type 3) } \\
\text{eq } \text{popr}(\text{push}(S)) = \text{popr}(\text{pushr}(S)). & \quad *** \text{ type 3) } \\
\text{eq } \text{pushr}(\text{pop}(S)) = \text{pushr}(\text{popr}(S)). & \quad *** \text{ type 3) } \\
\text{eq } \text{pushr}(\text{push}(S)) = \text{pushr}(\text{pushr}(S)). & \quad *** \text{ type 3) } \\
\text{eq } \text{top}(\text{popr}(S)) = \text{top}(\text{pop}(S)). & \quad *** \text{ type 4) } \\
\text{eq } \text{top}(\text{pushr}(S)) = \text{top}(\text{push}(S)). & \quad *** \text{ type 4) } \\
\text{eq } \text{pop}(\text{popr}(S)) = \text{pop}(\text{pop}(S)). & \quad *** \text{ type 4) } \\
\text{eq } \text{pop}(\text{pushr}(S)) = \text{pop}(\text{push}(S)). & \quad *** \text{ type 4) }
\end{align*}
\]

end

Now, we can reduce (or color) some terms, say \(t_1, t_2, t_3, t_4\), to their normal forms \(\varphi(t_1)\), \(\varphi(t_2)\), \(\varphi(t_3)\), and \(\varphi(t_4)\), as follows:

\[
\begin{align*}
\text{open PAINTER .} \\
\text{op } s : \rightarrow \text{Stack}. \\
\text{red } \text{push}(\text{pop}(\text{push}(s))). & \quad \text{*** should be: push}(\text{popr}(\text{pushr}(s))) \\
\text{red } \text{pop}(\text{push}(\text{pop}(\text{push}(s)))). & \quad \text{*** should be: pop}(\text{pushr}(\text{popr}(\text{pushr}(s)))) \\
\text{red } \text{pop}(\text{pop}(\text{push}(\text{pop}(\text{push}(s))))). & \quad \text{*** should be: pop}(\text{pop}(\text{pushr}(\text{popr}(\text{pushr}(\text{pushr}(s)))))) \\
\text{red } \text{push}(\text{pop}(\text{pop}(\text{push}(\text{pop}(\text{push}(\text{push}(s))))))). & \quad \text{*** should be: push}(\text{popr}(\text{popr}(\text{pushr}(\text{popr}(\text{pushr}(\text{pushr}(s)))))))
\end{align*}
\]

close

Notice that all operations following the operation \text{push} are colored with red, so the following rewriting rule of \(\mathcal{R}'\) (also written as a BOBJ standard theory) cannot be directly applied on them:

\[
\text{th } \mathcal{R}' \text{ is protecting PAINTER .} \\
\text{eq } \text{pop}(\text{push}(S)) = S. \\
\text{end}
\]

Then the normal forms of \(g(\varphi(t_1))\), \(g(\varphi(t_2))\), \(g(\varphi(t_3))\), and \(g(\varphi(t_4))\) advocated in Theorem 85, can be obtained as follows:
open R'.

op s : -> Stack .
red g(push(popr(pushr(s)))) .

***> should be: g(push(popr(pushr(s))))
red g(pop(push(popr(pushr(s)))) .

***> should be: g(s)
red g(pop(push(popr(pushr(s)))) .

***> should be: g(s)
red g(push(popr(pushr(pushr(s)))) .

***> should be: g(push(popr(pushr(pushr(s)))))

close

Then by Theorem 85, the behavioral normal forms of $t_1$, $t_2$, $t_3$, and $t_4$ in NDSTACK are $\text{push}(\text{popr}(\text{pushr}(s)))$, $s$, $s$, and $\text{push}(\text{pop}(\text{pop}(\text{push}(\text{push}(\text{pop}(\text{push}(s)))))))$, respectively.

Notice that if one only reduces $\varphi(t_2)$ or $\varphi(t_3)$ instead of $g(\varphi(t_2))$ or $g(\varphi(t_3))$, then one gets the normal form $\text{popr}(\text{pushr}(s))$ in $\mathcal{R}'$. It is the operation $g$ on top of this term which changes the color to green and then further reduces it to $g(s)$, obtaining the real behavioral normal form, $s$. ■

There may be at least three questionable hypotheses in our method:

Q1) Is the weak left linearity of $\mathcal{R}$ really needed?

Answer: Yes. We do not know any practical example where a non weakly linear rewriting system is needed, so we will devise an artificial counter-example. Let us imagine the following behavioral rewriting system (written as a BOBJ behavioral theory):

\[
\text{bth } \mathcal{R} \text{ is protecting TRIV-VISIBLE .}
\]

\[
\text{sort } h .
\]

\[
\text{op } a : h -> v .
\]

\[
\text{op } f : h h -> v .
\]

\[
\text{op } m : h -> h \ [\text{ncong] .}
\]

\[
\text{var } x : h .
\]

\[
\text{eq } f(m(x),x) = a(x) .
\]

end

Notice that $\mathcal{R}$ is not weakly linear because $c_1 = f(m(\bullet),x)$ is not safe, $c_2 = f(m(x),\bullet)$ is safe, and $f(m(x),x) = c_1[x] = c_2[x]$. Then PAINTER$_{\Sigma,\Gamma}$ becomes (written as a BOBJ standard theory):
th PAINTER is protecting TRIV-VISIBLE.

sort h.
op a : h -> v.
op f : h h -> v.
op m : h -> h.
op mr : h -> h.
op g : h -> h.
op g : v -> v.
vars x, y : h.
eq g (mr(x)) = g ( m(x)).
eq m ( m(x)) = m (mr(x)).
eq mr( m(x)) = mr(mr(x)).
eq a (mr(x)) = a ( m(x)).
eq f(mr(x),y) = f(m(x),y).
eq f(y,mr(x)) = f(y,m(x)).
end

Finally, $\mathcal{R}'$ is (also written as a BOBJ standard theory):

th $\mathcal{R}'$ is protecting PAINTER.
eq f(m(x),x) = a(x).
end

Now, suppose that we want to reduce the term $t = f(m(m(x)),m(x))$, whose behavioral normal form in $\mathcal{R}$ is $a(m(x))$. Applying the algorithm, we get in step 2 that $\varphi(t)$ is $f(m(mr(x)),m(x))$. Moreover, $g(f(m(mr(x)),m(x)))$ cannot be reduced anymore in step 3, so the returned term in step 4 is $f(m(m(x)),m(x))$, that is, no behavioral reduction has been found. This is obviously wrong, because $f(m(m(x)),m(x)) \Rightarrow a(m(x))$.

However, we think that there may be a possibility to handle the situation of non-weakly linear behavioral rewriting systems, namely to “linearize” them into conditional rewriting systems. More precisely, considering the previous example, the behavioral rewriting rule $(\forall x) f(m(x),x) \rightarrow a(x)$ might be replaced in $\mathcal{R}'$ by the conditional rule $(\forall x) f(m(x),x') \rightarrow a(x)$ if $\psi(x) = \psi(x')$, where $\psi$ can be naturally recurrently implemented by a set of non-conditional rewriting rules (it just forgets all the colors). If we implement the translation in this way, then we claim the following result, instead of Lemma 87:

**Conjecture.** $t$ is a bnf in $\mathcal{R}$ iff $\varphi(t)$ is an normal form in $\mathcal{R}'$. 

We expect the proof to be very tedious, but if it is indeed true, than it means that the weak left linearity of $\mathcal{R}$ in 2) of Theorem 85 is not needed anymore.

Q2) *Are the operations $g : s \to s$ really needed in $\text{PAINTER}_{\Sigma, \Gamma}$?*

Answer: Not all of them and not always. More exactly, the operations $g : v \to v$ for $v$ visible are not needed at all, but we opted for introducing them just to simplify writing. If we removed them, then we would have to introduce a case analysis each time a term of the form $g(t)$ occurs, on whether the sort of $t$ is visible or not; if visible, then $g(t)$ should be replaced by just $t$. On the other hand, the operations $g : h \to h$ for $h$ hidden can be needed. A good example is the nondeterministic stack above, where the normal form in $\mathcal{R}'$ of $\varphi(t_2)$ and $\varphi(t_3)$ was $\text{popr}(\text{pushr}(s))$, from which there is no way to extract $s$, the behavioral normal form of $t_2$ and $t_3$ in $\text{NDSTACK}$.

However, if there are no rewriting rules with only variables as right-hand sides, then the operations $g$ can be deleted.

Q3) *Is the required innermost rewriting strategy necessary?*

Answer: We do not know yet. All what we know is that just plain term rewriting in $\mathcal{R}'$, without any strategy, can lead to unsound simulated behavioral rewriting in $\mathcal{R}$. Let us imagine the following artificial example (in BOBJ):

```plaintext
bth R is protecting TRIV-VISIBLE
    sort h
    op a : h \to v
    op b c d : h \to h
    op m : h h \to h     [ncong]
    var x : h
    eq b(x) = m(x,x)
    eq c(x) = d(x)
    ...
end
```

Then $b(c(x)) \Rightarrow b(d(x)) \Rightarrow m(d(x), d(x))$. On the other hand, the following is a
non-innermost valid reduction in \( \mathcal{R}' \),

\[
g(\varphi(b(c(x)))) = g(b(c(x))) \\
\xrightarrow{\varphi(\mathcal{R})} g(m(c(x), c(x))) \\
\xrightarrow{\varphi(\mathcal{R})} g(m(c(x), d(x))) \\
\xrightarrow{\Sigma, \Gamma} g(m(c_r(x), d_r(x)))
\]

with \( g(m(c_r(x), d_r(x))) \) in normal form. Since \( m \) is not in \( \Gamma \), the behavioral reduction \( b(c(x)) \xrightarrow{\ast} m(c(x), d(x)) \) may not be sound.

Notice that the unsound behavioral rewriting above took place because \( \mathcal{R}' \) was not weakly right linear. We claim the following result:

**Conjecture.** If \( \mathcal{R} \) is weakly right linear, then Theorem 85 holds for arbitrary rewriting in \( \mathcal{R}' \).

**V.D Behavioral Coinductive Rewriting**

In this section, we instantiate the general extensional rewriting technique presented in Section V.B to the framework of behavioral specifications and behavioral rewriting. A non-standard feature of behavioral rewriting relation is that it might not be compatible with all the operations, so well-known term rewriting techniques do not always apply.

Since \((T, \Rightarrow)\) is an \( \Omega \)-ARS (see Proposition 77), all the results in Section V.B are applicable. In particular, the following theorem holds:

**Theorem 89** \( \Downarrow \subseteq \equiv_{Eq, \Delta} \) and if \( \Rightarrow \) is confluent then \( \Downarrow = \equiv_{Eq, \Delta} \).

**Proof:** The first inclusion follows by Theorem 69 and Propositions 76 and 77. The rest also follows by Theorem 69, noticing that \( \equiv_{Eq, \Delta} \) is the smallest extensional congruence on \( T \) including \( \Rightarrow \).

**V.D.1 Proving Behavioral Properties Automatically**

Even if behavioral coinductive rewriting is sound for behavioral satisfaction and consists in first doing coinduction steps and then rewritings, it is not at all clear
how many coinduction steps are needed. In this section, we provide an implementable
variation of behavioral coinductive rewriting, in which coinduction is applied “by need”.
The algorithm we describe takes two terms as input and if it terminates then it returns
either YES or a goal. If it returns YES then it means that it succeeded to prove the
behavioral equivalence of the two terms; if it returns a goal, it means that it didn’t
succeed to prove the behavioral equivalence of the two terms, but it reduced it to other
behavioral equivalences. The user can then try to prove them as lemmas.

We first show the correctness of the algorithm, in the sense that if it returns
YES for a pair of terms, then the two terms are indeed behaviorally equivalent. Then
we investigate conditions under which the algorithm terminates, and finally conditions
under which it is complete for $\equiv_{E,q,\Delta}$. There is no hope to find a complete algorithm
for behavioral satisfaction in the general case (see Section VI.E).

Before we present the main algorithm, let’s note that Definition 66 suggests
the following trivial algorithm: for all $n \geq 0$, check if $t \parallel_n t'$ doing all $n$ expansions
followed by rewritings of all terms in all pairs to normal forms, then remove all pairs
of equal terms; stop if the goal becomes empty. By Theorem 89 and Proposition 76,
this algorithm yields behavioral equivalence when stops. Moreover, if $\Rightarrow$ is canonical
then it terminates for any input in $\equiv_{E,q,\Delta}$. But unfortunately, besides its inefficiency,
it doesn’t terminate if the input is not in $\equiv_{E,q,\Delta}$, in particular when the two input
terms are not behaviorally equivalent. Therefore, if $\Rightarrow$ is canonical then this algorithm
is semi-decidable for $\equiv_{E,q,\Delta}$ and there is little hope to improve it in this way.

**Assumption:** $\Delta$ is a cobasis of $B$ with $\Delta \subseteq \Gamma$.

Now, let us consider the algorithm $\mathcal{A}$ depicted in Figure V.2.

**Proposition 90** If $\mathcal{A}(t,t')$ returns YES then $t \equiv_{E,q,\Delta} t'$.

**Proof:** If $\mathcal{A}(t,t')$ returns YES, then there is some trivial goal $G_\perp$ such that $[(t,t')] \Rightarrow_r$
$(\Rightarrow_{e}; \Rightarrow_r)^* G_\perp$. By 3 and 1 in Proposition 65, $\Rightarrow_r; (\Rightarrow_{e}; \Rightarrow_r)^* \subseteq \Rightarrow_{E,q,\Delta}, \Rightarrow_r$, so by 5 in
Proposition 65 and Definition 66, $t \parallel t'$. Then by Theorem 89, $t \equiv_{E,q,\Delta} t'$.

Therefore $\mathcal{A}$ is sound. The rest of this subsection is dedicated to syntactic
easy to check criteria for termination and completeness w.r.t. $\equiv_{E,q,\Delta}$ of $\mathcal{A}$, respectively.
**Algorithm $A(t, t')$**

1. initialize $G$ with $[(t, t')]$
2. rewrite all terms in all pairs in $G$ to normal forms (relation $⇒_r$)
3. remove from $G$ all pairs containing equal terms
4. if $(G = \emptyset)$ then return YES
5. if $(G$ has no hidden term rooted in $\Sigma - \Delta)$ then return $G$
6. expand a pair in $G$ containing a hidden term rooted in $\Sigma - \Delta$ (relation $⇒_e$)
7. goto 2.

Figure V.2: Behavioral coinductive rewriting algorithm.

These should be regarded only as a starting point toward stronger criteria; for now, there isn’t strong motivation to explore them more deeply.

**Proposition 91 Termination Criterion:** If $⇒$ terminates and each rewriting rule $(\forall X)\ l → r$ has the property that $r$ is a $\Delta$-term and $l$ is not a term $\delta(W)$ with $\delta \in \Delta$, then $A$ terminates.

**Proof:** We need the following

**Lemma 92** In the same context, if $s$ is a normal form and $\delta(W', s) ⇒^+ u$ for some $\delta \in \Delta$ and appropriate $W'$, then each subterm of $u$ rooted in $\Sigma - \Delta$ is a proper subterm of $s$.

**Proof:** The proof can be done by induction on the length of the derivation. If $\delta(W', s) ⇒ u$ then the only position where the rewriting can be done is at the top. Since the right hand side of every rewrite rule contains only operations in $\Delta$, every subterm of $u$ rooted in $\Sigma - \Delta$ is actually a subterm of $s$. On the other hand, $s$ cannot be a subterm of $u$ because there is no rewriting rule having $\delta(W)$ as its left hand side term.

Now, suppose that $\delta(W', s) ⇒^+ u$, each subterm of $u$ rooted in $\Sigma - \Delta$ is a proper subterm of $s$, and that $u ⇒ u'$. Since $s$ is a normal form and each subterm of $u$ rooted in $\Sigma - \Delta$ is a proper subterm of $s$, the position of $u$ where the rewriting to $u'$ is applied cannot be inside a subterm of $u$ rooted in $\Sigma - \Delta$. Therefore, there are only operations in $\Delta$ on the path from that position to the root of $u$. Since the rewriting rule applied has only operations in $\Delta$ in the right hand side, it is easy to observe that all subterms of $u'$ rooted in $\Sigma - \Delta$ are subterms of $u$. Thus, each subterm of $u'$ rooted in $\Sigma - \Delta$ is a proper subterm of $s$. □
Given a pair of terms $(t, t')$, $A(t, t')$ terminates either when $G = \emptyset$ (step 4) or when both terms in any pair in $G$ are distinct normal forms that have operations in $\Delta$ as roots (step 5). Suppose that $A(t, t')$ does not terminate. Then for each $n \geq 1$, there is a sequence $(t_1, t'_1), (s_1, s'_1), ..., (t_n, t'_n), (s_n, s'_n)$ such that $(t_1, t'_1) = (t, t')$, $(t_{i+1}, t'_{i+1}) = (\delta_i(W_i, s_i), \delta_i(W_i, s'_i))$ for some $\delta_i \in \Delta$, and $s_i$ and $s'_i$ are normal forms of $t_i$ and $t'_i$ such that at least one of $s_i$ and $s'_i$ is rooted in $\Sigma - \Delta$. By Lemma 92, there is an infinite sequence of normal forms rooted in $\Sigma - \Delta$, say $s_{i_1}, s_{i_2}, ...$, such that $s_{i_j+1}$ is a proper subterm of $s_{i_j}$ for all natural numbers $j$. Since any term has only a finite number of subterms, we deduce that $A(t, t')$ terminates. $\square$

The next proposition provides a completeness criterion for $A$ with respect to $\equiv_{\Delta}$.

**Proposition 93 Completeness Criterion:** Let $R$ be a behavioral rewriting system verifying the hypotheses in Proposition 91. If in addition,

- $\Rightarrow$ is confluent,
- if $\delta(W, \delta'(t_1, ..., t_n))$ is a left hand side of a rewriting rule with $\delta, \delta' \in \Delta$, then $t_1, ..., t_n$ are all variables,
- for each $\delta, \delta' \in \Delta$ (not necessarily distinct) there is some $\delta_1 \in \Delta$ such that neither $\delta_1(W_1, \delta(W))$ nor $\delta_1(W_1, \delta'(W'))$ appears as left hand side in any rewriting rule,

then $t \equiv_{\Delta} t'$ iff $A(t, t')$ returns YES.

**Proof:** By Proposition 90, if $A(t, t')$ returns YES then $t \equiv_{\Delta} t'$. $A$ terminates by Proposition 91. If $t \equiv_{\Delta} t'$ then by Theorem 89, $t \parallel t'$. Suppose that $A(t, t')$ does not return YES, that is, the returned goal contains some pairs of distinct normal forms $(s, s')$ such that both $s$ and $s'$ are rooted in $\Delta$. More precisely, there is a sequence $(t_1, t'_1), (s_1, s'_1), ..., (t_n, t'_n), (s_n, s'_n)$ such that $(t_1, t'_1) = (t, t')$ and $(s_n, s'_n) = (s, s')$, $s_i$ and $s'_i$ are normal forms of $t_i$ and $t'_i$, respectively, and for each $i = 0, ..., n-1$, $(t_{i+1}, t'_{i+1}) = (\delta_i(W_i, s_i), \delta_i(W_i, s'_i))$ for some $\delta_i \in \Delta$. Since $\Rightarrow$ is confluent, $\parallel$ is transitive (see 3 in Lemma 68), i.e., $\parallel; \parallel = \parallel$, and since $\Rightarrow$ is included in $\parallel$, one can easily deduce that $s_1 \parallel s'_1$; by the compatibility of $\parallel$ with the operations in $\Delta$ (see Theorem 69), $t_2 \parallel t'_2$. Then it can be easily obtained by induction that $s \parallel s'$; suppose that $s \parallel_n s'$. Let $\delta$ and $\delta'$ be the roots of $s$ and $s'$, respectively, and let $\delta_1$ be
a behavioral operation in $\Delta$ such that $\delta_1(W_1, \delta(W))$ and $\delta_1(W_1, \delta'(W'))$ do not appear as left hand side terms of any rewriting rule in $\mathcal{R}$. Since $s$ and $s'$ are normal forms and there is no rewriting rule with $\delta_1(W_1, \delta(t_1, \ldots, t_n))$ or $\delta_1(W_1, \delta'(t'_1, \ldots, t'_n))$ as left hand side, $\delta_1(W_1, s)$ and $\delta_1(W_1, s')$ are also normal forms. By Lemma 70, there is some $n_1 < n$ such that $\delta_1(W_1, s) \downharpoonleft_{n_1} \delta_1(W_1, s')$. Similarly, there is some $\delta_2 \in \Delta$ and $n_2 < n_1$ such that $\delta_2(W_2, \delta_1(W_1, s))$ and $\delta_2(W_2, \delta_1(W_1, s'))$ are normal forms and $\delta_2(W_2, \delta_1(W_1, s)) \downharpoonleft_{n_2} \delta_2(W_2, \delta_1(W_1, s'))$. Iterating this method, there are some $\delta_1, \ldots, \delta_k \in \Delta$ such that $\gamma_k$ and $\gamma_k'$ are normal forms and $\gamma_k \downharpoonleft_0 \gamma_k'$, where $\gamma_k = \delta_k(W_k, \ldots, \delta_1(W_1, s))$ and $\gamma_k' = \delta_k(W_k, \ldots, \delta_1(W_1, s'))$. Since $\gamma_k$ and $\gamma_k'$ are already normal forms, one gets that $\gamma_k = \gamma_k'$, that is, $s = s'$. This is a contradiction, because $s$ and $s'$ were supposed to be distinct. Consequently, $A(t, t')$ returns YES.

\[ \square \]

V.D.2 Other example: Extensible Lambda Calculus

Readers familiar with $\lambda$-calculus have probably felt or already seen a striking relationship between the rule of extensionality in $\lambda$-calculus and our extensional relations. The purpose of this subsection is not to solve hard problems of $\lambda$-calculus and type theory, or to come up with new unexpected techniques for reduction in this theories, but only to show that extensional rewriting can go beyond behavioral reasoning as we understand it today, and to suggest that, perhaps, part of the work in $\lambda$-calculus can be done at more abstract levels, such as $\Omega$-ARSs, for example.

We assume the reader familiar with basics of $\lambda$-calculus [5, 93] abbreviated $\lambda$; we just remind the reader some basic notions and results of $\lambda$ and combinators, and our notations for them. To keep the presentation simple, we only analyze untyped $\lambda$-calculus, hoping to extend the results and find deeper relationships with typed $\lambda$-calculus in subsequent research.

Combinatorial Logic, abbreviated CL, is the equational theory defined by the following axioms\^{11}:

\[
\text{th CL is sorts Var Exp . subscript Var < Exp .}
\]

\[
\text{ops S K : -> Exp .}
\]

\^{11}We use BOBJ notation in the following examples, but the code was actually executed using OBJ3 because some of the featured used have not been implemented in BOBJ yet.
Therefore, there is only one type Exp for all expressions, there are two constants \( S \) and \( K \), called \textit{combinators}, and an application operation “\( \_\_ \)” which is left associative\(^{12}\) verifying the two standard equations. The rewriting engine automatically interprets the two equations as rewriting rules, so the corresponding rewriting relation is the so called \textit{weak reduction}, denoted \( \rightarrow_w \), which is known to be confluent (the two rule rewriting system is orthogonal).

There are canonical mappings from \( \text{CL} \) to \( \lambda \) and vice versa, but the two theories are not entirely equivalent. For example, \( \lambda \vdash (S \ K \ K = S \ K \ S) \) but there is no way to show that \( \text{CL} \vdash (S \ K \ K = S \ K \ S) \). However, it is well-known that in presence of the rule of \textit{extensionality}

\[
\text{ext: } \quad \frac{Px = Qx, \text{ for some variable } x \text{ that is not free in } PQ}{P = Q}
\]

Figure V.3: Extensionality rule.

the two theories become equivalent to \( \lambda \) plus \( \eta \)-reduction:

\textbf{Theorem 94} \textit{The following theories:}

\[
\text{CL} + \text{ext, } \quad \lambda + \text{ext, } \quad \lambda + \eta
\]

are all equivalent.

By a mechanism similar to the one proposed in Section V.C (see Proposition 75), we can see both the \( \text{CL} \)-terms and the \( \lambda \)-terms as \( \Omega \)-algebras, where \( \Omega \) contains only one operation, \( \omega \), associated to application, defined as \( \omega(P) = Px \), for an arbitrary but fixed variable \( x \) which is not free in \( P \) but depends exactly on the free variables of \( P \). Let \( T_{\text{CL}} \) and \( T_\lambda \) be the two \( \Omega \)-algebras. Then

\(^{12}\)The attributes \texttt{[gather (E e)]} and \texttt{[prec 10]} stand for left associativity and precedence 10, and are needed for parser to disambiguate expressions which are not fully parenthesized.
Proposition 95 \((T_{\text{CL}}, \rightarrow_w)\) and \((T_{\lambda}, \rightarrow_{\beta})\) are \(\Omega\)-ARSs.

Since \(\rightarrow_w\) and \(\rightarrow_{\beta}\) are both confluent, and \(\equiv_{\text{CL}+\text{ext}}\) and \(\equiv_{\alpha\beta\eta}\) are the smallest extensional congruences on \(T_{\text{CL}}\) and \(T_{\lambda}\) including \(\rightarrow_w\) and \(\rightarrow_{\beta}\), respectively, if \(\Rightarrow_{\text{CL}}, \Downarrow_{CL}\) and \(\Rightarrow_{\lambda}, \Downarrow_{\lambda}\) are the extensional rewriting relations associated, then by 15 in Proposition 65 and Theorem 69,

Proposition 96 \(\Rightarrow_{\text{CL}}\) and \(\Rightarrow_{\lambda}\) are confluent, and \(\Downarrow_{\text{CL}} = \equiv_{\text{CL}+\text{ext}}\) and \(\Downarrow_{\lambda} = \equiv_{\alpha\beta\eta}\).

Therefore, \(\Rightarrow_{\text{CL}}\) and \(\Rightarrow_{\lambda}\) are complete for extensional \(\text{CL}\) and \(\lambda\), respectively.

Since \(\Omega\) has only one operation, all goals contain only one pair, and \([[s, t]] \Rightarrow_e [[s', t']]\) iff there are some distinct variables \(x_1, x_2, ..., x_n\) which are not free in \(st\) such that \([[s, t]] \Rightarrow^* e\) \([[sx_1x_2...x_n, tx_1x_2...x_n]]\) and \(sx_1x_2...x_n \rightarrow s'\) and \(tx_1x_2...x_n \rightarrow t'\), where \(\Rightarrow_e\) and \(\rightarrow\) can be either \(\Rightarrow_{\text{CL}}\) and \(\rightarrow_w\) or \(\Rightarrow_{\lambda}\) and \(\rightarrow_{\beta}\).

We end this section with two simple examples showing extensional rewriting at work. The first example just shows how to prove a simple equality in \(\text{CL} + \text{ext}\). It is known that \(\text{CL} + \text{ext}\) is actually equivalent to an equational theory which finitely extends \(\text{CL}\), in the sense that there is a finite set of equations \(A_{\beta\eta}\), such that \(\text{CL} + \text{ext}\) is equivalent to \(\text{CL} + A_{\beta\eta}\); \(A_{\beta\eta}\) can be the following four equations:

\[
\begin{align*}
\text{eq } S(S(K S)(S(K K)(S(K S)K)))(K K) &= S(K K) . \\
\text{eq } S(K S)(S(K K)) &= S(K K)(S(S(K S)(S(K K)(S(K K)))))(K S K K) . \\
\text{eq } S(K(S(K S)))(S(K S))(S(K S))) &= S(S(K S)(S(K K)(S(K S)(S(K S)))))(K S) . \\
\text{eq } S(K S K K) &= S(K S) K K . 
\end{align*}
\]

The difficult part is of course to show that these equations imply the soundness of extensionality, but this is beyond the goal of this paper. We only show by extensional rewriting that \(\text{CL} + \text{ext}\) satisfies the third equation, that is, that

\[
S(K(S(K S)))(S(K S)(S(K S)))) \Downarrow_{\text{CL}} S(S(K S)(S(K K)(S(K S)(S(K S)(S(K S))))))(K S).
\]

It can be automatically obtained that the two terms are not in the relations \(\Downarrow_{\text{CL},0}\), \(\Downarrow_{\text{CL},1}\), \(\Downarrow_{\text{CL},2}\) and \(\Downarrow_{\text{CL},3}\), but that they are in the relation \(\Downarrow_{\text{CL},4}\). The following is a proof score for the last two cases:
open.
ops x y z u : -> Var.

*** case 3
red S (K (S (K S))) (S (K S) (S (K S))) x y z.
red S (S (K S) (S (K K) (S (K S) (S (K S) (S (K S) S)))) (K S) x y z.

*** case 4
red S (K (S (K S))) (S (K S) (S (K S))) x y z u.
red S (S (K S) (S (K K) (S (K S) (S (K S) (S (K S) S)))) (K S) x y z u.
close

with the output:

reduce in CL: ((((S(K(S(K S))))((S(K S))(S(K S))))x)y)z
rewrites: 6
result Exp: (S((S((S(K S))x))y))z
==========================================
reduce in CL: ((((S((S(K S))((S(K K)) ... S(K S))))S))))(K S))x)y)z
rewrites: 15
result Exp: (S((S x)z)((S y)z)
==========================================
reduce in CL: ((((((S(K(S(K S))))((S(K S))(S(K S))))x)y)z)u
rewrites: 11
result Exp: ((x u)(z u))((y u)(z u))
==========================================
reduce in CL: ((((((S((S(K S))((S(K K)) ... )))S))))(K S))x)y)z)u
rewrites: 18
result Exp: ((x u)(z u))((y u)(z u))

Therefore, the pair formed of the two terms can be extensionally reduced in 4 extensional steps to a trivial goal, so by Proposition 96, they are equal in CL + ext. Notice that the proof score above can be automated.

The next example is concerned with Church numerals. Since it is a bit cumbersome to specify λ-calculus in BOBJ, we prefer to first map it to CL, and then do the proofs in CL + ext by extensional rewriting. By Theorem 94, this procedure is complete for λ + ext. A canonical mapping of λ-expressions to combinator expressions is given by the following algorithm for bracket abstraction:

th BRACKET-ABS is protecting CL.
sorts Var Fun. subsorts Var Fun < Exp.
op `_` : Var Exp -> Exp [ prec 20 ].
op `[_]` : Var Exp -> Exp.
vars X Y : Var. vars M N : Exp.
eq / X . M = [ X ] M.
The λ-expression 13 \( \lambda n m x . n(mx) \), for example, can be reduced as follows:

\[
\text{reduce in BRACKET-ABS : } / \ n . (/ \ m . (/ \ x . n(mx)))
\]

rewrites: 90

result Exp:\( (S((S(K S))((S((S(K S))((S(K K))(K S)))))(((S(S(K S))\((S((S(K K))((S(K K))(K S)))))(((S(S(K S))((S(S(K K))))((S(S(K S))((S(K K))((S(K K))((S(S(K S))((S(S(K S))((S(S(K S))((S(K K))(K S)))(S(S(K S))((S(S(K S))((S(S(K K))(K K))((S(S(K S))((S(S(K S))((S(S(K S))((S(K K))(K K)))))\right)\)

Natural numbers can be now defined as follows:

\[
\text{th LNAT is protecting BRACKET-ABS .}
\]

ops x y n m : -> Var.
ops 0 s + * : -> Exp.
ops 1 2 3 4 5 6 7 8 9 : -> Exp.

\[
eq 0 = (/ \ x . / \ y . y).
\]

\[
eq s = (/ \ n . / \ x . / \ y . x(n x y)).
\]

\[
eq + = (/ \ n . / \ m . / \ x . / \ y . n x(m x y)).
\]

\[
eq * = (/ \ n . / \ m . / \ x . n(m x)).
\]

\[
eq 1 = s \ 0 . \ eq 2 = s \ 1 . \ eq 3 = s \ 2 . \ eq 4 = s \ 3 . \ eq 5 = s \ 4 .
\]

\[
eq 6 = s \ 5 . \ eq 7 = s \ 6 . \ eq 8 = s \ 7 . \ eq 9 = s \ 8 .
\]

end

Our goal is to prove by extensional rewriting that, for example, \( (+ 4 5) = (+ (* 2 3) (* 2 1)) \). Notice that this kind of equalities can be easily proved in λ using only \( \alpha \beta \)-reduction, so extensionality is not needed, and that they can be also proved in CL, without extensionality, for proper combinator representations of numerals. It is not surprising that the combinator representation of the λ-numerals above is not good enough to allow us to prove simple equalities without extensionality; this is because \( \rightarrow_w \) is weaker than \( \beta \)-reduction. However, as we mentioned at the beginning of this section, our purpose is just to show extensional rewriting at work within a framework different from that of behavioral specifications:

\[
\text{open LNAT .}
\]

13 Which represents the Curry multiplication.
ops u v : -> Var .
red (+ 4 5)  == (+ (* 2 3) (+ 2 1)) .
red (+ 4 5) u  == (+ (* 2 3) (+ 2 1)) u .
red (+ 4 5) u v  == (+ (* 2 3) (+ 2 1)) u v .
close

The output of these reductions is:

reduce in LNAT : (+ 4) 5 == (+ ((* 2) 3)) ((+ 2) 1)
rewrites: 5999
result Bool: false
=================================================================
reduce in LNAT : ((+ 4) 5) u == ((+ ((* 2) 3)) ((+ 2) 1)) u
rewrites: 6061
result Bool: false
=================================================================
reduce in LNAT : (((+ 4) 5) u) v == (((+ ((* 2) 3)) ((+ 2) 1)) u) v
rewrites: 6812
result Bool: true

An interesting area of further research would be to efficiently automate extensional rewriting, which means, first to quickly find how many extensional steps are needed and then to apply the reductions efficiently.

V.E Circular Coinductive Rewriting

Circular coinductive rewriting is an algorithm for proving behavioral equalities which was introduced in [60]. It combines behavioral rewriting presented in Section V.C and circular Δ-coinduction presented in Section V.E. We give examples showing that this algorithm is surprisingly powerful in practice, even if we know that no such algorithm can be complete (see Section VI.E). Of course, incompleteness is more the rule than the exception for non-trivial theorem proving problem classes.

This section contains many examples reflecting the method, including: several equivalences of streams of values, such as might arise in verifying lazy functional programs; a behavioral refinement example, that stack can be implemented with a pointer into an array; and the equivalence of regular languages.

The specifications STREAM and ZIP presented in the Examples 31 and 33 are much used in our examples and should be considered to have been pre-loaded. BOBJ’s cobasis algorithm discovers that & and tt zip can be removed from the the cobasis:
The cobasis for sort Stream is:

\begin{align*}
\text{op head} : \text{Stream} & \rightarrow \text{Elt} \\
\text{op tail} : \text{Stream} & \rightarrow \text{Stream}
\end{align*}

V.E.1 Circular Coinductive Rewriting in BOBJ

The circular coinductive rewriting algorithm has as input a pair of terms, and returns true whenever it can prove the terms behaviorally equivalent, and otherwise returns false or fails to terminate, much as with proving term equality by rewriting. The detailed correctness proof is difficult and will appear later; here we describe the BOBJ implementation.

Given a behavioral specification $B = (\Sigma, \Gamma, \mathcal{E})$ with a cobasis $\Delta \subseteq \Gamma$, a set of pairs of terms$^{14}$ $\mathcal{C}$, and a $\Sigma$-term $s$, let $\text{bnf}_C(s)$ be the term derived from $s$ by rewriting with the equations in $\mathcal{E} \cup \mathcal{C}$ under the usual restrictions for behavioral rewriting [134] with $\mathcal{E}$, and then applying the equations in $\mathcal{C}$ at a term position if all (zero or more) operations on the path to that position are in $\Gamma - \Delta$.

Given a pair of $\Sigma$-terms $(t, t')$, the circular coinductive rewriting algorithm, hereafter denoted CCRW, can be described as follows:

1. let $\mathcal{C} = \emptyset$ and $\mathcal{G} = \{(t, t')\}$
2. for each $(s, s')$ in $\mathcal{G}$
3. move $(s, s')$ from $\mathcal{G}$ to $\mathcal{C}$
4. for each $\delta \in \Delta$
5. let $u = \text{bnf}_C(\delta[s, x])$ and $u' = \text{bnf}_C(\delta[s', x])$
6. if $u \neq u'$ then add $(u, u')$ to $\mathcal{G}$

Figure V.4: Circular coinductive rewriting algorithm.

$\mathcal{G}$ contains the still unproved goals and $\mathcal{C}$ contains the “circularities” to be used in proofs. By definition of cobasis, to prove $t \equiv t'$, it suffices to prove $\delta[t, x] \equiv \delta[t', x]$ for all $\delta \in \Delta$ and all appropriate $x$. Note that an equation $(t, t')$ in $\mathcal{C}$ can be used in the special way described above in proving an equation $(\delta[t, x], \delta[t', x])$, which is then used in proving $(t, t')$; this explains the word “circular.” Note also that the algorithm may fail $^{14}$BOBJ may reorder these to reduce the possibility of non-termination.
to terminate. Justification for the choice and use of equations in \( C \) is rather technical, and must be omitted here.

V.E.2 Lazy Functional Programming Examples

We give some simple examples from lazy functional programming. Many similar examples were done by Louise Dennis using a system called CoClam with a complex heuristic planning algorithm [35]; all her examples that we tried can be done algorithmically by CCRW without human intervention or machine heuristics. We thank Wolfram Schulte for getting us started in this direction.

**Example 97** We define a function \( \text{rev} \) taking an infinite stream of boolean values and returning a stream where each value is reversed:

\[
\text{bth REV is pr STREAM[BOOL] .}
\]
\[
\text{op rev : Stream -> Stream .}
\]
\[
\text{var S : Stream .}
\]
\[
\text{eq head(rev(S)) = not head(S) .}
\]
\[
\text{eq tail(rev(S)) = rev(tail(S)) .}
\]
\end

We show that \( \text{rev(rev}(S)) \) is behaviorally equivalent to \( S \):

\[
\text{BOBJ> set trace on}
\]
\[
\text{BOBJ> cred rev(rev(S)) == S .}
\]
\[
\text{c-reduce in REV : rev(rev(S)) == S}
\]
\[
\text{using cobasis for REV:}
\]
\[
\text{op head : Stream -> Bool}
\]
\[
\text{op tail : Stream -> Stream}
\]
\[
\text{reduced to: rev(rev(S)) == S}
\]
\[
\text{add rule (C1) : rev(rev(S)) = S}
\]
\[
\text{target is: rev(rev(S)) == S}
\]
\[
\text{expand by: op head : Stream -> Bool}
\]
\[
\text{reduced to: true}
\]
\[
\text{nf: head(S)}
\]
\[
\text{target is: rev(rev(S)) == S}
\]
\[
\text{expand by: op tail : Stream -> Stream}
\]
deduced using (C1) : true 
    nf: tail(S) 
-----------------------------------------
result: true
c-rewrite time: 82ms   parse time: 6ms

The first command tells BOBJ to display some high level information about the execution of CCRW; this often helps find errors in the spec, and to suggest needed lemmas. BOBJ’s cobasis algorithm discovers that rev is congruent, so the cobasis is just \{head, tail\}. These operations are applied to the two terms, giving the subgoals head(rev(rev(S))) == head(S) and tail(rev(rev(S))) == tail(S). The first follows directly by rewriting (the built in module BOOL contains the fact \text{not not} B == B for any boolean B). The second subgoal needs the circularity, since it reduces to rev(rev(tail(S))) == tail(S), which is an instance of its initial goal, with S replaced by tail(S). BOBJ reports circularity by displaying the keyword “deduced” instead of “reduced” with the normal form tail(S) of rev(rev(tail(S))). ■

Example 98 We define infinite streams zero, one, and blink, containing only 0’s, only 1’s, and alternations of 0 and 1, as follows:

\begin{verbatim}
bth BLINK is pr ZIP[NAT] .
    ops zero one blink : -> Stream .
    eq head(zero) = 0 .   eq tail(zero) = zero .
    eq head(one) = 1 .   eq tail(one) = one .
    eq head(blink) = 0 .   eq head(tail(blink)) = 1 .
    eq tail(tail(blink)) = blink .
end
\end{verbatim}

BLINK imports ZIP instantiated with the built in module NAT of natural numbers. The property that blink = zip(zero, one) is proved by circular coinduction as follows:

BOBJ> cred blink == zip(zero, one) .

producing the output

\begin{verbatim}
c-reduce in BLINK : blink == zip(zero , one) 
using cobasis for BLINK:
    op head : Stream -> Nat
    op tail : Stream -> Stream
---------------------------------------
\end{verbatim}
reduced to: blink == zip(zero , one)
-----------------------------------------
add rule (C1) : blink = zip(zero , one)
-----------------------------------------
target is: blink == zip(zero , one)
extend by: op head : Stream -> Nat
reduced to: true
nf: 0
-----------------------------------------
target is: blink == zip(zero , one)
extend by: op tail : Stream -> Stream
reduced to: tail(blink) == zip(one , zero)
-----------------------------------------
add rule (C2) : tail(blink) = zip(one , zero)
-----------------------------------------
target is: tail(blink) == zip(one , zero)
extend by: op head : Stream -> Nat
reduced to: true
nf: 1
-----------------------------------------
target is: tail(blink) == zip(one , zero)
extend by: op tail : Stream -> Stream
deduced using (C1) : true
nf: zip(zero , one)
-----------------------------------------
result: true

The cobasis algorithm found \{head, tail\} for BLINK, and the next four steps used this
for circular coinductive rewriting. First head is applied to the two streams, giving 0 in
each case. Then tail is applied, giving the new goal tail(blink) == zip(one, zero).
Again head is applied to the two new terms, giving 1, and then tail is applied, giving
a circularity, namely the subgoal blink == zip(zero, one), so the result is true. ■

Example 99 One obvious way to define the stream of natural numbers, that is, 0 1 2
3 4 5 ... is to define a function nat by nat(N) = N & nat(N + 1) for all N, and then
consider nat(0). Less obvious is to define a function succ incrementing all elements
in a stream by succ(S) = (head(S) + 1) & succ(tail(S)), and then define nat’ by
nat’(N) = N & succ(nat'(N)).

bth NAT-STREAM is pr STREAM[NAT] .
op nat : Nat -> Stream .
var N : Nat . var S : Stream .
eq head(nat(N)) = N.
eq tail(nat(N)) = nat(N+1).
op succ : Stream -> Stream.
eq head(succ(S)) = head(S) + 1.
eq tail(succ(S)) = succ(tail(S)).
op nat' : Nat -> Stream.
eq head(nat'(N)) = N.
eq tail(nat'(N)) = succ(nat'(N)).
end

We show these definitions equivalent, i.e., \( \text{n}(0) = \text{n}'(0) \), by proving the more general result \( \text{n}(N) = \text{n}'(N) \) for all \( N \); we omit some BOBJ output.

BOBJ> cred nat(N) == nat'(N).
==========================================
c-reduce in NAT-STREAM : nat(N) == nat'(N)
reduced to: nat(N) == nat'(N)
=========================================
add rule (C1) : nat(N) = nat'(N)
-----------------------------------------
.....
add rule (C2) : nat'(1 + N) = succ(nat'(N))
-----------------------------------------
.....
result: true

The new rule (C1) was generated and used for (C2), and (C2) for the final subgoal.

Example 100 We show that two definitions of the stream of Fibonacci numbers are equivalent.

dth FIBO-NAT is ex NAT.
var N : Nat.
op f : Nat -> Nat.
eq f(0) = 0.
eq f(1) = 1.
eq f(N + 2) = f(N + 1) + f(N).
end

bth FIBO-STREAM is pr ZIP[FIBO-NAT].
var N : Nat. var S : Stream.
op nat : Nat -> Stream.
eq head(nat(N)) = N.
eq tail(nat(N)) = nat(N + 1).
op f : Stream -> Stream.
eq head(f(S)) = f(head(S)).
eq tail(f(S)) = f(tail(S)) .
op fib : Nat -> Stream .
eq fib(N) = f(nat(N)) .
op add : Stream -> Stream .
eq head(add(S)) = head(S) + head(tail(S)) .
eq tail(add(S)) = add(tail(tail(S))) .
op fib' : Nat -> Stream .
eq head(fib'(N)) = f(N) .
eq head(tail(fib'(N))) = f(N + 1) .
eq tail(tail(fib'(N))) = add(zip(fib'(N), tail(fib'(N)))) .
end

Here is the BOBJ output:

BOBJ> cred fib(N) == fib'(N) .
==========================================
c-reduce in FIBO-STREAM : fib(N) == fib'(N)
using cobasis for FIBO-STREAM:
opt head : Stream -> NzNat
opty tail : Stream -> Stream
=========================================
reduced to: f(nat(N)) == fib'(N)
=========================================
add rule (C1) : f(nat(N)) = fib'(N)
=========================================
target is: f(nat(N)) == fib'(N)
expand by: opty head : Stream -> NzNat
reduced to: true
   nf: f(N)
=========================================
target is: f(nat(N)) == fib'(N)
expand by: op tail : Stream -> Stream
deduced using (C1) : fib'(1 + N) == tail(fib'(N))
=========================================
add rule (C2) : fib'(1 + N) = tail(fib'(N))
=========================================
target is: fib'(1 + N) == tail(fib'(N))
expand by: op head : Stream -> NzNat
reduced to: true
   nf: f(1 + N)
=========================================
target is: fib'(1 + N) == tail(fib'(N))
expand by: op tail : Stream -> Stream
reduced to: tail(fib'(1 + N)) == add(zip(fib'(N), tail(fib'(N))))
=========================================
add rule (C3) : tail(fib'(1+N)) = add(zip(fib'(N),tail(fib'(N))))
Example 101  Related to the previous examples of streams, we show that two behavioral specifications of stream are equivalent. First we declare their common signature:

\[
\text{bth SIGMA}[X :: \text{TRIV}] \text{ is sort Stream .}
\]

\[
\begin{align*}
\text{op head} : \text{Stream} & \rightarrow \text{Elt} . \\
\text{op tail} & : \text{Stream} \rightarrow \text{Stream} . \\
\text{op \_&_} & : \text{Elt Stream} \rightarrow \text{Stream} . \\
\text{op odd} & : \text{Stream} \rightarrow \text{Stream} . \\
\text{op even} & : \text{Stream} \rightarrow \text{Stream} . \\
\text{op zip} & : \text{Stream Stream} \rightarrow \text{Stream} .
\end{align*}
\]

end

Notice that all operations are behavioral, because they preserve the intended behavioral equivalence, which is “two streams are equivalent iff they have the same elements in the same order.”

\[
\text{bth STREAM1}[X :: \text{TRIV}] \text{ is using SIGMA}[X] .
\]

\[
\begin{align*}
\text{var E} & : \text{Elt} . \text{ var S S'} : \text{Stream} . \\
\text{eq head}(E \ & S) & = E . \\
\text{eq tail}(E \ & S) & = S . \\
\text{eq head}(\text{odd}(S)) & = \text{head}(S) . \\
\text{eq tail}(\text{odd}(S)) & = \text{even}(\text{tail}(S)) . \\
\text{eq head}(\text{even}(S)) & = \text{head}(\text{tail}(S)) . \\
\text{eq tail}(\text{even}(S)) & = \text{even}(\text{tail}(\text{tail}(S))) . \\
\text{eq head}(\text{zip}(S, S')) & = \text{head}(S) . \\
\text{eq tail}(\text{zip}(S, S')) & = \text{zip}(S', \text{tail}(S)) .
\end{align*}
\]

end

As we have seen in Subsection IV.E.1, this has the expected cobasis \{head, tail\}, but given any model, there is another interesting way to generate its behavioral equivalence,
namely using observations built with head, odd and even. The following specification is in this spirit:

```plaintext
bth STREAM2[X :: TRIV] is using SIGMA[X] .
  var E : Elt . var S S' : Stream .
  eq head(tail(S)) = head(even(S)) .
  eq odd(tail(S)) = even(S) .
  eq even(tail(S)) = tail(odd(S)) .
  eq head(E & S) = E .
  eq odd(E & S) = E & even(S) .
  eq even(E & S) = odd(S) .
  eq head(zip(S, S')) = head(S) .
  eq odd(zip(S, S')) = S .
  eq even(zip(S, S')) = S' .
  eq head(odd(S)) = head(S) .
end
```

BOBJ finds the cobasis \{head, odd, even\}, and also easily proves the two specifications behaviorally equivalent, in the sense of having the same models with the same behavioral equivalences [137, 71], by proving all equalities in each spec from those in the other (see also [139]). We encourage the reader investigate why the last equation in STREAM2 is needed (hint: without it there could be models of STREAM2 that are not models of STREAM1). Properties like zip(odd(S), even(S)) = S have much easier proofs by \{head, odd, even\}-coinduction. On the other hand, the equality head(S) & tail(S) == S is easy to prove with CCRW over \{head, tail\} in STREAM1, but cannot be proved with CCRW over \{head, odd, even\} in STREAM2. Thus neither cobasis is the best for all purposes, and so proving their equivalence has a practical value.

V.E.3 Behavioral Refinement

Some researchers have complained that stack examples have been overused, but they provide a very useful benchmark just because they have been used in so many different settings, and they often reveal something new and interesting. Here, we illustrate our approach to behavioral refinement with the familiar implementation of stack as a pointer into an array, with the indicated array element as the top, the pointer incremented and the element inserted into the array for push, and the pointer decremented when for pop.
bth STACK[X :: TRIV] is sort Stack .
    op top_ : Stack -> Elt .
    op pop_ : Stack -> Stack .
    op push : Elt Stack -> Stack .
    op empty : -> Stack .
    var E : Elt . var S : Stack .
    eq top push(E,S) = E .
    eq pop push(E,S) = S .
end

BOBJ find the cobasis \{top, pop\}. We have left top and pop undefined on empty to allow a larger class of models. The stack operations are implemented on pointer array pairs as expected:

bth ARRAY[X :: TRIV] is sort Arr . pr NAT .
    op nil : -> Arr .
    op put : Elt Nat Arr -> Arr .
    op \_[\_] : Arr Nat -> Elt .
    vars I J : Nat . var A : Arr . var E : Elt .
    eq nil [I] = 0 .
    cq put(E, I, A) [J] = E if eq(I, J) .
end

bth STACKIMP[X :: TRIV] is
    pr (NAT || ARRAY[X]) * (sort Tuple to Stack) .
    op top_ : Stack -> Elt .
    op pop_ : Stack -> Stack .
    op push : Elt Stack -> Stack .
    op empty : -> Stack .
    vars I J : Nat . var A : Arr . var E : Elt .
    eq empty = <0, nil>.
    eq push(E, <I, A>) = <s I, put(E, I, A)> .
    eq top <s I, A> = A [I] .
    eq top <0, A> = 0 .
    eq pop <s I, A> = <I, A> .
    eq pop <0, A> = <0, A> .
end

The operation eq used in ARRAY belongs to NAT, and returns true when its arguments can be proved equal, false when they can be proved unequal, and a normal form when BOBJ cannot prove them equal or unequal. The operation "||" on modules is also built in; it generates states which are parallel connections of the states of its components,
adding a new sort called Tuple, a tupling operation 
<.,.,.,.> : S1 S2 ... Sn 
-> Tuple, and projection operations i* : Tuple -> Si where i ranges from 1 to the 
number of modules connected, plus the “tupling equation” <1*(T),2*(T),...,n*(T)> 
= T, which says that all tuple states are tuples of component states.

To show refinement, we first prove a lemma, that <I, put(E, J, A)> = <I, A> if I <= J for any I,J,E,A, by induction on I, using circular coinductive rewriting:

open .
set cobasis of STACK

***> base case
cred <0, put(E, J, A)> == <0, A> .

***> induction step
ops i j : -> Nat . eq i < j = true .
cq <i, put(E, J, A)> = <i, A> if i <= J .
cred <s i, put(E, j, A)> == <s i, A> .
close

Because this is a refinement proof, we want to use the cobasis \{top, pop\} of STACK, not 
that of STACKIMP, which BOBJ correctly finds to include push. For this reason, BOBJ 
has a command “set cobasis of <mod-name>” to tell the system to use the cobasis of 
<mod-name> (there are other commands for setting cobases). Since both reductions give 
true, we can add the conditional equation:

bth STACKIMP is pr STACKIMP .
vars I J : Nat . var A : Arr . var E : Elt .
cq <I, put(E, J, A)> = <I, A> if I <= J .
end

The two equations of STACK now follow by just behavioral rewriting:

red pop push(E, <I, A>) == <I, A> .
red top push(E, <I, A>) == E .

V.E.4 Equivalent Regular Languages

We give a behavioral specification for regular expressions with several examples 
of their equivalence. It is perhaps surprising that no human intervention is required – 
BOBJ does all the work, using circular coinductive rewriting. We thank to Bogdan 
Warinschi, whose work inspired us. The behavioral spec for regular expressions contains
constants \texttt{empty} and \texttt{nil}, for the empty language and the language consisting of just the empty string, plus concatenation, union, Kleene star, and an operation \texttt{nil-in}, for asking if a language contains the empty string.

\begin{verbatim}
bth REGEXP is sort RegExp .
  ops empty nil : -> RegExp .
  vars L L' : RegExp .
  eq empty L = empty .
  eq nil-in L L' = nil-in L and nil-in L' .
  eq nil-in L + L' = nil-in L or nil-in L' .
  eq nil-in L* = true .
  eq nil-in nil = true .
  eq nil-in empty = false .
end
\end{verbatim}

Notice the attributes \texttt{assoc}, \texttt{comm}, \texttt{idem}, \texttt{id:}, \texttt{prec}, which declare an operation associative, commutative, idempotent, having an identity, and having a precedence, respectively; BOBJ can parse and rewrite modulo these attributes.

The following adds a letter \texttt{x} to the alphabet, where \texttt{x?(L)} is true iff there are words that start with \texttt{x} in \texttt{L}, and \texttt{-x(L)} gives the language obtained from \texttt{L} keeping only those words that start with \texttt{x} and then removing the leading \texttt{x}:

\begin{verbatim}
bth LETTER is pr REGEXP .
  sort Letter . subsort Letter < RegExp .
  op x : -> Letter .
  eq nil-in x = false .
  var Y : Letter .
  op x? : RegExp -> Bool .
  vars L L' : RegExp . var X : Letter .
  eq x?(L L') = x?(L) or nil-in L and x?(L') .
  eq x?(L + L') = x?(L) or x?(L') .
  eq x?(L*) = x?(L) .
  eq x?(X) = x == X .
  eq x?(nil)= false .
  eq x?(empty) = false .
  op -x : RegExp -> RegExp [prec 1] .
  eq -x(L L') = -x(L) L' +
      if (nil-in L) then -x(L') else empty fi .
  eq -x(L + L') = -x(L) + -x(L') .
\end{verbatim}
We can now create regular languages with as many letters as we want:

bth LANGUAGE is
   pr LETTER * (op x to a, op x? to a?, op -x to -a).
   pr LETTER * (op x to b, op x? to b?, op -x to -b).
end

BOBJ finds the correct cobasis, \{a?, b?, nil-in, -a, -b\}, and can prove equivalences of regular languages by circular coinductive rewriting:

cred a* + nil == a*.
cred aa* + nil == a*.
cred a*a* == a*.
cred (a+nil)a* == a*.
cred (a+nil)* == a*.
cred a*a == aa*.
cred a(ba)* == (ab)*a.
cred (a*b)*a* == a*(ba*)*.
cred (a+b)* == (a*b)*.
open.
   op L : -> RegExp.
>>> suppose L = aL + b
   eq a?(L) = true.
   eq -a(L) = L.
   eq b?(L) = true.
   eq -b(L) = nil.
   eq nil-in L = false.
>>> then we can prove that
   cred L == a*b.
close

Of course, no one should be surprised that the equivalence of regular expressions is decidable; what we found intriguing is that it can be decided using only circular coinductive rewriting. The correctness of this method is justified by the following easy to check result:

**Proposition 102** Two regular languages are equal iff they are behaviorally equivalent under the above specification, that is, iff they cannot be distinguished by experiments involving the operations nil-in, a?, -a, b?, -b, etc., as above.
Future research will investigate other examples, such as the equivalence of automata, finite state machines, processes in process algebra and similar formalisms, context-free grammars, lambda-expressions, linear temporal logic, concurrent connections of buffers and streams, and eventually, more complex examples like communication protocols.
Chapter VI

More Theoretical Aspects

This chapter is the most technical in the thesis, and consequently assumes the reader familiar with more mathematics.

VI.A Categories of Hidden Algebras

We have only defined and discussed hidden algebras over a hidden signature until now, ignoring any relationship between them. In this section, after a short introduction to category theory and some categorical constructions, we’ll define morphisms of hidden algebras, both loose data and fixed data versions, thus giving birth to categories of hidden algebras. Then we’ll see how certain hidden algebras are special coalgebras, and finally, two different ways of looking at hidden logic as an abstract logic are presented.

VI.A.1 A Little Category Theory

We assume the reader familiar with basic notions of category theory. The rôle of this subsection is to establish our conventions and notations rather than to rephrase well-known concepts. For more on category theory we refer the reader to [102, 91, 49].

We let $|C|$ denote the objects of a category $C$. The composition of morphisms is written in diagrammatic order, in the sense that if $f: A \to B$ and $g: B \to C$ are morphisms in $C$, then their composition is written $f; g: A \to C$. An object $I$ in $C$ is called initial iff there is a unique morphism, which we denote $\alpha_A: I \to A$, from $I$ to any object $A$. Dually, an object $F$ is final iff there is a unique morphism, which we denote
ω_A : A → F, from any object A to F. Given a family of objects \( \{A_j \mid j \in J\} \) in \( \mathcal{C} \), we let \( \prod_{j \in J} A_j \) and \( \coprod_{j \in J} A_j \) denote their product and coproduct, respectively, when they exist, and let \( \Pi_i : \prod_{j \in J} A_j \to A_i \) and \( \Pi_i : A_i \to \prod_{j \in J} A_j \) denote the corresponding projections and coprojections, for each \( i \in J \).

**Inclusion systems**

Inclusion systems are an alternative of factorization systems [91, 117], which promote the idea of *unique* factorization. Sometimes they are preferred to factorization systems both because they are more intuitive and because proofs tend to be smoother. Inclusion systems first appeared in [41] in the context of modularization, and were then developed and generalized in [92, 33] and also [34].

**Definition 103** \( (\mathcal{I},\mathcal{E}) \) is a **weak inclusion system** for a category \( \mathcal{C} \) iff \( \mathcal{I} \) and \( \mathcal{E} \) are subcategories of \( \mathcal{C} \) having the same objects as \( \mathcal{C} \), \( \mathcal{I} \) is a partial order in the sense that there is at most one morphism between any objects in \( \mathcal{I} \) and if there is one morphism from \( A \) to \( B \) and one morphism from \( B \) to \( A \) then \( A \) and \( B \) are equal, and every morphism \( f \) in \( \mathcal{C} \) admits a *unique* factorization \( f = e_f;i_f \) with \( e_f \) in \( \mathcal{E} \) and \( i_f \) in \( \mathcal{I} \). Morphisms in \( \mathcal{I} \) are called **inclusions** and they are often denoted \( A \hookrightarrow B \), and \( A \) is called a subobject\(^1\) of \( B \). Morphisms in \( \mathcal{E} \) are called **\( \mathcal{E} \)-morphisms**, and \( B \) is called a **quotient** of \( A \) whenever there are some \( \mathcal{E} \)-morphisms \( e : A \to B \). For a morphism \( f : A \to B \) in \( \mathcal{C} \), the factorization object (i.e. the target of \( e_f \)) is denoted \( f(A) \).

If we had required \( \mathcal{E} \) to contain only epimorphisms, then \( (\mathcal{I},\mathcal{E}) \) should have been called an inclusion system [41]. We prefer to use weak inclusion systems because they are more general and still have the same power in our context.

**Example 104** The most intuitive category admitting a weak inclusion system is probably \( \text{Set} \), the category of sets and functions, in which \( \mathcal{E} \) contains exactly the surjective functions and \( \mathcal{I} \) contains the inclusions of sets. The category of indexed sets admits an intuitive weak inclusion system, too, in which \( \mathcal{E} \) contains indexed families of surjective

\(^1\)Do not confuse this notion of subobject with the categorical notion of subobject as a coset of monomorphisms with the same target. Actually, there is some relationship between the two notions which is explored in [34].
functions and \( I \) contains indexed families of inclusions of sets. The category \( \text{Alg}_{\Sigma} \) of \( \Sigma \)-algebras has an obvious weak inclusion system (which is actually an inclusion system), where \( I \) contains inclusions of \( \Sigma \)-algebras and \( E \) contains surjective morphisms.

The following are basic properties of inclusion systems that care going to be used later in the other sections. Their proofs can be found in [33].

**Proposition 105** If \( \langle I, E \rangle \) is a weak inclusion system for \( C \), \( f \in C \) and \( i \in I \), then

1. \( i \) is a monomorphism,
2. **Right-cancellability**: If \( f; i \) is in \( I \) then \( f \) is in \( I \);
3. **Diagonal-fill-in**: For all morphisms \( g \in C \) and for all \( e \in E \), if \( f; i = e; g \) then there is a unique morphism \( h \in C \) such that \( e; h = f \) and \( h; i = g \);
4. If \( f; i \) is in \( E \) then \( i \) is an identity.

The following definition appeared many times in the literature under varied formulations. It basically provides a categorical framework under which all subobjects of an object can be put together in a coproduct.

**Definition 106** A weak inclusion system \( \langle I, E \rangle \) of \( S \) is called **well-powered** iff the class of subobjects of any object in \( S \) is a set.

**Coalgebra**

Coalgebra, as dual notion of (categorical) algebra [102], turned out to be appropriate to handle infinite data types and dynamic systems [98, 145, 129].

**Definition 107** Given a category \( C \) and a functor \( G : C \to C \), a \( G \)-coalgebra is a pair \((A, a)\) consisting of an object \( A \) and a morphism \( a : A \to G(A) \) in \( C \). A morphism of \( G \)-coalgebras from \((A, a)\) to \((B, b)\) is a morphism \( f : A \to B \) such that \( a; G(f) = f; b \).

\[
\begin{array}{ccc}
A & \xrightarrow{a} & G(A) \\
\downarrow{f} & & \downarrow{G(f)} \\
B & \xrightarrow{b} & G(B)
\end{array}
\]

\( G \)-coalgebras and morphisms of \( G \)-coalgebras form a category, \( \text{CoAlg}(G) \).
\(C\) is the category of indexed sets in most practical situations, and in many of them there is only one index. Software and hardware systems can be regarded under certain circumstances as coalgebras \(a : A \to \mathcal{G}(A)\), where \(A\) is the universe of states, \(\mathcal{G}\) provides the operations that can modify or observe a state, and \(a\) provides a particular implementation of those. This description is close in spirit to our approach to state systems based on hidden algebra, but it has its limitations (see Subsection VI.A.3). Most of the categories of coalgebras of interest have final objects (see [145] for a nice synthesis on the existence of final systems).

Using the proposition below, one can show that \(\text{CoAlg}(\mathcal{G})\) admits a weak inclusion system in almost all concrete situations. Its proof can be found in [34]:

**Proposition 108** If \((\mathcal{I}, \mathcal{E})\) is a weak inclusion system of \(C\) and \(\mathcal{G} : C \to C\) is an inclusion preserving\(^2\) functor, then \((\mathcal{I}, \mathcal{E})\) is also a weak inclusion system of \(\text{CoAlg}(\mathcal{G})\).

**Institutions**

The concept of institution was introduced by Goguen and Burstall [52, 53, 54, 55] to formalize the informal notion of "logical system". Having as basic idea the Tarski’s classic semantic definition of truth [152], institutions have the possibility of translating sentences and models along signature morphisms, with respect to an axiom called the satisfaction condition, which says that *truth is invariant under change of notation*.

**Definition 109** An institution \(\mathcal{Y} = (\text{Sign}, \text{Mod}, \text{Sen}, \models)\) consists of a category \(\text{Sign}\) whose objects are called *signatures*, a functor \(\text{Mod} : \text{Sign} \to \text{Cat}^{\text{op}}\) giving for each signature \(\Sigma\) a category of \(\Sigma\)-*models*, a functor \(\text{Sen} : \text{Sign} \to \text{Set}\) giving for each signature a set of \(\Sigma\)-*sentences*, and a \(\Sigma\)-indexed relation \(\models \subseteq |\text{Mod}(\Sigma)| \times |\text{Sen}(\Sigma)|\), such that for each signature morphism \(\varphi : \Sigma \to \Sigma'\), the following diagram commutes,

\[
\begin{array}{c}
\Sigma \\
\downarrow \varphi \\
\Sigma'
\end{array}
\begin{array}{c}
|\text{Mod}(\Sigma)| \\
\downarrow \text{Mod}(\varphi) \\
|\text{Mod}(\Sigma')|
\end{array}
\begin{array}{c}
\text{Sen}(\Sigma) \\
\downarrow \text{Sen}(\varphi) \\
|\text{Sen}(\Sigma')|
\end{array}
\]

\(^2\text{That is, } A \hookrightarrow B \text{ implies } \mathcal{G}(A) \hookrightarrow \mathcal{G}(B)\).
that is, the following \textit{Satisfaction Condition}

\[ m' \models_{\Sigma'} \text{Sen}(\varphi)(e) \text{ iff } \text{Mod}(\varphi)(m') \models_{\Sigma} e \]

holds for each \(m' \in |\text{Mod}(\Sigma')|\) and each \(e \in \text{Sen}(\Sigma)\).

We sometimes write only \(\varphi\) instead of \(\text{Sen}(\varphi)\) and \(\models_{\varphi}\) instead of \(\text{Mod}(\varphi)\); the functor \(\models_{\varphi}\) is called the \textit{reduct functor} associated to \(\varphi\). With these notations, the satisfaction condition becomes

\[ m' \models_{\Sigma'} \varphi(e) \text{ iff } m'|_{\varphi} \models_{\Sigma} e. \]

The satisfaction notation, \(\models_{\Sigma}\), is also used for a set of sentences in the right side, that is we write \(m \models_{\Sigma} E\) for \(E\) a set of \(\Sigma\)-sentences, meaning that \(m\) satisfies each sentence in \(A\). Moreover, we extend this notation for sets of sentences on both sides: \(E \models_{\Sigma} E'\) means that \(m \models_{\Sigma} E'\) for any \(\Sigma\)-model \(m\) with \(m \models_{\Sigma} E\). We forget the subscript \(\Sigma\) in \(\models\) whenever it can be inferred unambiguously from the context. The \textit{closure} of a set of \(\Sigma\)-sentences \(E\) is the set denoted \(E^*\) which contains all \(e\) in \(\text{Sen}(\Sigma)\) such that \(E \models_{\Sigma} e\). The sentences in \(E^*\) are often called the \textit{theorems} of \(E\). Obviously the closure operation is a closure operator, that is it is extensive, monotonic and idempotent.

\textbf{Closure Lemma:} For any signature morphism \(\varphi : \Sigma \to \Sigma'\) and any set of \(\Sigma\)-sentences \(E\), \(\varphi(E^*) \subseteq \varphi(E)^*\).

\textbf{Definition 110} A \textit{specification} or \textit{presentation} is a pair \((\Sigma, E)\) where \(\Sigma\) is a signature and \(E\) is a set of \(\Sigma\)-sentences. A \textit{specification morphism} from \((\Sigma, E)\) to \((\Sigma', E')\) is a signature morphism \(\varphi : \Sigma \to \Sigma'\) such that \(\varphi(E) \subseteq E'^*\). Specifications and specification morphisms give a category denoted \textbf{Spec}. A \textit{theory} \((\Sigma, E)\) is a specification with \(E = E^*\); the full subcategory of theories in \textbf{Spec} is denoted \textbf{Th}.

It can be readily seen that the categories \textbf{Th} and \textbf{Spec} are equivalent. The equivalence functor is just the inclusion of categories \(U_s : \textbf{Th} \to \textbf{Spec}\). It has a left-adjoint-left-inverse \(F_s : \textbf{Spec} \to \textbf{Th}\), given by \(F_s(\Sigma, E) = (\Sigma, E^*)\) on objects and identity on morphisms; note that \(F_s\) is also a right adjoint of \(U_s\), so \textbf{Th} is a reflective and coreflective subcategory of \textbf{Spec}.
VI.A.2 The Category $HAlg_{\Sigma}$

Definition 111 A morphism of loose data hidden $\Sigma$-algebras is any morphism of many-sorted $\Sigma$-algebras, while a morphism of fixed data $(\Sigma, D)$-algebras, say $f : A \to B$, is a morphism of many-sorted $\Sigma$-algebras which is identity on visible sorts, that is, $f|_V$ is $1_V : D \to D$. ■

Now it can be easily checked that hidden algebras together with morphisms of hidden algebras form a category, in both fixed data and loose data situations. Since loose data hidden algebra is nothing but many-sorted algebra, and since many-sorted algebra was intensively categorically studied by many scholars, we are going to put more emphasis on fixed data hidden algebra in this section.

Definition 112 Given a fixed data hidden signature $(\Sigma, D)$, abbreviated $\Sigma$, we let $HAlg_{(\Sigma, D)}$, abbreviated $HAlg_{\Sigma}$, denote the category of fixed data hidden $(\Sigma, D)$-algebras with their morphisms. ■

The category $HAlg_{\Sigma}$ admits an inclusion system $(I, E)$, where $I$ and $E$ contain inclusions and surjective morphisms of fixed data hidden $(\Sigma, D)$-algebras, respectively. The following proposition deals with preservation of satisfaction under morphisms.

Proposition 113 Given a $\Sigma$-equation $(\forall X) t = t'$, say $e$, and a morphism $f : A \to B$ in $HAlg_{\Sigma}$, then

1. If $f \in E$ then $A \models e$ implies $B \models e$, and
2. If $B \models e$ then $A \models e$.

Proof: 1. Let $\theta : X \to B$ be any map. Since $f$ is a surjection, there is a function $\tau : X \to A$ such that $\theta = \tau; f$. Since $A \models e$, it follows that $A_{\gamma}(\tau(t)) = A_{\gamma}(\tau(t'))$ for any $\Gamma$-experiment $\gamma$. On the other hand, $B_{\gamma}(\theta(t)) = B_{\gamma}(f(\tau(t))) = f(A_{\gamma}(\tau(t)))$ and similarly $B_{\gamma}(\theta(t')) = f(A_{\gamma}(\tau(t')))$, for any $\Gamma$-experiment $\gamma$. Therefore, $B_{\gamma}(\theta(t)) = B_{\gamma}(\theta(t'))$ for any map $\theta : X \to B$ and any $\Gamma$-experiment $\gamma$, that is, $B \models e$.

2. Let $\tau : X \to A$ be any map and let $\gamma$ be any $\Gamma$-experiment. Since $f$ is the identity on visible sorts, one gets that $A_{\gamma}(\tau(t)) = f(A_{\gamma}(\tau(t))) = B_{\gamma}(\tau; f(t))$ and similarly that
\[ A_\gamma(\tau(t')) = B_\gamma(\tau; f(t')). \] On the other hand, \( B \models_\Sigma e, \) that is, \( B_\gamma(\tau; f(t)) = B_\gamma(\tau; f(t')) \) for any \( \Gamma \)-experiment \( \gamma. \)

The preservation of satisfaction under quotients, i.e., 1. in the proposition above, is standard, but 2. does not hold for strict satisfaction\(^3\); it is a very distinctive feature of hidden algebra, strongly dependent on a fixed data universe. In Subsection VI.A.3, we’ll see that under some restrictions, coproducts of hidden algebras exist and they preserve a certain kind of equations.

One of the most frequent criticisms to standard algebraic specifications is that they always admit unwanted models, one of them being the final model which consists of only one element carriers. This means that those degenerated models are valid implementations of any algebraic specification, no matter how many equational constraints it has, which is of course unsatisfactory, because in this situation the specification can not serve as a concrete and complete interface between the designer of a system and its implementer.

A similar criticism works for loose data hidden algebra and its behavioral specifications, because its models are just ordinary algebras and its behavioral equations can be replaced by ordinary equations (see Proposition 21) in general. However, this criticism does not work for fixed data hidden algebra anymore. This is because its models preserve the data, so there are already many pairs of elements which are known to be distinct in all models. Therefore, the degenerated models are not accepted anymore. Moreover, in general \( HAlg_\Sigma \) has no “nice” categorical property, even for very simple hidden signatures:

**Proposition 114** If \( \Sigma \) is the fixed data hidden signature:

\[
V = \{v\}, \quad D_v = \{true, false\},
\]
\[
H = \{h\}, \quad \Sigma = \{a: hh \to v\},
\]
then \( HAlg_\Sigma \) has neither final objects nor (finite) coproducts. On the other hand, if \( \Sigma \) is the fixed data hidden signature:

\[
V = \{v\}, \quad D_v = \{true, false\},
\]
\[
H = \{h\}, \quad \Sigma = \{a: h \to v, b: \to h\},
\]

\(^3\)It holds only when \( f \) is an inclusion.
then $\mathcal{HAlg}_\Sigma$ has neither initial objects nor (finite) products.

Proof: Let us show that the first signature does not admit final models. Suppose that $\mathcal{F} = ((D_v, F_h), F_a : F_h \times F_h \to D_v)$ is a final hidden $\Sigma$-algebra. Since any set can be organized as a hidden $\Sigma$-algebra by interpreting $a$ in an arbitrary way, one gets that there exists a function from any set to $F_h$, so $F_h$ is nonempty. Let $\mathcal{P}$ be the hidden $\Sigma$-algebra $((D_v, \mathcal{P}(F_h)), P_a : \mathcal{P}(F_h) \times \mathcal{P}(F_h) \to D_v)$, where $\mathcal{P}(F_h)$ is the power set of $F_h$, and $P_a(A, B)$ is true if $A = B$ and false otherwise. Then there exists a unique\(^4\) morphism of hidden $\Sigma$-algebras $\alpha : \mathcal{P} \to \mathcal{F}$; suppose that $\alpha = (1_v : D_v \to D_v, \alpha_h : \mathcal{P}(F_h) \to F_h)$. Notice that $A \neq B$ implies $\alpha_h(A) \neq \alpha_h(B)$, since if $A \neq B$ and $\alpha_h(A) = \alpha_h(B)$ then:

\[
\begin{align*}
false &= P_a(A, B) \\
&= 1_v(P_a(A, B)) \\
&= F_a(\alpha_h(A), \alpha_h(B)) \\
&= F_a(\alpha_h(A), \alpha_h(A)) \\
&= 1_v(P_a(A, A)) \\
&= P_a(A, A) \\
&= true.
\end{align*}
\]

Consequently, $\alpha_h : \mathcal{P}(F_h) \to F_h$ is injective. But this is a contradiction, because $\mathcal{P}(F)$ always has a larger cardinal than $F$ for any set $F$.

Let us now show that the first signature does not have coproducts of models. Indeed, let $A = ((D_v, \{x\}), A_a : \{x\} \times \{x\} \to D_v)$ and $B = ((D_v, \{y\}), B_a : \{y\} \times \{y\} \to D_v)$ be two hidden $\Sigma$-algebras, where $A_a(x, x) = true$ and $B_a(y, y) = false$, and suppose that $C = ((D_v, C_h), C_a : C_h \times C_h \to D_v)$ is a coproduct of $A$ and $B$, where $\Pi_A : A \to C$ and $\Pi_B : B \to C$ are the two coprojections. It is clear that $C_a(\Pi_A(x), \Pi_A(x)) = true$ and that $C_a(\Pi_B(y), \Pi_B(y)) = false$. Therefore, $\Pi_A(x) \neq \Pi_B(y)$. Suppose that $C_a(\Pi_A(x), \Pi_B(y)) = true$ and let $C'$ be the hidden $\Sigma$-algebra $((D_v, C_h), C'_a : C_h \times C_h \to D_v)$ defined exactly as $C$, except that $C_a(\Pi_A(x), \Pi_B(y)) = false$. Notice that there are two morphisms of hidden $\Sigma$-algebras, $\Pi'_A : A \to C'$ and $\Pi'_B : B \to C'$ taking $x$ to $\Pi_A(x)$ and $y$ to $\Pi_B(y)$, respectively. Therefore, there exists a (unique) morphism of hidden $\Sigma$-algebras, say $c : C \to C'$, such that $\Pi_A; c = \Pi'_A$ and $\Pi_B; c = \Pi'_B$. It is then immediate

---

\(^4\)The unicity is not important here.
that $c(\Pi_A(x)) = \Pi_A(x)$ and that $c(\Pi_B(y)) = \Pi_B(y)$, so
\[
true = C_a(\Pi_A(x), \Pi_B(y)) \\
= c(C_a(\Pi_A(x), \Pi_B(y))) \\
= C'_a(c(\Pi_A(x)), c(\Pi_B(y))) \\
= C'_a(\Pi_A(x), \Pi_B(y)) \\
= false.
\]

Consequently, there is no coproduct for $A$ and $B$ in $HAlg_{\Sigma}$.

The second part is easier. Suppose that $HAlg_{\Sigma}$, where $\Sigma$ is the second signature, has an initial model $I$, and that $I_a(I_b) = true$. The let $A$ be any hidden $\Sigma$-algebra such that $A_a(A_b) = false$, and let $\alpha_A : I \to A$ be the unique morphism of $\Sigma$-algebras. Then
\[
true = I_a(I_b) \\
= \alpha_A(I_a(I_b)) \\
= A_a(A_b) \\
= false.
\]

Therefore, there is no initial model in $HAlg_{\Sigma}$.

Finally, let $A$ and $B$ be hidden $\Sigma$ algebras such that $A_a(A_b) = false$ and $B_a(B_b) = true$, and, for the sake of contradiction, suppose that $P$ is their product in $HAlg_{\Sigma}$, together with the two projections $\prod_A : P \to A$ and $\prod_B : P \to B$. It can be then easily seen that $P_a(P_b) = \prod_A(P_a(P_b)) = A_a(A_b) = false$ and that $P_a(P_b) = \prod_B(P_a(P_b)) = B_a(B_b) = true$, that is, that $false = true$, contradiction. $\square$

Therefore, $HAlg_{\Sigma}$ has neither limits nor colimits in general, thus suggesting that it may not be worth considering any categorical approach to hidden algebra. But what if hidden signatures are restricted such that operations with more than one hidden argument, like $a : hh \to v$ of the first signature in Proposition 114, and hidden constants, like $b : \to h$ of the second signature in Proposition 114, are not allowed?

VI.A.3 Hidden Algebra as Coalgebra

A special but theoretically very important case of fixed data hidden algebra occurs when all the operations in $\Sigma - \Sigma|_V$ are required to have exactly one argument
of hidden sort. Hence, neither operations with many hidden arguments nor hidden const-
stants are allowed. To be consistent with our overall notational and naming conventions,
we call the corresponding logic monadic fixed data hidden algebra, noting that various
other names can be found in the literature, such as coalgebraic or destructor hidden
algebra [95, 23, 24, 159].

As first noted by Malcolm [107] and Cîrstea [23], the category of hidden algebras
as above is isomorphic to a category of coalgebras (see the two references for proof):

**Theorem 115**  If $\Sigma$ is a monadic fixed data hidden signature, then there exists a functor
$G_\Sigma$ such that $HAlg_\Sigma$ is isomorphic to $CoAlg(G_\Sigma)$.

The functor $G_\Sigma: Set^H \to Set^H$ is defined as

$$(G_\Sigma(S))_h = \prod_{\sigma: wh \to h'} S_{h'}^{(D^{[w]})} \times \prod_{\sigma: wh \to v} D_v^{(D^{[w]})},$$

where $Set^H$ is the category of $H$-sorted sets and $S = \{S_h \mid h \in H\}$ is any $H$-sorted
set. The interested reader may consult [23, 24, 159] for more details on monadic hidden
algebra as coalgebra, as comonad, and even as topos. Since the functor $G_\Sigma$ is polynomial
[145], monadic fixed data hidden algebra has most of the “good” properties of coalgebra,
including the following:

**Theorem 116**  If $\Sigma$ is a monadic fixed data hidden signature, then $HAlg_\Sigma$ is cocomplete
and has a final hidden algebra, denoted $F_\Sigma$.

Next, we’ll show how coproducts and final hidden algebras can be constructed
in monadic fixed-data hidden algebra, and also how behavioral satisfaction relates to the
kernel of the unique morphism to the final model.

**On Coproducts**

For any family $\{A_j\}_{j \in J}$ of hidden $\Sigma$-algebras, their coproduct $A = \coprod_{j \in J} A_j$ is
defined as follows: $A_v$ is $D_v$ for visible sorts $v$ and $A_h$ is the set $\{(j, a) \mid j \in J, a \in (A_j)_h\}$
for hidden sorts $h$; if $\sigma: wh \to v$ is an attribute then $A_{\sigma}(\overline{d},(j,a)) = (A_j)_\sigma(\overline{d}, a)$, and if
$\sigma: wh \to h'$ is a method then $A_{\sigma}(\overline{d},(j,a)) = (j,(A_j)_{\sigma}(\overline{d}, a))$, for any appropriate data
$\overline{d} \in D^{[w]}$ and state $s \in (A_j)_h$. Notice that this construction works because the hidden
signature is monadic.
On Final Models

Elements of $F_{\Sigma}$ in Theorem 116 can be thought of as “abstract states” (see [20, 57, 65]) represented as functions on experiments, returning visible values as results of evaluating a state in a given context. However, there is a tough issue regarding the final hidden algebra which we’d like to discuss here.

First, let us introduce slightly more restrictive notions of contexts and experiments, taking into consideration the monadic fixed data framework:

**Definition 117** Given $(\Sigma, D)$ a monadic fixed data hidden signature, a $(\Sigma, D)$-context for sort $s$ is a term in $T_{\Sigma, D}(\{\bullet : s\})$ having exactly one occurrence of $\bullet$. A $(\Sigma, D)$-context which is a term of visible result is called a $(\Sigma, D)$-experiment. Let $C_{\Sigma, D}[\bullet : s]$ and $E_{\Sigma}[\bullet : s]$ denote the sets of $(\Sigma, D)$-contexts and $(\Sigma, D)$-experiments, respectively.

Notice that data is added as constants to $\Sigma$, thus being allowed to be used in experiments and contexts. In this way, any fixed data hidden $(\Sigma, D)$-algebra $A$ can be regarded as a $\Sigma \cup D$-algebra, where each constant $d \in D$ is interpreted in $A$ as the datum $d$. It can be relatively easily seen that $(\Sigma, D)$-experiments in $E_{\Sigma}[\bullet : s]$ generate exactly the same behavioral equivalence relation as the one generated by $\Sigma$-experiments in $E_{\Sigma}[\bullet : s]$ in Section III.C, that is:

**Proposition 118** Given a hidden $\Sigma$-algebra $A$ and $a, a' \in A_s$, then $a \equiv_{\Sigma, s} a'$ if and only if $A_{\gamma}(a) = v A_{\gamma}(a')$ for each $\gamma \in E_{\Sigma, v}[\bullet : s]$.

Notice that the contexts and experiments in Definition 117 and the proposition above are strictly dependent on the monadic flavor of fixed data hidden algebra in this section; for example, there is no $(\Sigma, D)$-experiment for the first hidden $(\Sigma, D)$-signature in Proposition 114.

The following construction of the final hidden algebra $F_{\Sigma}$ for a monadic fixed data hidden signature $(\Sigma, D)$ is common and it appeared many places in the literature:

- $(F_{\Sigma}|_{(\Sigma \cup V)}) = D$;
- $(F_{\Sigma})_h = \prod_{v \in V} [E_{\Sigma, v}[\bullet : h] \rightarrow D_v]$ for each $h \in H$;
- $(F_{\Sigma})_a(d, p) = p_v(a[d, \bullet])$ when $a : wh \rightarrow v$ is an attribute, $d$ is some data in $D^w$, and $p = \{p_v \mid v \in V\}$ is an element in $(F_{\Sigma})_h$;
\[
\bullet ((F_S)_m(d,p))_v(\gamma) = p_v(\gamma[m[d,\bullet]]) \text{ when } m : wh \rightarrow h' \text{ is a method, } d \in D^w, \text{ and } p = \{p_v \mid v \in V\} \text{ is an element in } (F_S)_h, \text{ for all } v \in V \text{ and } \gamma \in \mathcal{E}_{\Sigma,h'}^D[\bullet : h'].
\]

We warn the reader that the construction above has a subtle error and encourage him/her to find it; \(F_S\) is a correctly defined hidden algebra, but it is not final! Before unveiling the error and providing a simple solution, we encourage the reader to intuitively understand what is the real problem with the construction above by skimming the following example.

**Example 119** Let \((\Sigma, D)\) be the following simple monadic fixed data hidden signature:

\[
\begin{align*}
V &= \{\text{Nat}\}, \quad \Sigma|_V = \{\_ + \_,\}, \quad D = \mathbb{N}, \\
H &= \{h\}, \quad \Sigma - \Sigma|_V = \{a : \text{Nat} \rightarrow \text{Nat}\},
\end{align*}
\]

where \(\mathbb{N}\) is the set of natural numbers regarded as a \(\{\_ + \_,\}\)-algebra. Then \(a(2,\bullet)\), \(a(1 + 1,\bullet)\), and \(3 + a(2,\bullet)\) are three valid experiments in \(\mathcal{E}_{\Sigma,h'}^D[\bullet : h]\). According to the construction of \(F_S\) above, the hidden carrier contains all possible functions from experiments to natural numbers, including for example those assigning 7 to \(a(2,\bullet)\), 5 to \(a(1 + 1,\bullet)\), and 9 to \(3 + a(2,\bullet)\). Obviously this is not a correct behavior of any state, so one cannot expect the \(F_S\) above to be a final model. ■

Let us now prove that \(F_S\) is not final, by showing that there exists a morphism \(r : F_S \rightarrow F_S\) which is not the identity on \(F_S\). Intuitively, \(r\) “reduces” the abstract states to canonical ones, in which the constraints of \(D\) cannot be used anymore; the “incorrect” abstract state in the example above is “corrected” to one which assigns 7 to \(a(2,\bullet)\), 7 to \(a(1 + 1,\bullet)\), and 10 to \(3 + a(2,\bullet)\). Formally, \(r\) is of course identity on \(\text{Nat}\) and it is recurrently defined on functions \(p : \mathcal{E}_{\Sigma,h'}^D[\bullet : h] \rightarrow D\) by

\[
r(p)_v(\gamma) = \begin{cases} 
  p_v(r_D(\gamma)) & \text{if } r_D(\gamma) = \sigma(t_1,\ldots,t_n) \text{ for some attribute } \sigma \in \Sigma, \text{ and } \\
  \sigma(\overline{d}, r(p)_v'(\gamma')) & \text{if } r_D(\gamma) = \sigma(\overline{d},\gamma') \text{ for some } \sigma : \text{wv} \rightarrow v \text{ in } \Sigma|_V, \\
  d \in D^w \text{ and } \gamma' \in \mathcal{E}_{\Sigma,v'}^D[\bullet : s],
\end{cases}
\]

where \(r_D : T_{\Sigma \cup D}(\bullet : s) \rightarrow T_{\Sigma \cup D}(\bullet : s)\) is the function recurrently defined by

\[
r_D(t) = \begin{cases} 
  \epsilon_D(t) & \text{if } t \in T_{\Sigma \cup D}, \text{ and } \\
  \sigma(r_D(t_1),\ldots,r_D(t_n)) & \text{otherwise, where } t = \sigma(t_1,\ldots,t_n),
\end{cases}
\]
where $\epsilon_D : T_{\Sigma \cup D} \to D$ is the usual evaluation co-universal morphism. It can be easily seen now that $r$ is indeed a morphism of hidden $\Sigma$-algebras, despite the fact that $r_D$ is not a morphism of $(\Sigma \cup D)$-algebras. Therefore, $F_\Sigma$ is not final.

A simple correction to the construction of $F_\Sigma$ above such that to make it a final hidden $\Sigma$-algebra, would be to replace $E_D^\Sigma[\bullet : h]$ by $E_{\Sigma-\Sigma}^\Sigma \cdot h$. Thus, any experiment has the form $a(\overline{d}, m_1(\overline{d_1}, ..., m_k(\overline{d_k}, \bullet), ...))$ for appropriate $k$ and data $\overline{d}, \overline{d_1}, ..., \overline{d_k}$, so conflicts as in Example 119 can not appear anymore; these special experiments containing only attributes and methods are often called local in the literature [66, 23].

However, we do not advocate a final algebra semantics, because in general the most important implementations are neither final nor initial.

**On Satisfaction**

Behavioral satisfaction can be nicely related to the final model in monadic hidden algebra, thus providing a framework in which hidden logic can be treated fully categorically (see Section VI.B for further details). We let $\omega_A : A \to F_\Sigma$ denote the unique morphism from a hidden $\Sigma$-algebra $A$ to the final hidden $\Sigma$-algebra $F_\Sigma$ given by Theorem 116. The following result is well known and we refer the reader to, for example, [66] for a proof:

**Proposition 120** For any hidden $\Sigma$-algebra $A$, the behavioral equivalence on $A$ is exactly the kernel of $\omega_A$.

The importance of equations with at most one variable of hidden sort in the context of monadic fixed data hidden algebra seems to have been first noted by Goguen and Malcolm in various preliminary versions of [66]. It also has a coalgebraic flavor and it is related to Bart Jacobs’ mongruences [95]. Cîrstea and Worrell also needed this restriction in their destructor specifications to show that hidden algebra can be organized as comonad and topos [24, 159, 26]. A related condition appears in [90]. We also need it in Section VI.B, as a sufficient condition for the preservation of behavioral satisfaction under coproducts:

**Proposition 121** If $\Sigma$ is a monadic fixed data hidden signature, $e$ is a $\Sigma$-equation with at most one variable of hidden sort, and $\{A_j \mid j \in J\}$ is a family of hidden $(\Sigma, D)$-algebras
such that $A_j \models e$ for all $j \in J$, then $\prod_{j \in J} A_j \models e$.

**Proof:** Let $e$ be the equation $(\forall X) \ t = t'$ where $X$ has at most one hidden variable, let us say $x$, and let $h : X \to \prod_{j \in J} A_j$ be a function. Then there is a unique $j \in J$ such that $h(x) = (j, a)$ with $a \in A_j$, and so, there is a function $h_j : X \to A_j$ such that $h = h_j \Pi_j$. Since $A_j \models e$, we get $\omega_{A_j}(h_j(t)) = \omega_{A_j}(h_j(t'))$, and since $\omega_{A_j} = \Pi_j \omega_U$ where $U$ is the hidden algebra $\prod_{j \in J} A_j$, we obtain that $\omega_U(h(t)) = \omega_U(h(t'))$. Thus $\prod_{j \in J} A_j \models e$. □

It is known in the theory of coalgebra that the kernel of the unique morphism from a system to a final system is the greatest bisimulation on that system (see, for example, Section 9 on final systems in [145]). Thus, by Theorem 115 and Proposition 120, behavioral equivalence is the greatest bisimulation on $G_{\Sigma}$-coalgebras. The interested reader may consult [98, 145] for more details on coalgebra and bisimulation, and [104] for more on the relationship between bisimulation and behavioral satisfaction in monadic hidden algebra.

**VI.A.4 Hidden Logic as Institution**

A noticeable issue at this stage in our research on hidden logic, is that it can be organized as an abstract logic, or institution, in two interesting ways, depending on whether the declaration of an operation to be behavioral is considered part of the signature or a separated sentence. We first published the work in this subsection in [71].

The first construction follows the institution of hidden algebra initially presented in [50], the institution of observational logic in [87], and the coherent hidden algebra approach in [37, 40], while the second seems more promising for future research. Our approach also avoids the infinitary logic used in observational logic. Only the fixed-data case is investigated here, but we hope to extend it to the loose-data case soon. We fix a data $\Psi$-algebra $D$, and proceed as follows:

**Signatures:** The category $\text{Sign}$ has hidden signatures over a fixed data algebra $D$ as objects. A morphism of hidden signatures $\phi : (\Gamma_1, \Sigma_1) \to (\Gamma_2, \Sigma_2)$ is the identity on the visible signature $\Psi$, takes hidden sorts to hidden sorts, and if a behavioral operation $\delta_2$ in $\Gamma_2$ has an argument sort in $\phi(H_1)$ then there is some behavioral operation $\delta_1$ in $\Gamma_1$
such that $\delta_2 = \phi(\delta_1)$. $\text{Sign}$ is indeed a category, and the composition of two hidden signature morphisms is another. Indeed, let $\psi: (\Gamma_2, \Sigma_2) \to (\Gamma_3, \Sigma_3)$ and let $\delta_3$ be an operation in $\Gamma_3$ having an argument sort in $(\phi; \psi)(H_1)$. Then $\delta_3$ has an argument sort in $\psi(H_2)$, so there is an operation $\delta_2$ in $\Gamma_2$ with $\delta_3 = \psi(\delta_2)$. Also $\delta_2$ has an argument sort in $\phi(H_1)$, so there is some $\delta_1$ in $\Gamma_1$ with $\delta_2 = \phi(\delta_1)$. Therefore $\delta_3 = (\phi; \psi)(\delta_1)$, i.e., $\phi; \psi$ is also a morphism of hidden signatures.

**Sentences:** Given a hidden signature $(\Gamma, \Sigma)$, let $\text{Sen}(\Gamma, \Sigma)$ be the set of all $\Sigma$-equations. If $\phi: (\Gamma_1, \Sigma_1) \to (\Gamma_2, \Sigma_2)$ is a hidden signature morphism, then $\text{Sen}(\phi)$ is the function taking a $\Sigma_1$-equation $e = (\forall X) t = t'$ if $t_1 = t'_1, \ldots, t_n = t'_n$ to the $\Sigma_2$-equation

$$\phi(e) = (\forall X') \phi(t) = \phi(t')$$

if $\phi(t_1) = \phi(t'_1), \ldots, \phi(t_n) = \phi(t'_n)$,

where $X'$ is $\{x : \phi(s) | x : s \in X\}$. Then $\text{Sen}: \text{Sign} \to \text{Set}$ is indeed a functor.

**Models:** Given a hidden signature $(\Gamma, \Sigma)$, let $\text{Mod}(\Gamma, \Sigma)$ be the category of hidden $\Sigma$-algebras and their morphisms. If $\phi: (\Gamma_1, \Sigma_1) \to (\Gamma_2, \Sigma_2)$ is a hidden signature morphism, then $\text{Mod}(\phi)$ is the usual reduct functor, $\cdot |_{\phi}$. Unlike [12, 87], etc., this allows models where not all operations are congruent.

**Satisfaction Relation:** behavioral satisfaction, i.e., $|_{(\Gamma, \Sigma)} = \models_{\Sigma}^\Gamma$.

**Theorem 122 Satisfaction Condition:** Given $\phi: (\Gamma_1, \Sigma_1) \to (\Gamma_2, \Sigma_2)$ a hidden signature morphism, $e = (\forall X) t = t'$ if $t_1 = t'_1, \ldots, t_n = t'_n$ a $\Sigma_1$-equation, and $A$ a hidden $\Sigma_2$-algebra, then $A \models_{\Sigma_2}^\Gamma \phi(e)$ iff $A|_{\phi} \models_{\Sigma_1}^\Gamma e$.

**Proof:** There is a bijection between $(A|_{\phi})^X$ and $A^{X'}$ that takes $\theta: X \to A|_{\phi}$ to $\theta': X' \to A$ defined by $\theta'(x : \phi(s)) = \theta(x : s)$, and takes $\theta': X' \to A$ to $\theta: X \to A|_{\phi}$ defined by $\theta(x : s) = \theta'(x : \phi(s))$. Notice that for every term $t$ in $T_{\Sigma_1}(X)$, we have $\theta(t) = \theta'(\phi(t))$ where $\phi(t)$ is the term $t$ with each $x : s$ replaced by $x : \phi(s)$ and each operation $\sigma$ replaced by $\phi(\sigma)$. It remains to prove that $a \equiv_{\Sigma_1, h}^\Gamma a'$ iff $a \equiv_{\Sigma_2, \phi(h)}^\Gamma a'$ for each $a, a' \in A_{\phi(h)}$, where $\equiv_{\Sigma_1}^\Gamma$ is behavioral equivalence on $A|_{\phi}$ and $\equiv_{\Sigma_2}^\Gamma$ is behavioral equivalence on $A$. Since $\phi(c_1) \in LT_{\Gamma_2}(A|_{\phi} \cup \{\phi(h)\})$ whenever $c_1 \in LT_{\Gamma_1}(A|_{\phi} \cup \{h\})$, one gets $a \equiv_{\Sigma_2, \phi(h)}^\Gamma a'$ implies $a \equiv_{\Sigma_1, h}^\Gamma a'$. Now if $c_2 \in LT_{\Gamma_2}(A \cup \{h\})$ then because for every operation $\delta_2$ in $\Gamma_2$ having an argument sort in $\phi(H_1)$ there is some $\delta_1$ in $\Gamma_1$
with \( \delta_2 = \phi(\delta_1) \), we iteratively get a term \( c_1 \in LT_{\Gamma_1}(A|_\varphi \mathcal{H}_1 \cup \{ h \}) \) such that \( c_2 = \phi(c_1) \).

Therefore \( a \equiv^{\Gamma_1}_{\Sigma_1 A} a' \) implies \( a \equiv^{\Gamma_2}_{\Sigma_2 \phi(h)} a' \).

Our second institution views the declaration of a behavioral operation as a new kind of sentence, rather than part of a hidden signature. The notion of model also changes, adding an equivalence relation as in [12]. This is natural for modern software engineering, since languages like Java provide classes with an operation denoted \( \text{equals} \) which serves this purpose. Sentences in [12] are pairs \( \langle e, \Delta \rangle \), where \( \Delta \) is a set of terms (pretty much like a cobasis over the derived signature), which are satisfied by \( (A, \sim) \) iff \( (A, \sim) \) satisfies \( e \) as in our case below (actually \( e \) is a first-order formula in their framework) and \( \sim \subseteq \equiv_\Delta \). Fix a data algebra \( D \), and proceed as follows:

**Signatures:** The category \( \text{Sign} \) has hidden signatures over \( D \) as objects. A morphism of hidden signatures \( \phi: \Sigma_1 \rightarrow \Sigma_2 \) is identity on the visible signature \( \Psi \) and takes hidden sorts to hidden sorts.

**Sentences:** Given a hidden signature \( \Sigma \), let \( \text{Sen}(\Sigma) \) be the set of all \( \Sigma \)-equations unioned with \( \Sigma \). If \( \phi: \Sigma_1 \rightarrow \Sigma_2 \) is a hidden signature morphism, then \( \text{Sen}(\phi) \) is the function taking a \( \Sigma_1 \)-equation \( e = (\forall X) t = t' \) if \( t_1 = t'_1, ..., t_n = t'_n \) to the \( \Sigma_2 \)-equation \( \phi(e) = (\forall X') \phi(t) = \phi(t') \) if \( \phi(t_1) = \phi(t'_1), ..., \phi(t_n) = \phi(t'_n) \), where \( X' \) is the set \( \{ x : \phi(s) \mid x : s \in X \} \), and taking \( \sigma : s_1 ... s_n \rightarrow s \) to \( \phi(\sigma): \phi(s_1) ... \phi(s_n) \rightarrow \phi(s) \). Then \( \text{Sen}: \text{Sign} \rightarrow \text{Set} \) is indeed a functor.

**Models:** Given a hidden signature \( \Sigma \), let \( \text{Mod}(\Sigma) \) be the category of pairs \( (A, \sim) \) where \( A \) is a hidden \( \Sigma \)-algebra and \( \sim \) is an equivalence relation on \( A \) which is identity on visible sorts, with morphisms \( f: (A, \sim) \rightarrow (A', \sim') \) with \( f: A \rightarrow A' \) a \( \Sigma \)-homomorphism such that \( f(\sim) \subseteq \sim' \). If \( \phi: \Sigma_1 \rightarrow \Sigma_2 \) is a hidden signature morphism, then \( \text{Mod}(\phi) \), often denoted \( \downarrow_{\phi} \), is defined as \( (A, \sim) |_{\phi} = (A|_{\phi}, \sim |_{\phi}) \) on objects, where \( A|_{\phi} \) is the ordinary many-sorted algebra reduct and \( \sim |_{\phi} = \sim_{\phi(s)} \) for all sorts \( s \) of \( \Sigma_1 \), and as \( f|_{\phi}: (A, \sim) |_{\phi} \rightarrow (A', \sim') |_{\phi} \) on morphisms. Notice that indeed \( f|_{\phi} (\sim |_{\phi}) \subseteq \sim' |_{\phi} \), so \( \text{Mod} \) is well defined.

**Satisfaction Relation:** A \( \Sigma \)-model \( (A, \sim) \) satisfies a conditional \( \Sigma \)-equation \( (\forall X) t = t' \) if \( t_1 = t'_1, ..., t_n = t'_n \) iff for each \( \theta: X \rightarrow A \), if \( \theta(t_1) \sim \theta(t'_1), ..., \theta(t_n) \sim \theta(t'_n) \) then
\[ \theta(t) \sim \theta(t'). \] Also \((A, \sim)\) satisfies a \(\Sigma\)-sentence \(\gamma \in \Sigma\) iff \(\gamma\) is congruent for \(\sim\).

**Theorem 123 Satisfaction Condition:** Let \(\phi: \Sigma_1 \to \Sigma_2\) be a morphism of hidden signatures, let \(e\) be a \(\Sigma_1\)-sentence and let \((A, \sim)\) be a model of \(\Sigma_2\). Then \((A, \sim) \models_{\Sigma_2} \phi(e)\) iff \((A, \sim)|_{\phi} \models_{\Sigma_1} e\).

**Proof:** First suppose \(e\) is a \(\Sigma\)-equation \((\forall X) t = t'\) if \(t_1 = t'_1, \ldots, t_n = t'_n\). Notice that there is a bijection between functions from \(X\) to \((A|_{\phi})\) and functions from \(X'\) to \(A\) taking \(\theta: X \to A|_{\phi}\) to \(\theta': X' \to A\) defined by \(\theta'(x : \phi(s)) = \theta(x : s)\) and taking \(\theta': X' \to A\) to \(\theta: X \to A|_{\phi}\) defined by \(\theta(x : s) = \theta'(x : \phi(s))\). Because for every term \(t\) in \(T_{\Sigma_1}(X)\) we have \(\theta(t) = \theta'(\phi(t))\) where \(\phi(t)\) is the term \(t\) with each \(x : s\) replaced by \(x : \phi(s)\) and each operation \(\sigma\) replaced by \(\phi(\sigma)\), the result is immediate.

Second, suppose \(e\) is an operation \(\gamma \in \Sigma\). Then \((A, \sim)\) satisfies \(\phi(\gamma)\) iff \(\phi(\gamma)\) is congruent for \(\sim\), which is equivalent to \(\gamma\) being congruent for \(\sim|_{\phi}\). \(\square\)

This institution justifies our belief that asserting an operation behavioral is a kind of sentence, not a kind of syntactic declaration as in the “extended hidden signatures” of [40]. Coinduction now appears in the following elegant guise:

**Proposition 124** Given a hidden subsignature \(\Gamma\) of \(\Sigma\), a set of \(\Sigma\)-equations \(E\) and a hidden \(\Sigma\)-algebra \(A\), then

- \((A, \sim) \models_{\Sigma} E, \Gamma\) implies \((A, \equiv_{\Sigma}^{\Gamma}) \models_{\Sigma} E, \Gamma\).
- \((A, \equiv_{\Sigma}^{\Gamma}) \models_{\Sigma} \Gamma\).
- \(A \models_{\Sigma}^{\Gamma} E\) iff \((A, \equiv_{\Sigma}^{\Gamma}) \models_{\Sigma} E\) iff \((A, \equiv_{\Sigma}^{\Gamma}) \models_{\Sigma} E, \Gamma\).

There is a natural relationship between our two institutions:

- since congruent operations are declared with sentences, any signature in the first institution translates to a specification in the second;
- any model \(A\) of \((\Sigma, \Gamma)\) in the first institution gives a model of the second, namely \((A, \equiv_{\Sigma}^{\Gamma})\);
- any \((\Sigma, \Gamma)\)-sentence is a \(\Sigma\)-sentence;

\(^5\)However, the most recent version of [39] treats coherence assertions as sentences.
and we can see that for any \((\Sigma, \Gamma)\)-sentence \(e\) and any hidden \(\Sigma\)-algebra \(A\), we get \(A \equiv^\Gamma e \iff (A, \equiv^\Gamma) \models^\Sigma e\). This relationship suggests a new (as far as we know) kind of relationship between institutions (here \(\text{Th}(I)\) denotes the category of theories over \(I\), see [55]):

**Definition 125** If \(\mathcal{Y} = (\text{Sign}, \text{Mod}, \text{Sen}, \models)\) and \(\mathcal{Y}' = (\text{Sign}', \text{Mod}', \text{Sen}', \models')\) are two institutions, then an **institution theoroidal forward morphism**\(^6\), from \(\mathcal{Y}\) to \(\mathcal{Y}'\) is \((\Phi, \beta, \alpha)\) where:

- \(\Phi : \text{Sign} \to \text{Th}(\mathcal{Y}')\) is a map such that \(\Phi; \mathcal{U}' : \text{Sign} \to \text{Sign}'\) is a functor, where \(\mathcal{U}' : \text{Th}(\mathcal{Y}') \to \text{Sign}'\) is the forgetful functor; we ambiguously let \(\Phi\) also denote the functor \(\Phi; \mathcal{U}'\),
- \(\beta : \text{Mod} \Rightarrow \Phi; \text{Mod}'\) is a natural transformation, and
- \(\alpha : \text{Sen} \Rightarrow \Phi; \text{Sen}'\) is a natural transformation,

such that for any signature \(\Sigma \in \text{Sign}\), any sentence \(e \in \text{Sen}(\text{Sign})\) and any model \(m \in \text{Mod}(\text{Sign})\), the satisfaction condition, \(m \models^\Sigma e \iff \beta(m) \models^\Phi(\Sigma) \alpha(e)\), holds. ■

**Proposition 126** There is an institution theoroidal forward morphism from the first to the second institution defined above.

**Proof:** It follows immediately from the constructions and discussions above. □

### VI.B Axiomatizability

Starting with Birkhoff [17], mathematicians were interested in the definitional power of equations, i.e., in characterization results for classes of algebras containing exactly all algebras which satisfy a given set of equations. Birkhoff proved in 1935 that a class of algebras is definable by equations if and only if it is closed under common operations, such as subalgebra, quotient algebra and product algebra. He called such a class a **variety**. Later, the notion of algebra was defined categorically for any endofunctor (usually a monad [102]), and the Birkhoff axiomatizability result was also abstracted to

\(^6\)This terminology is a preliminary attempt to bring some order to the chaos of relationships among institutions, by using names that suggest the nature of the relationship involved.
catch other modern approaches to equational logics and even first order logic (for example [3, 118, 132]).

We published the results in this section in [133, 136]. Classes of certain coalgebras, including monadic fixed data hidden algebra (see Subsection VI.A.3), that are definable by equations are investigated. We show that a class of coalgebras is definable by equations if and only if it is closed under coproducts, quotients, sources of morphisms\(^7\) and representative inclusions\(^8\). Even if our result is not as general as might seem desirable (because it does not allow hidden constants and involves only a special kind of equation, having at most one hidden variable), it can be a starting point toward stronger equational characterization results for coalgebra.

**Related Work:** As far as the author knows, the first Birkhoff-like result for coalgebra belongs to Jan Rutten [145], who introduced the notion of *covariety* as a class of coalgebras closed under coproducts, quotients and subcoalgebras. He showed that a class of coalgebras \(\mathcal{K}\) is a covariety iff there are a set of “colors” and a subcoalgebra \(S\) of a cofree coalgebra over those colors such that \(\mathcal{K} = \mathcal{K}(S)\), where \(\mathcal{K}(S)\) is the class of all coalgebras \(U\) having the property that the unique coextension of any “coloring” of \(U\) factors through \(S\). On the other hand, Bart Jacobs [95] showed that for every set of equations \(E\) there is a subcoalgebra \(S\) as above such that \(\mathcal{K}(S)\) is exactly the class of all coalgebras satisfying \(E\). However, their results put together say only that a class of coalgebras defined by equations is closed under the three closure operations, but nothing about the other implication. Taking over Rutten’s result, Andrea Corradini [30] showed that the class of certain coalgebras (actually, certain hidden algebras) which satisfy a special\(^9\) coalgebraic equational specification, is closed under subalgebras, quotients and coproducts (or sums). Then he presented a counter-example showing that the other implication is not necessarily true. Working under a general categorical framework described in Section VI.B.1, we show that the notion of “closure under subalgebras” can

---

\(^7\)Closure under sources of morphisms means that the source of a morphism is in the class whenever its target is in the class. This notion is weaker than the closure under sub(co)algebras.

\(^8\)See Definition 140.

\(^9\)Special in the sense that equations in [30] are special cases of equations in the present paper. Actually, the main concern of Corradini’s paper [30] was to develop a complete set of inference rules for equational deduction in coalgebras. The result for which we cite his work here was only a side remark to Rutten’s covariety theorem [145].
be replaced by a weaker “closure under sources of morphisms” and that a new closure operation, namely “closure under representative inclusions” is needed in order to get a characterization result for classes of coalgebras definable by equations.

Peter Gumm and Tobias Schröder further investigated characterization properties of covarieties [83, 82]. They introduced the notion of complete covariety [83] (covariety closed under bisimulation) and gave a Birkhoff-like characterization making use of formulas over an appropriate language. Furthermore, Peter Gumm introduced the coequations [82] (elements of certain cofree coalgebras) and implications of coequations and provided characterization results for covarieties and quasicovarieties (classes of coalgebras closed only under quotients and coproducts) in a dual manner to those by Birkhoff. However, coequations as elements of cofree coalgebras are infinite structures and therefore, difficult to use in practical situations to specify systems. Our main concern is not to perfectly dualise the Birkhoff axiomatizability results, but to investigate the definitional power of the intuitive, well-understood and practical equations within a coalgebraic setting.

VI.B.1 Framework

In this subsubsection we establish the framework and the afferent notations for the rest of the section. It is based on previous observations that most categories of coalgebras have coproducts, final object and admit a weak inclusion system:

**Framework:** \[ \{ S \text{ is a category having coproducts, a final object } F, \text{ and} \]
\[ \text{a well-powered weak inclusion system } (I, \mathcal{E}) \}. \]

We claim that the framework above is general enough to be fulfilled by all categories of coalgebras of interest including those of monadic hidden algebras. The requirement that \((I, \mathcal{E})\) is well-powered is only a technical categorical one that is verified by all practical coalgebras. The objects in \(S\) can be thought of as systems and the subobjects of an object can be thought of as subsystems of that system.

**Notation:** Let \(\omega_A: A \to F\) denote the unique morphism from \(A\) to \(F\).

**Definition 127** For every class \(\mathcal{K}\) of objects in \(S\), let

- \(Q_{\mathcal{K}}\) be the set of all objects \(\omega_A(A)\) when \(A\) is in \(\mathcal{K}\),
• $U_K$ be the coproduct $\coprod_{Q \in \mathcal{Q}_K} Q$ and $\Pi^K_Q : Q \to U_K$ be the corresponding coprojection of each $Q \in \mathcal{Q}_K$.

• $F_K$ be the subobject $\omega_{U_K}(U_K)$ of the final object $F$.

Notice that $\mathcal{Q}_K$ is indeed a set because $(I, \mathcal{E})$ is well-powered and all $\omega_A(A)$ are subobjects of $F$.

**Proposition 128** If $\mathcal{K}$ is a class of objects in $\mathcal{S}$ then:

1. There is a unique morphism from each object in $\mathcal{K}$ to $F_K$, and
2. If $\mathcal{K}'$ is another class of objects in $\mathcal{S}$ such that $\mathcal{K} \subseteq \mathcal{K}'$ then $F_K \hookrightarrow F_{K'}$.

**Proof:**

1. For each $A$ in $\mathcal{K}$, the morphism $e_{\omega_A} : \Pi^K_{\omega_A(A)} e_{\omega_{U_K}}$ has the source $A$ and the target $F_K$. It is unique because the inclusion $F_K \hookrightarrow F$ is a monomorphism and $F$ is final in $\mathcal{S}$.

2. Obviously $\mathcal{Q}_K \subseteq \mathcal{Q}_{K'}$, so there is a unique morphism $f : U_K \to U_{K'}$ such that $\Pi^K_Q f = \Pi^K_{K'}$ for each $Q$ in $\mathcal{Q}_K$. Factor $\omega_{U_K}$ as $e; i$ and $\omega_{U_{K'}}$ as $e'; i'$. Notice that $\omega_{U_K} = f ; \omega_{U_{K'}}$, that is, $e; i = f ; e'; i'$. By the diagonal-fill-in property, there is a unique morphism $h : F_K \to F_{K'}$ such that $e; h = f ; e'$ and $h ; i' = i$. By the right-cancellable property it follows that $h$ is an inclusion. \qed

### VI.B.2 Closures

Closures under certain operations and relationships between their combinations are explored in this section. Lemma 130 shows that a class of objects closed under coproducts and quotients has a final object and Lemma 133 gives a characterization of classes of objects closed under coproducts, quotients and sources of morphisms.

**Definition 129** A class $\mathcal{K}$ of objects in $\mathcal{S}$ is **closed under coproducts** iff the coproduct of any set of objects in $\mathcal{K}$ is in $\mathcal{K}$, and is **closed under quotients** iff any quotient of an object in $\mathcal{K}$ is in $\mathcal{K}$. Given a class $\mathcal{K}$, let $\mathcal{C}(\mathcal{K})$ denote the smallest class including $\mathcal{K}$ which is closed under coproducts and let $\mathcal{H}(\mathcal{K})$ denote the smallest class including $\mathcal{K}$ which is closed under $\mathcal{E}$-morphisms. \qed
Actually, it is easy to observe that $C(K) (H(K))$ is exactly the class of objects in $S$ which are coproducts (quotients) of objects in $K$. Notice that $K$ is closed under coproducts (quotients) if and only if $K = C(K) (K = H(K))$.

**Lemma 130** If $K$ is closed under coproducts and quotients then it has a final object $F_K$ which is a subobject of $F$. Moreover, $F_K$ is unique with this property.

**Proof:** First, notice that $Q_K \subseteq H(K) = K$. Then $U_K \in C(K) = K$, so $F_K \in H(K) = K$. By 1 in Proposition 128, $F_K$ is final in $K$. If $F' \hookrightarrow F$ is another final object in $K$ which is a subobject of $F$ then there are two morphisms $f : F_K \rightarrow F'$ and $f' : F' \rightarrow F_K$, so by the right-cancelable property (2 in Proposition 105) $f$ and $f'$ are inclusions, that is $F' = F_K$. Therefore $F_K$ is unique. □

**Definition 131** A class $K$ of objects in $S$ is **closed under sources of morphisms** iff for any morphism $f : A \rightarrow B$ in $S$, if $B \in K$ then $A \in K$. Given a class $K$, let $S(K)$ denote the smallest class closed under sources of morphisms which includes $K$. Given a subobject $G$ of $F$, the **sink of $G$** is the class $S(G)$. $K$ is called a **sink** iff there is a subobject $G$ of $F$ such that $K = S(G)$. ■

It is not difficult to observe that $S(K)$ is the class of all sources of morphisms of target in $K$. Notice that $K$ is closed under sources of morphisms if and only if $K = S(K)$.

**Proposition 132** If $K$ is a sink then there exists a unique subobject $G$ of $F$ such that $K = S(G)$. Moreover, $G$ is final in $K$.

**Proof:** Let $G, G'$ be two subobjects of $F$ such that $K = S(G) = S(G')$. Then there are $g : G \rightarrow G'$ and $g' : G' \rightarrow G$ such that $g; \omega_{G'} = \omega_G$ and $g'; \omega_G = \omega_{G'}$. Since $\omega_G$ and $\omega_{G'}$ are inclusions, by the right-cancelable property (2 in Proposition 105), $g$ and $g'$ are inclusions, so $G = G'$. Since $K = S(G)$, there is a morphism from every object in $K$ to $G$. That morphism is unique because $F$ is final in $S$ and the inclusion $G \hookrightarrow F$ is a monomorphism. □

The following lemma establishes a necessary and sufficient condition in order for a class of objects to be a sink:
Lemma 133 A class $\mathcal{K}$ is a sink iff it is closed under coproducts, quotients and sources of morphisms.

**Proof:** First, assume that $\mathcal{K}$ is closed under coproducts, quotients and sources of morphisms. By Lemma 130, $F_\mathcal{K}$ is final in $\mathcal{K}$, so $\mathcal{K} \subseteq S(F_\mathcal{K})$. On the other hand, since $\mathcal{K}$ is closed under sources of morphisms, one gets $S(F_\mathcal{K}) \subseteq S(\mathcal{K}) = \mathcal{K}$. Therefore $\mathcal{K} = S(F_\mathcal{K})$, that is $\mathcal{K}$ is a sink.

Conversely, assume $\mathcal{K} = S(G)$ for some subobject $G$ of $F$. Then $\mathcal{K}$ is closed under sources of morphisms because for every morphism $f: A \to B$ with $B$ in $\mathcal{K}$ there exists the morphism $f; z_B: A \to G$, where $z_B: B \to G$ is the unique morphism from $B$ to $G$. If $e: A \to B$ is an $E$-morphism with $A$ in $\mathcal{K}$, then by the diagonal-fill-in property for the commutative diagram $e; \omega_B = z_A; \omega_G$ (where $z_A: A \to G$ is the unique morphism from $A$ to $G$ in $\mathcal{K}$), there is a (unique) morphism $h: B \to G$ such that $e; h = z_A$ and $h; \omega_G = \omega_B$. Hence $B$ is in $S(G) = \mathcal{K}$. Now, let us consider a set $\{A_j\}_{j \in J}$ of objects in $\mathcal{K}$, let $C$ be the coproduct $\coprod_{j \in J} A_j$ and let $\Pi_j: A_j \to C$ be the corresponding coprojections. Then there is a (unique) morphism $z_C: C \to G$ such that $\Pi_j; z_C = z_{A_j}$ for each $j \in J$. Therefore $C$ is in $S(G) = \mathcal{K}$. Consequently, $\mathcal{K}$ is closed under coproducts, quotients and sources of morphisms. □

We take the liberty to omit some parentheses when multiple closures are involved. For example, $SCSH(\mathcal{K})$ is nothing else than $S(C(S(H(\mathcal{K}))))$. $C$, $H$ and $S$ have all the properties of closure operators in set theory, that is:

**Proposition 134** For any class $\mathcal{K}$ of objects in $S$,

1. Extensivity: $\mathcal{K} \subseteq C(\mathcal{K})$, $\mathcal{K} \subseteq H(\mathcal{K})$ and $\mathcal{K} \subseteq S(\mathcal{K})$;
2. Monotonicity: If $\mathcal{K}'$ is another class of objects such that $\mathcal{K} \subseteq \mathcal{K}'$ then $C(\mathcal{K}) \subseteq C(\mathcal{K}')$, $H(\mathcal{K}) \subseteq H(\mathcal{K}')$ and $S(\mathcal{K}) \subseteq S(\mathcal{K}')$; and
3. Idempotency: $CC(\mathcal{K}) = C(\mathcal{K})$, $HH(\mathcal{K}) = H(\mathcal{K})$ and $SS(\mathcal{K}) = S(\mathcal{K})$.

**Proof:** This proof is very easy; we let it as an exercise for the interested reader. □

The following proposition shows some relationships between closures under combinations of operators:

**Proposition 135** For any class $\mathcal{K}$ of objects in $S$,
1. \( \textbf{CH}(\mathcal{K}) \subseteq \textbf{HC}(\mathcal{K}) \);
2. \( \textbf{CS}(\mathcal{K}) \subseteq \textbf{SC}(\mathcal{K}) \);
3. \( \textbf{HS}(\mathcal{K}) \subseteq \textbf{SH}(\mathcal{K}) \).

\textbf{Proof:} 1. Let \( A \) be an object in \( \textbf{CH}(\mathcal{K}) \). Then there is a family \( \{ B_j \}_{j \in J} \) of objects in \( \mathcal{K} \) and a family \( \{ e_j : B_j \to C_j \}_{j \in J} \) of \( \mathcal{E} \)-morphisms such that \( \{ \Pi_j : C_j \to A \}_{j \in J}, A \) is the coproduct of \( \{ C_j \}_{j \in J} \). Let \( \{ \gamma_j : B_j \to B \}_{j \in J}, B \) be a coproduct of the family \( \{ B_j \}_{j \in J} \). Notice that \( B \in \mathbf{C}(\mathcal{K}) \). Then there is a unique morphism \( f : B \to A \) such that \( \gamma_j ; f = e_j ; \Pi_j \) for each \( j \in J \). Factor \( f \) as \( e; i \) with \( e : B \to f(B) \) in \( \mathcal{E} \) and \( i : f(B) \to A \) in \( \mathcal{I} \).

![Diagram]

By the diagonal-fill-in property, there is a unique morphism \( h_j \) such that \( e_j ; h_j = \gamma_j ; e \) and \( h_j ; i = \Pi_j \) for each \( j \in J \). Since \( \{ \Pi_j : C_j \to A \}_{j \in J}, A \) is a coproduct, there is a unique \( h : A \to f(B) \) such that \( \Pi_j ; h = h_j \) for each \( j \in J \). Since \( \Pi_j ; (h; i) = \Pi_j ; 1_A \) for all \( j \in J \), one gets that \( h; i = 1_A \), so by Proposition 105, \( i \) is an identity. Therefore \( f = e \in \mathcal{E} \), that is, \( A \) is in \( \textbf{HC}(\mathcal{K}) \).

2. Let \( A \) be in \( \textbf{CS}(\mathcal{K}) \). Then there is a family \( \{ B_j \}_{j \in J} \) of objects in \( \mathcal{K} \) and a family \( \{ f_j : C_j \to B_j \}_{j \in J} \) of morphisms such that \( \{ \Pi_j : C_j \to A \}_{j \in J}, A \) is a coproduct of \( \{ C_j \}_{j \in J} \). Let \( \{ \gamma_j : B_j \to B \}_{j \in J}, B \) be a coproduct of the family \( \{ B_j \}_{j \in J} \). Notice that \( B \in \mathbf{C}(\mathcal{K}) \). Then there is a morphism \( f : A \to B \) such that \( \Pi_j ; f = f_j ; \gamma_j \) for each \( j \in J \). Therefore, \( A \) is in \( \textbf{SC}(\mathcal{K}) \).

3. Let \( A \) be in \( \textbf{HS}(\mathcal{K}) \). Then there are an object \( B \) in \( \mathcal{K} \), a morphism \( f : C \to B \) and an \( \mathcal{E} \)-morphism \( e : C \to A \). Factor \( \omega_B \) (the unique morphism from \( B \) to the final object \( F \)) as \( e_{\omega_B} ; i_{\omega_B} \), with \( e_{\omega_B} : B \to \omega_B(B) \) in \( \mathcal{E} \) and \( i_{\omega_B} : \omega_B(B) \to F \) in \( \mathcal{I} \). Notice that \( \omega_B(B) \) is in \( \textbf{H}(\mathcal{K}) \). By the diagonal-fill-in property, there is a unique \( h : A \to \omega_B(B) \) such that \( e ; h = f ; e_{\omega_B} \) and \( h ; i_{\omega_B} = \omega_A \). Therefore, \( A \) is in \( \textbf{SH}(\mathcal{K}) \). \( \square \)

The sink generated by a class of objects can be obtained by first taking the closure under coproducts, then the closure under quotients and finally the closure under sources of morphisms:
Theorem 136  For any class $\mathcal{K}$ of objects in $\mathcal{S}$,

1. $\text{SHC}(\mathcal{K})$ is the smallest sink which extends $\mathcal{K}$, and

2. $\mathcal{K}$ is a sink iff $\mathcal{K} = \text{SHC}(\mathcal{K})$.

Proof:  1. By Proposition 134 one gets $\mathcal{K} \subseteq \text{SHC}(\mathcal{K})$. It follows from Proposition 135 and Proposition 134 that $\text{SHC}(\mathcal{K})$ is closed under coproducts, quotients and sources of morphisms and therefore, by Lemma 133, $\text{SHC}(\mathcal{K})$ is a sink. If $\mathcal{K}'$ is another sink extending $\mathcal{K}$ then by monotonicity (see Proposition 134) and by the closure of $\mathcal{K}'$ under coproducts, quotients and sources of morphisms, $\text{SHC}(\mathcal{K}) \subseteq \text{SHC}(\mathcal{K}') = \mathcal{K}'$.

2. If $\mathcal{K} = \text{SHC}(\mathcal{K})$ then by 1, $\mathcal{K}$ is a sink. Conversely, if $\mathcal{K}$ is a sink then by Lemma 133, $C(\mathcal{K}) = H(\mathcal{K}) = S(\mathcal{K}) = \mathcal{K}$. Therefore, $\text{SHC}(\mathcal{K}) = \text{SH}(\mathcal{K}) = S(\mathcal{K}) = \mathcal{K}$.  \[\square\]

Similar closure properties have been investigated in [83]. We do not need to consider the closure under subobjects in our approach (it is replaced by the closure under sources of morphisms), but it is worth mentioning (see [83]) that the closure operators under subobjects and coproducts, and under subobjects and quotients, respectively, commute in the framework of coalgebra.

VI.B.3  The Main Result

Now, we introduce an axiomatization of sentences and satisfaction that has a dual flavor to Căzănescu’s truth systems [32]. We try to do it as generally as possible, to capture different versions equation and satisfaction of coalgebra and hidden algebra, abstracting the properties in Propositions 113 and 121 within the following

Assumption:  From now on in the section, let $E: |\mathcal{S}| \to \text{Set}$ be a map such that:

- $E(B) \subseteq E(A)$ whenever there is a morphism $f: A \to B$,
- $E(A) \subseteq E(B)$ whenever there is an $E$-morphism $e: A \to B$, and
- $\bigcap_{j \in J} E(A_j) \subseteq E(\prod_{j \in J} A_j)$ for each set of objects $\{A_j\}_{j \in J}$ in $\mathcal{S}$, where $\prod_{j \in J} A_j$ is their coproduct.

If it is more convenient, $E(A)$ can be read “all equations satisfied by $A$”, whatever the notions of equation and satisfaction are. Also, we can say “$A$ satisfies $\Gamma$” or “$\Gamma$ is satisfied
by $A^*$ whenever $\Gamma \subseteq E(A)$. Unlike institutions [55] which formalize categorically the notions of satisfaction, sentence and model keeping them distinct and providing the satisfaction condition as an interaction between them, the map $E : |S| \to Set$ catches all three important notions together, skipping the signatures. Thus it is more abstract from a certain point of view, but because of its three axioms there is little chance to be applied in this form to other non-coalgebraic situations.

Since we proved the validity of the three axioms for equations having at most one hidden variable (Proposition 121), they also hold for special cases of these equations, such as equations containing only transitions (terms of hidden sort having exactly one occurrence of the hidden variable), or only observations (terms of visible sort having exactly one occurrence of the hidden variable) as in [30].

**Proposition 137** The following stronger assertions hold:

1. $E(A) = E(B)$ whenever there is an $\mathcal{E}$-morphism $e : A \to B$, 
2. $\bigcap_{j \in J} E(A_j) = E(\coprod_{j \in J} A_j)$ for each set of objects $\{A_j\}_{j \in J}$ in $\mathcal{S}$.

**Proof:** Both are immediate from the assumption, noticing that the existence of the coprojections $\Pi_{A_j} : A_j \to \coprod_{j \in J} A_j$ implies that $E(\coprod_{j \in J} A_j) \subseteq E(A_j)$ for each $j \in J$. \(\square\)

**Definition 138** If $\Gamma \subseteq E(A)$ then $A$ is called a $\Gamma$-object. A class $\mathcal{K}$ of objects in $\mathcal{S}$ is **definable** iff there is a set $\Gamma$ such that $\mathcal{K}$ contains exactly all the $\Gamma$-objects. \(\blacksquare\)

Translated in the (co)algebraic language, the definition above says nothing else than $\mathcal{K}$ is definable iff there exists a set of equations such that $\mathcal{K}$ contains exactly the systems satisfying those equations.

The next example shows that the three closure operations introduced so far are not sufficient to characterize definable classes of objects, and so it motivates Definition 140.

**Example 139** Let us consider a modified version of the familiar Flag example [65], where $\Sigma$ has a hidden sort, three methods, $up$, $down$ and $rev$, and one attribute $up?$, but in which the data algebra has three values, $true$, $false$ and $unknown$, and a unary operation $not$ such that $not(true) = false$, $not(false) = true$ and $not(unknown) =$
unknown. Having in mind the standard three equations of the Flag specification, namely $(\forall x) \ up?(up(x)) = \text{true}$, $(\forall x) \ up?(down(x)) = \text{false}$ and $(\forall x) \ up?(rev(x)) = \text{not}(up?(x))$, we can define three functions $f_1, f_2, f_3: C_{\Sigma}[z] \to D$ as follows (a dot $\bullet$ in front of a context means that the three functions differ in that context):

<table>
<thead>
<tr>
<th>$C_{\Sigma}[z]$</th>
<th>$f_1$</th>
<th>$f_2$</th>
<th>$f_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bullet$ $up?(z)$</td>
<td>true</td>
<td>false</td>
<td>unknown</td>
</tr>
<tr>
<td>$up?(up(z))$</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>$up?(down(z))$</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>$\bullet$ $up?(rev(z))$</td>
<td>false</td>
<td>true</td>
<td>unknown</td>
</tr>
<tr>
<td>$up?(up(up(z)))$</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>$up?(up(down(z)))$</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>$up?(up(rev(z)))$</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>$up?(down(up(z)))$</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>$up?(down(down(z)))$</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>$up?(down(rev(z)))$</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>$up?(rev(up(z)))$</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>$up?(rev(down(z)))$</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>$\bullet$ $up?(rev(rev(z)))$</td>
<td>true</td>
<td>false</td>
<td>unknown</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
</tbody>
</table>

Since $F_{\Sigma,up}(f_1) = f_1$, $F_{\Sigma,up}(f_2) = f_1$, $F_{\Sigma,up}(f_3) = f_1$, $F_{\Sigma,down}(f_1) = f_2$, $F_{\Sigma,down}(f_2) = f_2$, $F_{\Sigma,down}(f_3) = f_2$, and $F_{\Sigma,rev}(f_1) = f_2$, $F_{\Sigma,rev}(f_2) = f_1$, $F_{\Sigma,rev}(f_3) = f_3$, we deduce that $A = \{f_1, f_2\}$ and $B = \{f_1, f_2, f_3\}$ can be organized as hidden subalgebras of $F_{\Sigma}$, and $A$ is a proper subalgebra of $B$.

We claim that $B$ satisfies every equation satisfied by $A$. Since the attribute $up?$ can distinguish the three functions $f_1, f_2, f_3$, it suffices to treat only the equations of visible sort. If $(\forall z) \ t = t'$ is such an equation then notice that $t$ and $t'$ are contexts in $C_{\Sigma}(z)$, so for every $\theta: \{z\} \to B$ (i.e., for every function in $\{f_1, f_2, f_3\}$), $\theta(t)$ and $\theta(t')$ are those values in the table above found on the columns labeled by $\theta(z)$ and on the lines labeled by $t$ and $t'$, respectively. Since there are no two lines in the table having the same values ($true$, $false$ or $unknown$) for $f_1$ and the same values for $f_2$, but distinct values for $f_3$, we deduce that $B$ satisfies every equation satisfied by $A$.

Now, let $\mathcal{K} = S(A)$ be the sink of $A$ and notice that $B \notin \mathcal{K}$ (otherwise there should be a morphism $B \to A$ which by right-cancellability would be an inclusion, contradiction). Since every system in $\mathcal{K}$ satisfies all equations satisfied by $A$ (see 2. in Proposition 113), we get that $B$ satisfies all equations satisfied by all systems in $\mathcal{K}$,
which is closed under the three closure operations introduced so far. Therefore, classes of objects closed under coproducts, quotients and sources of morphisms are not necessarily definable by equations.

The following introduces the notion of representative inclusion, needed to characterize definable classes.

**Definition 140** An inclusion $G \hookrightarrow G'$ of subobjects of $F$ is called **representative** iff $E(G) = E(G')$. A class $\mathcal{K}$ of objects in $S$ is **closed under representative inclusions** iff $G'$ is in $\mathcal{K}$ whenever $G \hookrightarrow G'$ is a representative inclusion and $G$ is in $\mathcal{K}$. Given a class $\mathcal{K}$, let $\mathbf{R}(\mathcal{K})$ be its closure under representative inclusions.

Notice that only the inclusion $E(G) \subseteq E(G')$ is important, the other one being immediately inferred from the Assumption. We prefer to let $E(G) = E(G')$ in the definition above to emphasize that $G$ and $G'$ can not be distinguished equationally. The inclusion $A \hookrightarrow B$ in Example 139 is representative.

**Lemma 141** Let $A, B \in |S|$ such that $E(B) \subseteq E(A)$. Then $A \in \text{SRH}(B)$.

**Proof:** Factor the unique morphism $\omega_B$ from $B$ to $F$ as $e_{\omega_B}; i_{\omega_B}$, with $e_{\omega_B}: B \to \omega_B(B)$ in $\mathcal{E}$ and $i_{\omega_B}: \omega_B(B) \hookrightarrow F$ in $\mathcal{I}$. Notice that $\omega_B(B) \in \mathbf{H}(B)$. Let $C$ be the coproduct $A \coprod \omega_B(B)$ and let $\Pi_A: A \to C$ and $\Pi_{\omega_B(B)}: \omega_B(B) \to C$ be the two coprojections. Factor $\omega_C: C \to F$ as $e_{\omega_C}; i_{\omega_C}$. Since there is exactly one morphism from $\omega_B(B)$ to $F$, one gets $i_{\omega_B} = \Pi_{\omega_B(B)}; e_{\omega_C}; i_{\omega_C}$, and by the right cancelable property, $\Pi_{\omega_B(B)}; e_{\omega_C}$ is an inclusion. Let $i$ denote this inclusion. We claim that $i$ is representative. Indeed, $\omega_B(B) \hookrightarrow F$, $\omega_C(C) \hookrightarrow F$, and

\[
E(\omega_C(C)) = E(\omega_B(B) \coprod A) \quad (1 \text{ in Fact 137})
\]
\[
= E(\omega_B(B)) \cap E(A) \quad (2 \text{ in Fact 137})
\]
\[
= E(B) \cap E(A) \quad (1 \text{ in Fact 137})
\]
\[
= E(B) \quad (1 \text{ in Fact 137})
\]
\[
= E(\omega_B(B)) \quad (1 \text{ in Fact 137})
\]
Therefore, \( \omega_C(C) \in \mathbf{RH}(B) \). Since there is a morphism from \( A \) to \( \omega_C(C) \), namely \( \Pi_A; e_{\omega_C} \), one obtains that \( A \in \mathbf{SRH}(B) \).

The following proposition gives inclusions between different closures under combinations of operators containing \( \mathbf{R} \):

**Proposition 142** For any class \( \mathcal{K} \) of objects in \( \mathcal{S} \),

1. \( \mathbf{RSR}(\mathcal{K}) \subseteq \mathbf{SRH}(\mathcal{K}) \),
2. \( \mathbf{CR}(\mathcal{K}) \subseteq \mathbf{SRHC}(\mathcal{K}) \),
3. \( \mathbf{HR}(\mathcal{K}) \subseteq \mathbf{SRH}(\mathcal{K}) \).

**Proof:**

1. Let \( A \) be an object in \( \mathbf{RSR}(\mathcal{K}) \). If \( A \in \mathbf{SR}(\mathcal{K}) \) then by Fact 134, \( A \in \mathbf{SRH}(\mathcal{K}) \). If \( A \in \mathbf{RS}(\mathcal{K}) \) then either \( A \in \mathbf{S}(\mathcal{K}) \), case in which obviously \( A \in \mathbf{SRH}(\mathcal{K}) \), or there exist \( B \in \mathcal{K} \) and \( f : C \rightarrow B \) such that \( C \hookrightarrow A \) is a representative inclusion. Then by Assumption, \( E(B) \subseteq E(C) \), and hence \( E(B) \subseteq E(A) \). Now, by Lemma 141, \( A \in \mathbf{SRH}(B) \). Now, suppose that \( A \notin \mathbf{RS}(\mathcal{K}) \) and \( A \notin \mathbf{SR}(\mathcal{K}) \). Then there exist \( B \in \mathcal{K} \), a representative inclusion \( B \hookrightarrow C \) and a morphism \( f : C' \rightarrow C \) such that \( C' \hookrightarrow A \) is a representative inclusion. Then \( E(B) = E(C) \subseteq E(C') = E(A) \), so by Lemma 141, \( A \in \mathbf{SRH}(B) \). Consequently, \( A \in \mathbf{SRH}(\mathcal{K}) \).

2. Let \( A \in \mathbf{CR}(\mathcal{K}) \). Then there exist a family \( \{B_j\}_{j \in J} \) of objects in \( \mathcal{K} \), a family \( \{i_j : B_j \rightarrow C_j\} \) of representative inclusions, and a family \( \{C_{j'}\}_{j' \in J'} \) of objects in \( \mathcal{K} \) such that \( A = \coprod_{j \in J} C_j \prod_{j' \in J'} C_{j'} \). Let \( B \) be the coproduct \( \coprod_{j \in J} B_j \prod_{j' \in J'} C_{j'} \), and notice that \( B \in \mathbf{C}(\mathcal{K}) \). Then

\[
E(B) = \bigcup_{j \in J} E(B_j) \bigcup_{j' \in J'} E(C_{j'}) \quad \text{(Fact 137)}
\]

\[
= \bigcup_{j \in J} E(C_j) \bigcup_{j' \in J'} E(C_{j'}) \quad \text{(} B_j \hookrightarrow C_j \text{ is representative)}
\]

\[
= E(A) \quad \text{(Fact 137)}
\]

By Lemma 141, \( A \in \mathbf{SRH}(B) \). Therefore, \( A \in \mathbf{SRHC}(\mathcal{K}) \).

3. Let \( A \in \mathbf{HR}(\mathcal{K}) \). Then either \( A \in \mathbf{H}(\mathcal{K}) \) with a trivial conclusion or there is a subobject \( B \) of \( F \) in \( \mathcal{K} \), a representative inclusion \( B \hookrightarrow C \) and an \( \mathcal{E} \)-morphism \( e : C \rightarrow A \). Thus \( E(B) = E(C) = E(A) \), and by Lemma 141, \( A \in \mathbf{SRH}(B) \). Consequently, \( A \in \mathbf{SRH}(\mathcal{K}) \). \( \square \)
Theorem 143  Given a class $\mathcal{K}$ of objects in $\mathcal{S}$,

1. $\text{SRHC}(\mathcal{K})$ is the smallest sink extending $\mathcal{K}$ which is closed under representative inclusions, and

2. $\mathcal{K}$ is a sink closed under representative inclusions iff $\mathcal{K} = \text{SRHC}(\mathcal{K})$.

Proof:  1. Obviously $\mathcal{K} \subseteq \text{SRHC}(\mathcal{K})$. Furthermore, $\text{SRHC}(\mathcal{K})$ is a sink and is closed under representative inclusions because, by Propositions 142, 135 and 134,

- $\text{S}(\text{SRHC}(\mathcal{K})) = \text{SSRHC}(\mathcal{K}) = \text{SRHC}(\mathcal{K})$,
- $\text{H}(\text{SRHC}(\mathcal{K})) = \text{HSRHC}(\mathcal{K}) \subseteq \text{SHRHC}(\mathcal{K}) = \text{SSRHHHC}(\mathcal{K}) = \text{SRHC}(\mathcal{K})$,
- $\text{C}(\text{SRHC}(\mathcal{K})) = \text{CSRHC}(\mathcal{K}) \subseteq \text{SCRHC}(\mathcal{K}) \subseteq \text{SSRHC}(\mathcal{K}) \subseteq \text{SSRHHHC}(\mathcal{K}) = \text{SRHC}(\mathcal{K})$,
- $\text{R}(\text{SRHC}(\mathcal{K})) = \text{RSHRC}(\mathcal{K}) \subseteq \text{SRHHHC}(\mathcal{K}) = \text{SRHC}(\mathcal{K})$.

If $\mathcal{K}'$ is a sink which extends $\mathcal{K}$ and is closed under representative inclusions, then $\text{SRHC}(\mathcal{K}) \subseteq \text{SRHC}(\mathcal{K}') = \mathcal{K}'$.

2. It is straightforward and follows immediately from 1. \qed

Definition 144  A subobject $G$ of $F$ is called maximal iff every subobject $G'$ of $F$ with $E(G) \subseteq E(G')$ is a subobject of $G$. A maximal sink is the sink of a maximal object. ■

Therefore, $G$ is maximal iff it is the greatest subobject of $F$ satisfying $E(G)$.

Lemma 145  A sink is maximal iff it is closed under representative inclusions.

Proof:  Let $S(G)$ be a maximal sink and let $A \hookrightarrow B$ be a representative inclusion with $A$ in $S(G)$. Since there is a morphism from $A$ to $G$ we get $E(G) \subseteq E(A)$, and since $A \hookrightarrow B$ is representative we get $E(A) = E(B)$. Hence $E(G) \subseteq E(B)$, so by the maximality of $G$, $B \hookrightarrow G$. Therefore $B \in S(G)$, that is $S(G)$ is closed under representative inclusions.

Conversely, let $S(G)$ be closed under representative inclusions and let $G'$ be a subobject of $F$ such that $E(G) \subseteq E(G')$. Let $C$ denote the coproduct $G' \amalg G$. Then by the Assumption, $E(G) = E(G) \cap E(G') = E(C)$. Moreover, $E(C) = E(\omega_C(C))$ because $\omega_C(C)$ is a quotient of $C$. Since there exists a morphism from $G$ to $\omega_C(C)$, by the right-cancelable property we obtain $G \hookrightarrow \omega_C(C)$. Therefore $G \hookrightarrow \omega_C(C)$ is a representative
inclusion, and so $\omega_C(C)$ is in $S(G)$. Since there exists a morphism from $G'$ to $\omega_C(C)$ and $S(G)$ is closed under sources of morphisms, $G'$ is also in $S(G)$. By the right-cancelable property, the unique morphism from $G'$ to $G$ is an inclusion. \hfill \Box

Now we are ready to formulate the main result of this section:

**Theorem 146** The following assertions are equivalent for any class of objects $\mathcal{K}$:

1. $\mathcal{K}$ is definable,
2. $\mathcal{K}$ is a maximal sink,
3. $\mathcal{K}$ is closed under coproducts, quotients, sources of morphisms and representative inclusions,
4. $\mathcal{K} = \text{SRHC}(\mathcal{K})$.

**Proof:** The equivalence between 2 and 3 follows immediately from Lemma 145 and Lemma 133. The equivalence between 3 and 4 follows from Theorem 143.

Let us show that 2 implies 1. For that, let $\mathcal{K} = S(G)$ with $G$ a maximal object. We claim that $\mathcal{K}$ is $E$-defined by $\Gamma = E(G)$. Indeed, if $B$ satisfies $\Gamma$ then $\omega_B(B)$ also satisfies $\Gamma$, and since $G$ is maximal we get that $\omega_B(B) \hookrightarrow G$. Hence there is a morphism from $B$ to $G$, i.e., $B \in S(G) = \mathcal{K}$.

Finally, let us show that 1 implies 3. Let $\mathcal{K}$ be $E$-defined by $\Gamma$. It is immediate from the Assumption that $\mathcal{K}$ is closed under sources of morphisms, quotients and coproducts. $\mathcal{K}$ is also closed under representative inclusions since $\Gamma \subseteq E(G') = E(G)$ for each representative inclusion $G \hookrightarrow G'$.

**Conclusion and Discussion:** Most categories of coalgebras of interest fall under our easy to check framework including monadic hidden algebra, but it is not restricted to only coalgebras. We explored closure properties, introducing the notion of sink of a subsystem $G$ of the final system as the class of all systems from which there is a morphisms to $G$. Of particular interest is a characterization of sinks (Lemma 133) saying that a class of systems is a sink if and only if it is closed under coproducts, quotients and sources of morphisms.

Then we presented an axiomatization for sentences as a map from systems to sets, giving for each system all the sentences it satisfies. Like in institutions, sentences...
can be basically everything satisfying our assumptions. Various kinds of equations with at most one hidden sorted variable verify our assumptions, giving us hope that other sentences of interest in coalgebra might also fall under our axiomatization.

Despite the fact that sinks are very large classes of objects (larger than “covarieties”) and are closed under many categorical operations (including coproducts, products, quotients, subobjects, sources of morphisms), they are still too small to characterize definable classes of systems. Example 139 shows a situation in which a sink (the sink of $A$) does not contain all the systems (for example $B$) satisfying the sentences satisfied by all its systems. Therefore, new non-standard closure operations were necessary in order to obtain Birkhoff-like results. Our solution was to introduce the notion of closure under representative inclusions which still involves equations; however, the situation in Example 139 does not give much hope in getting a clean categorical characterization. Then we showed that a class of systems is definable by sentences if and only if it is closed under coproducts, quotients, sources of morphisms and representative inclusions. Interestingly, the smallest class closed under the four operations which extends a given class $\mathcal{K}$, or in other words the class of all systems that satisfy the same sentences satisfied by all systems in $\mathcal{K}$, can be obtained from $\mathcal{K}$ taking its closure under coproducts, quotients, representative inclusions and sources of morphisms, in this order. This strengthens our hope that the results in this section can be used to prove the equational Craig interpolation for coalgebras, in the style of Rodenburg [141].

Nothing was done for conditional equations. It would be interesting to see which closures are not needed (if there are any) in order to characterize classes definable by conditional equations, or at least by equations with visible conditions.

Even if the coalgebraic aspect is destroyed if operations are allowed to have more than one hidden argument of hidden sort, some good properties of hidden algebra are still valid, such as, “the behavioral equivalence is the largest hidden congruence”. Can the results in this section, or at least part of them, be extended to the general framework of hidden logic?
VI.C Behavioral Abstraction is Information Hiding

Information hiding is an important technique in modern programming. Programmers and software engineers agree in unanimity that a crucial characteristic of the languages they use for implementations, such as C++, Java, etc., is the support these languages provide for both public and private features (types, functions). The public part is often called interface and is visible to all the other modules (classes, packages), while the private one is hidden outside the module, but can be internally used to implement the interface. Hiding implementation features allows not only an increased level of abstraction (thus avoiding details which are known to be a main source of confusion and errors), but also an increased potential to improve a given data representation without having to search through all of a large program for each place where details of the representation are used. We suggest work by Parnas [124] for more on the practical importance of hiding implementation details.

Information hiding is an important technique also in algebraic specification. Majster [105] suggested that algebraic specifications are practically limited because certain $\Sigma$-algebras cannot be specified as an initial $\Sigma$-algebra of a finite set of $\Sigma$-equations; later, Bergstra and Tucker [8] (see also work by Meseguer and Goguen [110] for a summary of related research) showed that any recursive $\Sigma$-algebra could be specified as the $\Sigma$-restriction of an initial $\Sigma'$-algebra of a finite set of $\Sigma'$-equations, for some finite $\Sigma'$ larger than $\Sigma$. Therefore, there are some interesting $\Sigma$-theories that do not admit finite $\Sigma$-specifications, but that are $\Sigma$-restrictions of finitely presented $\Sigma'$-theories for some $\Sigma \subseteq \Sigma'$. We suggest work by Diaconescu, Goguen and Stefaneas [41] and by Roșu [135] for logic paradigm independent approaches to information hiding and integration of it with other operations on modules. Work on module algebra by Bergstra, Heering and Klint [7] can also be of great interest.

In this section we show that any behavioral $\Sigma$-theory semantically is the $\Sigma$-restriction of an ordinary algebraic $\bar{\Sigma}$-specification for some $\Sigma \subseteq \bar{\Sigma}$, thus emphasizing once more the power of information hiding. More precisely, we show that for any behavioral specification $B = (\Sigma, \Gamma, E)$ there is some specification $\bar{B} = (\bar{\Sigma}, \bar{E})$ for some $\Sigma \subseteq \bar{\Sigma}$, such that a hidden $\Sigma$-algebra behaviorally satisfies $B$ iff it strictly satisfies $\Sigma \Box \bar{B}$ (which
is the \( \Sigma \)-theory of all \( \Sigma \)-theorems of \( \tilde{B} \), see [41, 135] for more detail). Moreover, \( \tilde{E} \) is finite whenever \( E \) is finite, and \( \tilde{B} \) can be generated automatically from \( B \). Thus an equational logic theorem prover, such as OBJ3 [80], can be used for behavioral proving. The examples in this section use OBJ3 notation on purpose, to avoid using BOBJ’s support for behavioral specification and verification.

The general idea of the work in this section is taken from [71], but the technical constructions in almost all definitions were radically changed; that’s because we wanted to emphasize the relationship between the results in this section and Hennicker’s context induction proof principle [86]. Previous work by Bidoit, Hennicker and Mikami [13, 14, 112] was a great source of inspiration for us. Before we proceed to the construction of \( \tilde{B} \), let us first introduce another notion of context\(^{10} \) which is dependent on a fixed hidden algebra:

Definition 147 Given a hidden subsignature \( \Gamma \) of \( \Sigma \) and a hidden \( \Sigma \)-algebra \( A \), a \((\Gamma, A)\)-context for \( s \) is a term in \( T_{\Gamma \cup A}(\bullet : s) \) having exactly one occurrence of \( \bullet \). A \((\Gamma, A)\)-context which is a term of visible result is called a \((\Gamma, A)\)-experiment. Let \( C^A_{\Gamma}[\bullet : s] \) and \( E^A_{\Gamma}[\bullet : s] \) denote the sets of \((\Gamma, A)\)-contexts and \((\Gamma, A)\)-experiments, respectively.

\( \blacksquare \)

Notice that, similarly to the \((\Sigma, D)\)-contexts introduced in Definition 117, the elements in \( A \) are added as constants\(^{11} \) thus being allowed to be used in contexts and experiments. Obviously, any hidden \( \Sigma \)-algebra can be regarded as a \((\Gamma \cup A)\)-algebra where the operations in \( \Gamma \) are interpreted as in \( A|_{\Gamma} \) and each constant \( a \in A \) is interpreted as the element \( a \in A \). Conceptually, the contexts in Definition 147 are instances of those in Definition 11, by replacing their variables different from \( \bullet \) with concrete values in \( A \). As expected, the \((\Gamma, A)\)-experiments generate the behavioral equivalence on \( A \):

Proposition 148 Given a hidden \( \Sigma \)-algebra \( A \) and \( a, a' \in A_s \), then \( a \equiv^A_{\Sigma, s} a' \) if and only if \( A_\gamma(a) =_v A_\gamma(a') \) for each \( \gamma \in E^A_{\Gamma,v}[\bullet : s] \).

Proof: We show that the relation \( \sim \) defined by \( a \sim_h a' \) iff \( A_\gamma(a) =_v A_\gamma(a') \) for each \( \gamma \in E^A_{\Gamma,v}[\bullet : s] \) the largest hidden \( \Gamma \)-congruence.

\(^{10}\)Besides those in Definitions 11 and 117.

\(^{11}\)It is interesting to observe that \( \Gamma \cup A \) can have more hidden sorts than \( \Gamma \).
Let $\sigma : h_1...h_kv_{k+1}...v_n \rightarrow s$ be any operation in $\Gamma$ (with its first $k$ arguments hidden), let $a_i, a'_i \in A_{h_i}$ such that $a_i \sim_{h_i} a'_i$ for $i = 1, ..., k$, let $d_i \in A_v$ for $i = k + 1, ..., n$, and let $v \in V$ and $\gamma \in E_{\Gamma,v}[\bullet : s]$ (if the sort $s$ is visible, delete all occurrences of $\gamma$ in the proof that follows and replace terms of the form $\gamma[t]$ by just $t$). Let $\gamma_i$ be the term $\gamma[\sigma(a'_1, ..., a'_{i-1}, \bullet : h_i, a_{i+1}, ..., a_k, d_{k+1}, ..., d_n)]$ for each $i = 1, ..., k$.

Because $a_i \sim_{h_i} a'_i$ one gets that $A_{\gamma_i}(a_i) = A_{\gamma_i}(a'_i)$. Letting $a$ and $a'$ denote the elements $A_\sigma(a_1, ..., a_k, d_{k+1}, ..., d_n)$ and $A_\sigma(a'_1, ..., a'_k, d_{k+1}, ..., d_n)$ respectively, notice that $A_\gamma(a) = A_{\gamma_1}(a_1), A_{\gamma_i}(a'_i) = A_{\gamma_{i+1}}(a_{i+1})$ for $i = 1, ..., k - 1$, and $A_{\gamma_k}a'_k = A_\gamma(a')$. Therefore $A_\gamma(a) = A_\gamma(a')$, and since $\gamma$ is arbitrary, we obtain that $a \sim s a'$, i.e., $\sim$ is preserved by $\sigma$, and so $\sim$ is a hidden $\Gamma$-congruence.

Because all operations in $\Gamma$ preserve hidden $\Gamma$-congruences, so do the terms in $E_{\Gamma,v}[\bullet : s]$. In particular, terms in $E_{\Gamma,v}[\bullet : s]$ take congruent elements to identities. Therefore any hidden $\Gamma$-congruence is included in $\sim$. \hfill $\Box$

### VI.C.1 Unhiding

Ordinary algebraic specifications can be associated to hidden specifications and behavioral specifications, and special ordinary algebras containing contexts as elements can be associated to hidden algebras. This subsection presents all these technical constructions and some of their basic properties, and the next subsection points their relationship with context induction.

#### Unhiding a Hidden Signature

An algebraic specification of “experiments” can be associated to any hidden signature as follows:

**Definition 149** Given a hidden signature $\Gamma$, let $\tilde{S}$ be the set $S \cup (H \rightarrow V)$, where $S$ is the set $V \cup H$ and $(H \rightarrow V)$ denotes the set of new sorts of the form $(h \rightarrow v)$ where $h \in H$ and $v \in V$. Let $\tilde{\Gamma}$ be the $\tilde{S}$-signature adding to $\Gamma$ the following operations:

- $\bullet \sigma : \tilde{S} \rightarrow (h \rightarrow v)$ for each\(^{12}\) $\sigma : \tilde{S} h \rightarrow v$ in $\Gamma$ with $h \in H$.

\(^{12}\)Since $\sigma$ can have more than one hidden argument, actually an operation $\sigma^k : \tilde{S} h_k \rightarrow (h_k \rightarrow v)$ is added for each $\sigma : \tilde{S} h_k \rightarrow v$ in $\Gamma$ and each $k = 1, ..., n$ s.t. $h_k \in H$. 

Grigore Rosu, PhD Thesis
University of California at San Diego, 2000
\( \sigma : (h \rightarrow v) \rightarrow (h \rightarrow v) \) for each \( v \in V \) and \( \sigma : h \rightarrow h' \) in the \( \Gamma \) with \( h, h' \in H \),

- \( \sigma : (h \rightarrow v) \rightarrow (h \rightarrow v) \) for each \( h \in H \) and \( v \in V \).

Furthermore, let \( E_{\Gamma} \) be the set of equations:

- \( (\forall Y, x : h) \sigma(Y)[x] = \sigma(Y, x) \) for each \( \sigma : h \rightarrow v \), and

- \( (\forall Y, z : (h' \rightarrow v), x : h) z[\sigma(Y)][x] = z[\sigma(Y, x)] \), for each \( \sigma : h \rightarrow h' \).

The ordinary specification \((\tilde{\Gamma}, E_{\Gamma})\) is called the unifying of \( \Gamma \). □

Intuitively, the sorts \((h \rightarrow v)\) stand for “experiments of sort \( v \) for sort \( h \)”. The operations \( \sigma : (h \rightarrow v) \rightarrow (h \rightarrow v) \) are curried versions of operations \( \sigma : h \rightarrow v \) of visible sorts in \( \Gamma \), their rôle being to produce elementary contexts \( \sigma(Y) \), where \( Y : h \) is an appropriate set of variables; the operations \( \sigma : h \rightarrow v \) generate experiments for sorts \( h \) from experiments for sorts \( h' \) by composing them with operations \( \sigma : h \rightarrow h' \); the operations \( \sigma : (h \rightarrow v) \rightarrow (h \rightarrow v) \) apply experiments to states. The first \( \tilde{\Sigma} \)-equation says that one-operation experiments evaluate exactly as the operation itself, while the second \( \tilde{\Sigma} \)-equation shows how a composed experiment \( z[\sigma(Y)] \) works: the state is first plugged into \( \sigma \) and then the whole thing into \( z \).

**Example 150** The following is the unifying of the hidden subsignature \( \Gamma \), containing only the membership operation \( \text{in} : \text{Elt Set} \rightarrow \text{Bool} \), of the signature of sets presented in Example 57, written in OBJ3 notation:

```obj
obj EXP-GAMMA-SET[X :: TRIV] is sort Set.
  op _in_ : Elt Set -> Bool.

sort (Set->Bool).
  op _in : Elt -> (Set->Bool).
  op _[.] : (Set->Bool) Set -> Bool.
var X : Set. var E : Elt.
  eq E in [X] = E in X.
endo
```

The operation \( \text{in} : \text{Elt} \rightarrow (\text{Set} \rightarrow \text{Bool}) \) is the curried version of the attribute membership, \( \text{in} \). Figure VI.1 represents the ADJ-diagram of the signature \( \tilde{\Gamma} \) of the specification above. Since there is no operation of hidden result, there is no operation of the form \( \sigma \cdot \] \) added to \( \tilde{\Gamma} \). Therefore, there is only one more operation, the “application” \( \sigma \cdot [.] \), and one equation which should be parenthesized like \( (\text{E in}) [X] = (\text{E in X}) \). □

\(^{13}\)Same observation as in footnote 12.
Example 151 The following OBJ3 specification is the unhiding of the hidden subsignature $\Gamma$ of streams (see Example 31) containing only two operations, the attribute head and the method tail:

```obj
EXP-GAMMA-STREAM[X :: TRIV] is sort Stream .
  op head : Stream -> Elt .
  op tail : Stream -> Stream .

sort (Stream->Elt) .
  op head : -> (Stream->Elt) .
  op _[tail] : (Stream->Elt) -> (Stream->Elt) .
  op _[_] : (Stream->Elt) Stream -> Elt .
  var Z : (Stream->Elt) . var X : Stream .
  eq head [X] = head(X) .
  eq Z[tail] [X] = Z[tail(X)] .
endo
```

with the graphical representation for its signature, $\tilde{\Gamma}$, depicted in Figure VI.2. Notice that since $\Gamma$ contains both attributes and methods, the signature $\tilde{\Gamma}$ has operations of each of the three kinds presented in Definition 149, and equations of both kinds.

Unhiding a Hidden Algebra

A many-sorted algebra can be associated to any hidden algebra as follows:

Definition 152 Given a hidden subsignature $\Gamma$ of $\Sigma$ and a hidden $\Sigma$-algebra $A$, let $\tilde{A}$ be the $(\Sigma \cup \tilde{\Gamma})$-algebra\(^{14}\) defined by:

- $\tilde{A}|_{\Sigma} = A$, that is $\tilde{A}$ extends $A$,

\(^{14}\)To keep the notation simple, $\Gamma$ does not occur in the notation of $\tilde{A}$.
\begin{itemize}
  \item \( \check{A}_{(h \rightarrow v)} = E^A_{\check{\Gamma},v}[\bullet : h] \),
  \item \( \check{A}_\sigma : A^\check{\pi} \rightarrow \check{A}_{(h \rightarrow v)} \) is defined by \( \check{A}_\sigma(\check{\pi}) = \sigma(\check{\pi}, \bullet) \), for each \( \sigma : \check{\pi} \rightarrow (h \rightarrow v) \),
  \item \( \check{A}_{\sigma}[\cdot] : \check{A}_{(h' \rightarrow v)} \times A^\check{\pi} \rightarrow \check{A}_{(h \rightarrow v)} \) is defined by \( \check{A}_{\sigma}[\cdot](\gamma, \check{\pi}) = \gamma(\sigma(\check{\pi}, \bullet)) \), for each \( \sigma : \check{\pi} \rightarrow h' \), and
  \item \( \check{A}_{\cdot}[\cdot] : \check{A}_{(h \rightarrow v)} \times A_h \rightarrow A_v \) is defined by \( \check{A}_{\cdot}[\cdot](\gamma, a) = A_\gamma(a) \), for each \( \gamma \in E^A_{\check{\Gamma},v}[\bullet : h] \), and \( a \in A_h \).
\end{itemize}

The \((\Sigma \cup \check{\Gamma})\)-algebra \( \check{A} \) is called the \( \Gamma \)-unhiding of \( A \).

Therefore, the carrier of sort \((h \rightarrow v)\) of the \( \Gamma \)-unhiding \( \check{A} \) of a hidden \( \Sigma \)-algebra \( A \) consists of all \((\Gamma, A)\)-experiments of sort \( v \) for sort \( h \), and the operations in \( \check{\Gamma} - \Sigma \) are interpreted as expected. The following proposition says that the \( \Gamma \)-unhiding of a hidden \( \Sigma \)-algebra is a model of the unhiding specification of \( \Gamma \):

**Proposition 153** Given \( \Gamma \subseteq \Sigma \) and a hidden \( \Sigma \)-algebra \( A \), then \( \check{A} \models_{\Sigma \cup \check{\Gamma}} E_\Gamma \). 

**Proof:** Let \((\forall Y, x : h) \sigma(Y)[x] = \sigma(Y, x)\) be an equation in \( E_\Gamma \), like in Definition 149, and let \( \theta : Y \cup \{x\} \rightarrow \check{A} \) be any map. Then

\[
\theta(\sigma(Y)[x]) = \check{A}_{\cdot}[\cdot](\check{A}_\sigma(\theta(Y)), \theta(x)) = \check{A}_{\cdot}[\cdot](\sigma(\theta(Y), \bullet), \theta(x)) = A_\sigma(\theta(Y), \bullet)(\theta(x)) = A_\sigma(\theta(Y), \theta(x)) = \theta(\sigma(Y, x)).
\]
Now, let \((\forall Y, z : (h' \to v), x : h) \ z[\sigma(Y)][x] = z[\sigma(Y, x)]\) be an equation of the other type in \(E_\Gamma\), and let \(\theta : Y \cup \{z : (h' \to v), x : h\} \to \tilde{A}\) be any map. Then

\[
\theta(z[\sigma(Y)][x]) = \tilde{A}_{\downarrow}(\tilde{A}_{\downarrow}(\theta(z), \theta(Y)), \theta(x)) = \tilde{A}_{\downarrow}(\theta(z)(\sigma(\theta(Y), \bullet)), \theta(x)) = A_{\theta(z)}(\sigma(\theta(Y), \bullet))(\theta(x)) = A_{\theta(z)}(\sigma(\theta(Y), \theta(x))) = A_{\theta(z)}(A_{\sigma}(\theta(Y), \theta(x))) = \tilde{A}_{\downarrow}(\theta(z), A_{\sigma}(\theta(Y), \theta(x))) = \theta(z[\sigma(Y, x)])
\]

Therefore, \(\tilde{A} \models_{\Sigma, \tilde{\Gamma}} E_\Gamma\).

Unhiding a Behavioral Abstraction

The following definition has a similar flavor to the construction of \(E_\Gamma[e]\) and \(E_\Gamma[E]\) in Definition 20:

**Definition 154** If \(\Gamma\) is a hidden subsignature of \(\Sigma\) and \(e\) is a conditional \(\Sigma\)-equation \((\forall X) \ t = t'\) if \(C\) of visible condition, then we let \(\tilde{e}\) denote either the set of conditional \((\Sigma \cup \tilde{\Gamma})\)-equations

\[
\{(\forall X, z : (h \to v)) \ z[t] = z[t'] \text{ if } C \mid v \in V\}
\]

when the sort \(h\) of \(t\) and \(t'\) is hidden, or the set \(\{e\}\) when the sort of \(t\) and \(t'\) is visible. If \(E\) is a set of conditional \(\Sigma\)-equations of visible condition, then let \(\tilde{E}\) be the set of conditional \((\Sigma \cup \tilde{\Gamma})\)-equations \(\bigcup_{e \in E} \tilde{e}\); the ordinary algebraic specification \(\tilde{B} = (\Sigma \cup \tilde{\Gamma}, \tilde{E} \cup E_\Gamma)\) is called the unhiding of \(B\).

Notice that \(\tilde{B}\) is finite whenever \(B\) is finite, and that if \(B\) has no conditional equations then neither does \(\tilde{B}\).

**Example 155** The following is the unhiding of the behavioral specification of sets in Example 57:

\[
\text{obj SET}^\sim[X :: TRIV] \text{ is pr EXP-GAMMA-SET[X].}
\]

\[
\text{op empty} : \to \text{Set}.\]
op add : Elt Set -> Set.
ops (_U_) (_&_) : Set Set -> Set.
vars E E' : Elt. vars X X' : Set.
eq E in empty = false.
eq E in add(E', X) = (E == E') or (E in X).
eq E in X U X' = (E in X) or (E in X').
eq E in X & X' = (E in X) and (E in X').
endo

Notice that we imported the unhiding of Γ, EXP-GAMMA-SET, presented in Example 150. Since all the equations are of visible sort, they are left unchanged.

Example 156 The unhiding of streams started in Example 151, unlike the one of sets, contains equations like those in Definition 154 because the behavioral specification of streams in Example 31 contains equations of hidden sort:

obj STREAM*[X :: TRIV] is pr EXP-GAMMA-STREAM[X].
op _&_ : Elt Stream -> Stream.
var E : Elt. var X : Stream. var Z : (Stream->Elt).
eq head(E & X) = E.
eq Z[tail(E & X)] = Z[X].
endo

The last equation in STREAM* above says that “for any experiment Z, any element E and any stream X, the experiment Z returns the same element when evaluated in the states tail(E & X) and X”, which conceptually is the same as when we write the equation “tail(E & X) = X” in a behavioral specification.

Proposition 157 If \( \mathcal{B} = (\Sigma, \Gamma, E) \) is a behavioral specification with E of visible condition, e is a \( \Sigma \)-equation of visible condition and A is a hidden \( \Sigma \)-algebra, then

1. \( A \models_{\Sigma} e \iff \tilde{A} \models_{\Sigma \cup \Gamma} \tilde{e} \),
2. \( A \models \mathcal{B} \iff \tilde{A} \models \tilde{\mathcal{B}} \), and
3. \( \tilde{\mathcal{B}} \models \tilde{e} \) implies \( \mathcal{B} \models e \).

VI.C.2 Practical Importance: Context Induction

Proposition 157 says that in order to prove that \( \mathcal{B} \models e \), it suffices to prove by ordinary equational reasoning that \( \tilde{\mathcal{B}} \models \tilde{e} \). However, it turns out that this method is not useful in practice; induction seems to be needed for the sorts \( (h \rightarrow v) \). Therefore,
we suggest the method depicted in Figure VI.3 for proving behavioral properties of behavioral specifications by unhiding, where we used the notation $|=_{Ind}$ for the equational satisfaction relation restricted to those models for which induction is valid for the sorts $(h \rightarrow v)$. The correctness of this method is an immediate consequence of the following corollary of Proposition 157:

**Corollary 158** If $B$ and $e$ are like in Proposition 157, then $\bar{B} |=_{Ind} \bar{e}$ implies $B \equiv e$.

**Example 159** Let us prove that union is a commutative operation on sets. By the method above, we have to prove that the equation

$$(\forall x, x': \text{Set}, z : (\text{Set} \rightarrow \text{Bool})) \ z[x \cup x'] = z[x' \cup x]$$

follows by equational reasoning and induction from $\text{SET}^-$ presented in Example 155. If one tries to prove it by just equational reasoning, for example by

```plaintext
open $\text{SET}^-$. 
red (Z [X U X'] == Z [X' U X]) . ***$>$ should be false 
close
```

one fails; it can be readily seen that there is nothing one can do to equationally show the two terms equal. However, analyzing the sort of $z$ and the equations in $\text{EXP-\text{GAMMA-SET}}$ in Example 150, one can easily see that, in any model for which induction is valid, the only way for an element of sort $(\text{Set} \rightarrow \text{Bool})$ to exist is to be of the form “$e \in$”, for some $e : \text{Elt}$. Thus, the following proof score ends the proof:
open SET^\sim .
  op z : \rightarrow (Set\rightarrow\text{Bool}) .
  op e : \rightarrow \text{Elt} .
  eq z = e \text{ in} .
  red (z [X \cup X'] == z [X' \cup X]) . \text{*** should be true}
close

Notice that a degenerated induction was used; the induction step was not needed. □

**Example 160** Let us prove now that the equation $(\forall x:\text{Stream}) \text{head}(x) \& \text{tail}(x) = x$ holds in any model of streams. With the method proposed in this section, it suffices to show that

$$(\forall x:\text{Stream}, z:(\text{Stream} \rightarrow \text{Elt})) z[\text{head}(x) \& \text{tail}(x)] = z[x]$$

follows by induction on $z$ and equational reasoning from STREAM~^\sim in Example 156. Since an experiment of sort $(\text{Stream} \rightarrow \text{Elt})$ can be either the constant head or a term $z[\text{tail}]$ for some other experiment $z$, the following OBJ3 proof score shows the desired property by induction on experiments:

open STREAM^\sim .
  red head [\text{head}(X) \& \text{tail}(X)] == head [X] . \text{*** should be true}
  op z : \rightarrow (\text{Stream} \rightarrow \text{Elt}) .
  eq z[\text{head}(X) \& \text{tail}(X)] = z[X] .
  red z[\text{tail}] [\text{head}(X) \& \text{tail}(X)] == z[\text{tail}] [X] . \text{*** true}
close

It is interesting to notice that, simply by chance, the induction hypothesis was not needed in the second reduction above. That’s because of the special form of the second equation in STREAM^\sim. □

**Example 161** In this example we show that a doubly reversed stream of boolean elements is behaviorally equivalent to the initial stream, the same property efficiently proved by circular coinductive rewriting in Example 97. Considering the behavioral specification REV from Example 97, an immediate method to build the ordinary specification REV~^\sim is:

obj REV~^\sim is
  pr STREAM~^\sim[\text{BOOL}] * (\text{sort } (\text{Stream} \rightarrow \text{Elt}) \rightarrow (\text{Stream} \rightarrow \text{Bool})) .
  op rev : \text{Stream} \rightarrow \text{Stream} .
  var X : \text{Stream} .
  var Z : (\text{Stream} \rightarrow \text{Bool}) .
  eq \text{head}(rev(X)) = \text{not head}(X) .
  eq Z[\text{tail}(rev(X))] = Z[rev(\text{tail}(X))] .
end
A lemma is needed for the inductive proof below, namely:

```
obj LEMMA-REV is pr REV.
  var X : Stream . var Z : (Stream->Bool) .
  eq Z[rev(tail(rev(X)))] = Z[rev(rev(tail(X)))] .
don
```

This lemma follows immediately from the fact that \texttt{rev} is a congruent operation. Now the desired property can be proved by induction on experiments:

```
open LEMMA-REV.
  red head [rev(rev(X))] == head [X] . *** should be true
  op z : -> (Stream->Bool) .
  eq z [rev(rev(X))] = z [X] .
  red z[tail] [rev(rev(X))] == z[tail] [X] . *** should be true
isclosed
```

Notice that the induction hypothesis was needed this time.

The inductive technique used in the three examples above was nothing but Hennicker’s context induction \cite{86} (see also \cite{12} for related work). In fact, any proof technique for the ordinary algebraic specification \(\widehat{B}\) is allowed, as far as it is sound at least for the models \(\widehat{A}\) associated to hidden algebras. The proofs above were simple because the properties to be proved were trivial, there was only one hidden sort and the experiments were simply generated, but in general such proofs would be as awkward as are proofs by context induction (see \cite{44} for a discussion on some difficulties to apply context induction in practice). We find circular coinduction a very elegant automatic method to prove behavioral properties, and we prefer it to context induction. Our reason for the previous constructions and results is to support and justify the theoretical result presented in the next subsection.

\textbf{VI.C.3 Behavioral Abstraction as Information Hiding}

We are now ready for the main theoretical result of the section, namely that any behavioral abstraction having only equations of visible sort semantically is a special case of information hiding:

\textbf{Theorem 162} \textit{Under the same hypothesis as Proposition 157,}
1. \( A \models B \iff A \models \Sigma \Box \tilde{B} \), and

2. In the loose data hidden algebra case, \( \mathcal{B} \) and \( \Sigma \Box \tilde{B} \) have the same models.

What’s the real importance of this result and how it can be used in practical situations to do behavioral proofs other than by context induction, are still open questions for us.
VI.D On Consistency or “To Be or not to Be”

Unlike algebraic specifications for abstract data types, where any specification has models (both initial and final), it is very easy to write inconsistent behavioral theories. We first analyze a few examples showing how difficult the problem can be, then we introduce some definitions and give a necessary condition for a hidden theory to have models, and finally give some sufficient conditions. Our main goal is to find conditions which are as syntactic as possible, to ease the process of automated verification of consistency. The work in this section is essentially a debugging of work in [67], but still should be considered the start of a quest for better consistency criteria, rather than a final answer.

Example 163 Consider a hidden theory having only one hidden sort \( h \) and an equation \((\forall \emptyset) 0 = 1\), where the data algebra consists of the two element set \{0,1\}, with no operations. This hidden theory clearly has no models. It is worth noticing that if the equation was replaced by \((\forall x : h) 0 = 1\) where \( x \) is a variable of hidden sort \( h \), then because there are no hidden constants of sort \( h \), it would admit exactly one model, namely that with the carrier of sort \( h \) empty.

A lesson one should learn from the above example is that one should avoid data conflicts in order to insure consistency. Since we find the degenerate models of no practical value, we will exclude them:

Definition 164 A hidden theory is **consistent** iff it has a model with all carriers non-empty.

Sometimes inconsistency can come from computational properties of \( D \) combined with direct data conflicts, rather than just from direct data conflicts. One such example is the equation \((\forall \emptyset) 0 + 0 = 1\). This motivates the following:

Definition 165 A set of \( \Sigma \)-equations \( E \) is **\( D \)-safe** if and only if for any data \( d, d' \in D \), if \( E \cup D^* \models_{\Sigma} (\forall X) d = d' \) then \( d = d' \), where \( D^* \) is the set of all ground \( \Psi \)-equations of the form \((\forall \emptyset) t = d \) that are satisfied by \( D \).
For the rest of this section suppose that $P = (\Sigma, E)$ is a hidden theory over a data algebra $D$ with no empty carriers. The following gives a natural necessary condition for consistency:

**Proposition 166** If $P$ is consistent then $E$ is $D$-safe.

**Proof:** Let $M$ be a non-empty carrier model of $P$. For the sake of contradiction, suppose that there are distinct $d, d' \in D$ such that $E \cup D^* \models_{\Sigma} (\forall X) d = d'$. Because equational deduction is sound, we get $E \cup D^* \models_{\Sigma} (\forall X) d = d'$, and because $M \models_{\Sigma} E \cup D^*$, we get $M \models_{\Sigma} (\forall X) d = d'$. This is a contradiction because $d \neq d'$ and the carriers of $M$ are not empty. □

As we'll shortly see, $D$-safety is not a sufficient condition for consistency, because the conflicts can be more complex than in the Example 163.

**Example 167** Consider a hidden theory where $D$ is the algebra of natural numbers with addition, there is one hidden sort $h$, one hidden constant $x$, an attribute $a : h \rightarrow v$, and $E$ consists of one equation $(\forall \emptyset) 1 + a(x) = a(x)$. This hidden theory is inconsistent because there is no natural number $n$ as a value of $a(x)$ such that $1 + n = n$. Notice that it is not the case that $E \cup D^* \models_{\Sigma} (\forall \emptyset) n = m$ for some distinct natural numbers $n, m \in D$: for example, the model with natural numbers plus a new element\(^{15}\) $\infty$ as carrier of sort $v$ such that $n + \infty = \infty$ for any $n$ and $\infty + \infty = \infty$, and with any one-element set $\{*\}$ as carrier of sort $h$ such that $x$ is interpreted as $*$ and $a(x)$ as $\infty$, satisfies $E$ but doesn’t satisfy any of the equations $(\forall \emptyset) n = m$ for distinct $n, m$. ■

What we should learn from the previous example is that the use of operations in $\Psi$ (such as the addition in the previous example) on top of attributes in equations is dangerous, in the sense that this may induce conflicts on data, thus making a hidden theory inconsistent. The next example shows that data conflicts can appear even if operations in $\Psi$ are disallowed in equations and the equations are $D$-safe.

**Example 168** Consider a hidden theory again over a set $\{0, 1\}$ (of sort $v$) without operations as data algebra, having one hidden sort $h$ and one hidden constant $x$ of sort

\(^{15}\)Note that $\models_{\Sigma}$ allows $\Sigma$-models which do not protect the data.
two attributes \( a : h \to v \) and \( b : h \to v \), and three equations in \( E \), 
\[
(\forall \emptyset) \ b(x, a(x)) = 0, \\
(\forall \emptyset) \ b(x, 0) = 1, \text{ and } (\forall \emptyset) \ b(x, 1) = 1. 
\]
This is also inconsistent because if a model existed then \( a(x) \) would be either 0 or 1 and thus, by the first equation and either the second or the third, one would get that the equation \((\forall \emptyset) 0 = 1\) would be satisfied by that model, which is a contradiction. Notice though, that it is not the case that \( E \cup D^* \models_\Sigma (\forall \emptyset) 0 = 1 \) because, as in Example 167, there may be models satisfying \( E \cup D^* \) in which \( 0 \neq 1 \). ■

**Definition 169** A \( \Sigma \)-term is **weakly local** iff each proper subterm of it is a \( \Psi \)-term. We let \( WL_\Sigma(X) \) denote the many-sorted set of weakly local terms with variables in \( X \).


**Proposition 170** For each \( l \in WL_\Sigma \) there is a unique \( \overline{t} \in L_\Sigma \) such that \( D^* \models_\Sigma (\forall \emptyset) l = \overline{t} \).

**Proof:** By the completeness of equational deduction, it is equivalent to show that there is a unique local ground term \( \overline{t} \) such that we can derive \((\forall \emptyset) l = \overline{t}\) using the equations in \( D^* \). Notice that the only positions in \( l \) where an equation in \( D^* \) can be used in derivations is within a ground \( \Psi \)-term. Since a ground local \( \Sigma \)-term has no proper \( \Psi \)-term different from an element in \( D \), the problem reduces to showing that for any ground \( \Psi \)-term \( t \) there is a unique \( d \in D \) such that \( D^* \models_\Sigma (\forall \emptyset) t = d \). But this is immediate, because one can take \( d \) to be \( D_t \), the evaluation of \( t \) in \( D \), and if \( d \) is not unique, say there is another \( d' \in D \) with the same property, then it follows that \( D^* \models_\Sigma (\forall \emptyset) d = d' \) which is a contradiction (one can take any \( \Sigma \)-algebra \( M \) with \( M|_\Psi = D \) and notice that it obviously satisfies \( D^* \) but does not satisfy \((\forall \emptyset) d = d'\)). ■

Examples 167 and 168 suggest that it may be difficult to state consistency criteria for equations with non weakly local terms even if they are \( D \)-safe, as in the previous two examples. We will shortly see (Theorem 174) that \( D \)-safety plus weak locality of terms in equations are sufficient for consistency, when the equations are non-conditional as in the examples above. In fact, Theorem 174 works even for conditional equations whose conditions contain only \( \Psi \)-terms. But first, we briefly consider some difficulties that may appear when conditional equations are allowed.
Example 171 Let $P$ be the hidden theory with one visible sort $v$ and one hidden sort $h$, $D$ the set $\{0, 1\}$, one attribute $a : h \rightarrow v$, four hidden constants $x, x', y, y'$, and the equations $(\forall \emptyset) a(x) = 0$, $(\forall \emptyset) a(x') = 1$, $(\forall \emptyset) a(y) = a(y')$, and $(\forall \emptyset) x = x'$ if $y = y'$. Then $P$ is inconsistent, because if it had a model $M$ then $M_y \equiv M_{y'}$ in that model (since there is only one experiment, $a(\bullet)$), the condition is satisfied and so $M_x$ is behaviorally equivalent to $M_{x'}$; but this would imply that the two give the same value under the experiment $a(\bullet)$, and therefore $0 = 1$. It can be easily seen that these equations are $D$-safe and involve only local terms. ■

One may complain that the inconsistency above occurred because of the hidden condition of the equation. The next example shows that inconsistencies can appear even if the conditions are visible and the terms in equations are local.

Example 172 Consider a hidden theory with one visible sort $v$ and one hidden sort $h$, $D = \{0, 1\}$, one hidden constant $x$, one attribute $a : h \rightarrow v$ and two conditional equations, $(\forall \emptyset) a(x) = 0$ if $a(x) = 1$ and $(\forall \emptyset) a(x) = 1$ if $a(x) = 0$. Since one of the two conditions must be satisfied in any model, it follows that $0 = 1$, and so this hidden theory is inconsistent. It can also be shown that the equations are $D$-safe; we encourage the reader find an appropriate model for this purpose. ■

Examples 171 and 172 indicate some difficulties with conditional equations where the conditions contain operations not in $\Psi$. Actually, the last four examples suggest the following notion:

Definition 173 A conditional $\Sigma$-equation $(\forall X) t = t'$ if $C$ is weakly local iff $t, t'$ are weakly local terms and $C$ is either empty or contains only $\Psi$-terms. $E$ is weakly local iff it contains only weakly local equations. ■

Theorem 174 If $E$ is $D$-safe and weakly local, then $P$ is consistent.

Proof: We build a model of $P$. For each sort $v \in V$, let $d_v$ be an arbitrary but fixed element in $D_v$. Note that we can assume that $P$ contains hidden constants: if it doesn’t, then we can just add an artificial one, noticing that it does not affect the $D$-safety property of $E$. Let $M$ be the following hidden $\Sigma$-algebra:
\[ M|_\Psi = D, \]
\[ M_h = L_{\Sigma,h}, \text{ the set of local ground terms of sort } h, \text{ for each hidden sort } h, \]
\[ M_\sigma : M_h \times D^w \rightarrow M_{h'} \text{ is defined as } M_\sigma(l,d) = \sigma(l,d) \text{ for any method } \sigma : h \rightarrow h' \text{ and any } l \in L_{\Sigma,h} \text{ and } d \in D^w \text{ (notice that the locality is preserved), and} \]
\[ M_\rho : M_h \times D^w \rightarrow D \text{ is defined by} \]
\[ M_\rho(l,d) = \begin{cases} 
  d', & \text{when there is some } d' \text{ such that } E \cup D^* \models_\Sigma (\forall \emptyset) \sigma(l,d) = d', \\
  d_v, & \text{otherwise.} 
\end{cases} \]

Since \( E \) is \( D \)-safe, if a \( d' \) as above exists, then it is unique. Therefore, \( M \) is a well-defined hidden \( \Sigma \)-algebra. Let \( \alpha \) be the unique \( \Sigma \)-morphism \( T_\Sigma \rightarrow M \); notice that \( \alpha(d) = d \) for each \( d \in D \). Then it can be relatively easily shown by structural induction that for any weakly local ground term \( l \in WL_\Sigma \),

\[ \alpha(l) = \begin{cases} 
  \overline{l}, & \text{when the sort of } l \text{ is hidden,} \\
  d', & \text{when there is some } d' \text{ such that } E \cup D^* \models_\Sigma (\forall \emptyset) l = d', \\
  d_v, & \text{otherwise,} 
\end{cases} \tag{VI.1} \]

where \( \overline{l} \) is the unique local ground term given by Proposition 170, and that \( d' \) is unique when it exists. Notice that \( \alpha \) is surjective.

We claim that \( M \) behaviorally satisfies all the equations in \( E \). Since behavioral satisfaction of a weakly local hidden equation \((\forall X) t = t' \) \text{ if } \( C \) is equivalent to the satisfaction of an enumerable set of equations \((\forall X) c[t] = c[t'] \) \text{ if } \( C \) and since \( c[t] \) is weakly local whenever \( t \) is weakly local, we can consider that \( E \) contains only weakly local visible equations. Let \((\forall X) t = t' \) \text{ if } \( t_1 = t'_1, \ldots, t_n = t'_n \) be a weakly local visible equation in \( E \) and let \( \theta : X \rightarrow M \) be any map such that \( \theta(t_i) = \theta(t'_i) = d_i \) for all \( i = 1, \ldots, n \). Since \( \alpha : T_\Sigma \rightarrow M \) is surjective and \( \alpha(d) = d \) for any \( d \in D \), there is some assignment \( \phi : X \rightarrow T_\Sigma \) such that \( \theta = \phi \circ \alpha \) and \( \phi(x) = \theta(x) \) for any variable \( x \) of visible sort in \( X \). Therefore, \( \alpha(\phi(t_i)) = \alpha(\phi(t'_i)) = d_i \) for each \( i = 1, \ldots, n \), that is, \( D_{\phi(t_i)} = D_{\phi(t'_i)} = d_i \) where \( D_{\phi(t_i)} \) and \( D_{\phi(t'_i)} \) are the evaluations in \( D \) of the ground \( \Psi \)-terms \( \phi(t_i) \) and \( \phi(t'_i) \), respectively. In other words, \( D \models_\Psi (\forall \emptyset) \phi(t_i) = d_i \) and \( D \models_\Psi (\forall \emptyset) \phi(t'_i) = d_i \), so that

\[ D^* \models_\Sigma (\forall \emptyset) \phi(t_i) = \phi(t'_i) \text{ for each } i = 1, \ldots, n. \tag{VI.2} \]
On the other hand, since $|=_{\Sigma}$ is closed under substitution, it follows that

$$E |=_{\Sigma} (\forall \emptyset) \phi(t) = \phi(t') \text{ if } \phi(t_1) = \phi(t'_1), \ldots, \phi(t_n) = \phi(t'_n). \quad (VI.3)$$

Therefore, by (VI.2) and (VI.3), it follows that $E \cup D^* |=_{\Sigma} (\forall \emptyset) \phi(t) = \phi(t')$. Since $\phi(t)$ and $\phi(t')$ are both weakly local visible ground terms, it follows by (VI.1) that

$$\theta(t) = \alpha(\phi(t)) = \begin{cases} d', & \text{when there is some } d' \text{ such that } E \cup D^* |=_{\Sigma} (\forall \emptyset) \phi(t) = d', \\ d_v, & \text{otherwise.} \end{cases}$$

and that

$$\theta(t') = \alpha(\phi(t')) = \begin{cases} d'', & \text{when there is some } d'' \text{ such that } E \cup D^* |=_{\Sigma} (\forall \emptyset) \phi(t) = d'', \\ d_v, & \text{otherwise.} \end{cases}$$

It is clear that a $d'$ as above exists iff a $d''$ exists and, if this is the case, by the $D$-safety of $E$ it follows that $d = d''$. Hence, $\theta(t) = \theta(t')$. \qed

Theorem 174 provides a sufficient condition for a hidden theory to be consistent. Unfortunately, it is not mechanically checkable because the notion of $D$-safety is semantic. For this reason, we explore sufficient conditions for $D$-safety.

**Definition 175** A $\Sigma$-term rewriting system $R$ is $D$-confluent iff $R \cup R_{D^*}$ is confluent, where $R_{D^*} = \{ t \rightarrow d \mid D |=_{\Psi} (\forall \emptyset) t = d \}$ is the $\Psi$-rewriting system associated to $D^*$. ■

So a $\Sigma$-term rewriting system is $D$-confluent iff it is confluent modulo evaluations of ground $\Psi$-terms in $D$. This concept seems very natural for hidden reasoning over a fixed data universe, and we strongly believe that confluence criteria, such as orthogonality, can be appropriately adapted to $D$-confluence.

**Proposition 176** If $E$ can be oriented as a $D$-confluent $\Sigma$-term rewriting system with no rule having a datum $d \in D$ as its left side, then $E$ is $D$-safe.

**Proof:** For the sake of contradiction, suppose that $E$ is not $D$-safe, that is, that there are two distinct data $d, d' \in D$ such that $E \cup D^* |=_{\Sigma} (\forall X) d = d'$. Since $E$ can be oriented as a $D$-confluent $\Sigma$-term rewriting system, say $R_E$, by the completeness of equational satisfaction, we can deduce that $d \rightarrow (\rightarrow \cup \leftarrow)^* d'$ where $\rightarrow$ is the rewriting
relation induced by the $\Sigma$-term rewriting system $R_E \cup R_D^*$ and $\leftarrow$ is its inverse. Since $R_E \cup R_D^*$ is confluent, we get that in fact $d \rightarrow^* \rightarrow^* d'$, which is a contradiction because there is no rule having $d$ or $d'$ as left side in $R_E$, and the only rules in $R_D^*$ having $d$ or $d'$ as their left sides are $d \rightarrow d$ and $d' \rightarrow d'$.

\[\square\]

VI.E Incompleteness

The results in this section were published in [22] and are a consequence of our and other scientists’ effort to find complete deduction systems for various versions of behavioral logics. The fact that equational reasoning was not strong enough to derive equalities provable by $\Delta$-coinduction (see Sections IV.A and IV.C.4) and $\Delta$-coinduction was not strong enough to prove equalities provable by circular coinduction (see Section IV.E), and the fact that the Birkhoff-like equational axiomatizability for the coalgebraic version of hidden algebra was not pure (see Section VI.B), made us believe that behavioral satisfaction might not admit a complete axiomatization and thus motivate the present work.

We show the incompleteness of all the hidden logics presented or mentioned so far. In order to show that a logic does not admit a complete axiomatization it is necessary and sufficient to show that the satisfaction problem is not recursively enumerable. That is to say, there is no algorithm taking as input a specification and a sentence, and returning “Yes” if and only if the sentence is satisfied by all specification’s models, otherwise looping forever. The intuition behind this technique is that if a complete deduction system existed, then an algorithm would be to just generate and check all possible proofs. We show a stronger result, that there are finite specifications in all hidden logics, for which the satisfaction problem is $\Pi_2^0$-hard and thus is not recursively enumerable. Moreover, in many cases, the behavioral satisfaction problem is in $\Pi_2^0$, and thus the satisfaction problem can be $\Pi_2^0$-complete. The fact that behavioral satisfaction can be $\Pi_2^0$-hard means not only that there is no complete axiomatization for the behavioral satisfaction problem, but also that the complement of the behavioral satisfaction problem is not recursively enumerable. This means there is no way to algorithmically refute the sentences which are not behavioral consequences of some finite behavioral specifications. Notice that
this result is even less intuitive than the incompleteness result, because for any fixed computable model (in which the interpretations of operations are computable functions), the satisfaction problem is co-r.e.

The fact that the hidden logics are incomplete does not mean that something is wrong with the hidden algebra notation or that the question we addressed is ill-posed. For an analogy, the standard first-order theory of the natural numbers is of course well-known to be incomplete [46, 123, 142], but it is nonetheless the correct first-order theory for the natural numbers. From considerations such as Rice’s theorem [130, 123, 142], we should expect that any formal system which is strong enough to capture a significant amount of the behavior of computer systems is likely to be incomplete in some way; namely, it may lack expressive power, or may lack a complete axiomatization, or may admit unintended nonstandard models. Even if our results may seem negative, in fact they motivate work on new automatic proof techniques and algorithms to prove behavioral equivalences, such as \( \Delta \)-coinduction (see Section IV.C.4) and/or circular coinduction (see Section IV.E), which may be applicable for large classes of problems, in the same way in which induction is a very useful proof technique for natural numbers.

VI.E.1 Some Recursion Theory

We assume the reader familiar with basics of recursion theory, such as recursive predicates, recursively and co-recursively enumerable predicates (abbreviated r, r.e and co-r.e, respectively) as well as the very basics of arithmetic and analytic hierarchies [142]. Figure VI.4 is intended to remind the reader the few concepts needed later in this section. \( \Pi^0_2 \) is the class in the arithmetic hierarchy which properly extends both classes r.e and co-r.e, and contains predicates of the form

\[
P(a) := (\forall x)(\exists y) \ R(a, x, y)
\]

where \( R \) is a recursive predicate. There are many other classes in the arithmetic hierarchy, denoted by \( \Pi^0_n \) and \( \Sigma^0_n \) for \( n \geq 0 \), which properly extend each other, but we are not interested in them here. However, they are all properly included in the analytic hierarchy, denoted by \( \Pi^1_n \) and \( \Sigma^1_n \) for \( n \geq 1 \). Recall that \( \Pi^1_1 \) is the first level of the analytic hierarchy [142], and thus properly contains the arithmetic hierarchy. An elementary
Figure VI.4: Complexity hierarchy.

canonical representation of the class $\Pi^1_1$ is obtained as follows. Let $f : \mathbb{N} \rightarrow \mathbb{N}$. We define $\mathcal{f}(n)$ to be the sequence $\langle f(0), f(1), \ldots, f(n-1) \rangle$. Then a predicate $P(a)$ is in $\Pi^1_1$ if and only if it can be expressed in the form

$$P(a) := (\forall f : \mathbb{N} \rightarrow \mathbb{N})(\exists x) R(a, x, \mathcal{f}(x)),$$

where $R$ is some primitive recursive predicate.

Turing Machines

There are many equivalent definitions of Turing machines in the literature. We prefer one adapted from [142], and describe it informally in the sequel. The reader is assumed familiar with basics of Turing machines, the role of the following paragraphs being to establish our notations and conventions for the rest of the section.

Consider a mechanical device which has associated with it a tape of infinite length in both directions, partitioned in spaces of equal size, called cells, which are able to hold either a “0” or an “1” and are rewritable. The device examines exactly one cell at any time, and can perform any of the following four operations (or commands):

1. Write an “1” in the current cell;
2. Write a “0” in the current cell;
3. Shift one cell to the right;
4. Shift one cell to the left.

The device performs one operation per unit time, and this performance is called a step. Formally,
Definition 177 Let $Q$ be a finite set of internal states, containing a starting state $q_s$ and a halting state $q_h$. Let $B = \{0, 1\}$ be a set of symbols (or bits) and $C = \{0, 1, \to, \leftarrow\}$ be a set of commands. Then a (deterministic) Turing machine is a mapping from $Q \times B$ to $Q \times C$. ■

If the pair $(q, b)$ is taken to $(q', c)$, then we sometimes write $(q, b) \rightarrow (q', c)$. We assume that the tape contains only 0’s (or blanks) before the machine starts performing.

Definition 178 A configuration of a Turing machine is a triple consisting of an internal state and two infinite strings\footnote{Notice that the two infinite strings contain only 0’s starting with a certain cell.}, standing for the cells on the left and for the cells on the right, respectively. We let $(q, L|R)$ denote the configuration in which the machine is in state $q$, with left tape $L$ and right tape $R$. ■

Given a configuration $(q, L|R)$, the content of the tape is $LR$, which is infinite at both ends. By convention, the current cell is the first cell of the right string. We also let $(q, L|R) \rightarrow (q', L'|R')$ denote the configuration transition under one of the four commands. Given a configuration in which the internal state is $q$ and the examined cell contains $b$, and if $(q, b) \rightarrow (q', c)$, then exactly one of the following configuration transitions can take place:

1. $(q, L|bR) \rightarrow (q', L|cR)$ if $c = 0$ or $c = 1$;
2. $(q, L|bR) \rightarrow (q', Lb|R)$ if $c = \to$;
3. $(q, Lb'|bR) \rightarrow (q', L|b'bR)$ if $c = \leftarrow$.

The machine starts performing in the internal state $q_s$, so the initial configuration is $(q_s, \cdots 0\cdots 0|0\cdots 0\cdots )$. Sometimes, we wish to run a Turing machine on a specific input, say $x = b_1b_2\cdots b_n$. In this case, its initial config is $(q_s, \cdots 0\cdots 0|b_1b_2\cdots b_n0\cdots 0\cdots )$.

Definition 179 A Turing machine stops when it first gets to its halting state, $q_h$. ■

Therefore, a Turing machine carries out a uniquely determined succession of steps, which may or may not terminate. It is well-known that Turing machines can compute exactly the partial recursive functions [142].
**Totality Problem**

We claim that there exist some Turing machines $M$ for which the following problem, called Totality:

\[
\text{TOTALITY} = \begin{cases}
\text{INPUT: An integer } k \geq 0; \\
\text{OUTPUT: Does } M \text{ halt on all inputs } 1^j01^k \text{ for all } j \geq 0?
\end{cases}
\]

is $\Pi^0_2$-complete. It is obvious that Totality is in $\Pi^0_2$ for any Turing machine $M$. To show that it is $\Pi^0_2$-hard, we may choose $M$ to be a universal Turing machine such that on input $1^j01^k$, $M$ computes $f_k(j)$, where $f_k$ is the (partial) function computed by Turing machine with Gödel number $k$ under some canonical assignment of Gödel numbers to Turing machines. By appropriately choosing conventions for Turing machines, $f_k(j)$ is defined if and only if the Turing machine numbered $k$ halts on input $j$. Therefore, Totality$(k)$ has positive solution if and only if the function $f_k$ is total. But the set \{ $k$ | $f_k$ is total \} is $\Pi^0_2$-complete [142]. It follows that Totality is $\Pi^0_2$-complete.

We henceforth fix some choice of $M$ that makes the Totality problem $\Pi^0_2$-complete.

**VI.E.2 Two Boolean Hidden Logics**

As seen in Chapter III, the different approaches to behavioral specification and satisfaction can be classified in two broad categories, depending on whether a fixed data algebra is assumed for all models or not. In this subsection we introduce two very restrictive hidden logics as references for the two categories. The other hidden logics currently in use can be derived from one of these two boolean logics, by relaxing their syntactic constraints. The fact that the following two logics are syntactically very restrictive is a positive issue w.r.t. incompleteness, since their incompleteness implies the incompleteness of the other more relaxed hidden logics. We have tried to prove our incompleteness results for the weakest (most restrictive) logics, so as to make our results as general as possible.
Fixed Data

Here, we present an over-simplified version of fixed data hidden algebra logic and refer to it as boolean fixed data hidden algebra in the rest of the section.

Definition 180 A hidden signature is a \(\{v, h\}\)-sorted signature \(\Sigma\), where \(v\) is the visible sort and \(h\) is the hidden sort, consisting of:

- two constants of visible sort, true and false, often called the data;
- one attribute, i.e., an operation \(a : h \to v\);
- a finite set of methods, i.e., operations \(m : h \to h\).

A behavioral specification is a pair \((\Sigma, E)\), where \(\Sigma\) is a hidden signature and \(E\) is a set of \(\Sigma\)-equations (so \(\Gamma = \Sigma\)). A hidden \(\Sigma\)-algebra is a \(\Sigma\)-algebra \(A\) such that \(A_v = \{\text{true}, \text{false}\}\). A morphism of hidden \(\Sigma\)-algebras is a morphisms of \(\Sigma\)-algebras which is identity on \(v\).

Note that, in keeping with our desire to restrict the systems as much as possible, all operations (attributes and methods) are unary. Given a hidden signature \(\Sigma\), then an experiment \(\gamma\) is a term \(a(m_1(m_2(...(m_j(\bullet)...)))\), for some \(j \geq 0\) (not necessarily distinct) methods \(m_1, m_2, \ldots, m_j\). Therefore, the experiments consist of a series of methods changing the state followed by the attribute which “observes” the state. Notice that the experiments and the two visible constants are the only (modulo renaming of variables) terms of visible sort. If \(t\) is a term of sort \(h\), i.e., a term \(m'_1(m'_2(...(m'_i(x)...)))\) over a variable \(x\) of sort \(h\), then \(\gamma[t]\) is the term \(a(m_1(m_2(...(m_j(m'_1(m'_2(...(m'_i(x)...))))))...)))\).

Moreover, by Proposition 21, given a hidden \(\Sigma\)-algebra \(A\) then \(A \models_\Sigma (\forall x) t = t'\) iff \(A \models_\Sigma (\forall x) \gamma[t] = \gamma[t']\) for all experiments \(\gamma\). Behavioral satisfaction is ordinary satisfaction on equations of visible sort, i.e., if \(t\) and \(t'\) have visible sort, then \(A \models_\Sigma (\forall x) t = t'\) iff \(A \models_\Sigma (\forall x) t = t'\).

Definition 181 Let \(\models^{fd}_\Sigma\) denote the satisfaction relation of boolean fixed data hidden algebra.

Equational reasoning is sound for \(\models^{fd}_\Sigma\) [66], but not complete. We will show in the next subsection that actually there is no complete deduction system for behavioral
satisfaction. More precisely, we show that for some behavioral specifications \( \mathcal{B} = (\Sigma, E) \), the following problem:

\[
BSAT_B^{fd} = \begin{cases} 
\text{INPUT: A } \Sigma\text{-equation } e; \\
\text{OUTPUT: } \mathcal{B} \models^{fd} e ?
\end{cases}
\]

is not recursively enumerable even for finite \( E \).

**Loose Data**

A loose data hidden logic can be easily adapted from the boolean fixed data one presented in Definition 180, by removing \textit{false} from the signature together with the restrictions that \( A_v = \{true, false\} \) on models, and that the morphisms are identities on visible sorts. The notions of “experiment” and “behavioral satisfaction” remain unchanged. We’ll refer to this logic as semi-boolean loose data hidden algebra for the rest of the section.

**Definition 182** Let \( \models^{ld}_\Sigma \) denote the behavioral satisfaction relation of semi-boolean loose data hidden algebra.  

Equational reasoning is also sound for this slightly different behavioral satisfaction [40, 139], but not complete as we will shortly see. More precisely, we show that for some behavioral specifications \( \mathcal{B} = (\Sigma, E) \), the following problem:

\[
BSAT_B^{ld} = \begin{cases} 
\text{INPUT: A } \Sigma\text{-equation } e; \\
\text{OUTPUT: } \mathcal{B} \models^{ld} e ?
\end{cases}
\]

is not recursively enumerable even for finite \( E \). As a side point, this is achieved for a specification with only one visible constant, \textit{true}. However, even \textit{true} can be replaced by an attribute if the reader finds it inconvenient to have visible constants in the signature.

**VI.E.3 Behavioral Satisfaction is \( \Pi^0_2 \)-Hard**

In this subsection we show that all versions of behavioral satisfaction discussed so far in the present work are \( \Pi^0_2 \)-hard, so in particular, the associated logics do not admit complete deduction systems.
The strategy used is the expected one: reduction from a $\Pi^0_2$-complete problem to behavioral satisfaction. We chose Totality (see Subsection VI.E.1), for a (fixed) Turing machine $M$ which makes it $\Pi^0_2$-complete. Next we define a behavioral specification like in Subsection VI.E.2. Let

- $\Sigma$ be the following hidden signature:
  - $v, h$ are a visible and a hidden sort, respectively;
  - $true, false$ are two visible constants\(^{17}\)

**Attributes:**
- $WillStop$: $h \rightarrow v$;

**Methods:**
- $blank$: $h \rightarrow h$;
- $q$: $h \rightarrow h$ for each state $q \in Q$;
- $0_l, 1_l, 0_r, 1_r$, all of the form $h \rightarrow h$;
- $always$: $h \rightarrow h$;
- $More$: $h \rightarrow h$;

- $E$ be the finite set of equations:
  1. $(\forall x) \ blank(m(x)) = blank(x)$ for all methods $m$;
  2. $(\forall x) \ m(always(x)) = always(x)$ for all methods $m \neq blank$;
  3. $(\forall x) \ m(q(x)) = always(x)$ for any method $m \neq blank$ and state $q \in Q$, such that $m \neq More$ or $q \neq q_s$;
  4. $(\forall x) \ More(q_s(x)) = q_s(1_r(x))$;
  5. $(\forall x) \ b_l'(b_r(x)) = b_r(b_l'(x))$ for all $b, b' \in \{0, 1\}$;
  6. $(\forall x) \ 0_q(blank(x)) = blank(x)$ for any $d \in \{l, r\}$;
  7. $(\forall x) \ WillStop(always(x)) = true$;
  8. $(\forall x) \ WillStop(q_h(x)) = true$;

Now, for every transition $(q, b) \rightarrow (q', c)$, add:

9a. $(\forall x) \ WillStop(q(b_r(x)))) = WillStop(q'(c_r(x)))$ when $c = 0$ or $c = 1$,

9b. $(\forall x) \ WillStop(q(b_r(x)))) = WillStop(q'(b_l(x)))$ when $c = \rightarrow$,

9c. $(\forall x) \ WillStop(q(b_l'(b_r(x)))) = WillStop(q'(b_r'(b_r(x))))$ for $b' = 0$ and $b' = 1$,

\(^{17}\)In fact, only $true$ is needed for the loose data case.
Let \( \mathcal{B} \) be the behavioral specification \((\Sigma, E)\). Notice that \( \mathcal{B} \) is finite; more precisely, the number of operations in \( \Sigma \) is \( O(|Q|) \), and the number of equations in \( E \) is \( O(|Q|^2) \), where \(|Q|\) is the number of internal states of \( M \).

Intuitively, the behavioral specification \( \mathcal{B} \) above describes the Turing machine \( M \), and behavioral deduction in \( \mathcal{B} \) simulates the execution of \( M \): the hidden sort \( h \) stands for configurations of \( M \); the attribute \( \text{WillStop} \) is true on a configuration \( x \) iff \( M \) terminates when started with the configuration \( x \); the method \( \text{blank} \) generates the initial configuration, with only blanks on the tape; \( q(x) \) changes the state of the configuration \( x \) to \( q \); \( 0_l, 1_l, 0_r, 1_r \) adds a 0 or an 1 on the left or on the right tape, respectively; \( \text{More} \) adds one more 1 on the right tape; the method \( \text{always} \) lets the machine in an artificial configuration which always terminates. We hope that the following result makes our intention clearer:

**Lemma 183** If \( M \) stops on an input \( b_1b_2\cdots b_n \), then

\[
\mathcal{B} \equiv_{\text{id}} (\forall x) \text{WillStop}(q_s(b_1\textit{r}(b_2\textit{r}(\cdots(b_{n\textit{r}}(\text{blank}(x)))\cdots)))) = \text{true}.
\]

**Proof:** It can be easily seen that every configuration transition in \( M \) can be simulated by equational deduction using the equations (9a), (9b), and/or (9c); notice that the equation (5) may be needed to bring the operations \( b_r \) on top of the terms as arguments of \( q \) operations in order to be allowed to use the equations (9a), (9b), and (9c), and that the equation (6) can be used to generate more \( 0_l \) and \( 0_r \) operations when needed. Iterating this procedure, we get that \( \mathcal{B} \) satisfies the equation

\[
(\forall x) \text{WillStop}(q_s(b_1\textit{r}(b_2\textit{r}(\cdots(b_{n\textit{r}}(\text{blank}(x)))\cdots)))) = \text{WillStop}(q_h(t(\text{blank}(x))))
\]

for some appropriate sequence \( t \) of methods \( b_d \in \{0_l, 1_l, 0_r, 1_r\} \). The rest follows by equation (8).

Unlike in equational logics where every specification has models, it is very easy to write inconsistent behavioral specifications under the fixed data hidden algebra logic. For example, the equation \((\forall\emptyset) \text{true} = \text{false} \) has no models; however, notice that the equation \((\forall x : h) \text{true} = \text{false} \) has one model, the one in which the hidden carrier is empty. For more on consistency of behavioral specifications in hidden algebra, the interested reader is
referred to [66]. The next result says that the behavioral specification above is consistent.

Let $M$ be the following $\Sigma$-algebra:

- $M_v = \{\text{true}, \text{false}\}$;
- $M_h = ((Q \cup \bot) \times \text{String} \times \text{String}) \cup \square$ where $\square$ is a special new element;
- $M_{\text{WillStop}}(\square) = \text{true}$ for any strings $S$ and $S'$;
- $M_{\text{WillStop}}((\bot, S, S')) = \text{true}$;
- $M_{\text{WillStop}}((q, S, S')) = \text{true}$ iff the Turing machine $M$ halts when starts with the state $(q, \cdots 0 \cdots 0 | S | 0 \cdots 0 \cdots)$; otherwise it is $\text{false}$;
- $M_{\text{blank}}(X) = (\bot, \epsilon, \epsilon)$ for any element $X$ in $M_h$;
- $M_q(\square) = \square$ for any $q \in Q$;
- $M_q((\bot, S, S')) = (q, S, S')$ for any $q \in Q$ and any strings $S, S'$;
- $M_q((q', S, S')) = \square$ for any $q, q' \in Q$ and any strings $S, S'$;
- $M_{b_d}(\square) = \square$ for any $b_d \in \{0_l, 1_l, 0_r, 1_r\}$;
- $M_{b_d}((q, S, S')) = \square$ for any $q \in Q$, strings $S, S'$, and $b_d \in \{0_l, 1_l, 0_r, 1_r\}$;
- $M_{b_d}((\bot, S, S')) = (\bot, Sb, S')$ for any $b \in \{0, 1\}$ and strings $S, S'$, such that $b \neq 0$ or $S \neq \epsilon$;
- $M_{0_l}((\bot, \epsilon, S')) = (\bot, \epsilon, S')$ for any string $S'$;
- $M_{b_r}((\bot, S, S')) = (\bot, S, bS')$ for any $b \in \{0, 1\}$ and strings $S, S'$, such that $b \neq 0$ or $S' \neq \epsilon$;
- $M_{0_r}((\bot, S, \epsilon)) = (\bot, S, \epsilon)$ for any string $S$;
- $M_{\text{always}}(X) = \square$;
- $M_{\text{More}}(\square) = \square$;
- $M_{\text{More}}((\bot, S, S')) = \square$ for all strings $S, S'$;
- $M_{\text{More}}((q, S, S')) = \square$ for all $q \in Q - \{q_s\}$ and strings $S, S'$;
- $M_{\text{More}}((q_s, S, S')) = (q_s, S, 1S')$ for all strings $S, S'$;

**Proposition 184** $M$ is a hidden $\Sigma$-algebra and $M \models^{fd} B$.

The following proposition reduces the problem TOTALITY to behavioral satisfaction, both fixed data and loose data versions:

**Proposition 185** Given $k \geq 0$, let $e_k$ be the equation

$$(\forall x) \ q_s(0_r(1_r(1_r(\cdots (1_r(\text{blank}(x))\cdots)))))) = \text{always}(x),$$

where the method $1_r$ occurs $k$ times. Then the following are equivalent:
1. **Totality**\((k)\) is positive;
2. \(B \vDash_{ld} e_k\);
3. \(B \vDash_{fd} e_k\).

**Proof:** (i) \(\Rightarrow\) (ii). Let \(s_k\) be the term on the left-hand side of \(e_k\). Suppose that there exists a context \(c\) such that \(B\) does not (loosely) behaviorally satisfy the equation \((\forall x)\ c[s_k] = c[\text{always}(x)]\). Clearly, the context \(c\) must be a term having the form \(\text{WillStop}(m_1(m_2(\cdots(m_j(\bullet)\cdots)))\), where \(m_1, m_2, \ldots, m_j\) are methods.

If \(\text{blank}\) is among \(m_1, m_2, \ldots, m_j\), then by iteratively using the equation (1), \(B \vDash_{ld} (\forall x)\ c[s_k] = c[\text{always}(x)]\), contradiction. Therefore, \(\text{blank}\) does not occur in as a subterm of \(c\). If \(\text{always}\) is in \(c\), then by (2), \(B \vDash_{ld} (\forall x)\ c[s_k] = \text{WillStop}(\text{always}(t))\) and \(B \vDash_{ld} (\forall x)\ c[\text{always}(x)] = \text{WillStop}(\text{always}(t))\) for some appropriate term \(t\) of sort \(\text{lt}\). Therefore, \(B \vDash_{ld} (\forall x)\ c[s_k] = c[\text{always}(x)]\), contradiction. Therefore, \(\text{always}\) does not occur in \(c\). By (2) and (7), one can immediately see that \(B \vDash_{ld} (\forall x)\ c[s_k] = \text{true}\), contradiction. Therefore, \(m_j\) is the method \(\text{More}\). For each \(i \leq j\), let \(s_k^i\) be the hidden sorted term \(q_s(1_r(1_r(\cdots(1_r(0_r(1_r(1_r(\cdots(1_r(\text{blank}(x))\cdots))))))))\), with \(j - i\) occurrences of the operation \(1_r\), followed by an operation \(0_r\), and followed by \(k\) operations \(1_r\); notice that \(s_k^1 = s_k\).

If there is an index \(i < j\) such that \(m_i \neq \text{More}\) and \(m_{i+1}, \ldots, m_j\) are all \(\text{More}\), then by equation (4), \(B \vDash_{ld} (\forall x)\ c[s_k] = c[i[s_k^i]]\), where \(c_i\) is \(\text{WillStop}(m_1(m_2(\cdots(m_i(\bullet)\cdots)))\). Since \(m_i \neq \text{More}\), by equations (3), (2), and (7) as above, \(B \vDash_{ld} (\forall x)\ c_i[s_k^i] = \text{true}\), that is, \(B \vDash_{ld} (\forall x)\ c[s_k] = \text{true}\), contradiction.

Therefore, \(c\) must have the form \(\text{WillStop}(\text{More}(\text{More}(\cdots(\text{More}(\bullet)\cdots)))\), for \(j \geq 0\) occurrences of \(\text{More}\). Then by (4), \(B \vDash_{ld} (\forall x)\ c[s_k] = \text{WillStop}[s_k^0]\). Since the answer of **Totality**\((k)\) is positive, the Turing machine \(M\) stops for any input \(1^j01^k\) with \(j \geq 0\). Then by Lemma 183, \(B \vDash_{ld} (\forall x)\ \text{WillStop}[s_k^0] = \text{true}\). Consequently, \(B \vDash_{ld} (\forall x)\ c[s_k] = c[\text{always}(x)]\), contradiction again.

Hence, \(B\) satisfies the equation \((\forall x)\ c[s_k] = c[\text{always}(x)]\) for any context \(c\), that is, \(B \vDash_{ld} e_k\).

(ii) \(\Rightarrow\) (iii). Obvious, since any hidden \(\Sigma\)-algebra which is a model of \(B\) is also a loose model of \(B\), with the same behavioral equivalence on it.
(iii) $\Rightarrow$ (i). If $\mathcal{B} \equiv^{fd} \epsilon_k$ then by equational reasoning as above, $\mathcal{B}$ behaviorally satisfies $(\forall x) \text{WillStop}[s^0_{s_0}] = \text{true}$ for all $j \geq 0$. By Proposition 184, $\mathcal{M}$ satisfies the equation $(\forall x) \text{WillStop}[s^0_{s_0}] = \text{true}$, that is, $\mathcal{M}_{\text{WillStop}}((q_s, \epsilon, 1^j01^k))$ is true, which means that the Turing machine $M$ terminates on the input $1^j01^k$. Hence, the answer of $\text{TOTALITY}(k)$ is positive. $\square$

Now the main result of the section follows obviously, showing that the behavioral satisfaction and non-satisfaction problems cannot be mechanized in any algorithmical way.

**Theorem 186** In all versions of behavioral logics mentioned, there are behavioral specifications for which the behavioral satisfaction problem is $\Pi^0_2$-hard. Therefore, these logics do not have complete deduction systems. Moreover, none of these logics admits algorithms to refute false statements.

**Proof:** Since the problem $\text{TOTALITY}$ is $\Pi^0_2$-complete, then by Proposition 185, $\text{BSAT}^{fd}_{\mathcal{B}}$ and $\text{BSAT}^{dd}_{\mathcal{B}}$ are $\Pi^0_2$-hard. Since the other versions of behavioral logics are generalizations of either basic fixed-data hidden algebra or basic loose-data hidden algebra, they are also $\Pi^0_2$-hard. Since the class $\Pi^0_2$ strictly includes both the set of r.e. predicates and the set of co-r.e. predicates, the behavioral satisfaction and behavioral non-satisfaction problems in all the mentioned logics are not r.e., so there is no algorithm to prove a true statement or to refute a false statement in behavioral logics. $\square$

**VI.E.4 Some Hidden Logics are $\Pi^0_2$-Complete**

Once we know that all hidden logics mentioned are $\Pi^0_2$-hard, a natural next step is to find a place for these logics in the arithmetic hierarchy, if such a place exists. In this subsection, we show that some behavioral logics are $\Pi^0_2$-complete, and in the next subsection we show that some fixed data versions of hidden algebra are not even in the arithmetic hierarchy, they being $\Pi^1_1$-hard. Our work in this section should be viewed only as a starting point toward stronger characterization results of hidden logics.

As opposed to the previous subsection, our goal now is to find as general hidden logics as possible which are in the class $\Pi^0_2$, since this would imply that all hidden logics...
obtained as special instances of them are also in $\Pi_2^0$. The general idea is to choose those logics admitting complete deduction for visible equations. If this is the case, then the behavioral satisfaction problem becomes of the form:

**Input:** A behavioral specification $B = (\Sigma, E)$ and a $\Sigma$-equation $(\forall X) t = t'$;

**Output:** Is it the case that for every context $c$, there exists a proof $p$, such that $p$ proves $(\forall X) c[t] = c[t']$ in $B$?

which is a $\Pi_2^0$ statement.

**Fixed Data**

We analyze two special cases of fixed data hidden logics which are $\Pi_2^0$-complete. The first one considers that the data algebra is finite, and for this reason we call it *finite fixed data hidden algebra*. It obviously includes the boolean fixed data hidden algebra logic we presented in Subsection VI.E.2 whose data contained only two elements, *true* and *false*.

**Proposition 187** If $B$ is any behavioral specification in the sense of finite fixed data hidden algebra, then $BSAT_B^{fd}$ is in $\Pi_2^0$.

**Proof:** Let $\varphi$ be a first-order sentence that completely characterizes the finite data algebra $D$ up to isomorphism. Then $B \models^{fd} e$ holds if and only if $\{\varphi, E[\Gamma]\} \vdash \Gamma[e]$ (see Definition 21) where $\vdash$ means provability in first-order logic with equality. This is a $\Pi_2^0$ statement.

Now we consider another hidden logic, which we call *flat fixed data hidden algebra*, that generalizes Corradini’s coalgebraic equational logic [30]: a hidden signature is a $(V \cup H)$-sorted signature such that each operation either has exactly one argument of sort in $H$ and zero or more arguments of sorts in $V$, or is a constant from a fixed recursively enumerable set of constants $D$. We call it “flat” because $D$ has no algebraic structure on it, it is just a set. Corradini’s approach is slightly more restrictive, in the sense that it does not admit visible sorts as arguments of operations. The deduction system presented in [30], generating a relation $\vdash_C$, and its completeness theorem can be easily generalized to operations allowing visible parameters, thus obtaining the following:
**Theorem 188** Given a behavioral specification $\mathcal{B} = (\Sigma, E)$ in the sense of flat fixed data hidden algebra and a $\Sigma$-equation of visible sort $e$, then $\mathcal{B} \models^{fd} e$ if and only if $E \vdash^C e$. Therefore, $\text{BSAT}_{fd}^\mathcal{B}$ is in $\Pi_2^0$.

Notice that Corradini’s framework allows only visible equations in $E$, but this is not an inconvenience since $E$ can be replaced by an r.e. set of visible equations $\mathcal{E}[E]$ (see Proposition 21). In order to make this operation part of the deduction systems, we need to add a congruence inference rule to those in [30].

**Loose Data**

We claim that all versions of loose data behavioral satisfactions for which the equational reasoning is sound are in $\Pi_2^0$; these include all problems $\text{BSAT}_{ld}^\mathcal{B}$, where $\mathcal{B} = (\Sigma, \Gamma, E)$, for which all operations in $\Sigma$ are congruent. The reason is that Proposition 21 is applicable, reducing behavioral satisfaction to equational standard satisfaction. We assume that the set $E$ of equations is r.e. and that it contains only equations of visible condition.

**Proposition 189** $\text{BSAT}_{ld}^\mathcal{B}$ is in $\Pi_2^0$ for any $\mathcal{B}$ as above.

**Proof:** Let $\mathcal{B} = (\Sigma, E)$ be any loose-data behavioral specification for which the equational reasoning is sound, and $e$ be any $\Sigma$-equation. By Proposition 21, $\mathcal{B} \models^{ld} e$ iff $\mathcal{E}[E] \models \mathcal{E}[e]$, and by equational completeness theorem, iff $\mathcal{E}[E] \vdash \mathcal{E}[e]$, where now $\vdash$ is the equational derivation relation. Therefore, $\text{BSAT}_{ld}^\mathcal{B}$ is in $\Pi_2^0$. \qed

**VI.E.5 Others are not in the Arithmetic Hierarchy**

We saw that for finite data algebras or for infinite flat data, with no operations on it, the satisfaction problem is in $\Pi_2^0$. We now consider the complexity of behavioral satisfaction for fixed data hidden algebra in general, where the data can be any infinite algebra.

**Proposition 190** Given a behavioral specification $\mathcal{B} = (\Sigma, \Gamma, E)$ such that $\Sigma$ is enumerable, $E$ is arithmetic, and $D$ is an enumerable infinite data algebra with an arithmetic representation, then $\text{BSAT}_{fd}^\mathcal{B}$ is in $\Pi_1^1$. 
Proof: Any model of $\mathcal{B}$ can be encoded by a function $f: \mathbb{N} \rightarrow \mathbb{N}$. Thus, $\mathcal{B} \models e$ holds if and only if every $f: \mathbb{N} \rightarrow \mathbb{N}$ which codes a valid model of $\mathcal{B}$ satisfies the equations $\mathcal{E}_f[e]$. Since equational validity for the model defined by $f$ is arithmetic, the condition on $f$ is arithmetic. Therefore $\mathcal{B} \models e$ is $\Pi^1_1$. \hfill \Box

Now we show that there are some natural infinite data algebras for which the behavioral satisfaction problem becomes $\Pi^1_1$-complete. Let the data algebra $D$ be $(\mathbb{N}, 0, s, \_)$, the set of integers with the successor function. Almost any infinite data algebra effectively contains $D$; further, $D$ is well-known to have a decidable first-order theory.

**Theorem 191** There is a finite\(^{18}\) behavioral specification $\mathcal{B} = (\Sigma, E)$ in the sense of fixed data hidden algebra (over $D$ above), such that $\text{BSAT}^f_{\mathcal{B}}$ is $\Pi^1_1$-complete.

**Proof:** Fix some $\Pi^1_1$-complete predicate $A(x)$ expressed in the form described in Subsection VI.E.1. We construct a hidden signature and hidden equations and give a many-one reduction from the predicate $A(x)$ to the behavioral satisfiability problem.

In addition, to the visible sort of $\mathbb{N}$, we let our hidden signature have a single hidden sort. This hidden sort will act only vacuously as a source of dummy variables — we use $h$ to denote a variable of hidden sort.

The language includes a finite list of function symbols which have one hidden input and several visible inputs. These function symbols represent primitive recursive functions augmented with an extra, ignored, hidden input. For example, the functions corresponding to addition and multiplication are $\text{Add}(h, m, n)$ and $\text{Mult}(h, m, n)$ and the set of equations contains their defining equations, namely,

\[
\begin{align*}
\text{Add}(h, x, 0) & = x \\
\text{Add}(h, x, S(y)) & = S(\text{Add}(h, x, y)) \\
\text{Mult}(h, x, 0) & = 0 \\
\text{Mult}(h, x, S(y)) & = \text{Add}(h, \text{Mult}(h, x, y), x)
\end{align*}
\]

The $\text{Not}$ function, which implements negation if one thinks of zero as denoting truth and non-zero as denoting falsity, is defined by

\[
\text{Not}(h, 0) = S(0)
\]

\(^{18}\) That is, $\Sigma$ has a finite number of sorts and operations, and $E$ is a finite set of equations.
\[ \text{Not}(h, S(y)) = 0 \]

In a similar fashion, we include the primitive recursive functions for coding sequences: \(\langle \rangle\) is the code for the empty sequence, and \(\text{Concat}(h, w, a)\) is the operation for appending an integer \(a\) to the end of a sequence \(w\). We also include enough primitive recursive functions in the language so as to include the function \(p_T\) defined by

\[ p_T(h, x, y, z) = 0 \text{ iff } T(x, y, z). \]

Finally, we add function symbols \(f(h, x, y)\) and \(g(h, x, y)\) and equations that make \(g(h, x, y)\) equal to \(\overline{f}(h, x, y)\), namely,

\[
\begin{align*}
  g(h, x, 0) & = \langle \rangle \\
  g(h, x, S(y)) & = \text{Concat}(h, g(h, x, y), f(h, x, y)).
\end{align*}
\]

We do not add any equations restricting the function \(f\).

One final function symbol is needed: \(a(h, x)\) which is intended to equal 0 if and only if \(f\) witnesses the truth of \(A(x)\). For this we add an equation

\[ \text{Mult}(h, \text{Not}(h, g(h, x, y)), a(h, x)) = 0 \]

which enforces the condition

\[ g(h, x, y) = 0 \rightarrow a(h, x) = 0. \]

We take as the set \(E\) all the above equations, and let \(B\) be \((\Sigma_D, E)\). For \(n \geq 0\), let \(\underline{n}\) denote the term \(S^n(0)\) with value \(n\). Let \(e_n\) be the equation \(a(h, \underline{n}) = 0\). Then it is straightforward to verify that

\[ A(n) \text{ is true } \iff B \models e_n. \]

It is an interesting observation that the proof of the previous proposition uses a \(B\) with no “methods”, but only “attributes.” This implies that it is not really a result about hidden logics, but is instead a result about equational logic over a fixed algebra. Indeed, our proof can be recast as showing that in ordinary equational logic, it can be
\(\Pi_2^1\)-hard to decide equational satisfaction in \(\omega\)-models. Moreover, this technique can be used to transfer any hardness result concerning ordinary equational logic over a fixed domain \(D\) into a hardness result for hidden algebra over the fixed data algebra \(D\).

**Conclusion and Discussion:** We showed that for some behavioral specifications in any hidden logic currently in use, the satisfaction problem is \(\Pi_2^0\)-hard. Since the class \(\Pi_2^0\) properly includes both the classes of r.e. problems and co-r.e. problems, our result has two major implications. The first one is that there is no algorithm to prove true statements, in particular the hidden logics are incomplete. The second is that there is no algorithm to reject false statements.

Then we showed the \(\Pi_2^0\)-completeness for the finite fixed data and flat fixed data hidden algebra logics, respectively, and also for those loose data hidden logics for which the equational reasoning is sound. The behavioral satisfaction problem was shown to be \(\Pi_1^1\)-complete in some cases of fixed data hidden algebra logics over infinite data algebras.

There are still many cases of hidden logics which can be derived from the two logics we presented in Chapter III, which we did not analyze in this section. For example, what is the upper bound on the complexity of behavioral satisfaction in loose data behavioral specifications in which equational reasoning is not sound? How about hidden algebra logic over infinite data algebra but with all operations having exactly one hidden argument and no visible arguments? How important is the restriction to not allow operations having more than one hidden argument?
Chapter VII

Conclusion and Further Research

Hidden logic was presented in this thesis, as a generic name for various logics derived from or closely related to hidden algebra, giving sound rules for behavioral reasoning in all approaches, with a high potential for automation. We investigated a series of coinduction inference rules and algorithms automating coinduction and behavioral rewriting. Then we presented some theoretical results.

There is much interesting work left. One important feature we’d like to add to hidden logic is the support for parallel connection of modules. We think that it can be integrated with the other more classical operations on modules, such as importing, renaming, aggregation, parameterization and hiding.

Another interesting direction of further research is to use BOBJ to do proofs in the context of Proof Carrying Code [116], where the real bottleneck is that the producer has to generate a correctness proof for his/her code. We wrote a report [140] showing how producer’s task can be easier if he/she uses OBJ3. We believe that BOBJ can ease the proof task further, by allowing the more abstract behavioral proofs.

A practically very useful extension of hidden logics would be to allow hidden subtyping, thus increasing both its expressive power and its area of applicability. In fact, it is already implemented in BOBJ and we tacitly used it in Subsection V.E.4 (Letter < RegExp), but the theory is not yet developed. Having the work on order-sorted algebra as a case study [48, 56, 131], we expect the theory of hidden subtyping to be very technical.

Circular coinductive rewriting can be further improved. Notice that the pairs
of new terms in the circularity pool $\mathcal{C}$ have to be oriented as rewriting rules in order
to be used in reductions. At this time, we implemented a simple heuristic in BOBJ to
orient them, but there is no guarantee that smarter techniques are not needed. It may
be possible that more complicated proof strategies, like in CoClam [35], will be needed
as we find new examples. Also, some theory on circular coinductive rewriting seems to
be needed, saying how strong the method in fact is.

The $\Omega$-abstract rewriting systems seem promising abstract tools for rewriting.
We only applied them on behavioral rewriting and $\lambda$-calculus in Chapter V, but we hope
that they have a larger area of applicability.

It is known that both equational logics and first order logics verify some versions
of Craig interpolation property [31, 138]. We think that hidden logics also have the Craig
interpolation property, but we were not able to prove or disprove it yet.
Bibliography


