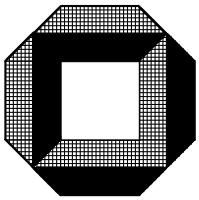
Universität Karlsruhe (TH)

Fakultät für Informatik

Institut für Programmstrukturen
und Datenorganisation



An Introduction to (Co)Algebras and (Co)Induction and their Application to the Semantics of Programming Languages

Sabine Glesner

August 2005

Technical Report No. 2005-22
University of Karlsruhe

Abstract

This report summarizes operational approaches to the formal semantics of programming languages and shows that they can be interpreted inductively by least fixed points as well as coinductively by greatest fixed points. While the inductive interpretation gives semantics to all terminating programs, the coinductive one defines moreover also a semantics for all non-terminating programs. This is especially important in areas where programs do not terminate in general, e.g. data bases, operating systems, or control software in embedded systems. The semantic foundations described in this report can be used to verify that transformations (e.g. in compilers) of such software systems are correct.

In the course of this report, coalgebras and coinduction are introduced, starting with a gentle intuitive motivation and ending with a detailed mathematical description within the notions of category theory.

Contents

1 Formal Semantics of Programming Languages

In mathematical logic, semantics is used to assign a formal meaning, a *semantics*, to syntactical elements. As a simple example, consider propositional logic. Propositional formulae are built from the logical constants **True** and **False**, from propositions denoted by capital letters $A, B, \ldots$, and from the logical connectives $\neg, \wedge, \vee$. One example for a propositional formula is $A \vee B$. Propositional semantics assigns meaning to such a formula by considering all possible assignments of truth values 0 and 1 to the propositions contained in that formula, typically by stating a truth table. For our simple example $A \vee B$, this truth table would be as follows:

$A \vee B$	$A = 0$	$A = 1$
$B = 0$	0	1
$B = 1$	1	1

This same principle underlies the formal semantics of programming languages. The term *formal* semantics indicates that this is a semantics in the mathematical way. Consider as example the following simple program from an imperative programming language:

$$x := 1;\ \textbf{while}\ x < 3\ \textbf{do}\ x := x + 1;$$

In principle, there are two possibilities to define the meaning of such a program. First, we can concentrate on its result and define its semantics as the value of x upon termination (the brackets "$[\![\ ]\!]$" are commonly used to denote the semantics of a syntactic construct):

$$[\![x := 1;\ \textbf{while}\ x < 3\ \textbf{do}\ x := x + 1;]\!] = (x = 3).$$

Secondly, we could consider the complete state transition sequence

$$[\![x := 1;\ \textbf{while}\ x < 3\ \textbf{do}\ x := x + 1;]\!] = (x =?, x = 1, x = 2, x = 3)$$

as program semantics. And, of course, there are many nuances between these two general possibilities. It depends on the intended purpose which semantics is to be chosen. For functional programming languages, the first choice seems to be well-suited while for imperative languages with their inherent concept of states, the second choice is better. In the context of this report, we apply the second possibility which understands programs as state transition systems. This is adequate for the verification of compilers because many programs in practice do not terminate (and are not designed to terminate, e.g. operating systems, control software, data base systems, etc.) and need to be compiled correctly by preserving their state transition behavior. This method of defining semantics of programming languages is called *operational semantics*.

Semantics is generally compositional. The semantics of a formula or a program is derived from the semantics of its direct subformulae or subprograms. In our example of the truth table for $A \vee B$, it does not matter if A and B are themselves formulae or if they are primitive propositions. This same principle is used in the semantics of programming languages but with one important extension: The semantics of a program is defined from the semantics of its direct

subprograms and, in recursive computations, also from its own semantics. Consider as example the syntactic construct of a while loop: "**while** b **do** S;". Its semantics is derived from the semantics of the conditioning expression b, from the semantics of the statements S in the loop body and, moreover, from its own semantics since it is executed again after the loop body has terminated. This is expressed with the following definition:

$$[\![\textbf{while } b \textbf{ do } S;]\!] = \text{if } [\![b]\!] \text{ then } ([\![S]\!]; [\![\textbf{while } b \textbf{ do } S;]\!]) \text{ else } ;$$

It is a difficult question if such a definition is well-founded because the definiens on the left-hand side appears also as definiendum on the right-hand side. If the computation terminates, then there exists a well-ordering, giving us an inductive definition. If the computation does not terminate, then there is no longer a well-ordering and, in turn, no inductive definition. In the course of this report, we show that for non-terminating programs, this is still a well-defined semantics, namely a coinductive or greatest fixed point semantics.

To develop coinductive semantics, we start in this chapter by reviewing characteristics in the definition of programming languages in Section 1.1 and by summarizing commonly used approaches for the operational semantics of programming languages, namely abstract state machines (ASMs) in Section 1.2 and inference rule-based specifications in the remaining sections by first giving an overview in Section 1.3 and by summarizing structural operational semantics (SOS) in Section 1.4 and natural semantics in Section 1.5. In Chapter 6, we use these foundations to develop a coalgebraic or greatest fixed point semantics which can be used to verify non-refining transformations even on non-terminating programs.

1.1 Characteristics in the Definition of Programming Languages

Programming languages are highly context-sensitive languages consisting of an infinite set of programs each of which is a sequence of characters. To obtain finite and easily comprehensible definitions of programming languages as well as efficient membership tests, programming languages are described by a two-stage process: A deterministic context-free grammar specifies a superset of the programming language. For each program, this context-free grammar defines a deriviation tree (the *concrete syntax*). These derivation trees and the corresponding context-free grammar can be simplified in many cases, e.g. by eliminating chain productions. The resulting grammar is called *abstract syntax* and is used in all subsequent semantic definitions. It defines an abstract syntax tree (AST) for each program. In the remainder of this report, we do not distinguish between a program and its abstract syntax tree. In the second stage of language definition, this superset of syntactically correct programs is restricted by context-sensitive conditions. Attributes are associated with the nodes in the abstract syntax tree, e.g. by specifying an attribute grammar. These attributes define the context-sensitive properties of programs. In general, these attributes cannot be computed locally within the scope of one production but need more sophisticated strategies which traverse the abstract syntax tree in special orders. The programming language itself is described as the set of programs whose attributes fulfill certain conditions defined within the context of one production of the abstract syntax, i.e., the conditions describe a relation between the attributes of a node and the attributes of its successors.

In general, the semantics of programs, i.e. their dynamic behaviour, is compositional. This means that the semantics of a node in an abstract syntax tree can be defined directly given its immediate successors. The principle of compositionality is necessary due to two reasons: Even if a programming language contains infinitely many programs, we need a finite description of their semantics. Using the principle of compositionality and the construction mechanism of

programs described in the concrete and abstract syntax, we can attach meaning to programs by specifying for each production how the semantics of the entire subtree, i.e. the left-hand side of the production, is defined given the semantics of the subtree's direct subtrees, i.e. the elements on the right-hand side of the production. Moreover, the principle of compositionality is essential since humans want to use programming languages. They would not be able to understand the meaning of their programs if their semantics was not compositional.

No rule without exception: Certain constructs in programming languages exhibit a semantics which is inherently not compositional. For example, `goto`-statements may leave a program part and go to some other place which cannot be described via the predecessor or successor relation in abstract syntax trees. To be able to define the semantics of such non-compositional program constructs, we need the concept of *continuations*. A continuation tells us where to proceed with the computation. If we are dealing with the standard case of compositional program constructs, the continuation specifies simply a child node or the parent node. In general, the continuation denotes an arbitrary program node. Already at compiling time, the continuations can be computed. Therefore, additional attributes are specified defining where to continue the computation. If the control flow branches at a node, then it is necessary to define several such continuations, each describing the succeeding computation depending on the branch direction. Typically, there are two ways of defining continuations, either as a pointer to the node at which the computation continues or as the subtree of the program where the computation proceeds. Both possibilities are equivalent.

1.2 Abstract State Machines (ASMs)

Abstract state machines (ASMs) [Gur95, ASM] have been used extensively in the definition of the semantics of programming languages, e.g. in the definition of C [GH93], of Java [SSB01] and of SDL [EGGP00]. Moreover, ASMs have been used successfully in proving the correctness of compilations, e.g. the correctness of the compilation of Prolog to the WAM [BR94], the correctness of the translation of Occam to transputer code [BD96] and in the Verifix project which deals with the construction of provably correct compilers [ZG97, GZ99]. During these projects, a remarkable engineering knowledge has emerged concerning the way in which specifications should be written to be useful for the purpose of semantics specification and translation verification. In this section, we summarize this particular use of ASMs. Our presentation generalizes the description in [GZ99].

Remark: Very few ASM semantics (e.g. [SSB01]) modify the AST during program execution. We do not consider this here as it is not the typical case.

Abstract state machines (ASMs) are used to describe the semantics of programming languages operationally as state transition systems based on the abstract syntax trees. Part of the current state is the current task, a pointer to the node in the abstract syntax tree which is currently executed. During program execution, states are transformed into new states, thereby also updating the pointer to the current task. States are regarded as algebras over a given signature. During a state transition, the interpretation $\mathcal{I}$ of some of the function symbols may change. For example, if a function symbol S specifies the state of memory, then a variable assignment `x:=v` changes the interpretation $\mathcal{I}(S)$ of the function symbol S for argument `x`: $\mathcal{I}(S(\mathtt{x})) := \mathcal{I}(\mathtt{v})$ holds in the new state. Each n-ary function symbol is interpreted with an n-ary mapping. For each state transition, the interpretation of some function values might change. In general, an ASM consists of four components $(\Sigma \cup \Delta, \mathcal{A}, \mathit{Init}, \mathit{Trans})$: The signature is composed of

two disjoint sorted signatures, the signature of the *static functions* Σ and the signature of the *dynamic functions* Δ. $\mathcal{A}$ is the *static algebra*, an order-sorted Σ-algebra interpreting the function symbols in Σ. *Init* is a set of equations over $\mathcal{A}$ which defines the *initial states* of $\mathcal{A}$. Finally, *Trans* is a set of *transition rules* for specifying the state transitions by defining or updating, resp., the interpretations of certain function values of functions in Δ. A $(\Sigma \cup \Delta)$-algebra is a state of the ASM iff its restriction to Σ is the static algebra $\mathcal{A}$. If q is a state, $f \in \Delta$ is a function symbol, and t_i are terms over $\Sigma \cup \Delta$ with interpretations x_i in q, then the update $f(t_1, \ldots, t_n) := t_0$ defines the new interpretation of f in the succeeding state q' as

$$q' \models f(x_1, \ldots, x_n) = \begin{cases} x_0 & \text{if for all } i,\ 0 \leq i \leq n,\ q \models t_i = x_i \\ f_q(x_1, \ldots, x_n) & \text{otherwise} \end{cases}$$

A transition rule defines a set of updates which are executed in parallel:

if *Cond* **then** $Update_1 \ldots Update_n$ **fi**

If $q \models Cond = true$ in state q, then $Update_1 \ldots Update_n$ are executed in q.

When defining the semantics of programming languages, we use the abstract syntax tree as basis and attach meaning to it, cf. Section 1.1. Thereby, we assume that the abstract syntax tree contains attributes defining all continuations, especially for the non-compositional changes of the control flow. The definition of the ASM models the program counter during program execution, thereby using the continuation attributes which might be split up according to the value of conditions (true case and false case). Here is the example of a transition rule defining the semantics of the while-loop, as stated in [GZ99]. CT (CT = current task) is the abstract program counter, $CT.TT$ (true task) is the true-continuation attribute of CT and $CT.FT$ (false task) is the false-continuation attribute of CT.

if $CT \in While$ **then**
  if $value(CT.cond) = true$ **then**
    $CT := CT.TT$
  else $CT := CT.FT$ **fi fi**

The semantics of each program node is described by a finite set of transition rules. Typically the condition of such a transition rule specifies the nodes in the abstract syntax tree (**While**-nodes in our example) for which the transition rule is applicable. The transition rules define updates, thereby employing child nodes (in our example $CT.cond$) as well as statically computed continuations ($CT.TT$ and $CT.FT$ in our above example). In the remainder of this report, we assume that in an ASM definition which specifies the semantics of a programming language, each transition rule is of the following general form:

if $CT \in X$ **then**
  if *applicability_conditions* **then**
    $CT := new\_CT$; *further_updates*
  else $CT := new\_CT'$; *further_updates'* **fi fi**

1.3 Inference Rule-Based Specifications in the Semantics of Programming Languages

Inference rule-based specifications have a long tradition in computer science for the specification of the formal semantics of programming languages. As a classical example, consider the rules of the simply-typed λ-calculus [Mit90] which specifies the definition and application of functions:

$$\text{(Ax)}\quad \{x:\tau\} \vdash x:\tau \qquad\qquad \text{(}\lambda\text{I)}\quad \frac{\Gamma \cup \{x:\tau\} \vdash t:\sigma}{\Gamma \vdash \lambda x:\tau.t:\tau\to\sigma}$$

$$\text{(}\lambda\text{E)}\quad \frac{\Gamma \vdash t:\tau\to\sigma \qquad \Gamma \vdash t':\tau}{\Gamma \vdash (t\ t'):\sigma} \qquad\qquad \text{(CI)}\quad \frac{\Gamma \vdash t:\sigma}{\Gamma \cup \{x:\tau\} \vdash t:\sigma}$$

The first rule (Ax) says that if the typing $x : \tau$ is contained in the context Γ, then we can conclude that $x : \tau$ holds. The second rule (λI) specifies how to build functions while (λE) defines the evaluation of functions. Finally, the fourth rule (CI) specifies that the assumptions in the context can be extended.

It is well-known that the structure of proofs for the well-typedness of a program corresponds directly with the structure of the program itself, cf. Figure 1.1. More exactly, the proof structure is already passed on by the program structure. To prove that a λ-term t is well-typed, it suffices to find a proof for the statement $\emptyset \vdash t : \tau$. Figure 1.1 shows such a proof and its correspondence with the program structure for the example program $\lambda x.\lambda y.(yx)$.

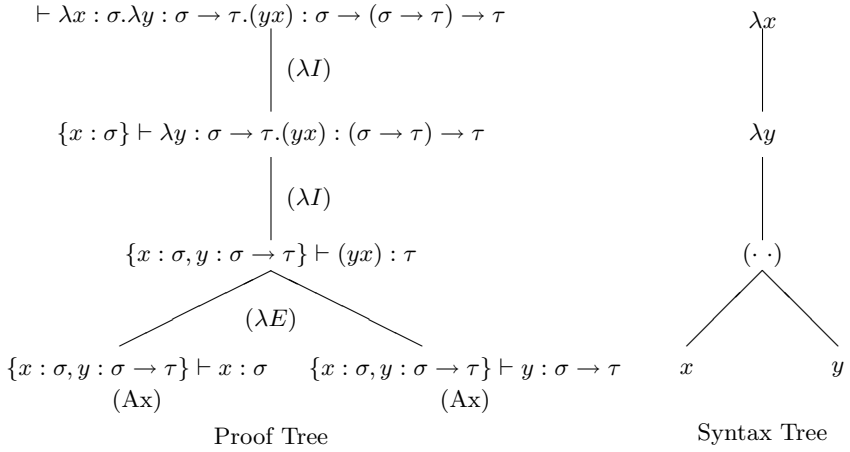


Figure 1.1: Proof of the Well-Typing of a λ-Term

Types are static semantic information which can be regarded as abstractions of dynamic behavior. In particular, recursions in the dynamic behavior do not show up in the proofs for well-typedness. Otherwise, there could be no strict correspondence between the program and the proof structure. In contrast, the full dynamic behavior is not related in a one-to-one correspondence with the program tree because recursion requires the program tree to be "unfolded" in the dynamic semantics. We define this process of unfolding formally in Chapters 4 and 6. For now we carry on by noticing that the dynamic behavior, i.e. the semantics, can be described by trees which are defined on top of the syntactic program structure by specifying axioms and inference rules.

Such specifications have been used extensively in the definition of the semantics of programming languages. Axioms and inference rules are used to specify semantic properties with respect to the abstract syntax. A prominent example is the complete specification of Standard-ML [MTH90]. Its revision [MTHM97] demonstrates that rule-based specifications are very stable. Most of the modifications changed the semantics of ML itself rather than correcting errors in the original specification. Further examples for language specifications are the dynamic semantics of Eiffel [Att96], Eiffel// (Eiffel Parallel) [ACEL96], Esterel [Ber90], and in general imperative and

object-oriented programming languages [GZ98, Gle99a, Gle99b, GZ04]. Inference-rule based semantics have also been used successfully to prove properties of programs: The investigations in [DE99, Sym99, vON99] prove the static type safety of subsets of the Java programming language whereby specifications based on natural and structural operational semantics are used.

There are two main variants of specifications based on inference rules, *big-step* or *natural semantics* and *small-step* or *structural operational semantics (SOS)*. While big-step semantics can only describe strictly compositional programming languages, small-step semantics is also able to handle non-compositional program constructs. We investigate both variants separately as they differ significantly on that score as well as in the way they treat the program during the formalized execution. In the following two sections, we describe both of them, for more details consult e.g. [NN99].

Remark: The terminology is not consistent throughout the literature. Sometimes natural semantics refers only to big-step semantics, sometimes it comprises big-step as well as small-step semantics. In this report, we use the terms natural and big-step semantics equivalently. The confusion in the use of these terms comes from the fact that in many specifications, small-step and big-step semantics are used together, big-step typically for the evaluation of expressions and small-step for the execution of control-flow sensitive parts such as statements.

1.4 Structural Operational Semantics (SOS)

Structural operational semantics (SOS), also called small-step semantics, concentrate on individual steps of program execution and how these single steps are integrated in the overall execution. Assumptions of inference rules formalize smaller steps while their embedding into the larger program context is defined in the conclusion. Individual steps are described in the axioms. Such an individual step is either termination of execution $< p, \sigma > \rightarrow \sigma'$ in the final state σ' or it is a state transition $< p, \sigma > \rightarrow < p', \sigma' >$ denoting that the execution of p in state σ yields a new program p' to be executed in the succeeding state σ'. p' is often called *continuation*. In this report, we call it *continuation program* to distinguish it from the statically computable continuation attributes described in Section 1.1. In most cases, p' is a direct subtree of p or composed from direct subtrees of p. The conclusions of inference rules define the embedding of such program parts into their larger context. In the case of compositional semantics, this context is simply the parent node in the abstract syntax tree. In general, arbitrary continuations are possible, allowing for the description of non-compositional semantics. As typical examples for small-step definitions, consider these inference rules:

$$\frac{<S_1,\sigma> \rightarrow <S_1',\sigma'>}{<S_1;S_2,\sigma> \rightarrow <S_1';S_2,\sigma'>} \qquad \frac{\mathsf{Eval}(cond)=true}{<\textbf{if } cond \textbf{ then } S_1 \textbf{ else } S_2,\sigma> \rightarrow <S_1,\sigma>}$$

$$\frac{<S_1,\sigma> \rightarrow \sigma'}{<S_1;S_2,\sigma> \rightarrow <S_2,\sigma'>} \qquad \frac{\mathsf{Eval}(cond)=false}{<\textbf{if } cond \textbf{ then } S_1 \textbf{ else } S_2,\sigma> \rightarrow <S_2,\sigma>}$$

$$< \textbf{skip}, \sigma > \rightarrow \sigma$$

$$< \textbf{while } cond \textbf{ do } S, \sigma > \rightarrow < \textbf{if } cond \textbf{ then } (S; \textbf{while } cond \textbf{ do } S) \textbf{ else skip}, \sigma >$$

The first two inference rules in the left column describe how the execution of a sequence of statements S_1 is integrated into a larger context, namely the sequence of statements $S_1; S_2$. The first two rules on the right-hand side specify the execution of the if-statement. The first

axiom defines the effect of the skip-statement. Finally the last axiom describes the while-loop by reducing its semantics to the semantics of the if-statement. These rules describe execution in a bottom-up style: execution of smaller program parts is integrated into the execution of larger parts of the program, without paying attention to the overall state transition performed by the entire program. The structure of the inference rules does not need to reflect the structure of the program. For example, the semantics of the while-statement is not defined in terms of its sub-statements but by a new program which has been built from the sub-statements. (Note that the if-statement in the last axiom is not part of the original program but created upon application of this axiom.) In general, also statically computed continuation attributes (cf. Section 1.1) can be used. As an example, consider the semantics of the goto-statement:

$$< \textbf{goto}\ L; \sigma > \rightarrow < L.\mathit{continuation}, \sigma' >$$

This axiom defines a non-compositional semantics since the control-flow of the program branches to another arbitrary part of the program, denoted by the continuation of L, $L.\mathit{continuation}$.

The general form of a small-step inference rule is as follows: Let $X_0 ::= X_1 \cdots X_n$ be a production of the abstract syntax, $X_{l_r} \in \{X_1, \ldots, X_n\}$, $1 \leq r \leq m$, m a natural number, X_i' is an arbitrary program built from X_i or its direct subprograms, i.e. direct subtrees of its abstract syntax tree, $1 \leq i \leq n$, X_0' is an arbitrary program built from X_0, from its subprograms $X_1 \cdots X_n$, from X_i' and from the continuations of $X_0, \ldots, X_n$.

$$\frac{\mathsf{Eval}(X_{l_1}, \sigma) = \mathit{value}_1, \ldots, \mathsf{Eval}(X_{l_m}, \sigma) = \mathit{value}_m, \quad < X_i, \sigma > \rightarrow < X_i', \sigma' >}{< X_0, \sigma > \rightarrow < X_0', \sigma' >}$$

In its evaluation part, $\mathsf{Eval}(X_{l_1}, \sigma) = \mathit{value}_1, \ldots, \mathsf{Eval}(X_{l_m}, \sigma) = \mathit{value}_m$, the inference rule describes conditions for which the rule is applicable. The state transition $< X_i, \sigma > \rightarrow < X_i', \sigma' >$ defines the execution of X_i in state σ and gives us a new continuation program X_i' to be executed in state σ'. The conclusion of the inference rule specifies how this single transition $< X_i, \sigma > \rightarrow < X_i', \sigma' >$ can be integrated into the larger context X_0 whose execution in state σ yields the new continuation program X_0' to be executed in the new state σ'.

Data structures are needed to define the values of the evaluation conditions and the states reached during program execution. These data structures are typically defined inductively by a term algebra over a fixed set of constructor functions. Additional (defined) functions are specified by equations defining recursively the effect of these functions on the constructor terms.

In a small-step semantics, the program to be executed is an explicit part of the state. Each state $< p, \sigma >$ contains a continuation program p. In the initial state, p is the original program while in the final state, p is simply the empty program. The axioms and inference rules of a small-step semantics define how to rewrite this program during each state transition.

1.5 Natural Semantics

Natural semantics [Kah87] is a deductive method to define the semantics of programming languages. Axioms and inference rules specify semantic properties wrt. the abstract syntax. The semantics of an abstract syntax tree is defined as a state transition from the initial state into the final state. This state transition is defined compositionally in terms of the state transitions of the direct subtrees of the abstract syntax tree. Consider e.g. the rules for the while-loop:

$$\frac{\mathsf{Eval}(\mathit{cond}, \sigma) = \mathit{false}}{< \textbf{while}\ \mathit{cond}\ \textbf{do}\ S\ \textbf{end}, \sigma > \rightarrow\ \sigma} \qquad \frac{\mathsf{Eval}(\mathit{cond}, \sigma) = \mathit{true}, \quad < S, \sigma > \rightarrow\ \sigma', \quad < \textbf{while}\ \mathit{cond}\ \textbf{do}\ S\ \textbf{end}, \sigma' > \rightarrow\ \sigma''}{< \textbf{while}\ \mathit{cond}\ \textbf{do}\ S\ \textbf{end}, \sigma > \rightarrow\ \sigma''}$$

These two rules express that the body S of the loop is executed depending on the value of the condition *cond*. If it is executed, then the entire loop is executed recursively again. In the traditional setting, which we revise in this report (cf. Chapter 4), this kind of semantic description is only used for terminating computations. In this case, the second rule says that there exists a state transition from σ to σ'' if the condition *cond* evaluates to true, if the body S is executed by a state transition from σ to σ' and if there is a state transition from σ' to σ'' describing the recursive execution of the loop.

We can regard a natural semantics as a recursive procedure defined by inference rules. Each inference rule belongs to a production $X_0 ::= X_1 \cdots X_n$ of the abstract syntax. It has the following general form, whereby $X_{l_k} \in \{X_1, \ldots, X_n\}$, $1 \leq k \leq m$ and $X_{i_j} \in \{X_0, X_1, \ldots, X_n\}$, $1 \leq j \leq r$:

$$\frac{\mathsf{Eval}(X_{l_1}, \sigma_0) = \mathit{value}_1, \ldots, \mathsf{Eval}(X_{l_m}, \sigma_0) = \mathit{value}_m, \quad < X_{i_1}, \sigma_0 > \rightarrow \ \sigma_1, \cdots, < X_{i_r}, \sigma_{r-1} > \rightarrow \ \sigma_r}{< X_0, \sigma_0 > \rightarrow \ \sigma_r}$$

The assumptions of an inference rule consist of two main parts, the evaluation conditions $\mathsf{Eval}(X_{l_k}, \sigma_0)$ and the "procedure calls" on direct successors of X_0, i.e. the state transitions of the direct subprograms $< X_{i_j}, \sigma_{j-1} > \rightarrow \sigma_j$. The evaluation conditions decide about the applicability of the rule in a given state σ_0. In the while-loop example, they express the value $\mathsf{Eval}(\mathit{cond}, \sigma_0)$ of the condition *cond*. The state transitions in the assumptions describe the semantics of the subprograms. If the evaluation conditions are fulfilled, then the procedures, i.e. inference rules, for $X_{i_1}, \ldots, X_{i_r}$ are called in this order, each with the corresponding initial state $\sigma_0, \ldots, \sigma_{r-1}$ as input value and with the corresponding final state $\sigma_1, \ldots, \sigma_r$ as result. The X_{i_j}, $1 \leq j \leq r$, are either X_0 or the roots of direct subtrees of X_0. A particular X_{i_j} might be called several times with possibly different initial values or might not be called at all. Since we assume a strictly compositional programming language to be described with such inference rules, the semantics of the abstract syntax tree can be concluded solely from the semantics of its direct subtrees (in recursive cases also from its own semantics). The entire state transition for the loop is expressed in the conclusion. An axiom is an inference rule with only evaluation conditions in its assumptions but no state transitions, i.e. $r = 0$.

The data structures necessary to define the values of the evaluation conditions and the states reached during program execution are defined in the same way as in small-step semantics definitions, cf. Section 1.4.

Natural semantics specifications describe derivation trees. Their root nodes are marked with the program to be executed and with the initial and final state of computation. The successors of the root are marked either with direct subtrees of the program or the program itself (in recursive definitions). Furthermore, the successors are marked with state transitions as defined by the inference rules: The entire state transition from the initial state σ into the final state σ' of the root node is split up into a sequence $\sigma = \sigma_0 \rightarrow \sigma_1 \rightarrow \cdots \rightarrow \sigma_r = \sigma'$ of state transitions. Each individual state transition $\sigma_{i-1} \rightarrow \sigma_i$, $1 \leq i \leq r$, is described by exactly one of the subtrees of the derivation tree. The order on these subtrees is specified implicitly by the linear order of the states $\sigma_0 \rightarrow \sigma_1 \cdots \rightarrow \sigma_r$.

Traditionally, natural semantics specifications are interpreted with finite derivation trees because only then, a unique final state exists. This traditional view corresponds to an inductive or least fixed point interpretation. In this report, we show that a greatest fixed point or coinductive interpretation is more appropriate. In particular, it also allows for a semantics for non-terminating programs while not changing the usual inductive semantics for terminating programs, cf. Chapter 4.

2 (Co)Algebras and (Co)Induction: A Motivation

In recent years, coalgebraic methods, in particular coinduction, have gained increased interest and importance in the specification of and reasoning about state-based systems. The semantics of imperative programming languages also models state-based computations because the execution of a given programs triggers a potentially infinite sequence of state transitions. Traditionally, approaches to the formal semantics of programming languages have mainly concentrated on the formalization of the computation of final results (as for example denotational semantics) but not on the state transition aspect. Nevertheless, this aspect is at least as important. Many programs as for example operating systems, data bases, or control software in reactive and embedded systems are not intended to terminate while still having a very special semantics. To compile such programs correctly, it is necessary to preserve their state transition behavior. Hence, if we want to verify the correctness of compilers, it is of utmost importance to model non-terminating program behavior appropriately. In this report, we show that semantics of programming languages needs to be defined based on greatest fixed points and that the theory of coalgebras is the method of choice for this purpose.

Most of the existing literature concerning coalgebras and coinduction (cf. [JR97] for an overview) is formulated in terms of category theory. While category-theoretical notions are well-appreciated among theorists, they are not equally respected by computer scientists working in practice. We believe that in the context of coalgebraic theory, the category-theoretical notation offers many advantages, ranging from better readable and much more elegant notation to better possibilities to express the inherent duality of induction and coinduction. It is the aim of our exposition of the topic here to demonstrate these points.

Therefore, in this Chapter, we first motivate in Section 2.1 why formal semantics of programming languages needs to model the potentially infinite state transition behavior of programs. Then, in Section 2.2, we start with an intuitive and gentle motivation of induction and coinduction. In Chapter 3 we define these informally introduced concepts in pure set-theoretical terms and apply them by interpreting natural semantics coinductively in Chapter 4. After that, in Chapter 5, we formulate coalgebraic theory in category-theoretical terms. In doing so, we hope to bridge the gap between the intuition behind coalgebras and coinduction and its clear and crisp notions within category theory.

Subsequently, in Chapter 6, we use the introduced notions of coalgebras and coinduction to develop a coalgebraic interpretation of the operational approaches to the semantics of programming languages. Furthermore, we use their coinductive interpretations to compare them with respect to two criteria, namely with respect to the structure of imperative programming languages whose semantics can be defined with them and with respect to the way programs are treated during the evaluation that is specified by the operational semantics. Our results in this comparison are very interesting as they are in contrast to the common understanding that natural semantics can only describe terminating programs. Moreover, we show that structural operational semantics and ASMs are equally expressive because each ASM semantics can be transformed into an equivalent structural operational semantics and vice versa. Parts of the results presented in this report have been published in [Gle03, Gle04a, Gle04b].

2.1 The Need for Greatest Fixed Point Semantics

The execution of a program triggers a potentially infinite state transition sequence. A formalism for the semantics of programming languages should model this aspect appropriately. This requirement is essential in practical applications. Many programs (e.g. operating systems, data bases, control software in embedded systems or reactive systems) are not intended to terminate while still having a very special semantics. Most of the existing methods for the formal semantics of programming languages are based on inductive definition and proof principles. In this section, we show that this approach is not able to model the state-based transition character of program execution appropriately. Afterwards, we introduce the theory of coalgebras and coinduction and show that the state transition behavior of programs can suitably be defined by coinduction, i.e. based on greatest fixed points.

The Insufficiency of Induction Proofs

Let us start with a motivation why induction is not the appropriate proof principle for infinite computations. Consider one of the well-known proof rules of the Hoare calculus [Hoa69].

```
{P}
proc p
  ...
    {P}
    p
    {Q}
  ...
endproc
{Q}
```

If one wants to prove that a recursive procedure `p` is correct with respect to a precondition P and a postcondition Q, then one assumes that for all recursive calls of `p` within the body of `p`, precondition P and postcondition Q hold. If `p` always terminates, then this is an induction proof. The recursion depth of the inner calls is always smaller than the recursion depth of `p` itself. If the procedure `p` does not terminate, it is no longer a valid induction proof. The state transition sequence in the inner procedure's body is infinitely long as well as the state transition sequence of the outer procedure. Hence, we do not have an induction premise about a strictly smaller state transition sequence. Both state transition sequences have the same set-theoretic size. Nevertheless, it is still a valid coinductive proof showing that at each procedure entry, the precondition is fulfilled. Thereby we assume that the precondition holds in the initial state and prove that it also holds when entering the inner procedure. In this case, we cannot say anything about the postcondition because the program point at which it should hold is never reached.

The proof for the validity of the Hoare calculus rule in the non-terminating case uses induction to show that no contradiction can be observed. This reasoning is the basis for coinduction. An inductive argument shows that, for all finite prefixes of the potentially infinite state transition sequence, the precondition P is valid at each entry of procedure `p`. Then it concludes that this property (P valid at each entry of `p`) holds also for the infinite state transition sequence. If this were not the case, then there would be a finite prefix not fulfilling P, hence contradicting the result of the induction proof.

The Hoare calculus rule for procedures is essentially an overlay of two rules. The first considers the terminating case with a postcondition. The second models the non-terminating case where the precondition holds at each procedure entry. We will see the same overlay of rules for natural semantics in Section 4.4.

The following section and succeeding chapters provide an introduction to the theory of coalgebras and coinduction and also show in the case study in Chapter 4 how this kind of reasoning in the Hoare calculus can be used to interpret natural semantics (also called big-step semantics) [Kah87] coinductively.

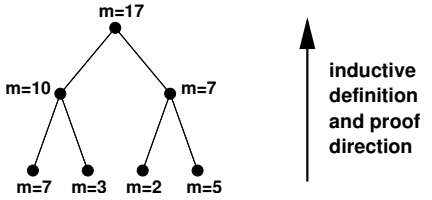
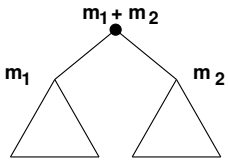


Figure 2.1: Abstract Data Types: A Simple Example

2.2 A Gentle Motivation for Coalgebras and Coinduction

Induction is a very well-known method in computer science to define abstract data types (ADTs). In general, ADTs are finite trees whose nodes and leaves are annotated with markings m, cf. Figure 2.1. Trees as the one in Figure 2.1 can be defined inductively:



- Each leaf marked with a natural number $n \in \mathbb{N}$ is a tree.
- If b_1 and b_2 are trees whose root nodes are marked with m_1 and m_2, then the tree consisting of the two subtrees b_1 and b_2 whose root node is marked with $m_1 + m_2$ (cf. figure next to this paragraph) is also a tree.

With this definition, we specify how leaves should look like and how we can construct larger trees from smaller trees. In the same sense we can prove properties of such trees by using the induction proof principle. For example, we could prove by induction that the marking of each inner node is the sum of the markings of its direct successor nodes. In an inductive proof, we would show this property in the base case for leaves (for which it holds trivially in this example) and then, in the induction step, we would show it for inner nodes.

Coinductive definitions are exactly the other way round: Starting from the markings of a given parent node, we define the markings of its children nodes. A typical example are the languages defined by context-free grammars $G = (\sigma, P, N, T)$ where σ is the start symbol, P is the set of production rules, N is the set of nonterminal symbols, and T is the set of terminal symbols, cf. also Figure 2.2. Assuming that P contains a production $\sigma \longrightarrow X_1 \cdots X_n$,

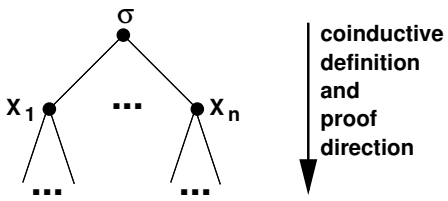


Figure 2.2: Context-free Grammars Coinductively

we could expand the node marked with σ as shown in Figure 2.2. Then we can continue this expansion process by letting $X_1, \ldots, X_n$ take the role of σ and by using appropriate productions $X_i \longrightarrow \cdots$, $1 \leq i \leq n$. The definition of context-free languages requires this expansion process to be finite. In the world of coalgebraic specifications, this expansion process is allowed to be infinite.

In essence, coalgebraic structures are potentially infinite trees whose nodes are annotated with markings (the nonterminals and terminals). Coalgebraic structures are well-suited to define the observable behavior of processes and to reason about it. In this view, σ takes the role of the

initial state. Successor nodes are possible successor states. If there is more than one successor node, then such structures can be used to model non-deterministic processes. (Note that this is not the only choice to interpret multiple successor nodes. In the coinductive interpretation of natural semantics, we use the existence of several successor nodes and, hence, of several direct subtrees to denote sequential composition of programs and their execution, cf. also Chapter 4.) There is no need to reach a leaf in a coinductive tree. If a process does not terminate, then no final state (=leaf) is reached.

Now let us assume that two coinductive structures t_1 and t_2 are given, cf. Figure 2.3. Coinduction can be used to show the equality of the trees t_1 and t_2 by using this proof rule:

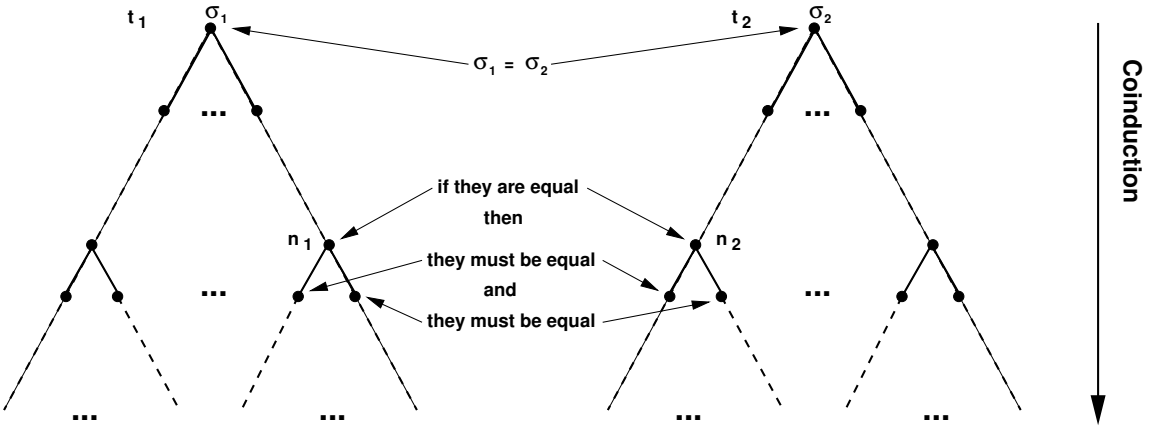


Figure 2.3: The Coinduction Principle Gently

Coinduction Proof Rule Gently: $t_1 = t_2$ if

- $\sigma_{t_1} = \sigma_{t_2}$ and
- for arbitrary nodes $n_1 \in t_1$ and $n_2 \in t_2$, if marking(n_1) = marking(n_2), then n_1 and n_2 have the same number of successor nodes and the markings of corresponding successor nodes are also equal. $\diamond$

In contrast to induction, we argue starting from the root node in direction to potential leaves (which do not need to exist in case the tree is infinite). With that we do not have a well-ordering any more as we have had it in the case of induction. If the depth of the entire tree is infinite, then the depth of at least one subtree is infinite as well. Nevertheless, one can use induction to verify that the coinduction principle is correct.

To see the correctness of the coinduction principle, let us sketch the following proof by contradiction. Assume that t_1 and t_2 are not equal, $t_1 \neq t_2$. Then one can show easily by induction that all finite prefixes of t_1 and t_2 are equal and that $t_1 \neq t_2$ cannot be observed in finite depth. Hence, one can conclude that t_1 and t_2 must be equal because otherwise, there would be nodes $n_1 \in t_1$ and $n_2 \in t_2$ in finite depth whose markings are identical but whose successor markings are different. Coinduction is the basis for bisimulation.

One can also use coinduction to show that the nodes in a given potentially infinite tree t fulfill a certain property P, cf. Figure 2.4. Therefore one needs to show that $P(\sigma)$ holds for the root node σ of t and that whenever $P(n)$ for $n \in t$ holds, then $P(m)$ also holds for all direct successor nodes m of n. In the coalgebraic literature [JR97], this principle is known as reasoning with greatest invariants. Greatest invariants define final subcoalgebras in which P holds.

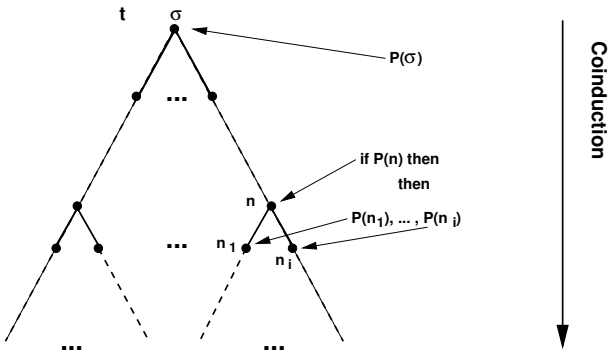


Figure 2.4: Reasoning with Greatest Invariants

To sum up, coinduction is – as also induction – a structural definition and proof principle. One argues about potentially infinite trees. In contrast to induction, one does not start at the leaves but at the root node. For readers being familiar with attribute grammars, this analogy might illustrate the difference: Inductive definitions correspond to synthesized attributes in attribute grammars while coinductively defined markings correspond to inherited attributes.

Now that we have given an intuitive introduction to coalgebras and coinduction, let us turn to the theory behind and motivate its use when reasoning about program and system behavior.

With induction, one defines abstract data types which are also called **initial algebras**. An initial algebra A is characterized by the fact that for each other algebra A' of the same signature, there is a unique homomorphism $h : A \longrightarrow A'$. (This homomorphism is for example used in completeness proofs for logical calculi when using Herbrand structures as most general models.)

Formally an initial algebra is defined as follows: Using the constructor symbols (which might also contain constant functions as 0 or "empty-list" [ ]) we build variable-free terms. The universe of the initial algebra is the smallest set which is closed under this construction process. In case of the constructor symbols 0 and s, we get exactly the set of natural numbers: $0, s(0), s(s(0)), \ldots$. For lists, we have the constructor functions

$$[\,] : \text{List (empty list) and } \mathit{append} : \text{List} \times A \longrightarrow \text{List}$$

A is the sort of list elements. It is interesting to note the mapping characteristics of the constructor functions. For example, append has several arguments on the "left-hand side" (List $\times$ A) but only one on its "right-hand side" (List).

In coalgebras, this mapping characteristic is exactly opposite. A coalgebra is a function that takes a single state and transforms it into a compound result consisting of possibly several states and observations. Let us consider a finite automaton *FA* as typical example for a coalgebra:

$$\mathit{FA} : \text{State} \longrightarrow \text{State} \times \text{Observation}$$

In comparison to the append function on lists, we have here a mapping characteristic that is exactly opposite. The coalgebra *FA* maps a given state into a compound result consisting of the successor state and of an observation. In case of non-deterministic automata, we would have several such compound results. In general, one would define a finite automaton as a coalgebra *FA* with this signature:

$$\mathit{FA} : \text{State} \longrightarrow \text{State} \times \text{Observation} + \{*\}$$

The tuple State × Observation denotes the mapping of a state into a successor state and an observation. "+" denotes disjoint union and the possible result "∗" denotes the arrival in the final state. In other words, a state transition in the finite automaton *FA* gives either a new state together with an observation or the final state where computation ends. Hence, the coalgebra *FA* is like a black-box yielding the new state and possibly an observation. It defines potentially infinite lists whose elements are observations.

As in the case of algebras and initial algebras, there are also special coalgebras, namely **final coalgebras**. They are characterized by the fact that for each coalgebra A' of the same signature, there exists a unique homomorphism $h : A' \longrightarrow A$ into the final coalgebra A. As initial algebras, final coalgebras are unique up to isomorphism. Final coalgebras represent the essence of specifications. We can think of the elements in a final coalgebra as finite and infinite trees which are set up according to the functions in the signature. Formally, final coalgebras are the greatest fixed points of signatures.

There is a large amount of theoretical results concerning coalgebras and coinduction, see [JR97] for an overview. Most of this work is formulated in terms of category theory and is, in spite of the intuitively clear results, not easily accessible. For practical applications, the signatures described by so-called **polynomial functors** are sufficient. These are basically those signatures whose functions have only finitely many arguments so that the nodes in the thereby defined finite and infinite trees have only finitely many successor nodes. In these cases, the final coalgebra exists (which is not the case in general).

We are interested in coalgebraic theory due to the following reasons:

- We can specify the semantics of potentially non-terminating state transition systems adequately with coalgebras. In particular, we can define the semantics of programming languages by assigning each program an element of a suitable final coalgebra. In doing so, we can reason not only about the final results of program computation but also about the state transition sequences which describe the behavior of programs in a more detailed fashion.

- Based on our coalgebraic program semantics, we are able to verify optimizing transformations, even when the transformed programs do not terminate.

- Coinductive definition and proof principles have a longstanding tradition in computer science, only that they are normally called differently. Just think of finite automata and classical proofs for their equivalence. These are pure coinductive proofs. Sometimes coinductive proofs are erroneously mixed up with induction proofs but the distinction between them becomes clear as soon as such proofs are formalized in theorem provers, cf. our discussion of related work in Section 4.5.

In the following Chapter 3, we derive the dual notions of induction and coinduction and in particular the basic principles of coalgebraic reasoning, namely coinduction and reasoning with greatest invariants, based on an easily comprehensible set-theoretic development. In Chapter 4, we use this development to derive a proof calculus for natural semantics analogously to the Hoare calculus rule for recursive procedures introduced in Section 2.1. In Chapter 5 we go one step further and show how the notions of (co)algebras and (co)induction can be defined in terms of category theory. Finally, in Chapter 6, we demonstrate that the operational semantics of programming languages can be interpreted coinductively.

3 A Set-Theoretic Formulation of Coalgebras and Coinduction

3.1 Abstract Data Types

Let $\mathcal{D}$ be an arbitrary but fixed abstract data type recursively defined as follows: Let $\mathcal{S}$ be some fixed set, $d_l \in \mathcal{D}$ for $l \in \{1, \ldots r_j\}$, $j \in \{1, \ldots, q\}$ and $s_k \in \mathcal{S}$ for $k \in \{1, \ldots, w_j\}$, $j \in \{1, \ldots, q\}$ or $k \in \{1, \ldots, t_i\}$, $i \in \{1, \ldots, p\}$. Then $d \in \mathcal{D}$ iff

$$\begin{aligned} d \quad ::= \quad & \mathit{Base}_1(s_1, \ldots s_{t_1}) \mid \cdots \mid \mathit{Base}_p(s_1, \ldots s_{t_p}) \mid \\ & \mathit{Con}_1(d_1, \ldots, d_{r_1}, s_1, \ldots s_{w_1}) \mid \cdots \mid \mathit{Con}_q(d_1, \ldots, d_{r_q}, s_1, \ldots s_{w_q}) \end{aligned}$$

This definition specifies a universe $\mathcal{D}$ of trees whose nodes are marked with one of the base or constructor symbols Base_i, $i \in \{1, \ldots, p\}$ or Con_j, $j \in \{1, \ldots, q\}$ and with the corresponding sequence of values $s_1, \ldots s_{t_i}$ or $s_1, \ldots s_{w_j}$. Formally, this set $\mathcal{D}$ of marked trees is defined by two recursive conditions: $d \in \mathcal{D}$ iff:

- If $d = \mathit{Base}_i(s_1, \ldots s_{t_i})$, $1 \leq i \leq p$, then $\mathit{root}(d)$ has no successor nodes. In this case, the marking of $\mathit{root}(d)$ is defined as $\mathit{mark}(\mathit{root}(d)) = (\mathit{Base}_i,\ s_1,\ \ldots\ s_{t_i})$.
- If $d = \mathit{Con}_j(d_1, \ldots, d_{r_j}, s_1, \ldots s_{w_j})$, $1 \leq j \leq q$, then $\mathit{root}(d)$ has r_j direct subtrees $d_1, \ldots, d_{r_j}$ such that $d_1, \ldots, d_{r_j} \in \mathcal{D}$. In this case, the marking of $\mathit{root}(d)$ is defined as $\mathit{mark}(\mathit{root}(d)) = (\mathit{Con}_j, s_1, \ldots s_{w_j})$.

This definition does not only specify trees of finite height but also trees of infinite height. For space reasons, we do not prove that the set $\mathcal{D}$ exists. Such a proof can be found e.g. in [CC92], showing that the closure ordinal of $\mathcal{D}$ is ω. For sake of readability, as an abbreviation, we write $\mathit{mark}(d)$ for a given tree d instead of $\mathit{mark}(\mathit{root}(d))$, where $\mathit{root}(d)$ denotes the root node of tree d.[1] The universe $\mathcal{D}$ of marked trees induces the complete lattice $(\mathcal{P}(\mathcal{D}), \subseteq)$ where $\mathcal{P}(\mathcal{D})$ denotes the powerset of $\mathcal{D}$ and $\subseteq$ the inclusion relation on sets.

Definition 3.1 (Specification *Spec***)** *A specification Spec defines a unary predicate Spec on the universe of an abstract data type $\mathcal{D}$ by stating exactly one equation for each base Base_i, $1 \leq i \leq p$, and each constructor Con_j, $1 \leq j \leq q$:*

$$\begin{aligned} & \mathit{Spec}(\mathit{Base}_i(s_1, \ldots, s_{t_i})) \equiv \mathit{true} \wedge \mathit{ok}_{\mathit{Base}_i}(s_1, \ldots, s_{t_i}) \\ & \mathit{Spec}(\mathit{Con}_j(d_1, \ldots, d_{r_j}, s_1, \ldots s_{w_j}) \equiv \mathit{Spec}(d_1) \wedge \cdots \wedge \mathit{Spec}(d_{r_j}) \\ & \quad \cdots \wedge \mathit{ok}_{\mathit{Con}_j}(s_1, \ldots, s_{w_j}, \mathit{mark}(d_1), \ldots, \mathit{mark}(d_{r_j})) \end{aligned}$$

Thereby, the predicates $\mathit{ok}_{\mathit{Base}_i}$ and $\mathit{ok}_{\mathit{Con}_j}$, $1 \leq i \leq p$ and $1 \leq j \leq q$, define restrictions on the allowed combinations of markings of neighbored nodes in the elements of $\mathcal{D}$. The exact definitions of $\mathit{ok}_{\mathit{Base}_i}$ and $\mathit{ok}_{\mathit{Con}_j}$ depend on the concrete specification. E.g. in the context of natural semantics, they are implicitly specified by the axioms and inference rules of the natural semantics, cf. also Section 4.1 where we state the corresponding details. For now, we only

[1] Note that $\mathit{mark}(d)$ is **not** a recursive function denoting the markings of all nodes in tree d. $\mathit{mark}(d)$ only specifies the marking of the root node of d.

require them to be decidable. The predicate Spec defines implicitly a function spec : $\mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$*:*

$$spec(X) = \{x \in \mathcal{D} \mid \exists i \in \{1,\ldots,p\}.x = Base_i(s_1,\ldots s_{t_i}) \ \wedge ok_{Base_i}(s_1,\ldots,s_{t_i}) \vee$$
$$\exists j \in \{1,\ldots,q\}.x = Con_j(d_1,\ldots,d_{r_j},s_1,\ldots,s_{w_j}) \ \wedge \ d_1 \in X \wedge \cdots \wedge d_{r_j} \in X \ \wedge$$
$$ok_{Con_j}(s_1,\ldots,s_{w_j},mark(d_1),\ldots,mark(d_{r_j})\} \qquad \diamond$$

Theorem 3.1 (Monotonicity of *spec***)** *The specification function spec:* $\mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$ *is monotone on lattice* $(\mathcal{P}(\mathcal{D}), \subseteq)$*: if* $X \subseteq Y$ *for* $X, Y \in \mathcal{P}(\mathcal{D})$*, then* $spec(X) \subseteq spec(Y)$*.* $\diamond$

Proof: By contradiction: Assume there exists $z \in spec(X)$, $z \notin spec(Y)$. Then there exists $x \in X$ such that $spec(\{x\}) = \{z\}$. Since $X \subseteq Y$, it follows that $x \in Y$ and $spec(\{x\}) = \{z\} \subseteq spec(Y)$, contradicting the assumption. ■

Tarski's fixed point theorem states that each monotone function f on a complete lattice has a least and a greatest fixed point, denoted by $lfp(f)$ and $gfp(f)$. Hence, we conclude that $lfp(spec)$ and $gfp(spec)$ exist. The least fixed point is called **initial algebra**, the greatest fixed point **final coalgebra**.

A specification *Spec* restricts the valid markings of the nodes of the trees in the universe $\mathcal{D}$ of an abstract data type. The least fixed point $lfp(spec)$ is the set of all finite trees whose markings are valid with respect to the specification. (Short outline of a proof: It is obviously a fixed point. Consider a set strictly smaller: Then the "missing element" can always be constructed by a finite construction sequence.) The greatest fixed point $gfp(spec)$ is the set of all trees with finite and infinite height whose markings are valid with respect to the specification. (Short outline of a proof: Each tree in $\mathcal{D}$ not contained in this set has at least two neighbored nodes whose markings are not valid with respect to the markings of its predecessor or successor nodes.)

3.2 Induction and Coinduction

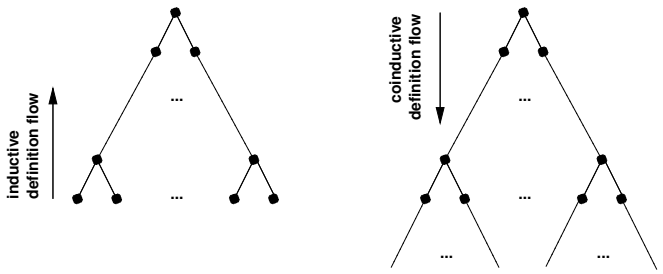


Figure 3.1: Induction versus Coinduction

A priori there is no direction in the specification. It is not determined if a marking is defined in terms of the markings of its successors or of its predecessor. In principle, two definition schemata are possible: In the inductive definition schema, we specify valid markings for the bases. Then we state how they are propagated through the entire tree by defining how the markings of a node are derived from the markings of its child nodes. The reverse direction is also possible and gives us the coinductive definition principle. Starting at the root node of a tree, we specify how its marking is propagated through the tree. Therefore we define how the marking of a node is derived from the marking of its predecessor. The first principle is structural induction and defines unique markings on finite trees. The second principle works also for infinite trees. Even though a tree might not be finite, the coinductive definition specifies a possibly infinite marking process well-defined at each step.

The inductive definition principle corresponds directly with the inductive proof principle. It states that some predicate Q holds for all elements in the least fixed point $lfp(spec)$. An inductive

proof is entirely constructive. Q can only be verified for elements which can be constructed.

There is also a coinductive proof principle which corresponds directly with the coinductive definition principle. It can be used to prove properties of elements in the greatest fixed point. We need these two versions:

Theorem 3.2 (Unary Coinduction Principle) *Let $d \in gfp(spec)$, Q a predicate on the markings of the nodes of d. $Q(mark(k))$ holds for all nodes $k \in d$ if*

- $Q(mark(root(d)))$ *and*
- *if* $\forall j \in \{1,\ldots,q\} \,.\, \forall d' \in gfp(spec) \,.\, (d' = Con_j(d_1,\ldots,d_{r_j},s_1,\ldots,s_{w_j}) \Rightarrow (Q(mark(Con_j(d_1,\ldots,d_{r_j},s_1,\ldots,s_{w_j}))) \Rightarrow Q(mark(d_1)) \wedge \cdots \wedge Q(mark(d_{r_j}))))$. ◇

The two conditions in Theorem 3.2 provide us with a proof principle to verify that all markings in a tree $d \in gfp(spec)$ fulfill a given predicate Q. Therefore we need to prove that Q holds for the marking of the root node of d (first condition) as well as for all nodes which can possibly be reached from this root node (second condition). This is achieved by proving that whenever Q holds for the marking of an inner node, then it also holds for the markings of its direct successor nodes. In contrast to Definition 3.1, there are no recursive proof obligations like $Spec(Con_j(\cdots)) \equiv Spec(d_1) \wedge \cdots \wedge Spec(d_{r_j}) \ldots$. Here we only need to prove a non-recursive statement about the finitely many constructors $Con_1,\ldots,Con_j$ of $\mathcal{D}$ and their possible direct successors. As a consequence of Theorem 3.2, we then get a statement about the infinitely many trees in the greatest fixed point $gfp(spec)$ (many of which are of infinite height) and their markings. In practical applications, we verify the two conditions of Theorem 3.2 by utilizing the specification *spec* and its definitions of the predicates ok_{Base_i} for $1 \leq i \leq p$ and ok_{Con_j} for $1 \leq j \leq q$, cf. also Chapter 4.

Proof: Proof of Theorem 3.2. By contradiction: Assume there exists a node $k \in d$ such that $\neg Q(mark(k))$. W.l.o.g. let k be a node with minimal distance to the root node of d such that $\neg Q(mark(k))$. Let *pos* be the position of this node k, i.e. $k = d\,|_{pos}$. (Each node in a tree can be specified by a list of navigation numbers denoting the path from the root on which it can be reached.) Since we assume that $Q(mark(root(d)))$ holds, the list *pos* contains at least one element: $pos = [l \mid pos']$. Since we assume that k is a smallest node such that $\neg Q(mark(k))$, $Q(mark(d\,|_{pos'}))$ follows. But $d\,|_{pos'} = Con_j(d_1,\ldots,d_{r_j},s_1,\ldots,s_{w_j})$ for some $j \in \{1,\ldots,q\}$ and $d\,|_{pos} \in \{d_1,\ldots,d_{r_j}\}$. From the second assumption in Theorem 3.2 we infer that $Q(mark(d_l))$ for all $l \in \{1,\ldots,r_j\}$, in particular $Q(mark(k))$ in contradiction to the assumption $\neg Q(mark(k))$. Hence, $Q(mark(k))$ for all $k \in d$. ■

Theorem 3.3 (Binary Coinduction Principle) *Let $d, d' \in gfp(spec)$. $d = d'$ if*

- *for some $i \in \{1,\ldots,p\}$: $d = Base_i(s_1,\ldots,s_{t_i})$ and $d' = Base_i(s_1,\ldots,s_{t_i})$ or if for some $j \in \{1,\ldots,q\}$: $d = Con_j(d_1,\ldots,d_{r_j},s_1,\ldots,s_{w_j})$ and $d' = Con_j(d'_1,\ldots,d'_{r_j},s_1,\ldots,s_{w_j})$ and*
- *if for all terms $t_1, t_2 \in gfp(spec)$ and for all $j \in \{1,\ldots,q\}$: if $t_1 = Con_j(d_1,\ldots,d_{r_j},s_1,\ldots,s_{w_j})$ and $t_2 = Con_j(d'_1,\ldots,d'_{r_j},s_1,\ldots,s_{w_j})$ implies that for all $l \in \{1,\ldots,r_j\}$: $mark(d_l) = mark(d'_l)$.* ◇

Proof: Analogous to the proof of Theorem 3.2: By contradiction: Assume that $d \neq d'$. Then there exists a position $pos = [l \mid pos']$ of minimal length such that $mark(d\,|_{pos}) \neq mark(d'\,|_{pos})$ and $mark(d\,|_{pos'}) = mark(d'\,|_{pos'})$. But then the second condition in Theorem 3.3 implies that $mark(d\,|_{pos}) = mark(d'\,|_{pos})$contradicting the assumption $d \neq d'$. Hence $d = d'$. ■

As Theorem 3.2, Theorem 3.3 states two **non**-recursive conditions which allow us to reason about recursive, possibly infinite structures. When reasoning about the semantics of programming languages, we use the unary coinduction principle to prove statements about possibly infinite state transition sequences of program executions. Moreover, we use the binary coinduction principle to compare programs by comparing their state transition sequences.

3.3 Related Work

We have based our development of (co)induction in this chapter on a simple exploitation of finite and infinite abstract data types. The set-theoretic basis for this straightforward development can be found e.g. in [CC92] which shows that coinductive interpretations of rule systems capture the behavior of finite and infinite state transition sequences. Most of the existing literature on algebras and coalgebras and their corresponding definition and proof principles induction and coinduction chooses a categorical setting, cp. e.g. [BCG02, JR97]. Nevertheless, in most situations one needs only polynomial functors going from the category of sets and functions to itself. The theory of algebras and coalgebras for polynomial functors can be stated in set theory, as we have demonstrated here. Then, the difference between an initial algebra and a final coalgebra is reduced to the distinction between finite and infinite data structures, i.e. least and greatest fixed points. We believe that this set-theoretical setting allows for a more intuitive understanding and in turn for better applicability in practical situations. Our notation in Section 3.1 is based on the exposition in Appendix B titled "Induction and Coinduction" in [NNH99]. While the explanations therein give a good understanding of least and greatest fixed points of specifications, they do not state proof rules like the unary or binary coinduction principle. Rather they state a proof rule that the elements of each post fixed point fulfill the specification. In the context of functional programming languages, coinduction and bisimulation (which corresponds to the binary coinduction principle stated in section 3.2) have been used to deal with non-terminating computations, cf. e.g. [Gor95].

3.4 Conclusions

In this chapter, we have shown that induction and coinduction and their respective definition and proof principles can be defined based on purely set-theoretic notions. Based on the definition of an abstract data type, we have introduced specifications which might put further restrictions on the valid structures. In a least fixed point setting, we consider only finite data structures. In a greatest fixed point setting, also infinite data structures are included. Induction proves that only correct structures can be constructed. Coinduction proves that no contradiction can be observed. We think that such an easily comprehensible description helps in bridging the gap between theoretical developments in the field of formal semantics of programming languages on one side and practical applications in reasoning about properties of programs and programming languages on the other side.

In the following chapter, we show how coinduction can be used to derive a greatest fixed point interpretation of natural semantics which is able to model terminating as well as non-terminating computations. With this development, we contradict the common understanding that natural semantics can only describe the semantics of terminating programs.

4 Case Study: A Coinductive Interpretation of Natural Semantics

Programming language semantics incorporates two dual aspects: The execution of a program triggers a potentially infinite state transition sequence. If this transition sequence terminates, then it defines the final result of program execution. A formalism for the semantics of programming languages should model both aspects simultaneously. If the execution of a program terminates, then its final result should be defined based on the finite state transition sequence. Moreover, a semantics formalism should specify a more meaningful semantics than just "undefined" for non-terminating programs. As already mentioned, this requirement is essential in practical applications since many programs (e.g. operating systems, data bases, control software in embedded systems or reactive systems) are not intended to terminate while still having a very special semantics.

In this chapter, we show that a greatest fixed point or, equivalently, coinductive interpretation of natural semantics is able to model both aspects simultaneously. This greatest fixed point interpretation gives rise to a proof calculus consisting of inductive and coinductive proof rules analogous to the Hoare calculus rule for recursive procedures discussed in Section 2.1. This proof calculus can be used in the formal reasoning about programming languages. As examples, we consider two applications. The first concerns the correctness proofs of translations, e.g. in compilers. Thereby one needs to prove that the observable behavior of the translated programs is preserved. This is a stronger requirement than just preserving their final results. The second example regards proofs for properties of programming languages, e.g. type safety, which need to consider terminating as well as non-terminating programs.

We start with the observation that each natural semantics defines an abstract data type. Then we show that each natural semantics is a specification in the sense of Definition 3.1. We prove that the least fixed point of such a specification describes the execution of all terminating programs while the greatest fixed point defines also a semantics for all non-terminating computations. Finally we show that the greatest fixed point interpretation gives rise to two simple proof rules which can be used to verify the correctness of translations as well as to verify properties of programming languages (e.g. type safety).

4.1 Derivation Trees of Natural Semantics

A natural (or big-step) semantics defines execution of programs in a top-down fashion: the state transitions of an entire abstract syntax tree are composed from the state transitions of its direct subtrees and, in recursive definitions, also from its own state transitions. It is important to observe that a big-step semantics defines individual state transitions only at the leaves of an abstract syntax tree. For all inner nodes, the inference rules specify how to compose the overall state transition sequence in the conclusion from the state transitions of the assumptions. Hence, we can regard each inference rule as a recursive procedure that is applicable if its evaluation conditions are fulfilled and that calls recursively further axioms or inference rules. The execution of a program defines a possibly infinite derivation tree. Its inner nodes correspond to the

application of inference rules and its leaves represent the application of axioms. We define this idea formally:

First we define the markings of the nodes in a derivation tree. Let $\overline{\mathbf{Prog}}$ be all abstract syntax trees. Let $\mathbf{Prog} = \{prog \mid \exists\ prog' \in \overline{\mathbf{Prog}}\ .\ prog = prog' \vee prog \text{ is a subtree of } prog'\}$ be all abstract syntax trees and their subtrees. Let $\mathcal{S}$ be the data structures used in a natural semantics to represent the states (cf. Section 1.5). In a derivation tree, each node is marked with $(P, prog, s, s')$ where P is its base or constructor, $prog \in \mathbf{Prog}$ is a program, $s \in \mathcal{S}$ is the initial state, and $s' \in \mathcal{S}$ the final state.

Let $A_1, \ldots, A_p$ be the axioms and $R_1, \ldots, R_q$ be the inference rules of a natural semantics specification, each belonging uniquely to one production $X_0 ::= X_1 \cdots X_n$ of the abstract syntax and each of the form

$$\frac{\mathsf{Eval}(X_{l_1}, \sigma_0) = value_1, \ldots, \mathsf{Eval}(X_{l_{m_i}}, \sigma_0) = value_{m_i}}{< X_0, \sigma_0 > \rightarrow\ \sigma_1} \quad \text{or} \quad \frac{\mathsf{Eval}(X_{l_1}, \sigma_0) = value_1, \ldots, \mathsf{Eval}(X_{l_{m_j}}, \sigma_0) = value_{m_j},\ < X_{i_1}, \sigma_0 > \rightarrow\ \sigma_1, \cdots, < X_{i_{r_j}}, \sigma_{r_j - 1} > \rightarrow\ \sigma_{r_j}}{< X_0, \sigma_0 > \rightarrow\ \sigma_{r_j}} \quad \text{resp.}$$

The abstract data type $\mathcal{D}$ of derivation trees is defined as follows whereby $prog \in \mathbf{Prog}$ and $s, s' \in \mathcal{S}$: $d \in \mathcal{D}$ iff

$$\begin{aligned} d \quad ::= \quad & A_1(prog, s, s') \mid \cdots \mid A_p(prog, s, s') \mid \\ & R_1(d_1, \ldots, d_{r_1}, prog, s, s') \mid \cdots \mid R_q(d_1, \ldots, d_{r_q}, prog, s, s') \end{aligned}$$

The predicate *Spec* is defined by stating one equation for each axiom and for each inference rule. This is the version for axioms A_i, $1 \leq i \leq p$:

$$\begin{aligned} & Spec(A_i(prog, s, s')) \equiv \\ & \quad root(prog) = X_0\ \wedge root(prog\ |_{[1]}) = X_1 \wedge \cdots \wedge root(prog\ |_{[n]}) = X_n\ \wedge \\ & \quad \exists \text{ substitution } \tau\ .\ (\tau(\sigma_0) = s \wedge \tau(\sigma_1) = s' \wedge \\ & \quad \mathsf{Eval}(prog\ |_{[l_1]}, \tau(\sigma_0)) = value_1 \wedge \cdots \wedge \mathsf{Eval}(prog\ |_{[l_{m_i}]}, \tau(\sigma_0)) = value_{m_i}) \end{aligned}$$

The first line specifies that axiom A_i belongs to production $X_0 ::= X_1 \cdots X_n$ and can only be applied to programs of that form. $prog\ |_{[i]}$ denotes the i-th direct subtree of $prog$. The second line describes that the general states σ_0 and σ_1 which may contain variables as placeholders can be mapped to the concrete states s and s' by applying a substitution τ. The last line specifies that the evaluation conditions must be fulfilled in the state $s = \tau(\sigma_0)$.

The version of *Spec* for inference rules R_j, $1 \leq j \leq q$ needs additional conditions for the recursive correctness requirements. The first line is the recursive constraint requiring that all subtrees fulfill the specification. The last three lines require that each direct subtree is marked either with the same program or a direct subtree of the program. Furthermore, it is specified that the final state of the k-th subtree is the initial state of the $k+1$-st subtree, $1 \leq k \leq j-1$. The remaining requirements are the same as for an axiom:

$$\begin{aligned} & Spec(R_j(d_1, \ldots, d_{r_j}, prog, s, s')) \equiv Spec(d_1) \wedge \cdots \wedge Spec(d_{r_j})\ \wedge \\ & \quad root(prog) = X_0\ \wedge root(prog\ |_{[1]}) = X_1 \wedge \cdots \wedge root(prog\ |_{[n]}) = X_n\ \wedge \\ & \quad \exists \text{ substitution } \tau\ .\ (\tau(\sigma_0) = s \wedge \tau(\sigma_{r_j}) = s' \wedge \\ & \quad \mathsf{Eval}(prog\ |_{[l_1]}, \tau(\sigma_0)) = value_1 \wedge \cdots \wedge \mathsf{Eval}(prog\ |_{[l_{m_i}]}, \tau(\sigma_0)) = value_{m_i}\ \wedge \\ & \quad \forall l \in \{1, \ldots, r_j\}\ .\ (mark(d_l) = (prog', s_1, s_2) \Rightarrow \\ & \quad (i_l = 0 \Rightarrow prog = prog' \wedge i_l > 0 \Rightarrow prog\ |_{[i_l]} = prog')\ \wedge \\ & \quad root(prog') = X_{i_l} \wedge \tau(\sigma_{l-1}) = s_1 \wedge \tau(\sigma_l) = s_2)) \end{aligned}$$

Spec is a specification with respect to Definition 3.1. Hence, there exists a monotone specification function *spec* with least and greatest fixed point, $lfp(spec)$ and $gfp(spec)$, on the set $\mathcal{D}$ of marked derivation trees. If $d \in \mathcal{D}$ is marked with $(P, prog, s, s')$ for $P \in \{A_1, \ldots, A_p, R_1, \ldots, R_q\}$,

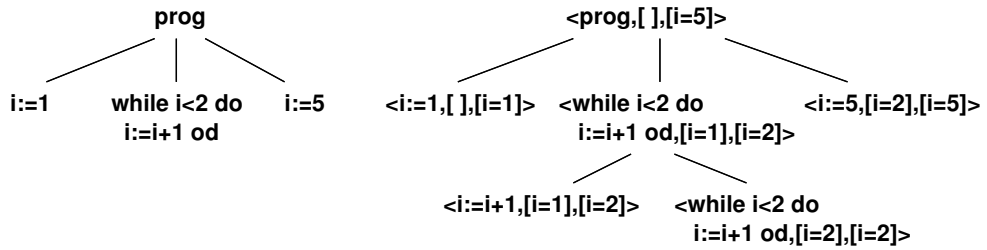


Figure 4.1: Semantics of a Terminating Program

then we say that d is a derivation tree for *prog* and s and that s is the initial and s' the final state of d.

A priori, there is no direction in the definition of the markings. Nevertheless, in all existing natural semantics specifications, such a direction exists. For each marking $(P, prog, s, s')$ of a derivation tree, $P \in \{A_1, \ldots, A_p, R_1, \ldots, R_q\}$, the program *prog* and the initial state s are defined coinductively while the final state s' is defined inductively. Even if the execution does not terminate and the final state is not uniquely defined, the state transitions performed during execution are still uniquely determined.

Definition 4.1 (Deterministic Natural Semantics Specification) *A natural semantics specification is deterministic if*

- *for all prog* $\in$ **Prog**, $s \in \mathcal{S}$, *there exists exactly one axiom or inference rule whose evaluation conditions are fulfilled in state s and which is applicable to prog (i.e. if the axiom or inference rule belongs to the production* $X_0 ::= X_1 \cdots X_n$, *then* $root(prog) = X_0$ *and for* $l \in \{1, \ldots, n\}$, $root(prog\,|_{[l]}) = X_l$*).*
- *For each axiom and inference rule, if prog and initial state s are known, then all evaluation conditions can be computed by a terminating computation.*
- *For each axiom, if prog and the initial state s are given, then the final state* s' *can be computed uniquely, also by a terminating computation.* $\diamond$

One can also consider the case that there are specifications such that no final state can be computed because, e.g., there might be no applicable rule. Such a case is called a *stuck computation*. To keep the presentation simple, we do not discuss such situations here.

4.2 Classical Inductive Interpretation

The classical interpretation of natural semantics defines semantics only for terminating programs. In this section, we give first an example for a terminating computation. Then we prove that for all terminating programs, the final state is unique.

Example 4.1 (Terminating Execution) *Assume that a state during execution is a list of pairs of variables with their current values. Assume further that the program prog as given on the left-hand side in Figure 4.1 is to be executed in state* [ ]*, i.e., no variable has been assigned a value yet. Then the semantics of the program is represented by the derivation tree d on the right-hand side in Figure 4.1. This example demonstrates the two-level hierarchy of coinductive and inductive structures in program semantics: The program prog and the initial state s are*

defined coinductively. Their definition starts at the root of the derivation tree and is propagated through the tree until its leaves are reached. At the leaves, the coinductive part of semantics, i.e. the state transition behavior, is connected with the inductive part, i.e. the computation of the final state. The definition of the final state is inductive since it starts at the leaves and proceeds along the derivation tree structure up to the root. ◇

Theorem 4.1 *Let Spec be a deterministic natural semantics, spec the corresponding specification function and lfp(spec) its least fixed point on the set $\mathcal{D}$ of marked derivation trees. Let $\mathcal{S}$ be the set of states and* **Prog** *the set of abstract syntax trees or subtrees thereof. For each program prog* $\in$ **Prog**, $s_0 \in \mathcal{S}$, *if the execution of prog starting in s_0 terminates, then there exists exactly one derivation tree $d \in \mathcal{D}$ for prog and s_0. The final state of d is uniquely determined.* ◇

Proof: By Induction on the (finite) structure of d:

Induction Base: If there is an axiom A_i such that its evaluation conditions are fulfilled in s_0 and which is applicable to *prog*, then there exists a unique final state $s' \in \mathcal{S}$ such that $< prog, s_0 > \rightarrow s'$. Because *Spec* is deterministic, no other axiom or inference rule is applicable, hence d is uniquely determined.

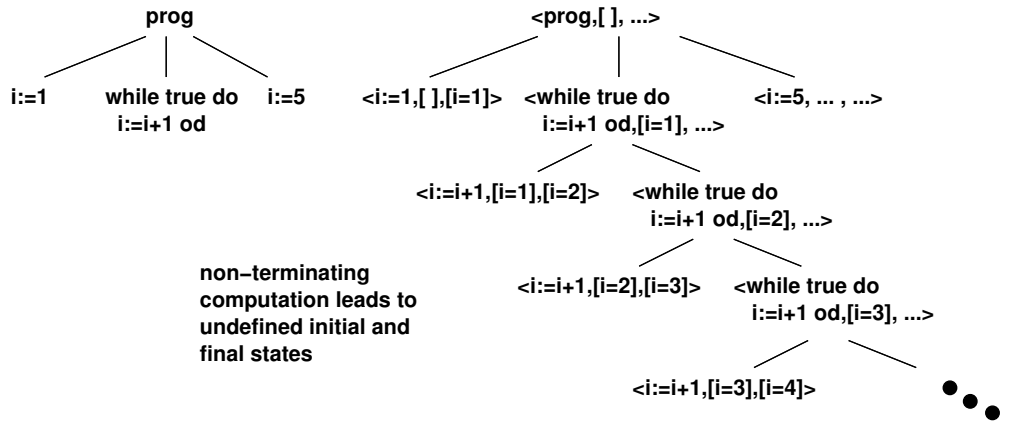
Induction Step: Let R_j, $1 \leq j \leq q$, be the unique inference rule applicable to *prog* whose evaluation conditions are fulfilled. If this rule has r_j assumptions, then the derivation tree d for *prog* and s_0 has r_j direct subtrees. The first subtree is uniquely determined because it is a derivation tree for some program *prog′* and s_0 where either *prog′* = *prog* or *prog′* is a direct subtree of *prog*, as specified by R_j. Due to the induction hypothesis, there exists a unique state s_1 which is the final state of the first direct subtree of d, $< X_{l_1}, s_0 > \rightarrow s_1$. s_1 is also the initial state for the second subtree of d. By repeating this reasoning, we conclude that all direct subtrees of d have unique initial and unique final states. The unique final state of the last subtree of d also serves as the final state of d. Hence, the derivation tree d for *prog* and s_0 is uniquely determined. ■

4.3 Coinductive Interpretation

If a program *prog* does not terminate when started in an initial state s_0, then the derivation tree for *prog* and s_0 has infinite height. This means that the coinductive and the inductive definition flow of the semantics cannot be connected since there are no leaves. In consequence, there is no unique derivation tree for *prog* and s_0. As an illustration, consider this example:

Example 4.2 (Non-Terminating Execution) *As in Example 4.1, each state is a list of pairs of variables and their current value. The semantics of the program with the non-terminating while-loop on the left-hand side is represented by the infinite derivation tree on the right-hand side. The annotation "..." means that the value of the respective initial or final state is not uniquely determined. Thus, there are several derivation trees for prog and $s_0 = [\ ]$, all for which the relation between the markings of parent and children nodes is valid with respect to the specification. Even though not all states are uniquely defined, these derivation trees define a unique infinite state transition sequence, as we prove below.* ◇

Definition 4.2 (State Transition Sequence) *Let Spec be a deterministic natural semantics specification, spec the corresponding specification function, prog be a program, and s_0 the initial state of computation. Let $d \in gfp(spec)$ be a derivation tree of prog and s_0. Then the state transition sequence of d, prog and s_0 is defined as follows:*



state_sequence$(R_j(d_1, \ldots, d_{r_j}, prog, s_0, s)) =$
append$([s_0], state\_sequence(d_1), \ldots, state\_sequence(d_n))$
state_sequence$(A_i(prog, s_0, s)) = [s_0, s]$ *where* s *is the uniquely determined final state (cf. third case in Definition 4.1).* $\diamond$

Lemma 4.1 *Let Spec be a deterministic natural semantics specification, spec the corresponding specification function, prog be a program, and* s_0 *the initial state of computation. Let* $d \in gfp(spec)$ *be a derivation tree of prog and* s_0*. Then the state transition sequence state_sequence*$(d,$ *prog,* $s_0)$ *of* d*, prog and* s_0 *is uniquely defined.* $\diamond$

Proof: If d has finite height, then the statement of Lemma 4.1 follows from Theorem 4.1. If d has infinite height and direct subtrees $d_1, \ldots, d_r$, then let d_i, $1 \leq i \leq r$ be the first subtree of infinite height. All subtrees $d_1, \ldots, d_{i-1}$ have finite height and unique initial and final states. d_i has a unique initial state. By using the unary coinduction principle (In Theorem 3.2, let Q be the property that the roots of all finite subtrees as well as the first subtree with infinite height have uniquely determined initial states), we conclude that d_i has a unique state transition sequence. Since this sequence is infinite, its concatenation with the state transition sequences of the subtrees $d_{i+1}, \ldots, d_r$ does not have any effect. (The concatenation of an infinite list l with any other list l' is again the list l.) Hence, d has a well-defined state transition sequence. ■

When programs do not terminate, then they have in general more than one derivation tree, cf. Example 4.2. Nevertheless, their state transition sequences are always the same. To prove this, we first define the effective part *eff_part*(d) of a derivation tree which includes only those parts which can be reached, if one spends enough time, during computation. As illustration, Figure 4.2 shows the effective part of the non-terminating derivation tree of Example 4.2.

Definition 4.3 (Effective Part of Derivation Tree) *The effective part of a derivation tree* d*, eff_part*(d)*, is the tree defined as follows:*

- *eff_part*$(d) = d$ *if* d *has finite height, and*
- *eff_part*$(R_j(d_1, \ldots, d_{r_j}, prog, s, s')) = R_j(d_1, \ldots, d_{l-1}, eff\_part(d_l), prog, s, \perp)$ *where* $l \in \{1, \ldots, r_j\}$*, and* $d_1, \ldots, d_{l-1}$ *have finite height.* $\diamond$

Theorem 4.2 (Unique Effective Parts) *Let Spec be a deterministic natural semantics specification, prog a program and* s *the initial state of program execution. Let spec be the corresponding specification function and* $d, d' \in gfp(spec)$ *derivation trees for prog and* s*. Then eff_part*$(d) =$ *eff_part*(d')*.* $\diamond$

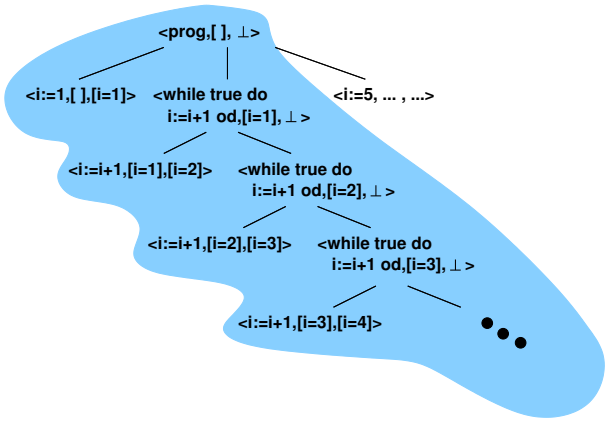


Figure 4.2: Effective Part of a Non-Terminating Derivation Tree

Proof: If d and d' both have finite height, then its follows directly from Theorem 4.1. Hence, assume that d or d' have infinite height and use the binary coinduction principle to prove that $\mathit{eff}\_\mathit{part}(d) = \mathit{eff}\_\mathit{part}(d')$. Therefore we prove the two conditions stated in Theorem 3.3 for $\mathit{eff}\_\mathit{part}(d)$ and $\mathit{eff}\_\mathit{part}(d')$.

First condition: The case that $d = A_i(\mathit{prog}, s, s')$ or $d' = A_i(\mathit{prog}, s, s')$, $i \in \{1, \ldots, p\}$, does not exist because then, both d and d' are equal to $A_i(\mathit{prog}, s, s')$ (and finite) because *Spec* is deterministic. Hence, $d = R_j(d_1, \ldots, d_{r_j}, \mathit{prog}, s, s')$ and $d' = R_l(d'_1, \ldots, d'_{r_l}, \mathit{prog}, s, s'')$, $j, l \in \{1, \ldots, q\}$. Because *Spec* is deterministic, it follows that $R_j = R_l$. Hence, without loss of generality, assume $d' = R_j(d'_1, \ldots, d'_{r_l}, \mathit{prog}, s, s'')$. Hence, we conclude that $\mathit{eff}\_\mathit{part}(d) = R_j(\mathit{eff}\_\mathit{part}(d_1), \ldots, \mathit{eff}\_\mathit{part}(d_{r_l}), \mathit{prog}, s, \bot)$ and $\mathit{eff}\_\mathit{part}(d') = R_j(\mathit{eff}\_\mathit{part}(d'_1), \ldots, \mathit{eff}\_\mathit{part}(d'_{r_l}), \mathit{prog}, s, \bot)$ which shows that the first condition of Theorem 3.3 is fulfilled.

Second condition: We need to show that those markings of the direct subtrees of $\mathit{eff}\_\mathit{part}(d)$ and $\mathit{eff}\_\mathit{part}(d')$ which do not denote trees are the same. These markings are the constructor symbols (i.e. the applied inference rules), the program annotations (element of **Prog**), and the initial states in the markings of the direct subtrees of $\mathit{eff}\_\mathit{part}(d)$ and $\mathit{eff}\_\mathit{part}(d')$.
d has at least one infinite subtree d_l, $1 \leq l \leq r_j$. The subtrees $d_1, \ldots, d_{l-1}$ have finite height. d_1 has the same initial state as d'_1, so it follows that $d_1 = d'_1$. (For a proof by contradiction, assume that $d_1 \neq d'_1$. Then assume that there is a first position when traversing d_1 and d'_1 in left-to-right order at which d_1 and d'_1 differ. But this is a contradiction to *Spec* being deterministic). In particular, we conclude that $\mathit{mark}(d_1) = \mathit{mark}(d_1{}')$. With the same reasoning repeated, we prove that $d_k = d'_k$ for $2 \leq k \leq l-1$. The markings of d_l and d'_l do not need to be equal as the final state of a non-terminating computation is not uniquely determined. Nevertheless, the parts of their markings which influence their effective parts are the same: The final state of d_{l-1} and d'_{l-1} are the same so that also the initial states of d_l and d'_l are equal; the programs $\in$ **Prog** in the marking of d_l and d'_l are the same because the same inference rule is applied at d and d' (*Spec* is deterministic); and because *Spec* is deterministic, there is exactly one inference rule R_l which is applicable at d_l and d'_l. Hence, we have $d_l = (R_l(\ldots), \mathit{prog}, s_l, s'_l)$ and $d_l = R_l(\ldots), \mathit{prog}, s_l, s''_l)$. From this, it follows that $\mathit{eff}\_\mathit{part}(d_l) = (R_l(\ldots), \mathit{prog}, s_l, \bot)$ and $\mathit{eff}\_\mathit{part}(d'_l) = (R_l(\ldots), \mathit{prog}, s_l, \bot)$ which completes the proof of the second condition of Theorem 3.3. Hence, we conclude that $\mathit{eff}\_\mathit{part}(d) = \mathit{eff}\_\mathit{part}(d')$. ■

Corollary 4.1 (Unique State Transition Sequence) *Let Spec be a deterministic natural semantics, prog a program, and s_0 the initial state of program execution. Let spec be the*

corresponding specification function and $d, d' \in gfp(spec)$ *be derivation trees for prog and* s_0*. Then* $state\_sequence(d, prog, s_0) = state\_sequence(d', prog, s_0)$. $\diamond$

Proof: This follows directly from Theorem 4.2 and the construction used in the proof of Lemma 4.1. ■

Definition 4.4 (Semantics of a Program) *Let Spec be a natural semantics, spec the corresponding specification function, and prog be a program. The semantics* $\mathbf{Sem}(prog)$ *of prog is defined as the set of all derivation trees in* $gfp(spec)$ *whose root is marked with prog:*

$$\mathbf{Sem}(prog) = \{d \in gfp(spec) \mid \exists s, s' \in \mathcal{S}, P \in \{A_1, \ldots, A_p, R_1, \ldots, R_q\} \; . \\ mark(root(d)) = (P, prog, s, s')\}$$

The semantics of prog for the initial state s_0 *is the set*

$$\mathbf{Sem}(prog, s_0) = \{d \in \mathbf{Sem}(prog) \mid \exists s' \in \mathcal{S}, P \in \{A_1, \ldots, A_p, R_1, \ldots, R_q\} \; . \\ mark(root(d)) = (P, prog, s_0, s')\} \quad \diamond$$

The set $\mathbf{Sem}(prog, s_0)$ might contain more than one derivation tree. In this case, the computation does not terminate. Subtrees of the derivation tree coming after (with respect to a depth-first left-to-right order) the non-terminating subtree do not contribute to the infinite state transition sequence since they will never be reached. Nevertheless, the effective parts of all derivation trees in $\mathbf{Sem}(prog, s_0)$ are the same and contain exactly those parts of the derivation trees which contribute to the state transition sequence of the program.

4.4 Applications of the Proof Calculus

Natural semantics, if interpreted coinductively, combines both aspects of programming language semantics in a very elegant and theoretically simple way. It defines a unique effective part for each program and each initial state. For all terminating executions, it also defines a unique final state. For all non-terminating executions, it describes uniquely the infinite state transition sequence of program execution. In this section, we show how the unary and binary coinduction principles can be applied for natural semantics and its coinductive interpretation.

4.4.1 Properties of Programming Languages

Assume that the semantics of a programming language is defined by a natural semantics specification. Assume furthermore that we want to prove a certain property Q for the states reached during execution of each program of that programming language. Such a property could e.g. be the type-safety of the language. We claim the following: To prove that Q holds in all states reached during execution of an arbitrary program, we need to verify the conditions stated in the proof rule in Figure 4.3.

Theorem 4.3 *Let Spec be a natural semantics specification and* Q *be a property such that the three requirements of the proof rule in Figure 4.3 hold. Then* Q *holds for all states reached during the execution of each – terminating and non-terminating – program of the programming language defined by Spec.* $\diamond$

Proof: Let p be an arbitrary program of the programming language defined by *Spec*, let d be a derivation tree for p for an arbitrary but fixed initial state σ such that $Q(\sigma)$ holds, and consider

Unary Proof Rule for Natural Semantics Specifications

To verify that a property Q holds for all states reached during execution of an arbitrary, potentially non-terminating program, prove these requirements:

- Show that $Q(\sigma)$ holds for all states σ in which program execution might start.
- Show for each axiom $< X, \sigma > \rightarrow \sigma'$: If $Q(\sigma)$, then $Q(\sigma')$.
- Show for each inference rule

$$\frac{\mathsf{Eval}(X_{l_1}, \sigma_0) = value_1, \ldots, \mathsf{Eval}(X_{l_m}, \sigma_0) = value_m, \quad < X_{i_1}, \sigma_0 > \rightarrow \ \sigma_1, \cdots, < X_{i_r}, \sigma_{r-1} > \rightarrow \ \sigma_r}{< X_0, \sigma_0 > \rightarrow \ \sigma_r} :$$

  If $\ Q(\sigma_0) \rightarrow Q(\sigma_1) \ \wedge \cdots \wedge \ Q(\sigma_{r-1} \rightarrow Q(\sigma_r), \ \text{then} \ \ Q(\sigma_0) \rightarrow Q(\sigma_r)$.

Figure 4.3: Proof Rule for Properties of Programming Languages

the effective part *eff_part*(d) of d. Then we need to verify that Q holds for all states contained in *eff_part*(d). Therefore we need an interleaved inductive and coinductive reasoning.

We use the unary coinduction principle (theorem 3.2) to verify that Q holds for the coinductively defined states (i.e. the initial states) in all markings in the effective part of d (by assuming that Q holds trivially for the state $\perp$). Therefore we need to verify the two conditions of the unary coinduction principle (Theorem 3.2) for *eff_part*(d) and the therein contained initial states. The first condition requires us to verify that Q holds in arbitrary initial states which is true because of the first requirement of the proof rule in Figure 4.3. To verify the second condition, we need an interleaved inductive and coinductive reasoning because the initial states are defined coinductively but the final states (which are also initial states in neighboured derivation subtrees) are specified inductively depending on the initial states. Consider the first l subtrees of d which have finite height. To prove that Q holds for all their initial and their final states requires an inductive proof along the axioms and inference rules of the specification (along the lines of the second and third requirement of the proof rule in Figure 4.3). If d has only l subtrees, then we are done at this point. Otherwise, consider the $l+1$-st subtree which is the last subtree contributing to the effective part of d. Its initial state is the uniquely determined final state of d_l, for which we have already shown that Q holds. Since *eff_part*(d) has no more subtrees, we have verified that the second condition of Theorem 3.2 holds which completes the proof. ■

As in case of the Hoare calculus rule for recursive procedures (cf. Section 2.1), the proof rule in Figure 4.3 is an overlay of two different proof rules: If the derivation tree is finite, i.e., if program execution terminates, then the proof rule describes an inductive proof showing that all initial and final states in the derivation tree fulfill property Q. If program execution does not terminate, i.e., if the derivation tree has an infinite subtree, then the proof obligations of the proof rule imply two facts: First, by an inductive argument they imply that the initial and final states in all finite subtrees coming "before" the infinite one fulfill Q. Furthermore, they imply coinductively that Q also holds for the initial states in the infinite subtree (structural argument along the third requirement of the proof rule in Figure 4.3).

4.4.2 Compiler Correctness

A correct compiler should preserve the observable behavior of the translated programs. This requirement is essential. In many practical applications, programs do not terminate and are not even intended to terminate (e.g. data bases, operating systems, software in embedded systems, reactive systems in general). If one wants to verify that software for such systems is translated correctly, the proof cannot be done by induction. The corresponding derivation trees and state transition sequences are not finite. Instead one needs a coinductive proof that the observable behavior, i.e. the state transition sequence is the same in the original and the translated program. The basis for coinductive reasoning is greatest fixed point semantics.

Example 4.3 (Verification of an Optimization) *Consider the non-terminating program from Example 4.2. An optimizing compiler might recognize that the while-loop does not terminate. Since the compiler is required to preserve the observable behavior, it cannot modify the while-loop. Nevertheless, the assignment* `i:=5` *will never be reached during any execution and can be eliminated. If the inference rules for the while-loop (cf. Section 1.5) are interpreted inductively with the least fixed point, then such a transformation cannot be verified as being semantics-preserving.* ◇

Proof Sketch: In the greatest fixed point interpretation, we can first show that the while-loop does not terminate and then do a coinductive proof showing that the state transition sequences in the original and in the optimized program are the same. The first part, proving non-termination, can be done by a simple contradiction argument.

For the second proof, the coinductive one, we need to use the binary coinduction principle to prove that the effective parts of the derivation trees d and d' of the original and the optimized program are the same. The reasoning is completely analogous to the one in the proof of Theorem 4.2. First we consider the case that d and d' are both finite (which does not hold since the original program does not terminate). Hence at least d or d' are of infinite height and we use the binary coinduction principle to prove that the effective parts of d and d' are the same. Therefore we need to verify the two conditions of Theorem 3.3: According to the first condition, we need to show that the markings (i.e. the applied inference rules, the initial states, and the annotated programs) at the root nodes of the derivation trees are the same. This holds trivially (assuming that the natural semantics specification is deterministic). According to the second condition we need to show that the markings of the direct subtrees of d and d' which contribute to their effective parts are marked with the same initial state, the same program, and the same applied inference rule or axiom. Therefore we first consider the first l finite subtrees of d and show that d' has the same first l finite subtrees (by induction). Then we finish the proof by showing that the first subtrees of infinite height in d and d' contribute with the same applied inference rule, the same initial state, and the same annotated program to the effective parts of d and d' which completes the proof of the two conditions in the binary coinduction principle. Therefore we conclude that the effective parts of the original loop-program and the optimized loop-program are the same and, hence, their state transition sequences are also the same. ■

4.5 Related Work

The results of this chapter contradict the common understanding that natural semantics can only describe terminating computations (cf. [NN99, Sch96] or any other textbook or lecture notes of your choice) while structural operational semantics, also called small-step semantics, is additionally suited to describe non-terminating programs. Rather it is the common least fixed

point interpretation of natural semantics that defines semantics only for terminating programs. Usually a greatest fixed point semantics is assumed implicitly for structural operational semantics but without drawing the conclusion that coinductive proof rules are necessary. For both specification formalisms, both interpretations are possible, cf. also Section 6.2. A least fixed point interpretation of structural operational semantics defines semantics only for terminating programs by assigning them a finite state transition sequence and an "undefined" to all non-terminating programs. The only related research attempting to widen the interpretations of natural semantics is described in [IS98]. It defines a coinductive interpretation of natural semantics by translating it into a small-step format. Induction is used to reason about the thereby defined finite and infinite state transition sequences. This is only a half-hearted approach as it does not separate between the coinductive character of the state transitions and the inductive nature of the final result defined on top of them. We want to emphasize that induction is not the appropriate proof method to reason about the state transition behavior of programs, see also our explanations about induction and coinduction for lists at the end of this section.

This insight has severe consequences as it reveals that most equivalence proofs for programs based on structural induction do hold only if the programs terminate. In particular, this holds for the research efforts in proving the static type safety of Java [DE99, Sym99, NvO98, vO01]. All proofs are based on inductive arguments and, hence, do only hold for terminating programs. Therefore, one would assume that the machine-checking approach needs extra assumptions when applying the inductive proof principle. Indeed, the inductive proofs did not work without further assumptions: In the machine-checking approach documented in detail in [vO01], the maximal recursive depth of evaluation is restricted to a finite number, cp. paragraphs 5.3.2 and 5.7.2 of [vO01]. The same assumption has been applied in the mechanical verification of the correctness of a compiling specification [DV01] using the PVS system [ORS92], cp. section 4 of [DV01].

Finally a remark about lists as degenerated trees: When we reason about state transition sequences, we start at the root node of the lists and infer properties for a child node from its parent node. This is coinduction. It is different from induction where we start at the leaf of the list and construct (finite) lists by using already constructed smaller lists. The inductive view is used for defining results of computations. Thereby we assume that the list-degenerated state transition tree has a leaf as base case. Since these dual definition and proof principles look so similar for lists, there is often no clear distinction between them. Nevertheless, in other areas this distinction is well-known and accepted. E.g. in functional programming languages, it can be found as the distinction between strict (inductive) and lazy (coinductive) evaluation. When dealing with formal semantics of programming languages, it is the difference between state transition sequences of unbounded but always finite length and sequences of infinite length. In the first case, one can deal with all finite sequences, no matter how large they are. In the second case, one can also deal with infinite sequences. To capture also the infinite sequences, one needs to use coinduction. Induction can only deal with finite state transition sequences of unbounded length but is not appropriate for infinite state transition sequences. This difference is strikingly documented in the machine-checking approaches discussed above which need extra assumptions restricting proofs to terminating computations only.

4.6 Conclusions of the Case Study

Our investigations are based on the observation that programming language semantics has two dual aspects, the state transition behavior and the computation of the final result. In consequence, programming language semantics needs to be defined by a two-layer hierarchy: First the potentially infinite state transition behavior is defined coinductively. On top of this coin-

ductive structure, an inductive definition specifies the final result of computation. It is unique only if the state transition sequence is finite. This connection between the coinductive and the inductive structure of program semantics seems to be essential and not only a characteristics of natural semantics, whereupon its greatest fixed point interpretation demonstrates it particularly clearly. In this sense, we have established natural semantics as a well-balanced formalism for the semantics of programming languages as it models both aspects sufficiently and evenly. Axiomatic semantics, in particular the Hoare calculus [Hoa69] is an equally balanced formalism. It defines the preconditions coinductively and the postconditions inductively. This implies that the postconditions do only hold if execution terminates. Especially the rule for recursive procedures demonstrates this interleaved coinductive/inductive reasoning (cf. Section 2.1).

Our developments in this chapter follow the insight that a combination of algebraic and coalgebraic methods can be used successfully in the specification of and reasoning about programming languages, especially for potentially non-terminating processes. We have shown that the state transition behavior of programs must be defined coinductively and that the final result is to be defined inductively on top of it. While automated theorem provers, e.g. Isabelle [Pau02], have the potential to reason coinductively, the standard practice does not use it. All automated as well as "paper and pencil" proofs based on natural semantics exploit induction and, hence, do only hold for terminating computations. The results of this chapter demonstrate that this is not sufficient and should be replaced by coinductive reasoning.

5 A Categorical Formulation of Coalgebras and Coinduction

Induction and coinduction are completely dual definition and proof principles. It is the aim of this chapter to expose this duality which can be demonstrated best when formalizing induction and coinduction in category-theoretical notions. As a motivation, we start with typical coalgebras in computer science in Section 5.1 and with a comparison between inductive and coinductive definitions which are widespread in many areas of computer science in Section 5.2. Then we proceed in Section 5.3 by introducing basic category theory, especially the concept of functors and in particular polynomial functors. In Section 5.4, we show that functors can be used to describe abstract data types. Based on this formalization, we explain the concept of algebras and induction in Section 5.5 and the concept of coalgebras and coinduction in Section 5.6. Finally we formalize bisimulation and the coinductive proof principle in Section 5.7.

Most of the material in this chapter can be found in existing literature as for example [JR97] which is a well-written introduction to the field and which can be highly recommended as a sequel to this chapter. In contrast to the presentation of coalgebras and coinduction in [JR97] (and basically all publications on coinduction), our exposition in this report strives to offer intuitive "pictures" in form of finite and infinite trees and to show their connection to the category-theoretical account of the topic via a relatively simple set-theoretic setting (which is given in Chapter 3). Therefore, at the end of this chapter, we come back to the pictures of finite and infinite trees and provide an intuitive understanding of coinduction which carries on where the gentle introduction in Section 2.2 left off.

5.1 Typical Coalgebras in Computer Science

Coalgebras are generally used to describe state-based computations in state transition systems. Formally coalgebras are functions which take a state as input and return one or several eventually modified successor states together with observations. Returning several successor states can serve different purposes. For example, they can express non-determinism or, as in the case of natural semantics derivation trees, cf. Chapter 4, sequential composition. The idea when using coalgebras and coinduction is to classify a system completely by its observable behavior. The description of this observable behavior is expressed by the unique homomorphism from an arbitrary coalgebra into the final coalgebra of the same signature. Remember that we have mentioned this unique mapping already in Section 2.2. In the course of this chapter, especially in the sections following the current one, we give a formal existence and uniqueness proof of this homomorphism. In this section, we discuss a series of typical examples and start by considering one of the most important coalgebras, namely that of non-terminating deterministic state transition systems:

Non-Terminating Deterministic State Transition Systems

Consider a system with two buttons, i.e. functions, value and next. When pressing the button value, we get some observation from a set A of possible observations of the system. The internal

state of the system is not modified by the function value. In contrast, the function next modifies the internal state and returns the new internal state of the system. Such a system can be described by the coalgebra consisting of the pair of functions value and next where X denotes the state space of the system:

$$\langle \mathsf{value}, \mathsf{next} \rangle : X \longrightarrow A \times X$$

From this system, we can observe the following behavior: By pressing the button value, i.e. by applying the function value to the current state, we observe some value val. Furthermore, we can pass over to the next state by the function next to the current state. And we can repeat this as often as we want. Hence, by doing so, we observe an eventually infinite list of observations $(\mathsf{val}_1, \mathsf{val}_2, \mathsf{val}_3, \ldots) \in A^{\mathbb{N}}$. $A^{\mathbb{N}}$ denotes the set of functions from $\mathbb{N}$ to A, and val_i is the observation after applying the function next i times. The behavior function $\mathsf{beh} : X \longrightarrow A^{\mathbb{N}}$ captures exactly these possible observations.

Possibly Non-Terminating Deterministic State Transition Systems

In practical situations, we often deal with systems for which we do not know whether they will terminate or not. If these systems are deterministic, then they can be described by the following coalgebra where A denotes the set of possible observations and A the state space of the system:

$$\langle \mathsf{value}, \mathsf{next} \rangle : X \longrightarrow (A \times X) \cup \{*\}$$

Let $x \in X$ be the current state. Then $\mathsf{next}(x)$ denotes the succeeding state. If $\mathsf{next}(x) = *$, then computation stops and the state transition system terminates. No observation can be obtained any more since value cannot be applied to $*$ but only to states in X. If $\mathsf{next}(x) \neq *$, then the computation can go on by applying the function next again.

As in the case of non-terminating deterministic state transition systems we get observations which can be described by elements from $A^{\mathbb{N}}$. Additionally, if the system terminates, the observations can be described by finite lists of elements of A, $(\mathsf{val}_1, \mathsf{val}_2, \mathsf{val}_3, \ldots, \mathsf{val}_n) \in A^*$ where A^* denotes the set of all finite lists of elements of A. The behavior function for such state transition systems is defined as a function $\mathsf{beh} : X \longrightarrow A^{\mathbb{N}} \cup A^*$.

In Section 6.1, we formalize the operational semantics of deterministic programming languages by defining such a function $\mathsf{beh}_{\mathsf{prog}} : X \longrightarrow A^{\mathbb{N}} \cup A^*$ for each program prog. Given an initial state $x \in X$, $\mathsf{beh}_{\mathsf{prog}}(x)$ returns either a finite list of observations in case the program terminates when started in state x, or, if program execution does not terminate, $\mathsf{beh}_{\mathsf{prog}}(x)$ returns an infinite list of observations.

Possibly Non-Terminating Non-Deterministic State Transition Systems

Now consider a state transition system which in every state might non-deterministically pass over to one of two possible successor states. Such a system can be described by the following coalgebra where A denotes again the set of possible oberservations, first the first of the two non-deterministic successor states, and second the second non-deterministic successor:

$$\langle \mathsf{value}, \mathsf{first}, \mathsf{second} \rangle : X \longrightarrow A \times (X \cup \{*\}) \times (X \cup \{*\})$$

For such a coalgebra, we get observations which can be described by finite and infinite trees whose nodes are marked with observations from A such that each node has either exactly two subtrees or is a leaf. Hence, if $T(A)$ denotes the set of these trees, the behavior function is a function $\mathsf{beh} : X \longrightarrow T(A)$.

Also, general non-deterministic state transition systems can be described by coalgebras. Assume that a state might have arbitrarily many possible successor states.

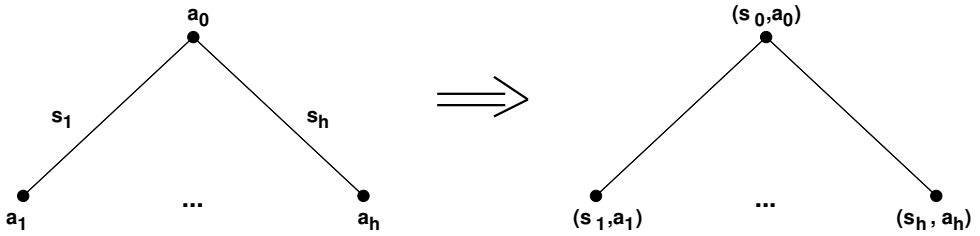


Figure 5.1: Behavior of Deterministic Finite Automata

Coalgebras for Deterministic Finite Automata

Finite automata are often used to describe state transition systems with only finitely many states. Given a current state, the successor state is determined depending on some input $s \in S$ for a finite input alphabet S. Assume furthermore that each state $x \in X$, X being the state space of the automaton, allows for an observation $\mathsf{val}(x) \in A$. A coalgebra describing such a deterministic finite automaton can be defined as a function as follows:

$$\langle \mathsf{value}, \mathsf{next} \rangle : X \longrightarrow A \times (X \cup \{*\})^S$$

The behavior of these deterministic finite automata can be defined by trees whose nodes are each marked with a pair consisting of an observation and an input. To see this, let us argue first that the behavior can be described by trees whose nodes and edges are marked: The root node of such a tree is labeled with the observation in the current state. If a node n is labeled with the observation of a state $x \in X$, then n has S outgoing edges which are marked with the elements $s_1, \ldots, s_h$ of S. Each such edge represents a possible input for the automaton. If an outgoing edge e of a node n, n representing state $x \in X$, is marked with $s \in S$ and if s is an admissible input for x, i.e., there exists a successor state for x and s, then the root node of the subtree of n reached via the edge e is marked with the observation in the successor state of x and s. If s is not an admissible input, then the root node of the subtree of n reached via edge e is marked with $*$ and is a leaf. Instead of having markings $s \in S$ at the edges and markings $a \in A$ at the nodes, we can equivalently push the edge markings into the markings of the nodes to which the corresponding edges are pointing. Note that this tree transformation corresponds to the set-theoretic equivalence of S^A and $A \times S$. Figure 5.1 illustrates the trees. Hence, the behavior of deterministic finite automata can be described by ordered $S \times A$-labeled trees.

This description can be generalized for non-deterministic finite automata as follows:

$$\langle \mathsf{value}, \mathsf{next} \rangle : X \longrightarrow A \times \mathcal{P}(X \cup \{*\})^S$$

The function value describes the observable behavior of the automaton depending on its current state. The function next maps the current state to a subset of $(X \cup \{*\})$, depending on the current input $s \in S$. This resulting subset represents the set of non-deterministic choices of the automaton with respect to the current state and the current input s.

In the above coalgebraic formulation of deterministic and non-deterministic automata, we have not used the fact that the state space is finite. Hence, the same kind of coalgebras can be used to describe general deterministic and non-deterministic state transition systems.

Coalgebras for Classes in Object-Oriented Programming Languages

Coalgebras have been successfully applied to describe the behavior of classes in object-oriented programming languages, cf. for example [Hui01]. Typically, one does not want to describe

the complete state space of the objects of a class but only those aspects that can be observed. Assume for example that we have a class Student which captures all the features of a student enrolled at a university. Then we have observer functions which return the relevant information for a given student in a specific state, his address, the courses he has taken so far, and so on. Furthermore we have functions which modify this state, for example a function change_address which updates the student's address information. The behavior of such a class can be constrained by equations, for example address(change_address(stud, addr)) = addr. In this example, stud is the student and represents the state to which we have access via the function address which returns the student's address. We can modify this address, and thereby also modify the current state stud, by the function change_address. This function takes a state and returns the modified state in which the address is set to the value of the second parameter.

Concludingly we see that in a coalgebraic description of object-oriented classes, we only specify their observable behavior by coalgebras, by possibly restricting it by constraints. The actual state space of a class is hidden and can only be observed and modified by those functions that are visible for the user of that class.

5.2 Inductive versus Coinductive Definitions

In the previous section, we have shown that many systems in computer science can be described by coalgebras. In particular, the behavior of such coalgebras is captured by a behavior function which maps each initial state to the complete observable behavior which is defined in terms of finite and infinite lists, trees, or abstract data types in general. In this section, we explain how such behavior functions can be given systematically. Therefore we introduce the coinductive way of defining functions and contrast it with its dual definition method which is induction.

For this purpose, it is helpful to recall the intuition for induction and coinduction given in Section 2.2 and Chapter 3: Assume that we have an abstract data type with its base and constructor functions. In the inductive view, we interpret this abstract data type with all finite trees constructed from the base and constructor symbols. We can then define a function inductively on these trees by specifying its values for the base functions, i.e. for the leaves in the trees of the abstract data type, and, furthermore, by specifying how the value of an inner node in these trees is constructed from the values of its direct successor nodes. Conversely in the coinductive view, we interpret the abstract data type with all finite as well as infinite trees consisting of the base and constructor symbols. A function is defined coinductively on these trees by specifying how the value of a successor node in the tree is derived from the value of its direct predecessor. These two dual definition directions are demonstrated in Figure 3.1.

Nomenclature: Up to now, we have called finite and infinite trees summarizingly *abstract data types*. From now on, to keep our presentation in line with the existing literature, we distinguish between them and call finite trees *abstract data types* and (potentially) infinite trees *process types*. In the same way, we call the base and constructor symbols which are used to build the trees *constructors* or *constructor functions* in case of abstract data types (i.e. for finite trees) and *destructor* or *observer (function)* in case of process types.

Constructor functions describe how elements are constructed by describing how larger trees are constructed from smaller ones while destructors tell us how trees are taken into pieces by specifying how a tree is disassembled into its direct subtrees. This duality is directly reflected in the difference between inductive and coinductive definitions:

Induction: An inductive definition of a function f defines the values on f on all constructors. Thereby, the result of f is defined for all trees in the abstract data type which can be constructed by the constructor functions.

Coinduction: A coinductive definition of a function f defines the values of all destructor functions on each possible result $f(x)$. Thereby, the observable behavior is defined for all possible $f(x)$.

To demonstrate these two principles, we consider inductive and coinductive function definitions for the abstract data type of finite lists, denoted by A^*, and for the process type of infinite lists, denoted by $A^{\mathbb{N}}$, whereby A is the type of the list entries. Finite lists are defined with the constructors nil and cons, infinite lists are defined with the destructors head and tail:

List Constructors				**List Destructors**			
nil:		$\longrightarrow$	A^*	head:	$A^{\mathbb{N}}$	$\longrightarrow$	A
cons:	$A \times A^*$	$\longrightarrow$	A^*	tail:	$A^{\mathbb{N}}$	$\longrightarrow$	$A^{\mathbb{N}}$

One can think of these functions in the intuitive way: nil returns an empty list, cons takes a list and an element and puts that element as additional entry to the list, head takes a list and returns its first element, and tail takes a non-empty list and returns the list which is the result of chopping off the first element.

Now we are ready to define functions inductively on finite lists and coinductively on infinite lists: For example, the function len which determines the length of each finite list is defined inductively as usual by these two equations:

$$\begin{aligned} \mathsf{len}(\mathsf{nil}) &= 0 \\ \mathsf{len}(\mathsf{cons}(a, l)) &= 1 + \mathsf{len}(l) \end{aligned}$$

As always in the case of inductive definitions, we see that the constructors of the abstract data type appear "inside" the function to be defined.

This is a typical example of a coinductive function definition: Assume that we have a function f which is defined for elements in A, $f : a \longrightarrow A$. This can be an arbitrary function. Then we can define its extension on lists of elements of A, $\mathsf{ext}(f) : A^{\mathbb{N}} \longrightarrow A^{\mathbb{N}}$, in the following way:

$$\begin{aligned} \mathsf{head}(\mathsf{ext}(f)(l)) &= f(\mathsf{head}(l)) \\ \mathsf{tail}(\mathsf{ext}(f)(l)) &= \mathsf{ext}(f)(\mathsf{tail}(l)) \end{aligned}$$

It should be intuitively clear that $\mathsf{ext}(f)$ is the function that applies f to all elements in a given list. In the rest of this chapter, we show that functions defined in this way exist and are uniquely defined by such equations. Therefore we need some basic category theory, especially the notion of functors, which we introduce in the next section.

5.3 Basic Category Theory and Polynomial Functors

Definition 5.1 (Category) *A category* **C** *is specified by:*

- *A collection of* objects*,*
- *A collection of* morphisms*, also called* arrows*,*

- *Two operations that assign to each morphism f an object $dom(f)$, also called its* domain, *and an object $cod(f)$, also called its* codomain. *The notation $f : A \longrightarrow B$ is used to express that f is a morphism with domain A, $dom(f) = A$, and codomain B, $cod(f) = B$.*
- *A composition operator $\circ$ for morphisms that assigns to each pair of morphisms f and g with $cod(f) = dom(g)$ a composite morphism $g \circ f : dom(f) \longrightarrow cod(g)$ such that $\circ$ is associative. That is, for all morphisms $f : A \longrightarrow B$, $g : B \longrightarrow C$, and $h : C \longrightarrow D$, it holds that $h \circ (g \circ f) = (h \circ g) \circ f$.*
- *For each object A, an identity morphism $id_A : A \longrightarrow A$ such that for any arrow $f : A \longrightarrow B$, $id_B \circ f = f$ and $f \circ id_A = f$.* $\diamond$

There are many examples for categories in computer science, cf. for example [Pie91] for an overview. We work mainly in the category **SET** whose collection of objects is the collection of all sets and whose morphisms are the total functions between sets.[1] It is easy to verify that **SET** is indeed a category: $\circ$ is the usual function composition, and id_A is the identity function on set A. Note that each morphism is also contained as an object in the category **SET** since in set theory, each function $f : X \longrightarrow Y$ from X to Y is itself a set, namely $f = \{(x, y) \mid x \in X \wedge y \in Y \wedge y = f(x)\}$.

An important concept in category theory are functors. These are functions that transform categories. Therefore a functor is applied to all objects as well as to all morphisms. Intuitively, a functor preserves the structure of a category. Formally, a functor is defined as follows [Pie91]:

Definition 5.2 (Functor) *Let* **C** *and* **D** *be categories. A functor $F : \mathbf{C} \longrightarrow \mathbf{D}$ is a mapping that maps each object A from* **C** *to an object $F(A)$ in* **D** *and each morphism $f : A \longrightarrow B$ from* **C** *to a morphism $F(f) : F(A) \longrightarrow F(B)$ such that for all objects A in* **C** *and all morphisms f, g with $cod(f) = dom(g)$, the following equations hold:*

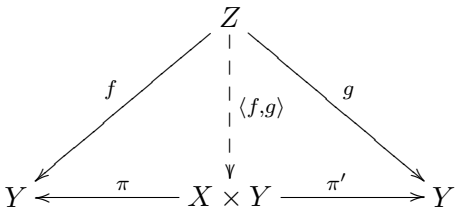
1. $F(id_A) = id_{F(A)}$
2. $F(g \circ f) = F(g) \circ F(f)$ $\diamond$

The requirements in this definition make sure that whenever a functor is applied to a category, identities and composites are preserved and that the result is again a category. If we are working in the category **SET**, then functors are functions that can be applied to sets as well as to functions between sets. For our development concerning induction and coinduction, we need in particular the functors product, coproduct, powerset, and constants, which are introduced in the following. The definitions given here are tailored to the category **SET** by considering set-theoretical products, coproducts, and powersets.

The Product Functor

Given two sets X and Y, their Cartesian product is defined as $X \times Y = \{(x, y) \mid x \in X \wedge y \in Y\}$. There are the usual projection functions $\pi : X \times Y \longrightarrow X$ and $\pi' : X \times Y \longrightarrow Y$ defined by $\pi(x, y) = x$ and $\pi'(x, y) = y$. This definition can be lifted to functions: Given two functions $f : Z \longrightarrow X$ and $g : Z \longrightarrow Y$, there is a unique product function $\langle f, g\rangle : Z \longrightarrow X \times Y$ with $\pi \circ \langle f, g\rangle = f$ and $\pi' \circ \langle f, g\rangle = g$, namely $\langle f, g\rangle(z) = (f(z), g(z))$ for all $z \in Z$. It is easy to see that $\langle \pi, \pi'\rangle = \mathsf{id} : X \times Y \longrightarrow X \times Y$ and that $\langle f, g\rangle \circ h = \langle f \circ h, g \circ h\rangle : W \longrightarrow X \times Y$ for functions $h : W \longrightarrow Z$.

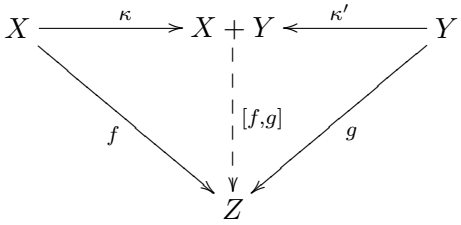
[1] Most of the theory applies to general categories as well.



The product operator can also be applied to functions. Given two functions $f : X \longrightarrow X'$ and $g : Y \longrightarrow Y'$, we define their product by $f \times g : X \times Y \longrightarrow X' \times Y'$ and $f \times g(x, y) = (f(x), g(y))$. It is easy to see that $\mathsf{id}_X \times \mathsf{id}_Y = \mathsf{id}_{X \times Y}$ and that $(f \circ h) \times (g \circ k) = (f \times g) \circ (h \times k)$. From these equations, it is easy to show that the Cartesian product operator is a functor on the category **SET** and subcategories thereof. (A category **B** is a subcategory of category **A** if each object and each morphism in **B** is also an object and a morphism, resp., in **A**, and if, furthermore, composite and identity arrows in **B** are the same as in **A**.)

The Coproduct Functor

The coproduct $X + Y$ (also called disjoint sum) of two sets X and Y is defined as $X + Y = \{\langle 0, x\rangle \mid x \in X\} \cup \{\langle 1, y\rangle \mid y \in Y\}$. $X + Y$ is kind of a unification of X and Y with the additional property that the 0s and 1s in the defined pairs allow for a unique identification from which original set, X or Y, the respective element is stemming. There are coprojections $\kappa : X \longrightarrow X + Y$ and $\kappa' : Y \longrightarrow X + Y$. Note that in contrast to the projection functions for Cartesian products, the coprojections do not take elements from the coproduct as arguments but instead map into the coproduct. They are defined by $\kappa(x) = \langle 0, x\rangle$ and $\kappa'(y) = \langle 1, y\rangle$. As in case of products, the coproduct mapping can also be lifted to functions. For functions $f : X \longrightarrow Z$ and $g : Y \longrightarrow Z$, there is a unique function $f + g : X + Y \longrightarrow Z$ with $[f, g] \circ \kappa = f$ and $[f, g] \circ \kappa' = g$.



It is easy to see that the following definition of $[f, g]$ fulfills these properties:

$$[f, g](w) = \begin{cases} f(x) & \text{if } w = \langle 0, x\rangle \\ g(y) & \text{if } w = \langle 1, y\rangle \end{cases}$$

From these properties it follows directly that $[\kappa, \kappa'] = id$ and $h \circ [f, g] = [h \circ f, h \circ g]$.

This definition of coproducts can be lifted to functions as follows: Given two functions $f : X \longrightarrow X'$ and $g : Y \longrightarrow Y'$, their coproduct $f + g : X + Y \longrightarrow X' + Y'$ is defined by

$$(f + g)(w) = \begin{cases} \langle 0, f(x)\rangle & \text{if } w = \langle 0, x\rangle \\ \langle 1, g(y)\rangle & \text{if } w = \langle 1, y\rangle \end{cases}$$

The coproduct $f + g$ could equivalently be defined as $f + g = [\kappa \circ f, \kappa' \circ g]$. It is easy to verify that the operation $+$ is a functor because it preserves identities and composition, namely $id_X + id_Y = id_{X+Y}$ and $(f \circ h) + (g \circ k) = (f + g) \circ (h + k)$.

The Powerset Functor

Given a set X, there exists the operation that maps it to its power set: $X \mapsto \mathcal{P}(X)$. This operation is also a functor as it can be lifted to operate on functions $f : X \longrightarrow Y$ as follows: We define $\mathcal{P}(f) : \mathcal{P}(X) \longrightarrow \mathcal{P}(Y)$ by $U \mapsto \{f(x) \mid x \in U\}$ for all $U \subseteq X$.

From this definition, it follows easily that $\mathcal{P}(id_X) = id_{\mathcal{P}(X)}$ and $\mathcal{P}(f \circ g) = \mathcal{P}(f) \circ \mathcal{P}(g)$ which shows that the powerset operation is indeed a functor.

For many practical applications, it suffices to consider the powerset functor $\mathcal{P}_{\text{fin}}(-)$ that maps a set to all its finite subsets. In contrast to the full powerset functor, it has nicer theoretical properties, as we discuss in Section 5.6.

Identity and Constant Functors

There are some trivial functors. For example, the identity map $X \mapsto X$ is a functor. It maps each function to itself: $f \mapsto f$. Furthermore, for a set C, there is the constant map $X \mapsto C$. It can be liftet to functions by mapping each function $f : X \longrightarrow Y$ to the identity function $id_C : C \longrightarrow C$. It is easy to verify that the identity and the constant map are functors.

Polynomial Functors

As already mentioned at the beginning of this section, we are only interested in the category **SET** and its subcategories. Therefore it suffices for us to consider only functors between these categories. These set-theoretical functors can be described by specifying their effect on sets. To illustrate this, consider the following functor, an example taken from [JR97]:

$$T(X) = X \ + \ (C \times X)$$

This functor maps each set X to the set $X+(C\times X)$ and each function $f : X \longrightarrow Y$ to a function $T(f) : T(X) \longrightarrow T(Y)$. $T(f)$ is the function $f + (id_C \times f) : X + (C \times X) \longrightarrow Y + (C \times Y)$ defined by:

$$w \mapsto \begin{cases} \langle 0, f(x)\rangle & \text{if } w = \langle 0, x\rangle \\ \langle 1, (c, f(x))\rangle & \text{if } w = \langle 1, (c, x)\rangle \end{cases}$$

In the rest of this chapter and this entire report, we are only considering polynomial functors:

Definition 5.3 (Polynomial Functors) *A polynomial functor is built from constants, the identity functor, products, coproducts, and finite powersets.* ◇

There are two kinds of objects in categories which can be characterized by the uniqueness of ingoing and outgoing arrows:

Definition 5.4 (Initial and Final Objects) *Let* **C** *be a category. An object o in* **C** *is called* initial *if there is exactly one morphism from o to each other object o' in* **C**. *An object o in* **C** *is called* final *if there is exactly one morphism from each object o' in* **C** *to o.* ◇

We denote a singleton set $\{*\}$ with 1, $*$ being its typical element. Since we abstract from concrete elements, the element's name does not matter here. For all sets X, there is exactly one function $X \longrightarrow 1$. This means that 1 is final in the category **SET**. Conversely, the empty set $\emptyset$ is initial in the category **SET** since for all sets X, there is exactly one function $\emptyset \longrightarrow X$ from $\emptyset$ into X. $\emptyset$ is also denoted by 0.

For polynomial functors, certain isomorphisms exist which involve the initial and final elements 0 and 1 and which can be easily verified using basic set theory:

$$
\begin{array}{lcl@{\qquad}lcl}
X \times Y & \cong & Y \times X & X + Y & \cong & Y + X \\
1 \times X & \cong & X & 0 + X & \cong & X \\
X \times (Y \times Z) & \cong & (X \times Y) \times Z & X + (Y + Z) & \cong & (X + Y) + Z \\
X \times 0 & \cong & 0 & X \times (Y + Z) & \cong & (X \times Y) + (X \times Z)
\end{array}
$$

With these identities, especially the last one describing disjunctivity of products over coproducts, we can transform each polynomial functor into a disjunctive normal form. In the following section, we use this fact to describe abstract data types as polynomial functors.

5.4 Abstract Data Types as Polynomial Functors

In this section we show that abstract data types can be described by polynomial functors in the sense that each abstract data type corresponds to a polynomial functor. Thereby we use the fact that abstract data types are characterized by the signatures of their operations whereas the names of these operations as well as the names of the carrier sets of the abstract data types do not matter. Given an operation $f : X \times \cdots \times X \longrightarrow X$ of an abstract data type, we represent it by the functor T with $T(X) = X \times \cdots \times X$, for all sets X. If an abstract data type has more than one operation, we combine these operations into a single one using the coproduct mapping:

Definition 5.5 (Functor of an Abstract Data Type) *Assume that an abstract data type has the operations $f_1, f_2, \ldots, f_n$ with the arities $\mathrm{ar}_{f_1}, \mathrm{ar}_{f_2}, \ldots, \mathrm{ar}_{f_n}$. Then we represent this abstract data type by the functor T defined by $T(X) = X^{\mathrm{ar}_{f_1}} + \cdots + X^{\mathrm{ar}_{f_n}}$ where X^n denotes the n-fold product $X \times \cdots \times X$. Constant operations are represented by the functor $T(X) = 1$. Operations involving elements from another set A (for example the sort of elements in a list) are represented for example by $T(X) = A \times X$ for the* cons*-operation for lists.* ◇

Example 5.1 (Natural Numbers and Lists) *The natural numbers $\mathbb{N}$ are defined as the term algebra built from the constructor functions $0 : \mathbb{N}$ (or equivalently $0 : 1 \longrightarrow \mathbb{N}$) and $succ : \mathbb{N} \longrightarrow \mathbb{N}$. These two functions are represented by the functor $T(X) = 1 + X$. The abstract data type* list *is an algebra $[\mathsf{nil}, \mathsf{cons}] : 1 + A \times X$.* ◇

From these examples, it becomes clear that each abstract data type can be characterized by a polynomial functor. This functor is unique modulo the isomorphisms stated at the end of the preceeding section. In the following two sections, we show how algebras and induction as well as coalgebras and coinduction can be classified using functors.

5.5 Algebras and Induction

In the classical definition, an algebra consists of a carrier set together with operations on this set. Here, we define an algebra equivalently as a carrier set together with one single function. This function corresponds to the coproduct of all the operations in the classical definition and is described by a functor.

Definition 5.6 (Algebra) *Let T be a functor. An* algebra *of T, or a T-algebra, is a pair (U, a) consisting of a set U and a function $a : T(U) \longrightarrow U$. U is called the* carrier *of the algebra, and the function a the* algebra structure *or the* operation *of the algebra.* ◇

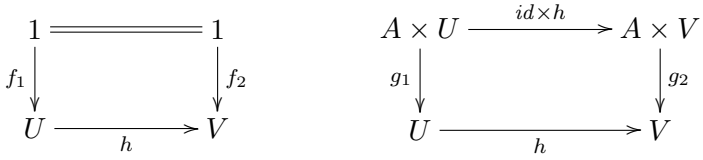
In classical and universal algebra, homomorphisms between algebras are defined as structure-preserving functions between algebras of the same signature. This definition is not changed

in principle here but looks somewhat different because it involves the functor that defines the structure of the algebras which are related by the homomorphism.

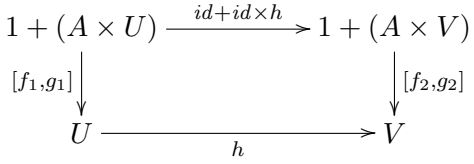
As an example, assume that we have a functor T with $T(X) = 1 + A \times X$ (the list functor, cf. Example 5.1) and two algebras defined by $a : T(U) \longrightarrow U$ and $b : T(V) \longrightarrow V$ of functor T with operations $f_1 : 1 \longrightarrow U$ and $g_1 : a \times U \longrightarrow U$ as well as $f_2 : 1 \longrightarrow V$ and $g_2 : a \times V \longrightarrow V$. A homomorphism h between the two algebras is a function $h : U \longrightarrow V$ such that these two requirements are fulfilled:

$$h \circ f_1 = f_2 \text{ and } h \circ g_1 = g_2 \circ (id \times h)$$

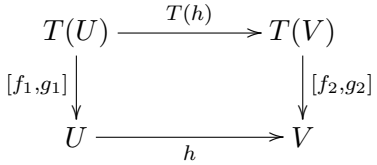
These equations are expressed with *commuting diagrams*, a notation commonly used in category theory (Diagrams like the two below are said to commute if for every pair of nodes U and V, all paths from U to V are equal. Thereby each path in the diagram denotes an arrow.):



By using the coproduct operation, these two diagrams can be combined into one:

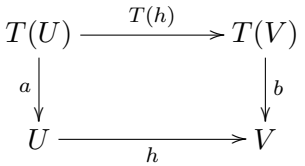


By employing the list functor T with $T(X) = 1 + A \times X$, we can write this diagram equivalently as follows, a characterization which is given completely in terms of the involved functor T:



This characterization of algebra homomorphisms in terms of the functor describing the underlying algebra structure is captured in the following definition:

Definition 5.7 (Algebra Homomorphism) *Let T be a functor with algebras $a : T(U) \longrightarrow U$ and $b : T(V) \longrightarrow V$. $h : U \longrightarrow V$ is an* algebra homomorphism *from (U, a) to (V, b) if $h \circ a = b \circ T(h)$, i.e., if the following diagram commutes:*



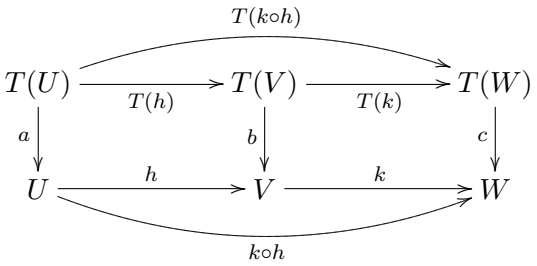
◇

Based on this definition, we can show that the concatenation of two algebra homomorphisms is again an algebra homomorphism.

Lemma 5.1 (Concatenation of Algebra Homomorphisms) *Let T be a functor with algebras $a : T(U) \longrightarrow U$, $b : T(V) \longrightarrow V$, and $c : T(W) \longrightarrow W$. Furthermore, let $h : U \longrightarrow V$ be an*

algebra homomorphism from (U, a) to (V, b), and let $k : V \longrightarrow W$ be an algebra homomorphism from (V, b) to (W, c). Then the concatenation of h and k, $k \circ h : U \longrightarrow W$, is again an algebra homomorphism. ◇

Proof: Consider this diagram:



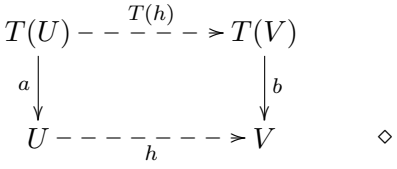
in which these equations hold by Definition 5.7 and by the defining property of functors to preserve composition, cf. Definition 5.2:

$$k \circ h \circ a = k \circ b \circ T(h) = c \circ T(k) \circ T(h) = c \circ T(k \circ h)$$

Hence, this completes the proof that $k \circ h$ is an algebra homomorphism from algebra $a : T(U) \longrightarrow U$ to algebra $c : T(W) \longrightarrow W$. ■

Initial algebras are a common concept in the theory of abstract data types. Here we define them not as usual as a term algebra but instead by their characterizing property that there exists a unique homomorphism from the initial algebra into each other algebra with the same algebra structure.

Definition 5.8 (Initial Algebra) *An algebra $a : T(U) \longrightarrow U$ is initial if for each algebra $b : T(V) \longrightarrow V$, there exists a unique algebra homomorphism from a to b, expressed with the following diagram:*

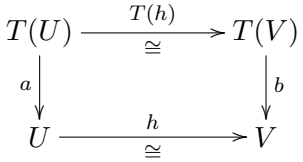


◇

This uniqueness property can be used to define homomorphic functions from one algebra to another as well as to prove existence of them. Existence corresponds to the inductive proof principle while uniqueness corresponds to the inductive definition principle. Even though initial algebras are often defined as the set of closed terms (= ground terms) that can be formed with the constructors, i.e. functions of the algebras, we do not need to know how the elements of the initial algebra look like. It completely suffices to know that the initial algebra with the properties stated in Definition 5.8 exists. Before looking at an illustrating example, let us state an important property of initial algebras:

Theorem 5.1 (Properties of Initial Algebras) *Let T be a functor.*

1. *Initial T-algebras, if they exist, are unique up to isomorphism of algebras. That means that if there are two initial algebras $a : T(U) \longrightarrow U$ and $b : T(V) \longrightarrow V$, then there exists a unique isomorphism of algebras $h : U \xrightarrow{\cong} V$:*

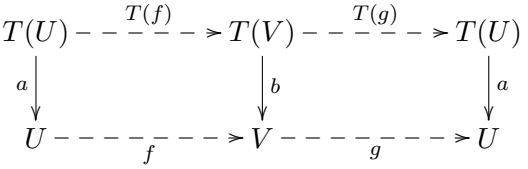


2. *The operation of an initial algebra is an isomorphism: If $a : T(U) \longrightarrow U$ is an initial algebra, then a has an inverse $a^{-1} : U \longrightarrow T(U)$.* $\diamond$

The first property states that functors have essentially at most one initial algebra since all initial algebras of a given functor are isomorphic to each other. Therefore we can speak of **the** initial algebra by identifying isomorphic ones. The second property states that an initial algebra $a : T(U) \longrightarrow U$ is a fixed point $T(U) \cong U$ of the functor T.

The characterization of initial algebras in Theorem 5.1 is different than the one which we gave in Chapter 3 where we described an initial algebra as the algebra whose carrier set contains all finite trees constructed with the constructor functions of the algebra, i.e. the carrier set containing all ground terms. The characterization here is purely in terms of the homomorphic properties of the initial algebra, no matter how its elements look like.

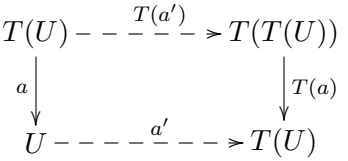
Proof: First we prove that initial algebras are unique up to isomorphism. Therefore, assume that there are two initial algebras $a : T(U) \longrightarrow U$ and $b : T(V) \longrightarrow V$. This implies, due to the initiality of the two algebras, that there exist unique algebra homomorphisms $f : U \longrightarrow V$ and $g : V \longrightarrow U$, represented in this diagram:



From this diagram, especially from the uniqueness of the functions f and g, it follows that the concatenation of f and g exists and is a homomorphism from (U, a) to (U, a). Also the identity function on U is a homomorphism from (U, a) to (U, a). Since (U, a) is an initial algebra, the homomorphism (U, a) to itself is uniquely determined and, hence, equals the identity map: $g \circ f = id$. Analogously we can show that the concatenation $f \circ g$ equals the identity map on V and is also the unique homomorphism from (V, b) to (V, b). Since $f \circ g = id = g \circ f$, it follows that f is an isomorphism of algebras and the initial algebra is uniquely determined up to isomorphism.

For the second part of Theorem 5.1, we need to show that the function $a : T(U) \longrightarrow U$ of the algebra (U, a) is an isomorphism. We show this by defining an inverse function $U \longrightarrow T(U)$. Thereby we use the initiality of (U, a) which allows us to define functions out of U into arbitrary algebras. In our case, we need a function into $T(U)$ (a function $U \longrightarrow T(U)$) so we put an algebra structure on the set $T(U)$. To complete the proof, we only need a suitable definition for it. The algebra with the operation $T(a) : T(T(U)) \longrightarrow T(U)$, which arises by applying T to the function a, will do this job for us as we demonstrate in the rest of this proof.

Due to the initiality of $a : T(U) \longrightarrow U$, there exists a function $a' : U \longrightarrow T(U)$ with $T(a) \circ T(a') = a' \circ a$, cf. the following diagram:



The function $a \circ a' : U \longrightarrow U$ is an algebra map $(U, a) \longrightarrow (U, a)$:

$$\begin{array}{ccccc} T(U) & \xrightarrow{T(a')} & T(T(U)) & \xrightarrow{T(a)} & T(U) \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle T(a)} & & {\scriptstyle a}\downarrow \\ U & \xrightarrow[a']{} & T(U) & \xrightarrow[a]{} & U \end{array}$$

Because algebra maps $(U, a) \longrightarrow (U, a)$ are unique, $a \circ a' = id$. Moreover, $a' \circ a = T(a) \circ T(a')$ (by definition of a') $= T(a \circ a')$ (since functors preserve composition) $= T(id)$ (as shown just above) $= id$. Hence, $a : T(U) \longrightarrow U$ is an isomorphism with a' being its inverse. The notation $a : T(U) \xrightarrow{\cong} U$ is used to express this isomorphism. ■

The preceeding proof has shown that initiality can be exploited by putting a suitable algebra structure on some set (in the proof, the set $T(U)$ and the structure $T(A) : T(T(U)) \longrightarrow T(U)$). This technique is also used in the following example which considers the natural numbers. In particular, we show in this example that the natural numbers are an initial algebra of the functor T with $T(X) = 1 + X$ and that functions can be defined inductively on other sets by defining a suitable mapping from such a set into the natural numbers.

Example 5.2 (Initiality of the Natural Numbers) *Consider the set of natural numbers* $\mathbb{N}$ *with its zero and successor function* $0 : 1 \longrightarrow \mathbb{N}$ *and* $s : \mathbb{N} \longrightarrow \mathbb{N}$. *These two functions can be combined into one, giving us the algebra* $[0, s] : T(\mathbb{N}) \longrightarrow \mathbb{N}$ *of the functor* T *with* $T(X) = 1 + X$. *In the following, we show that the map* $[0, s] : T(\mathbb{N}) \longrightarrow \mathbb{N}$ *is the initial algebra of the functor* T, *a fact that characterizes the natural numbers uniquely up to isomorphism, cf. Theorem 5.1. Moreover, we show how this fact can be used to define inductively functions from* $\mathbb{N}$ *to any other set.*

To prove initiality, we assume an arbitrary algebra $[u, h] : T(U) \longrightarrow U$ *and define a homomorphism (also called* mediating *homomorphism)* $f : \mathbb{N} \longrightarrow U$. *We do this in the intuitive way and show afterwards that this definition has the desired properties. So the intuitive way goes like this where we write* u *instead of* $u(*)$, $*$ *being the typical inhabitant of set* 1*:* $f(n) = h^n(u)$. *In other words, we define*

$$\begin{aligned} f(0) &= u \\ f(n+1) &= h(f(n)) \end{aligned}$$

These equations express that the diagram below commutes and that f *is a homomorphism:*

$$\begin{array}{ccc} 1 + \mathbb{N} & \xrightarrow{id+f} & 1 + U \\ {\scriptstyle [0,s]}\downarrow & & \downarrow{\scriptstyle [u,h]} \\ \mathbb{N} & \xrightarrow[f]{} & U \end{array}$$

Now we need to show that it indeed holds that f *is a homomorphism. Therefore we distinguish whether for an arbitrary element* $x \in 1 + \mathbb{N}$, $x = (0, *) = \kappa(*)$ *or* $x = (1, n) = \kappa'(n)$, $n \in \mathbb{N}$, *holds. For the first case* $x = (0, *) = \kappa(*)$, *the following holds:*

$$f([0, s](\kappa(*))) = f(0) = u = [u, h](\kappa(*)) = [u, h]((id + f)(\kappa(*)))$$

In the second case $x = (1, n) = \kappa'(n)$, $n \in \mathbb{N}$, *we get this:*

$$f([0, s](\kappa'(n))) = f(s(n)) = h(f(n)) = [u, h](\kappa'(f(n))) = [u, h]((id + f)(\kappa'(n)))$$

From these two cases, we conclude that $f([0,s](x)) = [u,h]((id+f)(x))$ for all $x \in 1+\mathbb{N}$ which makes the above diagram commute, i.e., $f \circ [0,s] = [u,h] \circ (id+f)$. This completes the proof that f is a homomorphism but we still need to show that it is the only one from $\mathbb{N}$ to U:

To show that f is the unique homomorphism between $(\mathbb{N},[0,s])$ and $(U,[u,h])$, we assume that there is another homomorphism $g : \mathbb{N} \longrightarrow U$ that also satisfies $g \circ [0,s] = [u,h] \circ (id+g)$. From this assumption, it follows directly that $g(0) = u$ and $g(n+1) = h(g(n))$. Hence, we conclude that $g(n) = f(n)$ by induction on n and, hence, that $g = f : \mathbb{N} \longrightarrow U$. (To avoid this argument "by induction on n" in order to prove uniqueness completely from the properties of the involved functions, the same kind of reasoning can be used as in the proof of Theorem 5.1. Here we simplify the proof for sake of brevity and because this proof is a special instance of the proof for Theorem 5.1).

Up to now we have shown how to define a function from $\mathbb{N}$ to some other algebra of the same structure inductively. This technique can be used to define functions from $\mathbb{N}$ into arbitrary other sets by putting a suitable algebraic structure on this set. Consider for example the definition of the function $f : \mathbb{N} \longrightarrow \mathbb{Q}$ with $f(n) = 2^{-n}$. This function can be defined inductively with

$$f(0) = 1 \quad \text{and} \quad f(n+1) = \tfrac{1}{2} f(n)$$

To define this function f by using the initiality of $[0,s] : 1+\mathbb{N} \longrightarrow \mathbb{N}$, we put an appropriate algebra structure $1+\mathbb{Q} \longrightarrow \mathbb{Q}$ on the set $\mathbb{Q}$. This algebra on $\mathbb{Q}$ is basically defined by the two above defining equations for f, namely:

$$\begin{array}{cc} 1 \xrightarrow{1} \mathbb{Q} & \mathbb{Q} \xrightarrow{\frac{1}{2}(-)} \mathbb{Q} \\ * \mapsto 1 & x \mapsto \frac{1}{2}x \end{array}$$

As usual, we combine these two functions into a single one:

$$1+\mathbb{Q} \xrightarrow{[1,\frac{1}{2}(-)]} Q$$

This function is an algebra on $\mathbb{Q}$. In this setting, the function $f(n) = 2^{-n}$ is uniquely determined by initiality of $\mathbb{N}$. This implies that the diagram below commutes:

$$\begin{array}{ccc} 1+\mathbb{N} & \xrightarrow{id+f} & 1+\mathbb{Q} \\ {\scriptstyle [0,s]}\downarrow & & \downarrow{\scriptstyle [1,\frac{1}{2}(-)]} \\ \mathbb{N} & \xrightarrow[f]{} & \mathbb{Q} \end{array}$$

This example has demonstrated that initiality of $(\mathbb{N},[0,s])$ can be used to define functions inductively into any other set U. Therefore one needs to put an algebra structure on the set U which corresponds to the induction clauses of the desired function. It should be clear that this does not only work for the natural numbers and the functor T with $T(X) = 1+X$ but for arbitrary functors and their initial algebras. ◇

5.6 Coalgebras and Coinduction

In the previous section, we have defined algebras as functions that take an input from a structured domain (the set $T(X)$ for a given functor T) and return an unstructured output. Coalgebras are exactly the dual notion of an algebra: they take an unstructured element as input and return a structured one, i.e., they are functions $X \longrightarrow T(X)$ for some functor T.

To fill this dual view on algebras and coalgebras with some intuition, recall the gentle motivation for initial algebras and final coalgebras and their set-theoretic foundation in Chapters 2 and 3: Initial algebras are the least fixed point of an abstract data type while final coalgebras

are the greatest fixed point of the abstract data type. Remember that we have described abstract data types by appropriate functors. We can think of the elements in the least fixed point as consisting of finite trees which are built with the constructors of the abstract data type and whose nodes carry labels (the observations). In contrast, elements in the greatest fixed point are all finite and infinite trees which are described with the constructors of the abstract data type and whose nodes also carry labels. Applying the function of a coalgebra to an element, i.e. a tree, of its domain corresponds to the destruction of this tree; we get all its direct subtrees as well as the labels of its root node (the observations). In contrast, the algebra function is an overlay of the constructor functions (defined by the coproduct operator) that can be used to build trees in the least fixed point. In the algebra case, a constructor function takes direct subtrees and uses them to construct an assembled tree.

While it is helpful to have the intuition of finite and infinite trees in mind, it is not necessary for the development in this and the previous section where we describe algebras and coalgebras completely in terms of their homomorphic mapping properties. Algebras are defined by functions going into the carrier set. These functions tell us how to construct elements. Dually, coalgebras are functions going out of the carrier set. They tell us some but not necessarily all information about the elements of the carrier set which means that we have only limited access to the elements of a coalgebra.

Definition 5.9 (Coalgebra) *Let T be a functor. A coalgebra of T, or a T-coalgebra, is a pair (U, a) consisting of a set U and a function $a : U \longrightarrow T(U)$. U is called the* carrier*. The function a is called the* structure *or the* operation *of the coalgebra. Since coalgebras are often used to describe state transition systems, U is also called the* state space *of the coalgebra.* ◇

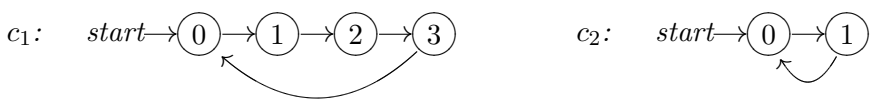
Example 5.3 (Infinite Lists) *Consider the coalgebras of non-terminating deterministic state transition systems in Section 5.1. We have described them by a function*

$$\langle \mathsf{value}, \mathsf{next} \rangle : X \longrightarrow A \times X$$

The operation value *gives us an observation, an element of some set A, while the function* next *produces a next element in the state space X. We can apply the functions* value *and* next *as often as we want. This defines an infinite list $(\mathsf{val}_1, \mathsf{val}_2, \mathsf{val}_3, \ldots) \in A^{\mathbb{N}}$ consisting of the observations which have been made by applying the functions* value *and* next*. Note that this list is also a tree, namely a degenerate infinite tree. This infinite sequence is all that we can observe about an element of this coalgebra.* ◇

It is important to notice that two coalgebras can behave identically without actually being equal. We call two coalgebras *observationally indistinguishable* or *bisimilar* if their behavior is identical. In Section 5.7 we show that final coalgebras are equal if they are bisimilar. This fact makes up the essence of coinduction.

Example 5.4 (Bisimilarity) *Consider the set of coalgebras of deterministic state transition systems with no final states and in particular the two automata c_1 and c_2 from this domain depicted with the following two diagrams:*



These two diagrams show two coalgebras of the functor T with $T(X) = \{0, 1\} \times X$ which we define as follows:

$c_1 = \langle \mathsf{value}, \mathsf{next} \rangle : \{0,1,2,3\} \longrightarrow \{0,1\} \times \{0,1,2,3\}$ *with*
$\mathsf{value}(x) = (x \bmod 2)$ *and* $\mathsf{next}(x) = ((x+1) \bmod 4)$, *and*

$c_2 = \langle \mathsf{value}, \mathsf{next} \rangle : \{0,1\} \longrightarrow \{0,1\} \times \{0,1\}$ *with*
$\mathsf{value}(x) = x$ *and* $\mathsf{next}(x) = ((x+1) \bmod 2)$.

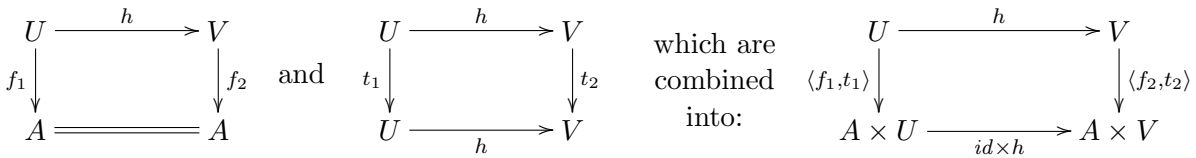
Both coalgebras have the same behavior, namely the infinite sequence $(0,1,0,1,0,1,\ldots)$ *of alternating* 0*s and* 1*s. Nevertheless, they are not equal.* $\diamond$

Before proceeding with theoretical results concerning coalgebras, let us look at one more common kind of coalgebras, namely those which describe possibly non-terminating deterministic state transition systems, cf. also Section 5.1.

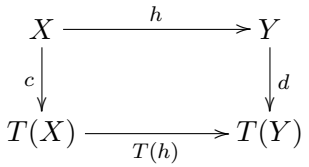
Example 5.5 (Possibly Non-Terminating Deterministic State Transition Systems) *The behavior of possibly non-terminating deterministic state transition systems consists of finite (in case of termination) and infinite (in case of non-termination) lists of observations. Such systems can be described as coalgebras* $c : X \longrightarrow 1 + A \times X$ *of the functor* T *with* $T(X) = 1 + A \times X$. *For each element* $x \in X$, $c(x) = \kappa(*)$ *or* $c(x) = \kappa'(a, x')$ *with* $(a, x') \in A \times X$. *The first case represents the situation that the state transition system stops because there is no succeeding state. In this case, the behavior of the system is described by a finite list of observations. In the second case, which represents the situation that the system does not hold, we get an observation* $a \in A$ *and a successor state* $x' \in X$ *in which the system can proceed. The observable behaviors are elements from the set* $A^* \cup A^{\mathbb{N}}$, *the union of finite and infinite lists over* A, *the set of observations.* $\diamond$

Up to now, we have seen observations which are lists. In general, observations are elements of the final coalgebra of the involved functors. In our examples, the final coalgebra of the functor T with $T(X) = A \times X$ consists of the infinite lists over A, $A^{\mathbb{N}}$. The final coalgebra of the functor T with $T(X) = 1 + A \times X$ contains in addition the finite lists over A, $A^* \cup A^{\mathbb{N}}$.

To define and prove these ideas and facts formally, we need the notion of a coalgebra homomorphism. A coalgebra homomorphism behaves analogously as an algebra homomorphism in the sense that it commutes with the coalgebra functions. For example, consider the functor $T(X) = A \times T(X)$ defining coalgebraically infinite lists and two T-coalgebras $\langle f_1, t_1 \rangle : X \longrightarrow A \times X$ and $\langle f_2, t_2 \rangle : Y \longrightarrow A \times Y$. A function $h : X \longrightarrow Y$ is a homomorphism of coalgebras if $f_2 \circ h = f_1$ and $t_2 \circ h = h \circ t_1$, i.e., the following two diagrams (which can be summarized as in the case of algebra homomorphisms into a single diagram, cf. Section 5.5) commute:



Definition 5.10 (Coalgebra Homomorphism) *Let* T *be a functor with coalgebras* $c : X \longrightarrow T(X)$ *and* $d : Y \longrightarrow T(Y)$. $h : X \longrightarrow Y$ *is a coalgebra homomorphism (or a coalgebra map) from* (X, c) *to* (Y, d) *if* $T(h) \circ c = d \circ h$, *i.e., if the following diagram commutes:*



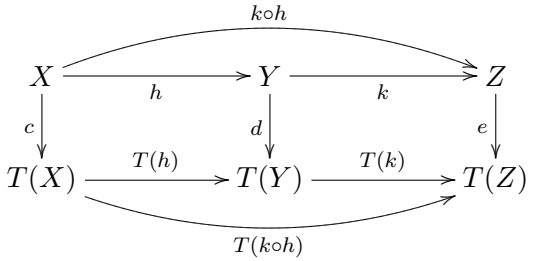
A final *coalgebra $d : W \to T(W)$ is a coalgebra such that for every other coalgebra $c : X \longrightarrow T(X)$, there is a unique map of coalgebras $(X, c) \longrightarrow (W, d)$.* ◇

It is easy to verify that the identity map is a coalgebra homomorphism. As in the case of algebra homomorphisms, the concatenation of two coalgebra homomorphisms is again a coalgebra homomorphism:

Lemma 5.2 (Concatenation of Coalgebra Homomorphisms) *Let T be a functor with coalgebras $c : X \longrightarrow T(X)$, $d : Y \longrightarrow T(Y)$, and $e : Z \longrightarrow T(Z)$. Furthermore, let $h : X \longrightarrow Y$ be a coalgebra homomorphism from (X, c) to (Y, d), and let $k : Y \longrightarrow Z$ be a coalgebra homomorphism from (Y, d) to (Z, e). Then the concatenation of h and k, $k \circ h : X \longrightarrow Z$, is again a coalgebra homomorphism.* ◇

The proof is completely analogous to the proof of the analogous property for algebra homomorphisms, cf. proof of Lemma 5.1.

Proof: Consider this diagram:



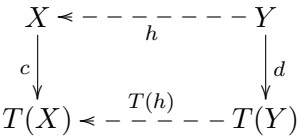
in which the following equations hold by Definition 5.10 and by the defining property of functors to preserve composition, cf. Definition 5.2:

$$e \circ k \circ h = T(k) \circ d \circ h = T(k) \circ T(h) \circ c = T(k \circ h) \circ c$$

This completes the proof that $k \circ h$ is a coalgebra homomorphism from coalgebra $c : X \longrightarrow T(X)$ to coalgebra $e : Z \longrightarrow T(Z)$. ■

Dually to the concept of initial algebras, we define final coalgebras in the definition below. Initial algebras are characterized by the fact that there exists a unique algebra homomorphism from the initial algebra of a given functor into any other algebra of the same functor. In the case of coalgebras, we have an exactly dual characterization which defines a final coalgebra of a given functor by the property that there exists a unique homomorphism from any other coalgebra of the same functor into the final coalgebra.

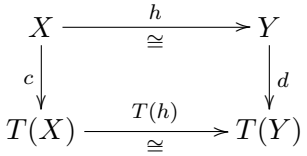
Definition 5.11 (Final Coalgebra) *A coalgebra $c : X \longrightarrow T(X)$ is final if for each coalgebra $d : Y \longrightarrow T(Y)$, there exists a unique coalgebra homomorphism $h : Y \longrightarrow X$ from (Y, d) to (X, c), expressed with the following commuting diagram:*



We can use the finality property of coalgebras to define functions that go into a final coalgebra. This is dual to the possibility that arises from the initiality property of algebras which allows us to define functions going out of the initial algebra into an arbitrary algebra of the same functor. Before looking at examples of final coalgebras and coinductive function definitions, let us state some important properties of final coalgebras which are completely dual to the properties of initial algebras stated in Theorem 5.1. Also the proof is completely analogous so that it is not really necessary to state it here, which we do nevertheless as a demonstration of that duality.

Theorem 5.2 (Properties of Final Coalgebras) *Let T be a functor.*

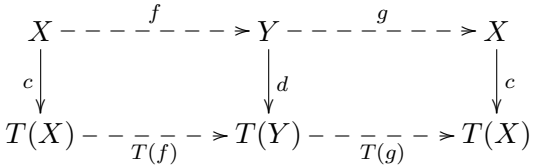
1. *Final T-coalgebras, if they exist, are unique up to isomorphism of coalgebras. That means that if there are two final coalgebras $c : X \longrightarrow T(X)$ and $d : Y \longrightarrow T(Y)$, then there exists a unique isomorphism of coalgebras $h : X \xrightarrow{\cong} Y$:*



2. *The operation of a final coalgebra is an isomorphism: If $c : X \longrightarrow T(X)$ is a final coalgebra, then c has an inverse $c^{-1} : T(X) \longrightarrow X$. In particular, the final coalgebra $X \longrightarrow T(X)$ is a fixed point $X \cong T(X)$ of the functor T.* ◇

As in the case of initial algebras, this theorem says that all final coalgebras of a given functor are isomorphic to each other which is why we speak of **the** final coalgebra. The preceeding theorem characterizes final coalgebras not by their elements but instead entirely by their homomorphic properties. The proof is completely dual to the proof of its dual (Theorem 5.1).

Proof: First we prove that final coalgebras are unique up to isomorphism. Therefore, assume that there are two final coalgebras $c : X \longrightarrow T(X)$ and $d : Y \longrightarrow T(Y)$. This implies, due to the finality of the two coalgebras, that there exist unique coalgebra homomorphisms $f : X \longrightarrow Y$ and $g : Y \longrightarrow X$, represented in this diagram:



From this diagram, especially from the uniqueness of the functions f and g, it follows that the concatenation of f and g exists and is a homomorphism from (X, c) to (Y, d). Also the identity function on X is a homomorphism from (X, c) to (X, c). Since (X, c) is a final coalgebra, the homomorphism (X, c) to itself is uniquely determined and, hence, equals the identity map: $g \circ f = id$. Analogously we can show that the concatenation $f \circ g$ equals the identity map on Y and is also the unique homomorphism from (Y, d) to (Y, d). Since $f \circ g = id = g \circ f$, it follows that f is an isomorphism of coalgebras and the final coalgebra is uniquely determined up to isomorphism.

For the second part of Theorem 5.2, we need to show that the function $c : X \longrightarrow T(X)$ of the coalgebra (X, c) is an isomorphism. We show this by defining an inverse function $T(X) \longrightarrow X$. Thereby we use the finality of (X, c) which allows us to define functions into X out of arbitrary coalgebras. In our case, we need a function out of $T(X)$ (a function $T(X) \longrightarrow X$) so we put a coalgebra structure on the set $T(X)$. To complete the proof, we only need a suitable definition for it. The coalgebra with the operation $T(c) : T(X) \longrightarrow T(T(X))$, which arises by applying T to the function c, will do this job for us as we demonstrate in the rest of this proof.

Due to the finality of $c : X \longrightarrow T(X)$, there exists a unique function $c' : T(X) \longrightarrow X$ with $T(c') \circ T(c) = c \circ c'$, cf. the following diagram:

$$\begin{array}{ccc} X & \xleftarrow{\quad c' \quad} & T(X) \\ {\scriptstyle c}\downarrow & & \downarrow{\scriptstyle T(c)} \\ T(X) & \xleftarrow[\quad T(c') \quad]{} & T(T(X)) \end{array}$$

The function $c' \circ c : X \longrightarrow X$ is a coalgebra map $(X, c) \longrightarrow (X, c)$:

$$\begin{array}{ccccc} X & \xleftarrow{\quad c' \quad} & T(X) & \xleftarrow{\quad c \quad} & X \\ {\scriptstyle c}\downarrow & & {\scriptstyle T(c)}\downarrow & & \downarrow{\scriptstyle c} \\ T(X) & \xleftarrow[\quad T(c') \quad]{} & T(T(X)) & \xleftarrow[\quad T(c) \quad]{} & T(X) \end{array}$$

Because coalgebra maps $(X, c) \longrightarrow (X, c)$ are unique, $c' \circ c = id$. Moreover, $c \circ c' = T(c') \circ T(c)$ (by definition of c') $= T(c' \circ c)$ (since functors preserve composition) $= T(id)$ (as shown just above) $= id$. Hence, $c : X \longrightarrow T(X)$ is an isomorphism with c' being its inverse. The notation $c : X \xrightarrow{\cong} T(X)$ is used to express this isomorphism. ■

The following two examples show how finality of coalgebras can be used to define finite and infinite lists consisting of elements from a set A. In Example 5.6, we prove first that the set of finite and infinite lists together with the operations empty, head, and tail is indeed the final coalgebra of the functor $T(X) = 1 + A \times X$. Secondly, in Example 5.7, we show that this finality can be exploited to define potentially infinite lists. We use this definition schema also in Chapter 6 where we define the operational semantics of programming languages coalgebraically by assigning each program and each initial state a potentially infinite state transition list, i.e., the set A represents the set of states that can be reached during computation.

Example 5.6 (Finality of Potentially Infinite Lists) *We claim that the coalgebra*

$$[\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail} \rangle] : A^\infty \longrightarrow 1 + A \times A^\infty$$

where A^∞ denotes $A^* \cup A^{\mathbb{N}}$, the set of finite and infinite lists over A, with the coalgebra operation $[\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail} \rangle]$ defined by

$$\alpha \mapsto \begin{cases} \kappa(*) & \text{if } \alpha = () \\ \kappa'(\langle a, \beta \rangle) & \text{if } \alpha = \mathrm{cons}(a, \beta) \end{cases}$$

is the final coalgebra of the functor $T(X) = 1 + A \times X$. To prove this claim, we need to show that there exists a unique homomorphism from any other coalgebra $[\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle] : U \longrightarrow 1 + A \times U$ into $[\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail} \rangle] : A^\infty \longrightarrow 1 + A \times A^\infty$. To prove this, we first define a function $h : U \longrightarrow A^\infty$, then we show that it is a coalgebra homomorphism and finally that it is unique.

Throughout this example, we denote the empty list with $()$ *and non-empty lists with* $\mathsf{cons}(a, \beta)$ *whereby* $a \in A$, $\beta \in A^\infty$. *Moreover, we describe lists as functions* $\mathbb{N} \longrightarrow A$ *which map natural numbers representing list indices to elements in* A.

To define the desired function $h : U \longrightarrow A^\infty$, *recall that intuitively all we can observe about the system* $[\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle]$ *can be represented by a potentially infinite list of individual observations which are the results of* value *in a given state* $u \in U$. *So the idea is to define the function* h *going into* A^∞ *by defining exactly this list of observations, in a step-by-step manner which turns out to be typical for coinductive definitions. In this definition, we distinguish two cases for a given* $u \in U$, *namely* $[\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle](u) = \kappa(*)$ *and* $[\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle](u) \neq \kappa(*)$. *In the first case, the system halts, in the second case, it goes on with state transitions.*

$$h(u) = \begin{cases} \lambda n.\mathsf{value}(\mathsf{next}^n(u)) & \text{if } [\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle](u) \neq \kappa(*) \\ () & \text{if } [\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle](u) = \kappa(*) \end{cases}$$

To prove that h *is a homomorphism, we need to show the following:*

1. *If* $[\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle](u) \neq \kappa(*)$, *we need to show that*
   a) $\mathsf{head} \circ h = \mathsf{value}$ *and*
   b) $\mathsf{tail} \circ h = h \circ \mathsf{next}$.
2. *If* $[\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle](u) = \kappa(*)$, *we need to show that* $\mathsf{empty} \circ h = \mathsf{stop}$.

Proof for Case 1.a): $(\mathsf{head} \circ h)(u) = \mathsf{head}(\lambda n.\mathsf{value}(\mathsf{next}^n(u))) = \mathsf{value}(u)$ *for all* $u \in U$ *with* $[\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle](u) \neq \kappa(*)$.
Proof for Case 1.b): $(\mathsf{tail} \circ h)(u) = \mathsf{tail}(\lambda n.\mathsf{value}(\mathsf{next}^n(u))) = \lambda n.\mathsf{value}(\mathsf{next}^{n+1}(u)) = \lambda n.\mathsf{value}\,(\mathsf{next}^n(\mathsf{next}^n(u))) = h \circ \mathsf{next}(u)$ *for all* $u \in U$ *with* $[\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle](u) \neq \kappa(*)$.
Proof for Case 2: *For all* $u \in U$ *with* $[\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle](u) = \kappa(*)$ *it holds that* $h(u) = ()$ *and* $\mathsf{empty}(h(u)) = \kappa(*) = \mathsf{stop}(u)$.

These case distinctions complete the proof that h *is a coalgebra homomorphism, i.e., the following diagram commutes:*

$$\begin{array}{ccc} U & \xrightarrow{\quad h \quad} & A^\infty \\ {\scriptstyle [\mathsf{stop},\langle \mathsf{value},\mathsf{next}\rangle]}\downarrow & & \downarrow{\scriptstyle [\mathsf{empty},\langle \mathsf{head},\mathsf{tail}\rangle]} \\ 1 + A \times U & \xrightarrow[\mathsf{id}+\mathsf{id}\times h]{} & 1 + A \times A^\infty \end{array}$$

It remains to show that h *is unique. Assume that there is another coalgebra homomorphism* $h' : U \longrightarrow A^\infty$. *We show that* $h = h'$. *In this proof, we do the same case distinction as before:*

First case: $[\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle](u) \neq \kappa(*)$. *It holds that* $(\mathsf{head} \circ h')(u) = \mathsf{value}(u)$ *and* $(\mathsf{tail} \circ h')(u) = (h' \circ \mathsf{next})(u)$ *because* h' *is a homomorphism. An easy induction over* $n \in \mathbb{N}$ *shows that* $h'(u) = \lambda n.\mathsf{value}(\mathsf{next}^n(u))$. *As a direct consequence, we conclude that* $h = h'$.
Second case: $[\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle](u) = \kappa(*)$. *It follows that* $([\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail} \rangle] \circ h')(u) = \kappa(*)$. *According to the definition of the coalgebra* $[\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail} \rangle] : A^\infty \longrightarrow 1 + A \times A^\infty$, *this happens only if* $h'(u) = ()$. *It follows that* $h(u) = h'(u)$.

This completes the proof that the coalgebra $[\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail} \rangle] : A^\infty \longrightarrow 1 + A \times A^\infty$ *is the final coalgebra of the functor* $T(X) = 1 + A \times X$ *because there exists a unique homomorphism from any other coalgebra of the same functor into* $[\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail} \rangle] : A^\infty \longrightarrow 1 + A \times A^\infty$. $\diamond$

The preceeding example shows that potentially infinite lists are the final coalgebra of the functor $T(X) = 1 + A \times X$ for an arbitrary set A. We can use this finality to define potentially infinite lists. The definition principle is analogous to the one used in the above example for the definition of the unique homomorphism into the final coalgebra: Define potentially infinite lists in a step-by-step manner, one element after the other. Whenever we define lists in this manner by exploiting the finality of an underlying coalgebra, we say that the definition is coinductive.

Example 5.7 (Defining Potentially Infinite Lists) *Consider the final coalgebra* $[\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail} \rangle] : A^\infty \longrightarrow 1 + A \times A^\infty$ *of the functor* $T(X) = 1 + A \times X$ *from Example 5.6. By exploiting the finality of this coalgebra, we can define potentially infinite lists. Whenever such a definition relies on the finality of some coalgebra, we also say that the definition is* **coinductive**.

As a simple example, consider the empty list $\mathsf{nil} : 1 \longrightarrow A^\infty$. *It is defined as the unique coalgebra homomorphism in this diagram where we have put a suitable coalgebra structure on the set* 1*:*

$$\begin{array}{ccc} 1 & \xrightarrow{\quad\mathsf{nil}\quad} & A^\infty \\ {\scriptstyle\kappa}\downarrow & & \downarrow{\scriptstyle[\mathsf{empty},\langle\mathsf{head},\mathsf{tail}\rangle]} \\ 1 + A \times 1 & \xrightarrow[\mathsf{id}+(\mathsf{id}\times\mathsf{nil})]{} & 1 + A \times A^\infty \end{array}$$

As a more complex example, let us consider the concatenation function $\mathsf{conc} : A^\infty \times A^\infty \longrightarrow A^\infty$ *on lists. Given two lists* $a : \mathbb{N} \longrightarrow A$ *and* $b : \mathbb{N} \longrightarrow A$*, their concatenation is the list which contains first all the elements from* a*, followed by the elements of* b*. To define this function coinductively, we consider the coalgebra* $[\mathsf{conc\_struct}] : A^\infty \times A^\infty \longrightarrow 1 + A^\infty \times A^\infty$ *defined by:*

$$(\alpha, \beta) \mapsto \begin{cases} \kappa(*) & \text{if } [\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail} \rangle](\alpha) = [\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail} \rangle](\beta) = \kappa(*) \\ \kappa'(a, (\alpha', \beta)) & \text{if } [\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail} \rangle](\alpha) = \kappa'(a, \alpha') \\ \kappa'(b, (\alpha, \beta')) & \text{if } [\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail} \rangle](\alpha) = \kappa(*) \text{ and } [\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail} \rangle](\beta) = \kappa'(b, \beta') \end{cases}$$

The function $\mathsf{conc} : A^\infty \times A^\infty \longrightarrow A^\infty$ *that we wish to define arises as the unique coalgebra homomorphism in this diagram:*

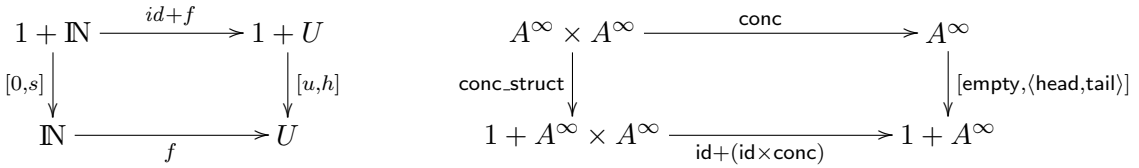
$$\begin{array}{ccc} A^\infty \times A^\infty & \xrightarrow{\quad\mathsf{conc}\quad} & A^\infty \\ {\scriptstyle\mathsf{conc\_struct}}\downarrow & & \downarrow{\scriptstyle[\mathsf{empty},\langle\mathsf{head},\mathsf{tail}\rangle]} \\ 1 + A \times A^\infty \times A^\infty & \xrightarrow[\mathsf{id}+(\mathsf{id}\times\mathsf{conc})]{} & 1 + A \times A^\infty \end{array}$$

This homomorphism conc *can equivalently be defined as follows, as can be easily verified:*

- *If* $[\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail} \rangle](\alpha) = [\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail} \rangle](\beta) = \kappa(*)$*:*
  $\mathsf{conc}(\alpha, \beta) = ()$.
- *If* $[\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail} \rangle](\alpha) = \kappa'(a, \alpha')$*:*
  $\mathsf{head}(\mathsf{conc}(\alpha, \beta)) = a$ *and* $\mathsf{tail}(\mathsf{conc}(\alpha, \beta)) = \mathsf{conc}(\alpha', \beta)$.
- *If* $[\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail} \rangle](\alpha) = \kappa(*)$ *and* $[\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail} \rangle](\beta) = \kappa'(b, \beta')$*:*
  $\mathsf{head}(\mathsf{conc}(\alpha, \beta)) = b$ *and* $\mathsf{tail}(\mathsf{conc}(\alpha, \beta)) = \mathsf{conc}(\alpha, \beta')$. ◇

The definition of the homomorphism conc in the preceeding example is analogous to the homomorphism $h : U \longrightarrow A^\infty$ in Example 5.6: In case that there is no next state ($\mathsf{conc\_struct}(\alpha, \beta) = \kappa(*)$), h maps the current state to the empty list $()$. Otherwise, the resulting list is characterized by defining its head and tail which is the typical pattern in coinductive definitions, cf. also

Section 5.2. The function to be defined appears "inside" the definition while the destructor operations (head and tail in this case) are "outside". In contrast, in inductive definitions as in Example 5.2, we find the function to be defined "outside" while the operations of the algebra are "inside". Recall the situations from Examples 5.2 and 5.7:



The opposite directions in inductive versus coinductive definitions are stemming from the opposite directions of the homomorphisms between the initial algebra and an arbitrary algebra of the same functor as well as between an arbitrary coalgebra and the final coalgebra of the same functor. In case of algebras, the unique homomorphism f is **going out** of the initial algebra. In particular, f is applied after the operation of the initial algebra ($[0, s]$ in the above example) which causes f to be outside in the inductive definition. In case of coalgebras, the unique homomorphism conc is **going into** the final coalgebra. This requires the homomorphism conc to be applied before the operation of the coalgebra ($[\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail} \rangle]$ in our example). Hence, the coalgebra homomorphism conc appears inside the coinductive definition. The "inside" versus "outside" definitions are the result of making the appropriate diagrams commute.

Finally a remark on the existence of final coalgebras: Theorem 5.2 states that final T-coalgebras, if they exist, are a fixed point of the functor T, i.e., there exists an isomorphism between X and $T(X)$. Clearly this is only possible if X and $T(X)$ have the same set-theoretic cardinality. For polynomial functors T_{poly} which we introduced in Section 5.3, in particular in Definition 5.3, it can be easily verified using basic set theory that X and $T_{poly}(X)$ have indeed the same cardinality. As a negative example, consider the power set functor which maps each set to its power set. For this functor, no final coalgebra exists because each set has a strictly smaller cardinality than its power set. Nevertheless, for the finite power set functor $\mathcal{P}_{finite}$ with $\mathcal{P}_{finite}(X) = \{X' \mid X' \subseteq X \wedge X \text{ is finite}\}$ the final coalgebra does exist [Bar93]. $\mathcal{P}_{finite}$ suffices to describe all *finitely branching* transition systems. These are all transition systems with the property that for each state, there is only a finite number of successor states. For practical reasons, this descriptive power is sufficient.

Coinduction is, as induction, a definition and a proof principle. The existence of homomorphisms allows us to define functions from arbitrary coalgebras into the final coalgebra. By exploiting the uniqueness of these homomorphisms, we can prove equality of functions, i.e. equality of observations of systems: Two systems are observationally equivalent if their observable behavior, i.e. their image in the final coalgebra obtained by applying the unique homomorphism into the final coalgebra, is the same. In the following section, we introduce bisimulations and the coinductive proof principle which is based on bisimulations. The coinductive proof principle does not mention the uniqueness aspect of finality explicitly but the proof for its correctness is based on it.

5.7 Bisimulations and the Coinductive Proof Principle

In this section, we introduce bisimulations and show that bisimulations on final coalgebras contain the equality relation. This important result, which is also known as the coinductive proof principle, can be used to prove observational equivalence of systems. We start this section

by introducing bisimulation on lists as introductory example. Then we proceed by giving a general definition for bisimulations. Finally we prove the coinductive proof principle and show its application for a typical example.

Definition 5.12 (Bisimulations on Lists) *Let T be the functor $T(X) = 1 + X$ with the final coalgebra $[\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail}\rangle] : A^\infty \longrightarrow 1 + A \times A^\infty$ defined and discussed in detail in Example 5.6. A* bisimulation *on this carrier set A^∞ is a relation $R \subseteq A^\infty \times A^\infty$ satisfying these requirements:*

$$R(\alpha,\beta) \Rightarrow \begin{cases} \alpha = \beta = () \quad or \\ \alpha \neq () \quad and \quad \beta \neq () \quad and \quad \mathsf{head}(\alpha) = \mathsf{head}(\beta) \quad and \quad R(\mathsf{tail}(\alpha), \mathsf{tail}(\beta)) \end{cases} \qquad \diamond$$

This definition expresses that the relation R is closed under the operations of the final coalgebra. It requires that two lists in relation R are either both empty or both non-empty. In the first case, the two lists are mapped to $\kappa(*)$ by the coalgebra operation $[\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail}\rangle]$, and the system halts. In the second case, the operations head and tail can be applied. Thereby it is required that the lists obtained by applying tail are again in relation R. The coinductive proof principle for lists states the following:

Lemma 5.3 (Coinduction for Lists) *Let R be a bisimulation on A^∞, the set of finite and infinite lists. If $R(\alpha,\beta)$, then $\alpha = \beta$.* $\diamond$

The proof of the coinduction principle for lists exploits the finality of A^∞ by putting a suitable coalgebra structure on R:
Proof: Consider R as a set of pairs and define a coalgebra structure on it:

$$\gamma : R \longrightarrow 1 + A \times R \qquad \text{with} \qquad (\alpha,\beta) \mapsto \begin{cases} \kappa(*) & \text{if } \alpha = \beta = () \\ \kappa'(\mathsf{head}(\alpha), (\mathsf{tail}(\alpha), \mathsf{tail}(\beta))) & \text{otherwise} \end{cases}$$

γ is well-defined because either α and β are both empty or both non-empty. Moreover, in the second case, $(\mathsf{tail}(\alpha), \mathsf{tail}(\beta))$ is again contained in R because R is a bisimulation. Now consider the two projection functions π_1 and π_2 on R (for a definition of π_1 and π_2, cf. Section 5.3):

$$\pi_1 : R \longrightarrow A^\infty \quad \text{and} \quad \pi_2 : R \longrightarrow A^\infty$$

It is easy to verify that π_1 and π_2 are coalgebra homomorphisms from (R,γ) to $(A^\infty, [\mathsf{empty}, \langle \mathsf{head}, \mathsf{tail}\rangle])$, displayed in the following commuting diagram:

$$\begin{array}{ccccc} A^\infty & \xleftarrow{\pi_1} & R & \xrightarrow{\pi_2} & A^\infty \\ \downarrow{\scriptstyle [\mathsf{empty},\langle \mathsf{head},\mathsf{tail}\rangle]} & & \downarrow{\scriptstyle \gamma} & & \downarrow{\scriptstyle [\mathsf{empty},\langle \mathsf{head},\mathsf{tail}\rangle]} \\ 1 + A \times A^\infty & \xleftarrow[\mathsf{id}+\mathsf{id}\times\pi_1]{} & 1 + A \times R & \xrightarrow[\mathsf{id}+\mathsf{id}\times\pi_2]{} & 1 + A \times A^\infty \end{array}$$

Therefore, it follows from the uniqueness aspect of the finality of A^∞ that $\pi_1 = \pi_2$. ■

The following example illustrates how the coinduction principle can be applied to prove the equality of potentially infinite lists.

Example 5.8 (Coinduction on Lists) *Consider the set A^∞ of finite and infinite lists. The function* odd *takes such a list as input and outputs the list which contains all the elements*

occurring in oddly numbered places of the original list in the same order. $\mathsf{odd} : A^\infty \longrightarrow A^\infty$ *is coinductively defined as follows:*

$$\mathsf{odd}(\alpha) = \begin{cases} () & \text{if } \alpha = () \\ (\mathsf{head}(\alpha), \mathsf{odd}(\mathsf{tail}(\mathsf{tail}(\alpha)))) & \text{otherwise} \end{cases}$$

The function even *behaves analogous to the function* odd *but keeps only the list elements in evenly numbered places:*

$$\mathsf{even}(\alpha) = \begin{cases} () & \text{if } \alpha = () \\ \mathsf{odd}(\mathsf{tail}(\alpha)) & \text{otherwise} \end{cases}$$

Moreover, consider the function merge *which takes two lists and merges them by taking alternately elements from both of the lists:*

$$\mathsf{merge}(\alpha, \beta) = \begin{cases} () & \text{if } \alpha = () \text{ and } \beta = () \\ \alpha & \text{if } \beta = () \\ \beta & \text{if } \alpha = () \\ (\mathsf{head}(\alpha), \mathsf{merge}(\beta, \mathsf{tail}(\alpha))) & \text{otherwise} \end{cases}$$

Using the coinduction principle for potentially infinite lists, we can prove that $\mathsf{merge}(\mathsf{odd}(\alpha), \mathsf{even}(\alpha)) = \alpha$. *Therefore we need to define a suitable bisimulation relation* R. *We choose*

$$R = \{(\mathsf{merge}(\mathsf{odd}(\alpha), \mathsf{even}(\alpha)), \alpha) \mid \alpha \in A^\infty\}$$

Now it remains to show that R *is indeed a bisimulation, i.e., it is closed under the operations of the coalgebra* A^∞. *Therefore we distinguish four cases whereby only the fourth considers the coinductively relevant case:*

First case: $\alpha = ()$. *In this case,* $\mathsf{odd}(\alpha) = \mathsf{even}(\alpha) = \mathsf{merge}(\mathsf{odd}(\alpha), \mathsf{even}(\alpha)) = () = \alpha$.
Second case: $\alpha = (a)$. *In this case,* $\mathsf{odd}(\alpha) = (a)$, $\mathsf{even}(\alpha) = ()$, *and* $\mathsf{merge}(\mathsf{odd}(\alpha), \mathsf{even}(\alpha)) = (a) = \alpha$.
Third case: $\alpha = (a, a')$. *In this case,* $\mathsf{odd}(\alpha) = (a)$, $\mathsf{even}(\alpha) = (a')$, *and* $\mathsf{merge}(\mathsf{odd}(\alpha), \mathsf{even}(\alpha)) = \alpha$.
Fourth case: $\alpha \neq ()$, $\alpha \neq (a)$, *and* $\alpha \neq (a, a')$. *In this case, we need to show that whenever we have a pair* $(\mathsf{merge}(\mathsf{odd}(\alpha), \mathsf{even}(\alpha)), \alpha) \in R$, *then applying the coalgebra operation* tail *on both elements yields again an element in* R, *i.e. that* $(\mathsf{tail}(\mathsf{merge}(\mathsf{odd}(\alpha), \mathsf{even}(\alpha))), \mathsf{tail}(\alpha)) \in R$. *This holds due to these rewrite transformations:*

$$\begin{aligned} \mathsf{tail}(\mathsf{merge}(\mathsf{odd}(\alpha), \mathsf{even}(\alpha))) &= \mathsf{merge}(\mathsf{even}(\alpha), \mathsf{tail}(\mathsf{odd}(\alpha))) \\ &= \mathsf{merge}(\mathsf{odd}(\mathsf{tail}(\alpha)), \mathsf{odd}(\mathsf{tail}(\mathsf{tail}(\alpha)))) \\ &= \mathsf{merge}(\mathsf{odd}(\mathsf{tail}(\alpha)), \mathsf{even}(\mathsf{tail}(\alpha))) \end{aligned}$$

Hence, $(\mathsf{tail}(\mathsf{merge}(\mathsf{odd}(\alpha), \mathsf{even}(\alpha))), \mathsf{tail}(\alpha)) \in R$ *which shows that* R *is a bisimulation which completes the proof that* $\mathsf{merge}(\mathsf{odd}(\alpha), \mathsf{even}(\alpha)) = \alpha$ *for all* $\alpha \in A^\infty$. $\diamond$

Now we are ready to generalize the notions which we have introduced for lists to the general case considering polynomial functors, i.e. functors T whose final T-coalgebra exist. We start with a general definition for bisimulations as relations which are closed under the operation of the underlying coalgebra:

Definition 5.13 (Bisimulation) *Let T be a functor and $c : U \longrightarrow T(U)$ be a T-coalgebra. A* bisimulation *on U is a relation $R \subseteq U \times U$ such that there exists a T-coalgebra structure $\gamma : R \longrightarrow T(R)$ such that the two projection functions $\pi_1 : R \longrightarrow U$ and $\pi_2 : R \longrightarrow U$ are homomorphisms of T-coalgebras:*

$$\begin{array}{ccccc} U & \xleftarrow{\pi_1} & R & \xrightarrow{\pi_2} & U \\ {\scriptstyle c}\downarrow & & {\scriptstyle \gamma}\downarrow & & \downarrow{\scriptstyle c} \\ T(U) & \xleftarrow[T(\pi_1)]{} & T(R) & \xrightarrow[T(\pi_2)]{} & T(U) \end{array} \qquad \diamond$$

Theorem 5.3 (Coinduction Principle) *Let $c : X \xrightarrow{\cong} T(X)$ be the final coalgebra of the functor T. Let R be a bisimulation on X. For all $x, x' \in X$, $R(x, x')$ implies that $x = x'$.* $\diamond$

Proof: The proof is completely analogous to the proof of Lemma 5.3. We take the coalgebra structure on R, $\gamma : R \longrightarrow T(R)$ (which exists by definition of bisimulations), and consider the projection functions $\pi_1 : R \longrightarrow X$ and $\pi_2 :\longrightarrow X$ which are homomorphisms into the final coalgebra (X, c):

$$\begin{array}{ccccc} X & \xleftarrow{\pi_1} & R & \xrightarrow{\pi_2} & X \\ {\scriptstyle c}\downarrow & & {\scriptstyle \gamma}\downarrow & & \downarrow{\scriptstyle c} \\ T(X) & \xleftarrow[T(\pi_1)]{} & T(R) & \xrightarrow[T(\pi_2)]{} & T(X) \end{array}$$

Because of the finality of (X, c), $\pi_1 = \pi_2$ and, hence, $x = x'$ for all $(x, x') \in R$. ■

5.8 Conclusions

This chapter has introduced the dual notions of algebras and coalgebras as well as their dual definition and proof principles induction and coinduction in terms of category theory. In this setting, coalgebras are functions that take an input, representing the state of a system, and output a structured result which typically contains one or several successor states together with observations of the system. In contrast, algebras are represented as functions that take a structured input (the parameters for their construction functions) and return a single output which is the newly constructed element. Functors are used to uniformly represent the signatures of these coalgebra and algebra operations.

There are special algebras and coalgebras, namely initial algebras and final coalgebras. Initial algebras are characterized by the existence of a unique homomorphism from the initial algebra into an arbitrary algebra of the same functor (i.e. of the same signature). We can think of this homomorphism as a function which calculates values: The elements of the initial algebra can be regarded as finite trees (cf. also Chapter 2 and 3), and the homomorphism tells us how to map such a tree to an element of an arbitrary algebra by calculating this element inductively along the tree. This has been demonstrated particularly clearly in Example 5.2 in which we defined the function $F : \mathbb{N} \longrightarrow \mathbb{Q}$ with $f(s^n(0)) = 2^{-n}$ by induction, i.e. by defining how to map a tree $s(s(\cdots(0)))$ defining a natural number to a number in $\mathbb{Q}$.

Dually, final coalgebras are defined by the existence of a unique homomorphism from an arbitrary coalgebra into the final coalgebra of the same functor. If we regard coalgebras as state transition systems, we can think of this homomorphism as a function that unfolds the observable behavior of a system in a normalized form by representing it in form of finite or infinite trees. Each node in such a tree corresponds to a state and is decorated with the observations of that state. Successor nodes represent successor states. We have seen an example of two different

coalgebras with the same observable behavior in Example 5.4 where we considered two different, yet bisimilar automata with the same observable behavior. Mapping these two automata into the final coalgebra corresponds to defining the (infinite) sequence of observations occurring during their state transitions. This mapping into the final coalgebra unfolds the observable behavior in a normalized form.

Bisimulations are binary relations on coalgebras which are closed under the operations of the coalgebra and which contain those pairs of elements, i.e. states, which show the same observable behavior. The coinductive proof principle states that for final coalgebras, bisimulations are contained in the equality relation. So if we want to prove that two systems have the same observable behavior, we define their behavior by suitable coalgebras and, furthermore, by mapping these coalgebras into the final coalgebra of the same functor. Then we need to show that two systems which we want to prove observationally equivalent are mapped to the same element in the final coalgebra. Therefore we need to define a relation which contains those pairs of elements that we want to prove to be equivalent and need to verify that this relation is indeed a bisimulation.

In the following chapter, we apply this principle to define the operational semantics of programming languages coalgebraically by assigning each program an element of a suitable final coalgebra.

6 Programming Language Semantics in a Coalgebraic Setting

Operational approaches to programming language semantics (cf. Chapter 1) define for each program a state transition system. Hence, it is a natural consequence to regard programs as coalgebras, i.e. as functions that take a state as input and output a new state. We develop this idea in Section 6.1. Thereby we also argue that this view is completely in line with the intention of the three approaches to operational semantics, namely with abstract state machines (ASMs) and with the two incarnations of inference rule-based specifications which are natural (or big-step) semantics and structural operational (or small-step) semantics. In Section 6.2, we compare these three specification frameworks for the operational semantics of programming languages with respect to two criteria: the range of imperative programming languages to which they are applicable and the way the program is used in the specifications and treated during the thereby defined executions. To reveal the fundamental differences between these three mechanisms, we investigate if there are automatic transformations between them. As a side effect, this leads to new insights concerning the classification of big-step and small-step semantics. A preversion of the results of Section 6.2 have been published in [Gle03].

6.1 Programs as Coalgebras

Coalgebras model state-based systems. Since operational approaches for the semantics of programming languages define the semantics of programs as state transition systems, coalgebras are a natural way to express operational semantics. In this chapter we consider deterministic programming languages to keep notation as clear as possible but the extension to non-deterministic semantics can be expressed as well. Deterministic programming languages are characterized by the fact that, for each state reached during program execution which is not a final state in which computation terminates, there exists exactly one successor state. We define semantics of such programming languages by a function that maps each program together with an initial state into the final coalgebra of the functor $T(X) = 1 + A \times X$ where A is the set of data structures used to define the states reached during computation. The carrier set of this coalgebra is A^∞, the set of all finite and infinite lists with elements of A. This means that the semantics of a program is described by a finite state transition list if program execution terminates and by an infinite state transition list in case of non-termination. In this section, we describe this mapping separately for each of the three formalisms for the operational semantics of programming languages which are abstract state machines, structural operational semantics, and natural semantics. In the succeeding section, we compare the three formalisms to reveal the fundamental differences between them, i.e. those differences which do not stem from only notational differences.

6.1.1 Coalgebraic Semantics for Abstract State Machines (ASMs)

Abstract state machines are a general formalism to describe state transition systems. In Section 1.2 we have introduced ASMs and shown how they are used in the definition of the operational

semantics of programming languages. In this section, we show that each such ASM also defines an element in a suitable final coalgebra.

Abstract state machines define state transition systems by defining each state as an algebra over a given signature. During a state transition, the interpretation of some of the function symbols might change, cf. Section 1.2. Initial states are defined by a set of equations which hold in the initial states. This implies that each ASM defines a state transition system with several valid runs, namely all those that start with an initial state. In the context of programming language semantics, a state is given by the program being executed and by the current state which contains also the pointer CT, *current task*, which indicates which program part is currently being executed.

Each ASM defines directly a coalgebra with the coalgebra operation $[\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle]$: If the run of the ASM terminates, then its current state is mapped to $\kappa(*)$; otherwise we can observe the current state denoted by the operation value and the succeeding state is given by applying the function next to the current state.

Given an ASM, we define coinductively a function

$$[\![ASM]\!] : A \times P \longrightarrow A^\infty.$$

This function takes a state a in A, initially one from the initial states of the ASM, and a program p in the programming language P and maps them to an element in A^∞. A^∞ contains all finite and infinite lists with elements of A and is the carrier set of the final coalgebra of the functor $T(X) = 1 + A \times X$. To define this function $[\![ASM]\!]$ coinductively, we put a suitable coalgebra structure on its domain $A \times P$:

$$\begin{array}{ccc} A \times P & \xrightarrow{\quad [\![ASM]\!] \quad} & A^\infty \\ {\scriptstyle [\mathsf{stop},\langle \mathsf{value},\mathsf{next}\rangle]}\downarrow & & \downarrow{\scriptstyle [\mathsf{empty},\langle \mathsf{head},\mathsf{tail}\rangle]} \\ 1 + A \times (A \times P) & \xrightarrow[\mathsf{id}+(id \times [\![ASM]\!])]{} & 1 + A \times A^\infty \end{array}$$

As in Example 5.7, the function $[\![ASM]\!]$ arises as the unique coalgebra homomorphism in the above diagram. $[\![ASM]\!]$ is defined by a case distinction, as has been shown in Examples 5.6 and 5.7:

$$[\![ASM]\!](a,p) = \begin{cases} \mathsf{cons}(\mathsf{value}(a,p), \mathsf{next}(a,p)) & \text{if } [\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle](a,p) \neq \kappa(*) \\ & \text{i.e. there is a successor state in the run of the ASM} \\ () & \text{if } [\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle](a,p) = \kappa(*) \\ & \text{i.e. there is no successor state in the run of the ASM} \end{cases}$$

This definition expresses the following: If there is no successor state, i.e. $[\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle](a, p) = \kappa(*)$, then $[\![ASM]\!](a,p) = (a)$ which means that no further state transition sequence can be observed. Otherwise, if $[\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle](a,p) \neq \kappa(*)$, then $[\![ASM]\!](a,p) = \mathsf{cons}(\mathsf{value}(a,p), \mathsf{next}(a,p))$ which is the state transition sequence consisting of the current state a concatenated with the state transition sequence encountered afterwards. Note that in this context, $\mathsf{value}(a,p) = a$ holds for all $a \in A$ and $p \in P$.

With this definition, we assign each program and each initial state an element of the final coalgebra A^∞ which is the final coalgebra of the functor $T(X) = 1 + A \times X$. Hence, we define the operational semantics of a program as the finite or infinite state transition sequence which is run through during program execution.

6.1.2 Coalgebraic Semantics for Structural Operational Semantics (SOS)

Structural operational semantics defines semantics of programming languages as a function that maps tuples $< p, a >$ to tuples $< p', a' >$, thereby denoting that the execution of p in state a yields a new program p' to be executed in the succeeding state a', or, in case that program execution terminates, as a mapping of $< p, a >$ to a' which is a final state (cf. Section 1.4). Hence, each structural operational semantics is a function that takes a program p and an initial state a as input and iteratively defines a finite or infinite list with elements of A where A is as in Subsection 6.1.1 the set of states that can be reached during program execution.

As in the case of ASMs, each structural operational semantics corresponds to a coalgebra with the coalgebra operation $[\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle]$. If program execution terminates, then the coalgebra operation returns $\kappa(*)$. Otherwise the current state is observable and denoted by the function value, and the rest of the observable state transition sequence is obtained by applying the function next to the current state. Given a structural operational semantics, we define a function $[\![SOS]\!] : A \times P \to A^\infty$ coinductively as the unique coalgebra homomorphism in the diagram below:

$$\begin{array}{ccc} A \times P & \xrightarrow{\quad [\![SOS]\!] \quad} & A^\infty \\ {\scriptstyle [\mathsf{stop},\langle \mathsf{value},\mathsf{next}\rangle]}\downarrow & & \downarrow{\scriptstyle [\mathsf{empty},\langle \mathsf{head},\mathsf{tail}\rangle]} \\ 1 + A \times (A \times P) & \xrightarrow[\mathsf{id}+(id \times [\![SOS]\!])]{} & 1 + A \times A^\infty \end{array}$$

As in the case of ASMs, we define the function $[\![SOS]\!]$ by a case distinction:

$$[\![SOS]\!](a,p) = \begin{cases} \mathsf{cons}(\mathsf{value}(a,p), \mathsf{next}(a,p)) & \text{if } [\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle](a) \neq \kappa(*) \\ & \text{i.e. if } < p, a > \to < p', a' > \text{ or } < p, a > \to\ a' \\ () & \text{if } [\mathsf{stop}, \langle \mathsf{value}, \mathsf{next} \rangle](a) = \kappa(*) \\ & \text{i.e. if no successor state exists} \end{cases}$$

As in the case of ASMs, with this definition, we assign each program and each initial state an element of the final coalgebra A^∞ which is the final coalgebra of the functor $T(X) = 1 + A \times X$. Hence, we define the operational semantics of a program as the finite or infinite state transition sequence which is run through during program execution.

6.1.3 Coalgebraic Semantics for Natural Semantics

In Chapter 4, we have shown that natural semantics specifications can be interpreted coinductively. In particular, in Definition 4.4, we have defined the semantics of a program as the set of all derivation trees whose root nodes are marked with the initial state. Moreover, in Corollary 4.1, we have shown that assuming a deterministic semantics, each program semantics defines a unique finite or infinite state transition sequence. Hence, in Chapter 4, we have already defined the semantics coinductively based on natural semantics by a unique mapping into the final coalgebra of the functor $T(X) = 1 + A \times X$.

Now that we have seen that all three formalisms for the operational semantics of programming languages describe semantics as finite and infinite state transition sequences which can be defined by the unique coalgebra homomorphism into the final coalgebra, we turn to the question which, if any, differences exist between these three formalisms. We answer this question in the succeeding section.

6.2 Semantic Equivalence of Program Coalgebras

Abstract state machines (ASMs) and inference rule-based semantics with its two incarnations big-step and small-step semantics are competing specification frameworks for the operational semantics of programming languages. In the ASM as well as in the inference rule community, there exists an extensive engineering knowledge of how to use these specification mechanisms appropriately. This raises the widely debated question if there are fundamental differences between them. We compare ASMs and inference rule-based semantics according to two main criteria: First, we characterize them regarding the structure of imperative programming languages whose semantics can be defined with them. Secondly, we evaluate them with respect to the way the programs are treated in the specifications and during their thus defined execution. While both criteria are certainly coupled, it turns out that the second criterion really shows where the fundamental differences are.

To accomplish the desired comparison between ASMs and inference rule-based semantics, we define, if possible, automatic semantics-preserving transformations from one mechanism into the other. This proceeding is particularly helpful as it separates non-relevant discrepancies in notation from essential differences. Since semantics is defined operationally in these frameworks, each program is regarded as a coalgebra. Hence, semantic equivalence means in our context that for each program, the state transitions with the respective observable behavior during execution are the same. If all specification mechanisms, ASMs as well as big-step and small-step semantics, could be applied for arbitrary programming languages, then we could hope to find transformations in any desired direction. This is not the case and leads us directly to our classification of imperative programming languages. We distinguish between strictly compositional and non-strictly compositional programming languages. In a strictly compositional programming language, the semantics of each part of the program, which we regard in form of its abstract syntax tree, can be defined solely in terms of the semantics of its direct parts, i.e. subtrees. Big-step semantics defines semantics of programs recursively in terms of the semantics of their direct subtrees. This implies that big-step semantics can only define the semantics of strictly compositional programming languages. This does not hold for small-step semantics and ASMs.

Concerning the second criterion, treatment of programs, there are more similarities between big-step semantics and ASMs while small-step semantics is the outsider. Both big-step semantics and ASMs use the abstract syntax trees of programs in their specifications but do not modify it during program execution. In contrast, small-step semantics explicitly rewrites the abstract syntax trees during execution. In general, a small-step semantics defines a term-rewriting system. Starting with the original program as initial continuation, during each state transition, the current continuation program is rewritten until the empty program is reached. In each state, the continuation represents the computation which still needs to be done. In contrast, ASMs represent the remaining computation in a given state as a pointer to the node in the abstract syntax tree which is executed next. In each state transition, this pointer is updated.

In this section, we show that each small-step semantics can be transformed automatically into an equivalent ASM semantics and vice versa. We also prove that each big-step semantics can be transformed automatically into an equivalent ASM.

6.2.1 Transformations between ASMs and Inference Rule-Based Semantics

Since big-step semantics can only define strictly compositional semantics, we cannot hope for an automatic transformation from ASMs or small-step semantics to big-step semantics. The reverse direction is possible. We prove that we can transform each big-step semantics into an

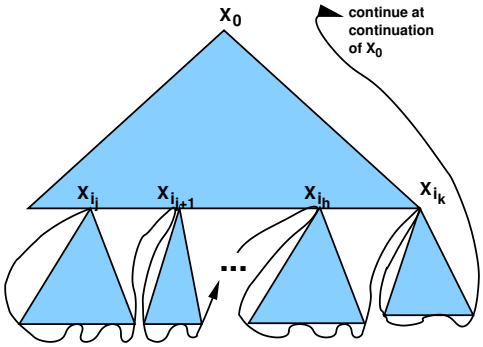


Figure 6.1: Dynamic Continuations

equivalent ASM. Furthermore, we show that each ASM can be transformed automatically into an equivalent small-step semantics and vice versa. This implies that each big-step semantics can also be transformed into an equivalent small-step semantics.

Data Structures in the Specifications

ASMs as well as big-step and small-step semantics define state transitions by exploiting (more or less strictly) the structure of abstract syntax trees. Thereby data structures are defined to represent the states and values which are computed during program execution. In the ASM case, these data structures are defined by the signatures $\Sigma \cup \Delta$ of the static and dynamic functions, the set *Init* of equations defining the initial states, and implicitly by the transition rules which specify how to change their interpretation from one state to another. The signatures define a Herbrand universe. The set *Init* maps all terms into the same equivalence class which are equal under these equations. The transition rules define how to modify this Herbrand structure, i.e. the interpretation, from one state to the next. In natural semantics, the data structures are defined also as a term algebra based on *constructor functions*. Additional *defined functions* can be introduced by stating inductively how they operate on constructor terms. These data structures correspond directly to the states of an ASM and vice versa as they can be interpreted also by the same Herbrand structures.

From Big-Step Semantics to ASMs

A big-step semantics defines execution of programs top-down: the state transitions of an entire abstract syntax tree are composed from the state transitions of its direct subtrees and, in recursive definitions, also from its own state transitions. When transforming a big-step semantics into an ASM specification, we need to explicitly define the continuation attributes which are specified only implicitly by the top-down style of the big-step semantics. Therefore we define a continuation attribute *cont* for each node in the abstract syntax tree. Since a node X_0 may be called recursively, these continuation attributes must also contain the continuations of all active calls of this node X_0. We organize the continuations in a stack (with the usual stack operations). We attach a dynamic stack attribute to each node in the abstract syntax tree. Its value during program execution is part of the current state.

It is important to observe that a big-step semantics defines individual state transitions only at the leaves of an abstract syntax tree. For all inner nodes, the inference rules specify how to compose the overall state transition sequence in the conclusion from the state transitions of the assumptions. When defining an equivalent ASM, the idea is to define rules modifying

the state for the leaves of the abstract syntax tree. Thereby we use the function *update* taking two arguments σ and σ'. It maps the current ASM state σ to the new state σ' and can be defined easily (cf. remarks in Subsection 6.2.1). Moreover, the rules for the inner nodes of the abstract syntax tree adjust the continuations. The idea is that the most right leaves (wrt. to the ordering of the nodes in the assumptions of the applied inference rules) of each subtree contain the continuation of the root of this subtree, cf. Figure 6.1. We need to update the continuations sequentially from "right to left" wrt. the ordering of the assumptions. Since the ASM rules allow only for the specification of updates to be executed in parallel, we need to introduce several ASM rules per inference rule. The current task CT is a pair (X, n) where X is a pointer to the current node in the AST and n denotes the n-th update rule which needs to be executed next.

Definition 6.1 *Let* **Spec** *be a big-step semantics as defined in Section 1.5 consisting of a set of axioms and inference rules. Then the corresponding ASM* $\mathbf{ASM_{Spec}}$ *is defined by the following transition rules:*

- *For each axiom* $< X, \sigma > \rightarrow \sigma'$*, the corresponding transition rule is defined as:*

  if $CT \in (X, 0)$ **then**
  $update(\sigma, \sigma')$; $CT := (cont(X).top, 0)$; $cont(X) := cont(X).pop$ **fi**

- *For each inference rule of the general form*

$$\frac{\mathsf{Eval}(X_{l_1}, \sigma_0) = value_1, \ldots, \mathsf{Eval}(X_{l_m}, \sigma_0) = value_m, \quad < X_{i_1}, \sigma_0 > \rightarrow \sigma_1, < X_{i_2}, \sigma_1 > \rightarrow \sigma_2, \cdots, < X_{i_k}, \sigma_{k-1} > \rightarrow \sigma_k}{< X_0, \sigma_0 > \rightarrow \sigma_k}$$

  the corresponding transition rules are defined as:

if $CT \in (X_0, 0)$ **then**
  if $\mathsf{Eval}(X_{l_1}) = value_1$ **and** ... **and** $\mathsf{Eval}(X_{l_m}) = value_m$ **then**
    $cont(X_{i_k}) := cont(X_{i_k}).push(cont(X_0).top)$; $cont(X_0) := cont(X_0).pop$;
    $CT := (X_0, k-1)$ **fi fi**
if $CT \in (X_0, k-1)$ **then**
  $cont(X_{i_{k-1}}) := cont(X_{i_{k-1}}).push(X_{i_k}); CT := (X_0, k-2)$ **fi**
...
if $CT \in (X_0, 2)$ **then**
  $cont(X_{i_1}) := cont(X_{i_1}).push(X_{i_2}); CT := (X_0.X_{i_1}, 0)$ **fi** $\quad \diamond$

To prove that the semantics of the ASM $\mathbf{ASM_{Spec}}$ defines the same semantics as the big-step semantics **Spec**, we need to show that for each program, the state transitions are the same in both specifications $\mathbf{ASM_{Spec}}$ and **Spec**.

Theorem 6.1 *Let* **Spec** *be a big-step semantics and* $\mathbf{ASM_{Spec}}$ *the corresponding ASM. The state transitions are the same for each program in both specifications.*

Proof: State transitions happen only at the leaves of the AST. The continuation of a node X is a reference to the node where the computation is to be continued after the computation of X is finished. The computation of a node and subtree X is finished when all its leaves are computed. Therefore the leaf processed at last has a continuation pointing to the continuation of X. Since a node might call itself recursively, the different calls and their continuations are superimposed recursively in the abstract syntax tree. Since the continuations are organized in a

stack, they represent the nested recursive structure properly. To prove that the state transitions of **Spec** and $\mathbf{ASM_{Spec}}$ are the same, we distinguish between terminating and non-terminating programs. For the terminating case, we do induction on the height of the abstract syntax tree X and its run-time expansion.

Base case: X is a leaf described by axiom $< X, \sigma > \rightarrow \sigma'$. Clearly the state transition $update(\sigma, \sigma')$ gives us the same new state σ'. The deletion of the top continuation reference removes the current recursive frame.

Induction step: For the computation of X_0, $X_{i_1}, \ldots, X_{i_k}$ need to be computed. We can assume that for $X_{i_1}, \ldots, X_{i_k}$, the state transitions are the same in the big-step semantics **Spec** and in $\mathbf{ASM_{Spec}}$ (induction hypotheses). It remains to show that the continuations are correct. Due to the updates $cont(X_{i_j}) := cont(X_{i_j}).push(X_{i_{j+1}})$ for $1 \leq j \leq k-1$, $X_{i_{j+1}}$ is computed directly after X_{i_j}, $1 \leq j \leq k-1$. The adjustment of the continuations from "right to left" makes sure that the stacking of the continuations is correct for the case that there exists $j \leq h, j, h \in \{1, \ldots, k\}$ such that $X_{i_j} = X_{i_h}$. When processing the subtree marked with X_{i_J} (which equals X_{i_h}), first the continuation of the i_j-th subtree needs to be taken, then the continuation of the i_h-th subtree. Finally, after X_{i_k}, $cont(X_0)$ is executed so that computation either stops if X_0 is the root of the program or continues at the continuation of X_0. The deletion of the continuation of X_0, $cont(X_0).pop$, removes the current recursive frame at X_0. Since the continuation of X_0 is stored in the right most leaf of the subtree located at X_0, this continuation $cont(X_0)$ is not needed any more. Whenever this right-most leaf is reached, the execution will continue directly at $cont(X_0)$.

Coinduction Step for Non-Terminating Programs: The above proof is only valid if the programs terminate. Only then the expansions of $X_{i_1}, \ldots, X_{i_k}$ are truly smaller than the expansion of X_0. If the program does not terminate, then there is one l, $1 \leq l \leq k$, such that $X_{i_1}, \ldots, X_{i_{l-1}}$ are truly smaller than X_0 and such that X_{i_l} is the first subtree with infinite height. The computation will get stuck in X_{i_l}. To prove that both specifications **Spec** and $\mathbf{ASM_{Spec}}$ show the same state transition behavior, we need to show that the execution at the root of X_{i_l} starts with the same state. This state is σ_{l-1}. The state at the root of X_{i_l} is the same for both specifications. Hence, we can conclude that both specifications enforce the same state transition behavior. If this were not the case, then there would be a smallest number of state transitions after which they are different. In the state before they would be the same. But then they must also be the same in the proceeding state. ■

From Small-Step Semantics to ASMs

Small-step semantics define the execution of programs by recursively defining how to transform an initial state as well as a given program itself stepwise into a new state and a new program. This means that a small-step semantics does not define execution by a (eventually recursive) walk through the abstract syntax tree as it is the case in a big-step semantics. Rather the program is treated as a term which is rewritten during execution until it is reduced to the empty tree. This term to be reduced is also called *continuation*.

The axioms and inference rules in a specification define a recursive rewriting procedure. An axiom $< X, \sigma > \rightarrow \sigma'$ or $< X, \sigma > \rightarrow < X', \sigma' >$, resp., states that the current continuation X is to be replaced by the empty tree or the new program X', resp. An inference rule of the general form, cf. Section 1.4, calls the rewriting procedure recursively on one of the direct

proc $eval\_AST(Cur\_AST, state) : (New\_AST, new\_state)$;
  if $Cur\_AST = nil$ **then**
    $(New\_AST, new\_state) := (nil, state)$;
  fi;
  if $Cur\_AST \in X$ **and** $< X, \sigma > \rightarrow \sigma' \in Spec$ **then**
    $new\_state := update(\sigma, \sigma')[\sigma/state]$;
    $New\_AST := nil$;
  fi;
  if $Cur\_AST \in X$ **and** $< X, \sigma > \rightarrow < X', \sigma' > \in Spec$ **then**
    $new\_state := update(\sigma, \sigma')[\sigma/state]$;
    $New\_AST := X'[X/Cur\_AST]$;
  fi;
  if $Cur\_AST \in X_0$
  and $\dfrac{\mathsf{Eval}(X_{l_1}, \sigma) = value_1, \ldots, \mathsf{Eval}(X_{l_m}, \sigma) = value_m,\ < X_i, \sigma > \rightarrow < X_i', \sigma' >}{< X_0, \sigma > \rightarrow < X_0', \sigma' >} \in Spec$
  and $\exists$ *direct subtree* $X_i(Cur\_AST)$ *of* Cur_AST *such that* $X_i(Cur\_AST) \in X_i$
  and $Eval(X_{l_1}(Cur\_AST, state)) = value_1$ **and** $\cdots$
  and $Eval(X_{l_m}(Cur\_AST, state)) = value_m$ **then**
    $(Cur\_AST', state') := eval\_AST(X_i(Cur\_AST), state)$;
    $(New\_AST, new\_state) := eval\_AST(X_0'[X_0/Cur\_AST, X_i'/Cur\_AST'], state')$;
  fi;
  return (New_AST, new_state);
end_proc

(AST stands for *abstract syntax tree*.)

Figure 6.2: Meaning of a Small-Step Semantics

subtrees X_i of the current program. After its completion, the rewriting procedure modifies its continuation and state by possibly integrating the results of the recursive call. The detailed recursive algorithm is stated in Figure 6.2 in a pseudo-Pascal notation.

In general, this is not a pure term rewriting procedure since nodes may have static continuation attributes, cf. the goto-definition in Section 1.4, which might point to arbitrary nodes in the original program tree. So whenever we talk about a subtree of the original program, we do not only mean the subtree itself but the transitive closure of all subtrees to which static continuations point.

The algorithm in Figure 6.2 can easily be transformed into an ASM definition. The recursion is eliminated by transforming the recursive procedure into a while-loop which runs until the program is reduced to the empty tree. In the usual way, the nested recursive calls at run-time are modelled by a stack whose entries are tuples of the current continuation (i.e. program) and the current state, carrying the state of computation of the individual recursive calls. This is a trivial standard proceeding to eliminate recursion. The resulting while-loop can easily be restated as an ASM: The while-loop still contains the four if-statements as the original recursive procedure. It is straightforward to transform these if-statements into four corresponding ASM transition rules. The content of the stack of the while-loop during execution becomes the state of the ASM.

This resulting ASM is different from the ASMs typically defined when specifying the semantics of programming languages, cf. Section 1.2. It does not define how to traverse the abstract syntax tree, e.g. by using a current task CT. Even though in many practical cases, given a small-step semantics, a human might easily be able to define walks through abstract syntax trees and

a corresponding ASM, in general we cannot hope to find such an automatic transformation. This is because a small-step semantics has many degrees of freedom in transforming a given program. It might duplicate subtrees, move subtrees from one part of the program to others by using the static continuation attributes, etc. This demonstrates the fundamental difference between small-step semantics on one hand and big-step semantics and ASM semantics on the other hand: While in big-step and ASM semantics, the abstract syntax tree is a constant during program execution, it is modified during the execution defined by a small-step semantics.

From ASMs to Small-Step Semantics

An ASM specification defines the semantics of a programming language operationally based on the abstract syntax trees of the programs. The state of the ASM contains a reference CT to some node in the abstract syntax tree, pointing to the current task to be executed. ASM semantics specifications are able to describe not strictly compositional semantics. Therefore we cannot expect to find a transformation from ASM semantics specifications to big-step semantics because big-step semantics can only define strictly compositional semantics. But we can define a transformation from ASM semantics to small-step semantics.

The idea is to take the abstract syntax tree as (constant) dynamic continuation. The current task CT becomes part of the state: If σ is a state of the ASM and CT the current task during some point of program execution, then (CT, σ) is the state at the same point of program execution wrt. to the corresponding small-step semantics. Formally, for each transition rule in an ASM semantics,

if $CT \in X$ **then**
  if *applicability_conditions* **then**
    $CT :=$ *new_CT*; *further_updates*
  else $CT :=$ *new_CT′*; *further_updates′* **fi fi**

the corresponding inference rules are defined as follows:

$$\frac{\mathit{applicability\_conditions}(AST, (\sigma, CT))}{< AST, (\sigma, CT) > \rightarrow < AST, (\mathit{further\_updates}(\sigma), \mathit{new\_CT}) >}$$

$$\frac{\neg \mathit{applicability\_conditions}(AST, (\sigma, CT))}{< AST, (\sigma, CT) > \rightarrow < AST, (\mathit{further\_updates'}(\sigma), \mathit{new\_CT'}) >}$$

Again, as in the preceeding transformations, we assume that the data structures of the ASM specification can be transformed easily into corresponding data structures of a small-step semantics. To prove that the defined transformation is correct, we need to show that for each program, the state transitions are the same wrt. to both specifications. Since the inference rules are not recursive, i.e., there are no state transitions specified in their assumptions, each execution will directly undertake the state transition of the conclusion of some inference rule whose assumptions are valid. This is the same as saying that some transition rule whose conditions (which are equivalent to the assumptions of the matching inference rule) are fulfilled will be executed. Therefore it follows immediately that the original ASM specification and the defined small-step semantics are equivalent.

6.2.2 Related Work

Again, as already discussed in Chapter 4, in particular in Section 4.5, our results here are in contrast to the common understanding (cf. [NN99] or any other textbook or lecture notes of

your choice) that big-step semantics can only describe terminating programs while small-step semantics is also suited for the description of non-terminating computations. This view is not adequate as our investigations show. Rather, it is the common interpretation of big-step semantics which fits only for terminating computations. In the traditional interpretation of big-step semantics, the assumptions must be true (terminating) in order to infer the conclusion. In our view, we ask if a (terminating or non-terminating) state transition sequence is consistent with the rules of the big-step semantics, allowing us to deal with non-terminating computations as well. This seems to be more appropriate anyway. Otherwise one could not determine whether a program has a semantics at all because this would be equivalent to solve the halting problem.

There are no other works comparing ASMs and big-step and small-step semantics wrt. the "applicability to imperative programming languages" and "treatment of the AST". In particular, no transformations between the three mechanisms have been proposed. Only [MCK+00] proposes a mechanism to generate action notation environments from montages descriptions.

6.2.3 Conclusions of the Comparison

The three specification frameworks ASMs and small-step and big-step semantics vary significantly wrt. our two criteria "applicability to imperative programming languages" and "treatment of the abstract syntax tree". While big-step semantics can only define strictly compositional programming languages, ASMs and small-step semantics can also specify non-strictly compositional program constructs. Furthermore, big-step and most ASM semantics do not modify the abstract syntax tree during computation, in contrast to small-step semantics which explicitly defines a term-rewriting system that rewrites the program during execution until the empty program is reached. These differences are reflected in the transformations between them. We have shown that each ASM semantics can be transformed into an equivalent small-step semantics and vice versa. Furthermore we have proved that each big-step semantics can be transformed into an equivalent ASM semantics while the reverse direction cannot be expected.

From a theoretical point of view, these transformations are interesting as they reveal the unexpressed interpretations of the specification frameworks. ASMs and small-step semantics follow the idea that a program defines a state transition system whose execution can be observed. In contrast, the usual interpretation of big-step semantics defines how to construct finite state transition sequences. Our transformations indicate that for each specification mechanism, both interpretations are possible. The traditional classification – big-step only for terminating programs, small-step also for non-terminating programs – is not a classification of the specification frameworks but rather of their usual interpretations.

These transformations are also interesting from a practical point of view. In the ASM as well as in the natural semantics community, a remarkable engineering knowledge has emerged concerning the way in which specifications should be written to be useful for the purpose of semantics specification and translation verification. Having the transformations between these mechanisms in mind, we can transfer engineering knowledge from one community to another.

In practice, most small-step semantics do not have the intention to define a term-rewriting system but rather incorporate the idea of defining a current task as in the ASM semantics. Therefore it would be interesting to define a simplified small-step semantics which allows for recursive semantic definitions on the tree structure of the program but does not permit to rewrite it. This seems to be sufficient for the usual applications. Moreover, it should be investigated if the three specification mechanisms ASMs and small-step and big-step semantics deal differently with multi-threaded and parallel programming languages.

Bibliography

[ACEL96] Isabelle Attali, Denis Caromel, Sidi O. Ehmety, and Sylvain Lippi. Semantic-Based Visualization for Parallel Object-Oriented Programming. In *Proceedings OOPSLA'96 (Object-Oriented Programming: Systems, Languages, and Applications)*, San Jose, CA, October 1996. ACM Press, Sigplan Notices, Vol. 31, No. 10.

[ASM] Website ASM. http://www.eecs.umich.edu/gasm/.

[Att96] Isabelle Attali. A Natural Semantics for Eiffel Dynamic Binding. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(5), November 1996.

[Bar93] M. Barr. Terminal coalgebras in well-founded set theory. *Theor. Comp. Sci.*, 114(2):299–315, 1993. Corrigendum in *Theor. Comp. Sci.* 124:189-192, 1994.

[BCG02] Roland Backhouse, Roy Crole, and Jeremy Gibbons, editors. *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. Springer, Lecture Notes in Computer Science, Vol. 2297, 2002. International Summer School and Workshop, Oxford, UK, April 10-14, 2000, Revised Lectures.

[BD96] Egon Börger and Igor Durdanovic. Correctness of compiling Occam to Transputer Code. *Computer Journal*, 39(1):52–92, 1996.

[Ber90] Yves Bertot. Implementation of an Interpreter for a Parallel Language in Centaur. In *Proceedings of the 3rd European Symposium on Programming*, Copenhagen, Denmark, May 1990. Springer Verlag, LNCS 432.

[BR94] Egon Börger and Dean Rosenzweig. The WAM - definition and compiler correctness. In L.C. Beierle and L. Pluemer, editors, *Logic Programming: Formal Methods and Practical Applications*. North-Holland Series in Computer Science and Artificial Intelligence, 1994.

[CC92] Patrick Cousot and Radhia Cousot. Inductive Definitions, Semantics and Abstract Interpretation. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–94, Albuquerque, New Mexico, USA, January 19-22 1992. ACM.

[DE99] Sophia Drossopoulou and Susan Eisenbach. Describing the Semantics of Java and Proving Type Soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, page 41 ff. Springer Verlag, LNCS 1523, 1999.

[DV01] Axel Dold and Vincent Vialard. A Mechanically Verified Compiling Specification for a Lisp Compiler. In *Proceedings of the 21st Conference on Software Technology and Theoretical Computer Science (FSTTCS 2001)*, pages 144–155, Bangalore, India, December 13-15 2001. Springer Verlag, Lecture Notes in Computer Science, Vol. 2245.

[EGGP00] Robert Eschbach, Uwe Glässer, Reinhard Gotzhein, and Andreas Prinz. On the Formal Semantics of Design Languages: A compilation approach using Abstract State Machines. In *Abstract State Machines – ASM 2000, International Workshop on Abstract State Machines*. Local Proceedings, TIK-report 87, Swiss Federal Institute of Technology (ETH) Zurich, 2000.

[GH93] Yuri Gurevich and James K. Huggins. The Semantics of the C Programming Language. In *Selected papers from CSL'92 (Computer Science Logic)*, pages 274–308. Springer Verlag, Lecture Notes in Computer Science, Vol. 702, 1993.

[Gle99a] Sabine Glesner. Natural Semantics for Imperative and Object-Oriented Programming Languages. In K. Beiersdörfer, G. Engels, and W. Schäfer, editors, *Informatik '99 - Informatik überwindet Grenzen, Proceedings der 29. Jahrestagung der Gesellschaft für Informatik*, pages 370–379, Paderborn, October 1999. GI-Gesellschaft für Informatik e.V., Springer Verlag, ISBN: 3-540-66450-5.

[Gle99b] Sabine Glesner. *Natürliche Semantik für imperative und objektorientierte Programmiersprachen*. PhD thesis, Universität Karlsruhe, Fakultät für Informatik, February 1999. Shaker Verlag, Aachen, ISBN: 3-8265-6388-3.

[Gle03] Sabine Glesner. ASMs versus Natural Semantics: A Comparison with New Insights. In Egon Börger, Angelo Gargantini, and Elvinia Riccobene, editors, *Abstract State Machines - Advances in Theory and Applications, Proceedings of the 10th International Workshop, ASM 2003*, Taormina, Italy, March 2003. Springer Verlag, Lecture Notes in Computer Science, Vol. 2589.

[Gle04a] Sabine Glesner. A Proof Calculus for Natural Semantics Based on Greatest Fixed Point Semantics. In *Proceedings of the Workshop Compiler Optimization meets Compiler Verification (COCV 2004), 7th European Conferences on Theory and Practice of Software (ETAPS 2004)*, Barcelona, Spain, April 2004. Elsevier, Electronic Notes in Theoretical Computer Science (ENTCS).

[Gle04b] Sabine Glesner. Verification of Optimizing Compilers, 2004. Habilitationsschrift (Habilitation Thesis), Universität Karlsruhe.

[Gor95] Andrew D. Gordon. A Tutorial on Co-Induction and Functional Programming. In *Proceedings of the 1994 Glasgow Workshop on Functional Programming, September 1994*, Ayr, Scotland, 1995. Springer Workshops in Computing.

[Gur95] Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.

[GZ98] Sabine Glesner and Wolf Zimmermann. Using Many-Sorted Natural Semantics to Specify and Generate Semantic Analysis. In R. Nigel Horspool, editor, *Proceedings of the Systems Implementation Conference (SI2000)*, pages 249–262, Berlin, February 1998. IFIP Working Group 2.4, Chapman & Hall, ISBN: 0-412-83530-4.

[GZ99] Gerhard Goos and Wolf Zimmermann. Verification of Compilers. In E.-R. Olderog and B. Steffen, editors, *Correct System Design*, pages 201–230. Springer-Verlag, Lecture Notes in Computer Science, Vol. 1710, 1999.

[GZ04] Sabine Glesner and Wolf Zimmermann. Natural Semantics as a Static Program Analysis Framework. *ACM Trans. Program. Lang. Syst.*, 26(3):510–577, 2004.

[Hoa69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576 – 580, October 1969.

[Hui01] Marieke Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. PhD thesis, Faculty of Science, University of Nijmegen, 2001.

[IS98] Husain Ibraheem and David A. Schmidt. Adapting Big-Step Semantics to Small-Step Style: Coinductive Interpretations and "Higher-Order" Derivations. In C. Talcott, editor, *Proceedings 2nd Workshop on Higher-Order Techniques in Operational Semantics (HOOTS2)*, Stanford, 1998. Elsevier ENTS.

[JR97] Bart Jacobs and Jan Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *EATCS Bulletin*, 67:222–259, 1997.

[Kah87] Gilles Kahn. Natural Semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science (STACS'87)*, pages 22–39, Passau, Germany, February 1987. Springer, LNCS 247.

[MCK+00] M.Anlauff, S. Chakraborty, P.W. Kutter, A. Pierantonio, , and L. Thiele. Generating an Action Notation Environment from Montages Descriptions. *International Journal on Software Tools for Technology Transfer (STTT)*, 2000.

[Mit90] John C. Mitchell. *Type Systems for Programming Languages*, volume B of *Handbook of Theoretical Computer Science*, chapter 8, pages 365–458. MIT Press/Elsevier Science Publishers B.V., 1990.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[NN99] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Published in 1992 by John Wiley & Sons, revised edition available at http://www.daimi.au.dk/~ hrn, 1999.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[NvO98] Tobias Nipkow and David von Oheimb. Java_{light} is Type-Safe – Definitely. In *Proceedings of the 25th ACM Symposium on the Principles of Programming Languages*, pages 161–170, San Diego, California, USA, January 19–21 1998. ACM Press.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Proceedings 11th International Conference on Automated Deduction CADE*, pages 748–752, Saratoga, NY, October 1992. Springer-Verlag, Lecture Notes in Artificial Intelligence, vol. 607.

[Pau02] Lawrence C. Paulson. A Fixedpoint Approach to (Co)Inductive and (Co)Datatype Definitions, 2002. Computer Laboratory, University of Cambridge, England.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, 1991.

[Sch96] David Schmidt. Programming Language Semantics. In Allen Tucker, editor, *CRC Handbook of Computer Science*, Boca Raton, USA, 1996. CRC Press.

[SSB01] Robert Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer Verlag, 2001.

[Sym99] Don Syme. Proving Java Type Soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, page 83 ff. Springer Verlag, LNCS 1523, 1999.

[vO01] David von Oheimb. *Analyzing Java in Isabelle/HOL*. PhD thesis, Technische Universität München, Germany, 2001.

[vON99] David von Oheimb and Tobias Nipkow. Machine-Checking the Java Specification: Proving Type-Safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, pages 119–156. Springer Verlag, Lecture Notes in Computer Science, Vol. 1523, 1999.

[ZG97] Wolf Zimmermann and Thilo Gaul. On the Construction of Correct Compiler Backends: An ASM Approach. *Journal of Universal Computer Science*, 3(5):504–567, 1997.