

Introducing Modal Fixed-Point Operators into CCSL

Christian Kissig

February 13, 2005

Bachelor Thesis Assignment (Computer Science)
Student: Christian Kissig

Modal Fixed Point Operators for Coalgebras

CCSL (short for Coalgebraic Class Specification Language) is a formal specification language, developed jointly at the university of Nijmegen and at TU Dresden. CCSL permits to specify the interface of classes of object-oriented languages by declaring their methods with argument and result types. Further, it is possible to restrict the behavior of the methods with a special form of higher order logic. The master thesis of Jan Rothe enhanced the logic of CCSL with the (infinitary) modal operators Box and Diamond.

Assignment: The aim of this project is to enrich CCSL with arbitrary modal fixedpoint operators as they are known from the μ -calculus. For this it is necessary:

1. To extend CCSL's logic such that one can express formulas from the modal μ -calculus in it. The extension can also be done in the form of "syntactic sugar".
2. To describe a semantics for the new syntax.
3. To develop an example specification that demonstrates the usefulness of the new operators.

Literature CCSL is documented in [Tew02b, Tew01, Rot00, Tew02a]. The modal μ -calculus is described in [Sti92].

Dresden, 29th Oktober 2004

Contents

1	Introduction	4
2	Preliminaries	5
2.1	Coalgebras for polynomial Functors	5
2.2	Predicates, Relations, Predicate- and Relation-Lifting, Bisimulations, and Bisimilarity	6
2.3	Fixed-Points	7
3	A simplified Type-System for CCSL	9
3.1	Kinds	9
3.2	Types	10
4	Class Signatures	12
4.1	Coalgebraic Class Signatures	12
4.2	Semantics of Signatures	14
5	Specifying the Behaviour	17
5.1	Terms, Formulae and Types	17
5.1.1	Syntax of CCSL-Terms	18
5.1.2	Syntactic Sugar	22
5.1.3	Class Specifications	22
5.2	Semantics of Class Specifications	23
6	Introducing single-step Modal Operators	25
7	Introducing modal Fixed-Point Operators	29
7.1	Discussion of Monotonicity	30
7.2	Defining the modal Fixed-Point Operators	33
7.3	Examples and Applications	34
8	Related Work and Conclusions	37

Chapter 1

Introduction

With the increasing complexity of today's software systems, reliability of software became a vital issue not only in safety critical applications. Erroneous programs may expose the users not only to the risk of financial losses, but also to physical threats, not talking about the loss of reputation.

In order to reduce as much costs as possible, the design of software should be validated as early as possible in the development process. Currently, there are three fundamental approaches to ensure correctness of software. There are still veteran techniques like overengineering or simulation and testing widely used, although they do not guarantee the complete correctness especially of larger systems. Model checking arose as an attempt to show the systems correctness by proving it against some specifying properties. However, those properties are hard to formulate in most model checkers. In this thesis we will focus on a verification system based upon a (co-)algebraic representation.

Today, most of the complex software-systems are written in object-oriented languages like C++ and Java. In recent years the formal semantics of object-orientation has been in focus of research. In 1995, Reichel recognized that terminal coalgebras precisely capture many aspects of object-orientation.

Modal logics have been shown to be an appropriate means to reason about dynamic systems. However, a user of CCSL might want to specify recursive properties. The semantics of such a recursive formula in the modal μ -calculus is given by fixed-point construction.

In this thesis I will extend the logic of CCSL by the modal fixed-point operators. Furthermore, I will introduce the single-step modal operators to potentiate a meaningful application of modal fixed-point formulae.

Chapter 2

Preliminaries

In this section I introduce the central mathematical basics that we will find and use throughout the rest of this thesis. Most of the interesting things happen in Category Theory, although only a little understanding thereof is necessary. Furthermore, I will use Set Theory as being taught in undergraduate courses in mathematics or computer science.

2.1 Coalgebras for polynomial Functors

In CCSL, systems designed in the object oriented approach are modelled with *Algebras* and *Coalgebras*. Both can be understood as morphisms in the category **Set** of sets and functions between them. Their domain or codomain, respectively, are described by endofunctors T on **Set**. Then an algebra a is considered as a T -algebra and a coalgebra c as a T -coalgebra, such that $a : TX \rightarrow X$ and $c : Y \rightarrow TY$ for X and Y being objects in **Set**. In CCSL those functors T are assumed to be polynomial in the sense of the following definition.

Definition 1 (Polynomial Functors) *Polynomial Functors $T : \mathbf{Set} \rightarrow \mathbf{Set}$ are defined inductively as follows :*

- *The identity functor id , i.e. $id(X) = X$ is polynomial.*
- *The constant functor $TX = A$ for some A is polynomial.*
- *The product $TX = T_1X \times T_2X$ of two polynomial Functors T_1 and T_2 is polynomial.*
- *The coproduct $TX = T_1X + T_2X$ of two polynomial Functors T_1 and T_2 is polynomial.*
- *The exponent $TX = A \rightarrow T_1X$ of a polynomial functor T_1 and a set A is polynomial.*

Actually, the notions for *Products* and *Coproducts* of sets stem from category theory as certain Limit constructions and thus they require a deeper explanation. However, as we have already indicated, most of the foundations for

CCSL take place in the category **Set**, such that we can consider the product of two sets X and Y as $X \times Y = \{(x, y) \mid x \in X \text{ and } y \in Y\}$. The coproduct, or the disjoint sum, of two sets X and Y can be considered as $X + Y = \{(x, 1) \mid x \in X\} \cup \{(y, 1) \mid y \in Y\}$. The exponent Y^X of two sets X and Y is then the set $\{f : X \rightarrow Y\}$ of functions from X to Y . In the standard literature on algebraic category theory like [Bor94] the proofs are shown that those product, coproduct and exponent constructions do indeed satisfy the properties of the corresponding limit constructions.

In the next chapters we will see that constructors of a class are interpreted to algebras and methods to coalgebras w.r.t. a functor that is determined by the typing of those constructors and methods involved.

2.2 Predicates, Relations, Predicate- and Relation-Lifting, Bisimulations, and Bisimilarity

In this thesis we treat predicates P as subsets $P \subseteq X$ and relations as subsets $R \subseteq X \times Y$. For a set X the predicate \top_X denotes the truth predicate on X and thus holds for all $x \in X$. The equality relation $=_X$ on a carrier X is defined as the diagonal relation $\{(x, x) \mid x \in X\}$ on X . For general type relations $R \subseteq X \times Y$ and $S \subseteq Y \times Z$ we define the opposite R^{-1} of R as $R^{-1} = \{(x, y) \in X \times Y \mid (y, x) \in R\}$, and the composition of R and S as $R \circ S = \{(x, z) \in X \times Z \mid \text{there is some } y \in Y \text{ such that } (x, y) \in R \text{ and } (y, z) \in S\}$

The following definitions describe the lifting of predicates and relations along a polynomial functor T . These notions stem from fibration semantics of predicate logic as introduced in [Jac99]. But, we will not need the abstract aspects of fibration theory in this thesis.

Definition 2 (Predicate and Relation Lifting)

Let T be a polynomial functor, $P \subseteq X$ a predicate, and $R \subseteq X \times X$ a relation. Then, the predicate lifting $\text{Pred}(T)(P) \subseteq TX$ for P along T and the relation lifting $\text{Rel}(T)(R) \subseteq TX \times TX$ for R along T are defined inductively over the structure of T :

- *If $T = \text{id}$, i.e. $TX = X$, then $\text{Pred}(T)(P) = P$ and $\text{Rel}(T)(R) = R$*
- *If $TX = A$ for a fixed set A , then $\text{Pred}(T)(P) = \top_A$ and $\text{Rel}(T)(R) = =_A$.*
- *If $T = T_1 \times T_2$, i.e. $TX = T_1X \times T_2X$, then*
 $\text{Pred}(T)(P) = \{(x_1, x_2) \mid \text{Pred}(T_1)(P)(x_1) \text{ and } \text{Pred}(T_2)(P)(x_2)\}$ *and*
 $\text{Rel}(T)(R) = \{((x_1, x_2), (y_1, y_2)) \mid \text{Rel}(T_1)(R)(x_1, y_1) \text{ and } \text{Rel}(T_2)(R)(x_2, y_2)\}$.
- *If $T = T_1 + T_2$, i.e. $TX = T_1X + T_2X$, then*
 $\text{Pred}(T)(P) = \{\kappa x_1 \mid \text{Pred}(T_1)(P)(x_1)\} \cup \{\kappa' x_2 \mid \text{Pred}(T_2)(P)(x_2)\}$ *and*
 $\text{Rel}(T)(R) = \{(\kappa x_1, \kappa y_1) \mid \text{Rel}(T_1)(R)(x_1, y_1)\} \cup \{(\kappa' x_2, \kappa' y_2) \mid \text{Rel}(T_2)(R)(x_2, y_2)\}$.
- *If $TX = A \rightarrow T_1X$, then*
 $\text{Pred}(T)(P) = \{f \mid \forall a \in A. \text{Pred}(T_1)(P)(f(a))\}$ *and*
 $\text{Rel}(T)(R) = \{(f, g) \mid \forall a \in A. \text{Rel}(T_1)(R)(f(a), g(a))\}$.

Intuitively the predicate lifting brings a predicate on the state space forward to a predicate on the codomain of the coalgebra. The following additional results shall give a deeper understanding of the relation lifting.

Introducing the following notions of *Invariance*, *Bisimulation* and *Bisimilarity* we make use of the Predicate Lifting.

Definition 3 (Invariance, Bisimulation and Bisimilarity for Coalgebras)

For a (polynomial) functor T consider two T -coalgebras $c : X \rightarrow TX$ and $d : Y \rightarrow TY$, then

- A relation $R \subseteq X \times Y$ is called a *bisimulation* for c and d , if $\forall x \in X, y \in Y. R(x, y) \Rightarrow \text{Rel}(T)(R)(c(x), d(y))$.
- The *bisimilarity* relation ${}_c \overset{\leftrightarrow}{-}{}_d \subseteq X \times Y$ is defined as $x {}_c \overset{\leftrightarrow}{-}{}_d y \Leftrightarrow \exists R \subseteq X \times Y. R(x, y)$ and R is a bisimulation.

For the bisimilarity ${}_c \overset{\leftrightarrow}{-}{}_c$ we drop the subscript if c is understood from context.

Again, we will give some standard results to support the understanding of the previously defined notions.

Lemma 4 (Additional results on Invariance, Bisimulation, and Bisimilarity)

For a (polynomial) functor T assume three T -coalgebras c, d , and e over X, Y , and Z respectively. Then

- $=_X$ is a bisimulation
- \top_X is an invariant
- invariants are closed under intersection and union, i.e. for an indexed collection $(P_i)_{i \in I}$ both $\bigcup_{i \in I} P_i$ and $\bigcap_{i \in I} P_i$ are invariants.
- bisimulations are closed under intersection and union in a similar manner
- ${}_c \overset{\leftrightarrow}{-}{}_d \subseteq ({}_d \overset{\leftrightarrow}{-}{}_c)^{-1}$
- ${}_d \overset{\leftrightarrow}{-}{}_e \circ {}_c \overset{\leftrightarrow}{-}{}_d \subseteq {}_c \overset{\leftrightarrow}{-}{}_e$
- ${}_c \overset{\leftrightarrow}{-}{}_d$ is a bisimulation for c and d
- ${}_c \overset{\leftrightarrow}{-}{}_c$ is an equivalence relation.

For proofs of these results consult [Rot00].

2.3 Fixed-Points

Consider an arbitrary carrier set S , then a function $g : 2^S \rightarrow 2^S$ mapping a predicate over S to a predicate over the same carrier is called a predicate transformer. In a fixed-point X of such a predicate transformer it holds that

$X = g(X)$. In general a predicate transformer may have more than one fixed-point. Especially in modal logics one restricts the considerations on the least and greatest fixed-point w.r.t. set inclusion. Knaster and Tarski have shown that a predicate transformer has a least and greatest fixed point if it is monotonic, i.e. for arbitrary subsets X and Y of S it holds that $X \subseteq Y \Rightarrow g(X) \subseteq g(Y)$.

Chapter 3

A simplified Type-System for CCSL

In CCSL, the appearance of a class to be validated is given by its signature. Such a coalgebraic class signature describes the constructors and methods of a class together with their types. The interpretation of those types will determine the functor T for those (co-)algebras modelling this signature.

In order to represent polymorphic classes like templates in C++, CCSL has to be based upon a polymorphic type theory. In such a type theory the typing depends not only on the regarded term, but also on an instantiation of some type and term variables. A term judgements thus consists of four ingredients, a type variable context (a set of type variables), a term variable context (a set of terms together with their respective types), the regarded term and its type. Actually, CCSL is founded on a specialized version of the polymorphic $\lambda \rightarrow$ type-theory(see [Bar92] for details) enriched by product, coproduct, and exponent types together with the special types **Self**, representing the state space of classes and abstract data types, and **Prop**, being the type of propositions.

However, a type system as it underlies CCSL is rather complex and brings several ambiguities. Since, the focus of this thesis shall be on the modal fixed-point operators rather than the the discussion of all the corner stones of CCSL's type system, we will use a slightly simplified version that does not allow for binary methods in classes. We ensure the exclusion of binary methods by only allowing polynomially typed methods.

3.1 Kinds

In CCSL *Kinds* are used to distinguish type constructors from types by the denoting them by natural numbers representing their respective number of arguments. So, kinds are basically finite ordinals, namely natural numbers in CCSL. The kind of ordinary types is 0.

We denote kinds by outlined letters, such that kind judgments look like $\vdash \mathbb{k} : \mathit{Kind}$, meaning \mathbb{k} is a valid kind, i.e. a natural number.

3.2 Types

Types in CCSL are built over a set of type parameters \mathcal{P} and type constructors \mathcal{C} . We denote type variables with lowercase Greek letters α, β, \dots . In a type context all type variables are supposed to be distinct. Furthermore, type variables in a type context shall be placeholders for types only, and not type constructors. To emphasize this type contexts are denoted by lists of associations $\alpha_i : Type$ with explicit kind $Type$.

Arbitrary types are denoted by with lowercase Greek letters like σ and τ , and derived formally in type judgements of the form

$$\Xi \vdash \tau : \mathbb{k} \text{ or } \Xi \vdash \tau : Type$$

for Ξ being a type variable context and \mathbb{k} a valid kind. Such a type judgements state that τ is a type constructor of kind \mathbb{k} or an ordinary type respectively. If \mathbb{k} is 0 then τ is called a constant type and denoted by an uppercase C .

Definition 5 ((Polynomial) Types in CCSL) *Let \mathcal{C} be a set of type constructors, then the set of types generated over \mathcal{C} is defined inductively by*

- α for a type variable $\alpha : Type$
- C for a type constant $(\vdash C : Type) \in \mathcal{C}$
- $C[\tau_1, \dots, \tau_k]$ for a type constructor $(\vdash C : \mathbb{k}) \in \mathcal{C}$.
- the special type **Self** for the state space of the class or abstract data type
- the special type **Prop** of propositions
- the unit type 1, and the empty type 0
- the product $\tau \times \sigma$ and coproduct $\tau + \sigma$ for types τ and σ
- the exponent $\tau \rightarrow \sigma$ for types τ and σ whereby τ does not contain any occurrence of **Self**

Product and coproduct constructions are supposed to be associative, i.e. $(\tau_1 \times \tau_2) \times \tau_3 = \tau_1 \times (\tau_2 \times \tau_3)$. And, the exponent associates to the right, such that $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 = \tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$ for types τ_1, τ_2 , and τ_3 . For all three constructions I will omit parenthesis, when they do not contribute to the readability.

The semantics of a type is given by a functor. Because in this thesis we will deal with polynomial types only such a functor will be polynomial, too. More formally, every type $\Xi \vdash \tau : Type$ in a variable context Ξ gives rise to a collection of functors indexed by an interpretation U_1, \dots, U_n of the n type parameters in Ξ

$$([\tau]_{U_1, \dots, U_n} : Set \rightarrow Set)_{U_i \in |\mathbf{Set}|}$$

Definition 6 (Interpretation of Types)

1. Type constructors C of arity \mathbb{k} are interpreted to functors

$$[[C]] : \text{Set}^{\mathbb{k}} \rightarrow \text{Set}$$

for $(-)^{\mathbb{k}}$ being the \mathbb{k} -fold product.

2. Let $\alpha_1, \dots, \alpha_n \vdash \tau : \text{Type}$ be a type with type constructors C from \mathcal{C} each having an interpretation $[[C]]$. The interpretation of τ is inductively defined as follows

- $[[\alpha_i]]_{U_1, \dots, U_n}(X) = U_i$
- $[[\mathbf{Self}]]_{U_1, \dots, U_n}(X) = X$
- $[[1]]_{U_1, \dots, U_n}(X) = 1 = \{*\}$
- $[[0]]_{U_1, \dots, U_n}(X) = 0 = \emptyset$
- $[[\mathbf{Prop}]]_{U_1, \dots, U_n}(X) = \text{bool} = \{\top, \perp\}$
- $[[\sigma_1 \times \sigma_2]]_{U_1, \dots, U_n}(X) = [[\sigma_1]]_{U_1, \dots, U_n}(X) \times [[\sigma_2]]_{U_1, \dots, U_n}(X)$
- $[[\sigma_1 + \sigma_2]]_{U_1, \dots, U_n}(X) = [[\sigma_1]]_{U_1, \dots, U_n}(X) + [[\sigma_2]]_{U_1, \dots, U_n}(X)$
- $[[\sigma_1 \rightarrow \sigma_2]]_{U_1, \dots, U_n}(X) = [[\sigma_1]]_{U_1, \dots, U_n}(X) \rightarrow [[\sigma_2]]_{U_1, \dots, U_n}(X)$ with σ_1 not containing an occurrence of \mathbf{Self} (see definition 5 on page 10)
- $[[C[\sigma_1, \dots, \sigma_k]]]_{U_1, \dots, U_n}(X) = [[C]]([[\sigma_1]]_{U_1, \dots, U_n}, \dots, [[\sigma_k]]_{U_1, \dots, U_n})$

Chapter 4

Class Signatures

One of the central subjects in coalgebraic class specification is a *Signature*. Signatures give information about constructors and methods of a class together with their respective types. Since, the user of CCSL does not only want to specify stand-alone classes, but rather systems of classes, a CCSL class specification consists of many signatures both of classes and of abstract data types. However, we will not explicitly consider such a kind of structuring.

4.1 Coalgebraic Class Signatures

A class signature contains the types of both constructors and methods of a class. Like templates in C++, classes can be generalized by definition w.r.t. a set of type parameters. In CCSL class signatures are thus defined in dependence on a set \mathcal{P} of type parameters.

As we have indicated in the beginning of this chapter the user might wish to define hierarchies of many signatures both for classes and ADTs. This kind of interrelations is manifested in CCSL signatures by introducing a set \mathcal{C} of type constructors.

Definition 7 (Coalgebraic Class Signatures) *A class signature is a tuple $\langle \Sigma_M, \Sigma_C \rangle$ defined over a finite set \mathcal{C} of type constructors and a finite set \mathcal{P} of parameter types consisting of sets Σ_M and Σ_C of types built from \mathcal{C} and \mathcal{P} . Σ_C is restricted in that **Self** must not occur in it. Σ_M associates a type to all methods, while Σ_C associates the constructors with a type.*

Subsignatures are a means to select parts of class signatures. They can be used for inheritance, specifying the visibility modifiers PUBLIC or PRIVATE as known from C++ or Java, and for defining a semantics of modal operators. Rothe showed the latter in his diploma thesis [Rot00]. For a coalgebraic class signature $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$ the signature $\Sigma' = \langle \Sigma'_M, \Sigma'_C \rangle$ is a subsignature if $\Sigma'_M \subseteq \Sigma_M$.

CCSL signatures regularly share a special kind of common subsignature, namely a *ground signature*. A signature is called ground if Σ_M is empty. Furthermore,

a ground signature is called plain if Σ_C contains only constant type constructors and proper if for each type constructor C in \mathcal{C} there are at least two symbols $Pred_C$ and Rel_C for predicate and relation lifting, respectively.

The following examples shall clarify how to write coalgebraic class signatures.

Example 8 (Scheduler) *The central example of this thesis will be a general-type scheduler. Such a scheduler is requested by clients for resources. If a resource is available it is assigned to a requesting client. Otherwise the client is enqueued until a resource becomes available. After the client finished its operations on the resource it is released to the scheduler.*

Obviously, the methods of our Scheduler class should have the following types

$$\begin{aligned} \text{request} &: \mathbf{Self} \times \text{Client} \rightarrow \mathbf{Self} \\ \text{assign} &: \mathbf{Self} \times \text{Client} \times \text{Resource} \rightarrow \mathbf{Self} \\ \text{finished} &: \mathbf{Self} \times \text{Client} \times \text{Resource} \rightarrow \mathbf{Self} \\ \text{get_resource} &: \mathbf{Self} \rightarrow \text{Resource} + 1 \end{aligned}$$

Comparing these results with Definition 5 on page 10, one can easily see that these types are not polynomial, since they \mathbf{Self} is contained in the exponent. Stripping off the \mathbf{Self} component from the argument side of every method solves this problem, but forbids partial functions. Therefore, CCSL provides us with the abstract data type *Lift* which I sketch in the following definition for demonstrative purposes.

Example 9 (Lift) *Before we can formulate a complete signature of such a scheduler, we have to define the abstract data type Lift. In theoretic context Lift provides us with a sum construction $1 + A$ for a set A . For the scheduler example we need Lift to construct the partial function `get_resource`. The following figure demonstrates a CCSL-signature of Lift as an abstract data type*

```
BEGIN Lift[A:TYPE]: ADT
Constructor
  bot: Carrier;
  up: X -> Carrier
END Lift
```

Recognize that to each constructor a respective recognizer is associated, i.e. `bot?` is associated to `bot` and `up?` to `up`.

Example 10 (Scheduler - continued) *Using the recently defined ADT in our Scheduler example enables us to define the methods separately and even the partial function `get_resource` as follows*

Methods

$$\begin{aligned} \text{request} &: \text{Client} \rightarrow \mathbf{Self} \\ \text{assign} &: \text{Client} \times \text{Resource} \rightarrow \mathbf{Self} \\ \text{finished} &: \text{Client} \times \text{Resource} \rightarrow \mathbf{Self} \\ \text{get_resource} &: \text{Lift}[\text{Resource}] \end{aligned}$$

Constructors

$$\text{new_scheduler}: 1$$

The intuition behind the construction of `get_resource` is, that if `get_resource` returns `bot` as a result, the method fails, meaning that there is no free resource available. If on the other hand `get_resource` returns `up(r)` for a resource `r`, then `r` is available.

Fitting the results gathered so far into the scheme of class signatures gives us

- *Client and Resource being the only parameter types in \mathcal{P}*
- *Lift[Resource] being the only type constructor in \mathcal{C}*
- $\sigma_M = (\text{Client} \rightarrow \mathbf{Self}) \times (\text{Client} \times \text{Resource} \rightarrow \mathbf{Self}) \times \text{Lift}[\text{Client}]$
- *and $\sigma_C = 1$*

Since CCSL is designed with a functional view on class specifications the user specifies the interface of a class instead of its signature. This requires to give functional types for methods and constructors. Notice the difference between the signature and the types in the CCSL class interface.

```
BEGIN Scheduler[Client,Resource:TYPE]: CLASSSPEC
METHOD
    request: Self x Client -> Self;
    assign: Self x Client x Resource -> Self;
    finished: Self x Client x Resource -> Self;
    get_resource: Self -> Lift[Resource];
CONSTRUCTOR
    new_scheduler: Self;
END Scheduler
```

4.2 Semantics of Signatures

In this section we will define the semantics of class signatures. Expectedly, methods will be interpreted in terms of coalgebras, and types set-theoretically.

Definition 11 (Model of a Class Signature) Let $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$ be a class signature with n type parameters $\alpha_1, \dots, \alpha_n$. A model \mathcal{M}_Σ of Σ consists of an indexed collection of triples $(\langle X, c, a \rangle_{U_1, \dots, U_n})$, such that for each interpretation U_1, \dots, U_n of the type parameters X is the state space, c is the coalgebra, and a is the algebra as in the bi-algebra

$$[[\sigma_\Sigma]]_{U_1, \dots, U_n} \xrightarrow{a} X \xrightarrow{c} [[\tau_\Sigma]]_{U_1, \dots, U_n}(X)$$

A model of a class signature thus gives a coalgebraic interpretation of the methods, an algebraic interpretation of the constructors, and a set-theoretic interpretation of the state-space of the class. The following model for our Scheduler example shall illustrate this definition further.

Example 12 (Scheduler - continued) In our scheduler example the interface functor for σ_M looks like

$$|\sigma_M| : S \mapsto S^{[[Client]]} \times S^{[[Client]] \times [[Resource]]} \times S^{[[Client]] \times [[Resource]]} \times [[Lift[Resource]]]$$

such that the coalgebra for Scheduler is of type

$$X \rightarrow X^{[[Client]]} \times X^{[[Client]] \times [[Resource]]} \times X^{[[Client]] \times [[Resource]]} \times [[Lift[Resource]]]$$

It is still left open how the state space S looks like. As an intuitive interpretation of **Self** I choose a function assigned from $[[Resource]]$ to the disjoint sum $1 + [[Client]]$ with the idea that if assigned is $\kappa_1 \perp$ for some resource then it is not assigned to a client. If it is assigned the respective client is returned. Furthermore we need a list of clients having requested a resource. Therefore we just take the set $request_list = [[Client]]^*$ of all words over the alphabet $[[Client]]$. The state space S is just the product assigned $\times request_list$.

Now, we can give a semantics to each of the methods in the scheduler class according to the previous intuition as given in example 8 (page 13)

Methods

$$\begin{aligned} f_{request} &= \lambda(x : X, c : [[Client]]).let(v = get_resource(x)) \text{ in} \\ &\quad (\text{cases } v \text{ of } \left\{ \begin{array}{l} \kappa_1 \perp : \pi_1(x) \times (\pi_2(x) \circ c) \\ \kappa_2 r : assign(x, c, r) \end{array} \right\}) \\ f_{assign} &= \lambda(x : X, c : [[Client]], r : [[Resource]]).(\lambda s : [[Resource]]. \\ &\quad \text{if } s = r \left(\begin{array}{l} \text{then } \kappa_2 c \\ \text{else } (\pi_1 x)(s) \times (\pi_2 x) \end{array} \right)) \\ f_{finished} &= \lambda(x : X, c : [[Client]], r : [[Resource]]).(\lambda s : [[Resource]]. \\ &\quad \text{if } s = r \left(\begin{array}{l} \text{then } \kappa_1 \perp \\ \text{else } (\pi_1 x)(s) \times (\pi_2 x) \end{array} \right)) \\ f_{get_resource} &= \lambda(x : X).choice(\{s : [[Resource]] \mid (\pi_1 x)(s) = \kappa_1 \perp\}) \end{aligned}$$

with \circ being the concatenation of words here, π_1 and π_2 , i.e. left and right projection of pairs, and choice : $2^{[[Client]]} \rightarrow [[Client]]$ a choice function on $[[Client]]$ as known from set theory.

It appears convenient to have access to selected methods. Then, the necessity of semantics for single methods arises. In the next definition the connections between σ_M and the single method types becomes obvious. Recall previous considerations on the semantics of a class signature that methods are interpreted by a T -coalgebra c mapping states $x \in X$ to elements in $[[\tau_{m_1}, \dots, \tau_{m_k}]]_{U_1, \dots, U_n}(X)$. The coalgebra interpreting the single i -th method is then just a mapping $X \rightarrow [[\tau_{m_i}]]_{U_1, \dots, U_n}(X)$. The obvious way to gather such a coalgebra is projection, as constituted by the following definition.

Definition 13 (Coalgebraic Interpretation of Methods) *Assume a class with n methods of type τ_{m_i} for $i \leq n$ and therefor a class signature $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$. A model $M_\Sigma = \langle X, c, a \rangle$ of Σ contains the coalgebra c interpreting the methods in this class. Thus c maps states $x \in X$ to elements of the interpretation $[[\tau_{m_1} \times \dots \times \tau_{m_k}]]_{U_1, \dots, U_n}$ of all methods for U_1, \dots, U_n being the interpretation of the typeparameters. A coalgebra interpreting the i -th method m_i is again a coalgebra $c_{m_i} = \pi_i \circ c$ with π_i being the i -th projection and \circ being functional composition.*

In the next chapter we will see how to restrict this class of models to those that show an intended behaviour.

Chapter 5

Specifying the Behaviour

In the last chapter we have seen the definition of models of class signatures. By now we have no means to restrict this class of models to those showing an intended behaviour. For example the following is a model for our Scheduler class signature though it contradicts our intuition of a Scheduler.

Example 14 (Scheduler - continued) *Let $\Sigma_{\text{Scheduler}}$ be the signature for our Scheduler example as given in example 10 from page 14.*

The state space X is interpreted by the unit set $\{\}$. Further, consider the following interpretation of each single method*

$$\begin{aligned}c_{\text{request}} &: x \mapsto (c \mapsto x) \\c_{\text{assign}} &: x \mapsto ((c, r) \mapsto x) \\c_{\text{finished}} &: x \mapsto ((x, r) \mapsto x) \\c_{\text{get_resource}} &: x \mapsto \kappa'x\end{aligned}$$

for arbitrary $c : \llbracket \text{Client} \rrbracket_{C,R}$ and $r : \llbracket \text{Resource} \rrbracket_{C,R}$. The only constructor in Σ_C is interpreted to $a : \{\} \mapsto \{*\}$.*

Deciphered, this interpretation describes a system behaving perfectly passive. No client will ever get a resource; no resource will every be associated to any client.

In CCSL the behaviour of a class is specified in a higher order logic equipped with modal operators. In this thesis I will furthermore introduce fixed-point operators. First we will introduce syntax and semantics of the underlying logic. Later we will show how to specify the behaviour in assertions and creation conditions.

5.1 Terms, Formulae and Types

Following the “formulae are terms” paradigm from higher order logics, we treat formulae and terms equally. Such that formulae are terms of the special type **Prop** embodying the type of truth values. In the following I give a context free grammar of CCSL terms.

5.1.1 Syntax of CCSL-Terms

Each CCSL term lives in a term variable context Γ over a type variable context Ξ . Such a term variable context is a finite list of distinct variable declarations $x : \tau$ such that τ is a type in context Ξ . A term is given by a term judgement of the form

$$\Xi \mid \Gamma \vdash t : \tau$$

All free (term) variables of t must be declared in Γ and all type variables occurring in t, τ and Γ must be declared in Ξ .

The following rules describe how terms and formulae are built over a coalgebraic class signature. Later we will add single-step modal operators and the modal fixed-point operators.

Definition 15 (Terms and Formulae in CCSL) *Let Σ be a coalgebraic class signature over a proper ground signature Ω . The set of terms over Σ is the least set containing*

- $x : \tau$ for a variable x and a type τ .
- $*$: $\mathbf{1}$ the only inhabitant of $\mathbf{1}$.
- \perp : **Prop** and \top : **Prop** representing the truth values true and false
- $f : \sigma$ for constants $f \in \Omega_\sigma$
- $m : \mathbf{Self} \times \sigma \rightarrow \rho$ for all method declarations $m \in \Sigma_M$
- $c : \sigma \rightarrow \mathbf{Self}$ for all constructor declarations $c \in \Sigma_C$
- $(t_1, t_2) : \sigma \times \tau$ for the binary product of $t : \sigma$ and $t_2 : \tau$
- $\pi_1 t : \sigma$ and $\pi_2 t : \tau$ for the projections for a term $t : \sigma \times \tau$.
- $\kappa_1 s : \sigma + \tau$ and $\kappa_2 s : \sigma + \tau$, the injections for terms $s : \sigma$ and $t : \tau$.
- $(\text{cases } t \text{ of } \kappa_1 x : r, \kappa_2 y : s) : \tau$, being the cases analyses for terms $t : \sigma_1 + \sigma_2$, $r : \tau$, and $s : \tau$ with x being free in r and y being free in s . x and y are bound in the complete cases expression.
- $\text{if } r \text{ then } s \text{ else } t : \tau$, the conditional for a term $r : \mathbf{Prop}$ and two terms s and t of the same type τ .
- $\lambda x : \sigma. t : \sigma \rightarrow \tau$, the lambda abstraction for a variable $x : \sigma$ and a term $t : \tau$.
- $t_1 t_2 : \tau$ is the application of a term $t_1 : \sigma \rightarrow \tau$ to a term $t_2 : \sigma$.
- $t_1 = t_2 : \mathbf{Prop}$ denotes the equality of two terms of the same type
- $t_1 \sim t_2 : \mathbf{Prop}$ is the behavioural equality of two terms of the same type
- $\neg t : \mathbf{Prop}$ being the negation of a term $t : \mathbf{Prop}$

- $t_1 \wedge t_2 : \mathbf{Prop}$ and $t_1 \vee t_2 : \mathbf{Prop}$ are the conjunction and disjunction of two terms of type \mathbf{Prop} , respectively
- $\forall x : \tau. t : \mathbf{Prop}$ is the universal quantification for a variable $x : \tau$ and a term $t : \mathbf{Prop}$.

Additionally, we will make use of the following abbreviations

Definition 16 (Abbreviations for Terms) Let t_1 and t_2 be terms and τ a type in context $\Xi \mid \Gamma$, then the following are abbreviating terms

- implication $t_1 \Rightarrow t_2 \stackrel{def}{=} \neg t_1 \vee t_2$
- logical equivalence $t_1 \Leftrightarrow t_2 \stackrel{def}{=} (t_1 \Rightarrow t_2) \wedge (t_2 \Rightarrow t_1)$
- the let binding let $x : \tau = t_1$ in $t_2 \stackrel{def}{=} (\lambda x : \tau. t_2) t_1$
- the existential quantification $\exists x : \tau. t \stackrel{def}{=} \neg \forall x : \tau. \neg t$

Figure 5.1 shows how to derive the typing of terms.

For defining the semantics of the logic, let $\Xi \mid \Gamma \vdash t : \tau$ be a term. Fix an interpretation U_1, \dots, U_n for the type parameters in Ξ and a set X interpreting \mathbf{Self} . Then, such a term t is mapped to a function mapping any instantiation of the variables in Γ to a value in τ , i.e. to a function

$$[[\sigma_1]] \times \dots \times [[\sigma_k]] \longrightarrow [[\tau]]$$

with $\Gamma : \{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$. Unfixing the interpretation of the type variables and \mathbf{Self} we get the following complete semantics of t

$$[[\sigma_1]]_{U_1, \dots, U_n}(X) \times \dots \times [[\sigma_k]]_{U_1, \dots, U_n}(X) \longrightarrow [[\tau]_{U_1, \dots, U_n}(X)]$$

with X, U_1, \dots, U_n being sets interpreting the type parameters.

If $t : \tau$ is a formula then $\tau = \mathbf{Prop}$ and thus τ is interpreted to the set of truth values and interpretation function returns true for exactly those elements of $[[\sigma_i]]$ fulfilling t . Equivalently, one might consider a formula $x : \sigma \vdash \phi : \mathbf{Prop}$ as a (collection of) predicate(S) $[[\phi]] \subseteq [[\sigma]]$.

Definition 17 (Semantics of CCSL-Terms) Let $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$ be a coalgebraic class signature and let M_Σ be a model for Σ . The interpretation of a term $\Xi \mid \Gamma \vdash t : \tau$ with $\Xi = \{\alpha_1, \dots, \alpha_n\}$ and $\Gamma = \{x_1 : \sigma_1, \dots, x_k : \sigma_k\}$ w.r.t. M_Σ is denoted by $[[t]]^{M_\Sigma}$, where I omit the superscript if obvious from the context. The interpretation $[[t]]$ is defined by induction on the structure of terms. Fix an interpretation U_1, \dots, U_n of the type-variables α_i and an interpretation X of \mathbf{Self} . Let $\bar{x} : \bar{\sigma}$ denote the tuple of arguments $x_1 : [[\sigma_1]], \dots, x_k : [[\sigma_k]]$.

- $[[x_i]] = \pi_i$

Figure 5.1: Derivation System for Terms over a coalgebraic class signature Σ and a ground signature Ω

$$\begin{array}{c}
\frac{\Xi \vdash \sigma : \text{Type}}{\Xi | \Gamma \cup (x : \sigma) \vdash x : \sigma} \textit{ground} \qquad \frac{\Xi \vdash \sigma : \text{Type}}{\Xi | \Gamma \vdash f : \sigma} \textit{ground} \text{ for } f \in \Omega_\sigma \\
\frac{\Xi \vdash \tau : \text{Type}}{\Xi | \Gamma \vdash m : \tau} \textit{ground} \text{ for } m : \tau \in \Sigma_M \qquad \frac{\Xi \vdash \tau : \text{Type}}{\Xi | \Gamma \vdash c : \tau} \textit{ground} \text{ for } c : \tau \in \Sigma_C \\
\frac{}{\Xi | \Gamma \vdash * : \mathbf{1}} \textit{ground} \\
\frac{}{\Xi | \Gamma \vdash \perp : \mathbf{Prop}} \textit{ground} \qquad \frac{}{\Xi | \Gamma \vdash \top : \mathbf{Prop}} \textit{ground} \\
\frac{\Xi | \Gamma \vdash s : \sigma \quad \Xi | \Gamma \vdash t : \tau}{\Xi | \Gamma \vdash (s, t) : \sigma \times \tau} \textit{product} \\
\frac{\Xi | \Gamma \vdash t : \sigma \times \tau}{\Xi | \Gamma \vdash \pi_1 t : \sigma} \textit{left projection} \qquad \frac{\Xi | \Gamma \vdash t : \sigma \times \tau}{\Xi | \Gamma \vdash \pi_2 t : \tau} \textit{right projection} \\
\frac{\Xi | \Gamma \vdash s : \sigma \quad \Xi \vdash \tau : \text{Type}}{\Xi | \Gamma \vdash \kappa_1 s : \sigma + \tau} \textit{left coprojection} \qquad \frac{\Xi \vdash \sigma : \text{Type} \quad \Xi | \Gamma \vdash t : \tau}{\Xi | \Gamma \vdash \kappa_2 t : \sigma + \tau} \textit{right coprojection} \\
\frac{\Xi | \Gamma \vdash t : \sigma_1 + \sigma_2 \quad \Xi | \Gamma, x : \sigma_1 \vdash r : \tau \quad \Xi | \Gamma, y : \sigma_2 \vdash s : \tau}{\Xi | \Gamma \vdash \textit{cases } t \textit{ of } \kappa_1 x : r \kappa_2 y : s : \tau} \textit{cases} \text{ for } x \notin \Gamma, y \notin \Gamma \\
\frac{\Xi | \Gamma \vdash r : \mathbf{Prop} \quad \Xi | \Gamma \vdash s : \tau \quad \Xi | \Gamma \vdash t : \tau}{\Xi | \Gamma \vdash \textit{if } r \textit{ then } s \textit{ else } t : \tau} \textit{conditional} \\
\frac{\Xi \vdash \sigma : \text{Type} \quad \Xi | \Gamma, x : \sigma \vdash t : \tau}{\Xi | \Gamma \vdash \lambda x : \sigma, t : \sigma \rightarrow \tau} \textit{abstraction} \\
\frac{\Xi | \Gamma \vdash t : \sigma \rightarrow \tau \quad \Xi | \Gamma \vdash s : \sigma}{\Xi | \Gamma \vdash t s : \tau} \textit{application} \\
\frac{\Xi | \Gamma \vdash s : \tau \quad \Xi | \Gamma \vdash t : \tau}{\Xi | \Gamma \vdash s = t : \mathbf{Prop}} \textit{equality} \\
\frac{\Xi | \Gamma \vdash s : \tau \quad \Xi | \Gamma \vdash t : \tau}{\Xi | \Gamma \vdash s \sim t : \mathbf{Prop}} \textit{equality} \\
\frac{\Xi | \Gamma \vdash s : \mathbf{Prop} \quad \Xi | \Gamma \vdash t : \mathbf{Prop}}{\Xi | \Gamma \vdash s \wedge t : \mathbf{Prop}} \textit{conjunction} \\
\frac{\Xi | \Gamma \vdash s : \mathbf{Prop} \quad \Xi | \Gamma \vdash t : \mathbf{Prop}}{\Xi | \Gamma \vdash s \vee t : \mathbf{Prop}} \textit{disjunction} \\
\frac{\Xi | \Gamma \vdash t : \mathbf{Prop}}{\Xi | \Gamma \vdash \neg t : \mathbf{Prop}} \textit{negation}
\end{array}$$

Figure 5.2: Derivation System for Terms over a coalgebraic class signature Σ and a ground signature Ω

$$\frac{\Xi \vdash \tau : \text{Type} \quad \Xi | \Gamma \vdash t : \mathbf{Prop}}{\Xi | \Gamma \vdash \forall x : \tau. t : \mathbf{Prop}} \text{ universal quantification for } x \notin \Gamma$$

$$\frac{\Xi | \Gamma \vdash t : \tau}{\Xi, \alpha : \text{Type} | \Gamma \vdash t : \tau} \text{ weakening for } \alpha \notin \Gamma$$

$$\frac{\Xi \vdash \sigma : \text{Type} \quad \Xi | \Gamma \vdash t : \tau}{\Xi | \Gamma, x : \sigma \vdash t : \tau} \text{ weakening}$$

$$\frac{\Xi \vdash \sigma : \text{Type} \quad \Xi | \Gamma \vdash t : \tau}{\Xi | \Gamma[\sigma/\alpha] \vdash t[\sigma/\alpha] : \tau[\sigma/\alpha]} \text{ type substitution for } \alpha \notin \Xi$$

$$\frac{\Xi | \Gamma \vdash s : \sigma \quad \Xi | \Gamma, x : \sigma \vdash t : \tau}{\Xi | \Gamma \vdash t[s/t] : \tau} \text{ term substitution}$$

- $[[m : \mathbf{Self} \times \sigma' \rightarrow \tau]] = \lambda \bar{x} : \bar{\sigma}. (\lambda x : X, p : [[\sigma']] . \pi_m(cx)(p))$
- $[[c]] = \lambda \bar{x} : \bar{\sigma}. \kappa_c$
- $[[*]] = \lambda \bar{x} : \bar{\sigma}. *$
- $[[\perp]] = \lambda \bar{x} : \bar{\sigma}. \perp$
- $[[\top]] = \lambda \bar{x} : \bar{\sigma}. \top$
- $[[\langle t_1, t_2 \rangle]] = \langle [[t_1]], [[t_2]] \rangle$
- $[[\pi_1 t]] = \pi_1 \circ [[t]]$
- $[[\pi_2 t]] = \pi_2 \circ [[t]]$
- $[[\kappa_1 t]] = \kappa_1 \circ [[t]]$
- $[[\kappa_2 t]] = \kappa_2 \circ [[t]]$
- $[[\text{cases } t \text{ of } \kappa_1 x : r, \kappa_2 y : s]] = \lambda \bar{x} : \bar{\sigma}. \begin{cases} [[r]](\bar{x}, z_1) & \text{if } [[t]]\bar{x} = \kappa_1 z_1 \\ [[r]](\bar{x}, z_2) & \text{if } [[t]]\bar{x} = \kappa_2 z_2 \end{cases}$
- $[[\text{if } r \text{ then } s \text{ else } t]] = \lambda \bar{x} : \bar{\sigma}. \begin{cases} [[s]](\bar{x}) & \text{if } [[r]]\bar{x} = \top \\ [[t]](\bar{x}) & \text{if } [[r]]\bar{x} = \perp \end{cases}$
- $[[\lambda x : \rho. t]] = \lambda \bar{x} : \bar{\sigma}. (\lambda y : [[\rho]]. [[t]](\bar{x}, y))$
- $[[t_1 t_2]] = \lambda \bar{x} : \bar{\sigma}. [[t_1]](\bar{x})([[t_2]]\bar{x})$
- $[[t_1 = t_2]] = \lambda \bar{x} : \bar{\sigma}. [[t_1]]\bar{x} = [[t_2]]\bar{x}$
- $[[t_1 \sim t_2]] = \lambda \bar{x} : \bar{\sigma}. \text{Rel}([[\rho]]) \overset{\leftrightarrow}{(-)}_{M_\Sigma} ([[t_1]]\bar{x}, [[t_2]]\bar{x})$ for $(t_1 \text{ and } t_2 \text{ of type } \rho)$
- $[[\neg t]] = \lambda \bar{x} : \bar{\sigma}. \neg [[t]]\bar{x}$
- $[[t_1 \wedge t_2]] = \lambda \bar{x} : \bar{\sigma}. [[t_1]]\bar{x} \wedge [[t_2]]\bar{x}$

- $[[t_1 \vee t_2]] = \lambda \bar{x} : \bar{\sigma}. [[t_1]]\bar{x} \vee [[t_2]]\bar{x}$
- $[[\forall x : \tau. t]] = \lambda \bar{x} : \bar{\sigma}. \begin{cases} \top & \text{if } [[t]](\bar{x}, y) = \top \text{ for all } y \in [[\tau]] \\ \perp & \text{otherwise} \end{cases}$

5.1.2 Syntactic Sugar

The user of our specification language may want to use arbitrary, but finite products, and thereby finds an iterated construction of binary products inconvenient. Therefore, we define the following abbreviation.

For terms $t_1, t_2, \dots, t_{n-1}, t_n$ the arbitrary (but finite) product $\langle t_1, t_2, \dots, t_n \rangle$ abbreviates the successive application $\langle t_1, \langle t_2, \dots \langle t_{n-1}, t_n \rangle \dots \rangle$ of the binary product. The corresponding projections are abbreviated by π_1, \dots, π_n . Note that even the binary product is an arbitrary one. Hence the projections in a binary product may be denoted with π_1 and π_2 as well.

Usually, when working with abstract data types, one uses recognizer predicates. The following definition shows, how recognizers can be expressed within our logic.

Assume a data type dt with n constructors $cons_1, \dots, cons_n$, variable identifiers $x_{1,1}, \dots, x_{1,k(1)}, \dots, x_{n,1}, \dots, x_{n,k(n)}$ and a term $t : dt$ in context Γ . Then for each $i \leq n$, $cons_i?(t)$ is a term of type $bool$ in context Γ and abbreviates

$$cons_i?(t) = \text{cases } t \text{ of } \begin{cases} cons_1(x_{1,1}, \dots, x_{1,k(1)}) : \perp \\ \vdots \\ cons_{i-1}(x_{i-1,1}, \dots, x_{i-1,k(i-1)}) : \perp \\ cons_i(x_{i,1}, \dots, x_{i,k(i)}) : \top \\ cons_{i+1}(x_{i+1,1}, \dots, x_{i+1,k(i+1)}) : \perp \\ \vdots \\ cons_n(x_{n,1}, \dots, x_{n,k(n)}) : \perp \end{cases}$$

with $k(i)$ being the arity of the i -th constructor $cons_i$.

Intuitively, the recognizers describe which constructor a given term t has been constructed with.

5.1.3 Class Specifications

In the following we will see how to restrict the class of models for a class signature by stating properties in terms as given above.

Definition 18 (Class Specification) *A class specification \mathcal{S} is a triple $\langle \Sigma, A_M, A_C \rangle$ consisting of*

1. a coalgebraic class signature Σ with type parameters $\Xi = \alpha_1, \dots, \alpha_n$.
2. a set A_M of method assertions $\Xi \mid x : \mathbf{Self} \vdash \phi$ with at most one free variable, namely x
3. a set A_C of creation conditions A_C of the form $\Xi \mid \emptyset \vdash \psi$

for ϕ and ψ being formulae.

The following example demonstrates the use of assertions

Example 19 (Scheduler - continued) *When a new Scheduler is instantiated, it should reasonably be a resource available. We specify this property in the following class specification for the coalgebraic class signature $\Sigma_{\text{Scheduler}}$ as given in example 10 (page 14).*

$$\text{Scheduler} = \langle \Sigma_{\text{Scheduler}}, \{\dots\}, \{F_{\text{new}}\} \rangle$$

with some method assertions ... and the creation condition F_{new}

$$F_{\text{new}} = \{up?(get_resource(new_scheduler))\}$$

Analogously to the subsignatures for coalgebraic class signatures, one can define the notion of a subspecification for class specifications. Then a class specification $\mathcal{S}' = \langle \Sigma', A'_M, A'_C \rangle$ is a subspecification of $\mathcal{S} = \langle \Sigma, A_M, A_C \rangle$ if Σ' is a subsignature of Σ and $A'_M \subseteq A_M$.

5.2 Semantics of Class Specifications

In section 4.1 we gave a semantics to class signatures by describing their models. In this section we will proceed in a similar manner, i.e. we describe the semantics of class specifications in terms of their models. The models of class specifications shall be models for the underlying class signatures, too. Thereby, assertions impose restrictions on those signature models.

Models of a Class Specification

Method assertions and creation conditions in a class specification establishes a predicate on the state space of the according class. To determine whether a model of a class signature behaves properly in the sense of specified assertions, one has to check whether each element of the state space together with the interpretations of the methods fulfills the assertions. The following definition captures this idea formally :

Definition 20 (Models of a Class Specification) *Let $\mathcal{S} = \langle \Sigma, A_M, A_C \rangle$ be a class specification. A model $M = (\langle X, c, a \rangle_{U_1, \dots, U_n})$ of \mathcal{S} is a model of the underlying class signature Σ , such that for all interpretations U_1, \dots, U_n of the n parameter types in \mathcal{P} the coalgebra $c : X \rightarrow \llbracket \sigma_M \rrbracket_{U_1, \dots, U_n}(X)$ and the algebra $a : \llbracket \sigma_C \rrbracket_{U_1, \dots, U_n}(X) \rightarrow X$ of this model satisfy the following conditions.*

- *For all $x \in X$, $\llbracket A_M \rrbracket_{v:\mathbf{Self}}(x)$. This means that every possible state x from the state space X has to fulfill the behaviour assertion. Since A_M contains at most one variable of type **Self**, namely v . The type context of A_M contains exactly this variable. Then $\llbracket A_M \rrbracket_{v:\mathbf{Self}}$ is a function mapping an element of X , i.e. a state, to the a truth value.*

- Analogously $\llbracket A_C \rrbracket_{alg:\Sigma_C \rightarrow X}$ is functions mapping algebras a to truth values describing whether a fulfills the property given as A_C . alg is the only variable in the variable context of A_C .

Furthermore, for every class type C among the constant types, $\llbracket C \rrbracket$ must be a model of the specification of the respective class.

Chapter 6

Introducing single-step Modal Operators

As it has been argued in [Rut97] and [Rut96] a coalgebra determines a general type labelled transition system (LTS). Thus it appears natural to use constructions from Modal Logics in reasoning about Coalgebras. In ([Rot00]) Rothe introduced the infinitary modal operators into CCSL. Although their use has been shown, e.g. for specifying safety properties, they are too coarse when being applied with μ -calculus operators.

In modal logics the formula $\Box\phi$ is true for those states such that all direct successor states thereof satisfy ϕ . Respectively, $\Diamond\phi$ separates those states such that there is a direct successor state satisfying ϕ . In multi-modal logics those two modal operators are defined indexed by a set of actions Δ . So, does for instance $\Box^{\Delta}\phi$ describe those states that all successor states reachable in one step by any action $a \in \Delta$; similar for \Diamond .

However, it is not easy to determine such actions in a T -coalgebra for a signature model $M = \langle X, c, a \rangle$. A single action shall result in precisely one state from X . However, as we have seen in the preliminaries a T -coalgebra maps a X to TX , i.e. a set of complex structures. Therefore it is necessary to define a selector function to decompose those structures. We will follow Rothe in using paths through such polynomial functors T .

Definition 21 (Paths through polynomial Functors) *For a polynomial functor T the set P_T of valid paths through T is defined as follows*

- $P_T = 1 = \{*\}$ if T is the identity or constant functor
- $P_T = \hat{\kappa}_1 P_{T_1} \cup \hat{\kappa}_2 P_{T_2}$ if T is the (co-)product of two functors T_1 and T_2
- $P_T = A \times P_{T_1}$ if T is an exponent $T(X) = T_1(X)^A$ for A being a set independent from X

$\hat{\kappa}_1$ is the natural extension of κ_1 to sets, i.e. $\hat{\kappa}_1(X) = \{\kappa_1(x) \mid x \in X\}$.

Intuitively, these injections represent pointers describing the position of the actual successor states from X inside the complex structures TX . Beside those injections paths may contain products if the regarded functor contains exponents. In this case a path also respects the dependency of the successor state on the arguments handed over to a method for instance.

Using this path construction we can define a successor function taking a coalgebra and a path to return the concrete successor state.

Definition 22 (Successor Function) *For a set X , T a functor and P_T its set of paths we define the successor function $succ_T : T(X) \times P_T \rightarrow X + 1$ as follows*

- $succ_T(x, *) = \kappa_1 x$ if $T(X) = X$
- $succ_T(x, *) = \kappa_2 *$ if T is a constant functor
- If T is a product, i.e. $T(X) = T_1(X) + T_2(X)$ then

$$\begin{aligned} succ_T(x, \kappa_1 p) &= succ_{T_1}(\pi x, p) \\ succ_T(x, \kappa_2 p) &= succ_{T_2}(\pi' x, p) \end{aligned}$$

- If T is a coproduct $T(X) = T_1(X) + T_2(X)$ then

$$\begin{aligned} succ_T(\kappa_1 x, \kappa_1 p) &= succ_{T_1}(x, p) \\ succ_T(\kappa_2 x, \kappa_2 p) &= succ_{T_2}(x, p) \end{aligned}$$

In case the injections do not match, we define

$$succ_T(\kappa_1 x, \kappa_2 p) = succ_T(\kappa_2 x, \kappa_1 p) = \kappa_2 *$$

- If T is an exponent $T(X) = A \rightarrow T_1(X)$ then

$$succ_T(x, (a, p)) = succ_{T_1}(x(a), p)$$

Note that $succ_T$ can be extended in the natural way to return sets of successor states for sets of paths through T . We define this extension $\hat{succ}_T : T(X) \times 2^{P_T} \rightarrow 2^X$ of $succ_T$ as

$$\hat{succ}_T(x, P) = \{x' \mid \exists p \in P. \kappa_1 x' = succ_T(x, p)\}$$

The following examples shall demonstrate the previous definitions

Example 23 (Scheduler - continued) Remember from example 10 (page 14) the class signature of our Scheduler and its polynomial functor

$$T(X) = \begin{array}{ll} (C \rightarrow X) \times & (\text{request}) \\ ((C \times R) \rightarrow X) \times & (\text{assign}) \\ ((C \times R) \rightarrow X) \times & (\text{finished}) \\ (R + 1) & (\text{get_resource}) \end{array}$$

with the set P_T of paths through T given as

$$P_T = \left\{ \begin{array}{ll} \hat{\kappa}_1(C \times \{*\}) \cup & (\text{request}) \\ \hat{\kappa}_2((C \times R) \times \{*\}) \cup & (\text{assign}) \\ \hat{\kappa}_3((C \times R) \times \{*\}) \cup & (\text{finished}) \\ \{\kappa_4*, \kappa_5*\} & (\text{get_resource}) \end{array} \right\}$$

for sets C and R interpreting the instantiated type parameters *Client* and *Resource*, respectively.

Note that in the above Definition 21 on paths through polynomial functors we implicitly assumed (co-)products to be in binary form. However, in the preliminaries we have seen a general form of arbitrary but finite (co-)products. In a similar way one can define paths for arbitrary but finite (co-)product functors. In this example we implicitly assume such a definition.

Then assume we want to select the successor state for the method “finished”. So, we only have to determine the path to the method which is

$$\hat{\kappa}_3((C \times R) \times \{*\})$$

Note that we have not only selected a single successor state, rather a set of successor states each depending on arguments from C and R handed over to the method “finished”.

Now, that we have a means to select a distinct successor state by its path, we can define modal operators quantifying over the set of such selected successor states.

Definition 24 (Syntax of Modal Operators - extends Definition 15)

Let $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$ be a coalgebraic class signature then for a predicate $\phi : \mathbf{Self} \rightarrow \mathbf{Prop}$ the following formulae represent the single-step modal operators in CCSL

- $\Box^P \phi : \mathbf{Self} \rightarrow \mathbf{Prop}$
- $\Diamond^P \phi : \mathbf{Self} \rightarrow \mathbf{Prop}$

with P being a set of paths, i.e. a subset $P \subseteq P_T$ for $T = [[\sigma_M]]_{U_1, \dots, U_m}$ being the polynomial functor describing Σ_M .

Additionally, we give the following two rules for deriving the modal operators for a coalgebraic class signature Σ and T describing Σ_M . These two rules extend figure 5.2.

$$\frac{\Xi|\Gamma \vdash \phi : \mathbf{Self} \rightarrow \mathbf{Prop}}{\Xi|\Gamma \vdash \square^P \phi : \mathbf{Self} \rightarrow \mathbf{Prop}} \text{ box}$$

$$\frac{\Xi|\Gamma \vdash \phi : \mathbf{Self} \rightarrow \mathbf{Prop}}{\Xi|\Gamma \vdash \diamond^P \phi : \mathbf{Self} \rightarrow \mathbf{Prop}} \text{ diamond}$$

with P being a set of valid paths through T .

In [Rot00] and [Tew02a] the infinitary box and diamond operators are denoted with \square and \diamond respectively. Nonetheless I will stay with the notation common in modal logics in that I denote the single-step modal operators with \square and \diamond . To discriminate the infinitary modal operators from the single-step ones I suggest to denote the infinitary modal operators by the resp. black symbols \blacksquare and \blacklozenge .

Note that the \diamond -operator is only an abbreviation in the sense that $\diamond^P \phi \stackrel{def}{=} \neg \square^P \neg \phi$, such that we will consider the semantics for the \square -operator only.

Definition 25 (Semantics of \square) *Let $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$ be a coalgebraic class signature and let $M = \langle X, c, a \rangle$ be a model thereof. Let T be the polynomial functor $[[\Sigma_M]]_{U_1, \dots, U_n}$. Furthermore, let us fix contexts $\Xi = \alpha_1, \dots, \alpha_n$ and $\Gamma = x_1 : \sigma_1, \dots, x_k : \sigma_k$, and an interpretation U_1, \dots, U_n for the type parameters. Then*

$$[[\square^P \phi]] = \lambda x : \mathbf{Self}. \lambda \bar{x} : \bar{\sigma}. \forall x' \in X. (x' \in \text{succ}(cx, P) \Rightarrow [[\phi]](\bar{x})(x'))$$

The previous definition just formalizes our intuition that $[[\square^P \phi]]$ is true exactly for those states x such that all successor states x' reachable from x via paths in P satisfy ϕ .

In this chapter we have extended the CCSL logic by single-step modal operators. However, their introduction should not be central in this thesis. Their purpose is to facilitate the the application of the modal fixed-point operators that we will introduce in the next chapter. Therefore, I refer for examples for the application of the single-step modal operators to the next chapter.

Chapter 7

Introducing modal Fixed-Point Operators

In software specification it is often not enough to have static predicates like the property $\diamond\top$. Rather, the user might wish to define recursive properties like the safety-property $P = \phi \wedge \Box P$. However, the semantics of such a property P may not be clear. Is P true for all finite runs containing only states satisfying P or is it even true for an infinite run with the invariant P ?

In order to solve this question, one may regard such a recursive predicate P as a predicate transformer F mapping $P \mapsto \phi \wedge \Box P$. The set of states satisfying the equation $P = \phi \wedge \Box P$ is given in the fixed-point of the predicate transformer.

In most cases such a F has more than one fixed-point. In standard μ -calculus one usually regards the least and greatest fixed points, only. In this thesis we will follow this rule in defining the μ -calculus operators μ and ν representing the least and greatest fixed-points, respectively.

However, neither the least nor the greatest fixed-point may exist. Knaster and Tarski have shown that this can only happen if the predicate transformer under consideration is not monotonic. In this thesis I circumvent any exceptions by explicitly requiring the predicate transformers to be monotonic. It is, however, not trivial to show monotonicity. In the following I will preprend a short discussion thereof.

7.1 Discussion of Monotonicity

A predicate transformer $f : 2^X \rightarrow 2^X$ is monotonic iff for subsets X_1 and X_2 of X the subset relation $X_1 \subseteq X_2$ implies $f(X_1) \subseteq f(X_2)$. In this chapter we mainly regard transformers of predicates over the state space **Self**, i.e. of predicates of type $(\mathbf{Prop} \rightarrow \mathbf{Self})$.

Therefore, each such predicate has the form $\lambda(x : \mathbf{Self}).t : \mathbf{Prop}$. Then we say a predicate $P : (\mathbf{Self} \rightarrow \mathbf{Prop})$ is contained in a predicate $Q : (\mathbf{Self} \rightarrow \mathbf{Prop})$ iff for each x of type **Self** it holds that $P(x) \Rightarrow Q(x)$. Then a predicate transformer $F : (\mathbf{Self} \rightarrow \mathbf{Prop}) \rightarrow (\mathbf{Self} \rightarrow \mathbf{Prop})$ is monotonic iff $\forall(x : \mathbf{Self}).(P(x) \Rightarrow Q(x))$ implies $\forall(x : \mathbf{Self}).(FP(x) \Rightarrow FQ(x))$.

As it has been argued in literature there are in general two ways to ensure monotonicity of such a predicate transformer F . A semantical approach consists in adding the above conditions as a constraint to the specification. In a syntactic approach one may use a criterion to syntactically discriminate such a predicate transformer as monotonic or not.

In CCSL the validity of CCSL formulae for models of class signatures is computed by a theorem prover. The semantic approach adds the monotonicity constraints to the set of formulae that have to be proved. Those monotonicity constraints are of a rather simple structure and can thus be proved mechanically in most cases.

To find a syntactic criterion of monotonicity is not that trivial if one considers that the logic of CCSL contains higher order constructions like λ -abstraction and -application. In the following I will adapt a syntactic monotonicity criterion from the first order logic μ FO enriched by standard μ -calculus operators as it has been described by Leivant in [Lei94].

This criterion requires formulae to be in a certain normalform, namely in negation normal form requiring the formulae to be β normalform. In the following we will prepend the introduction of both normalforms. However, therefor syntactic abbreviations are assumed to be unfolded, i.e.

- if-condition
- \Rightarrow and \Leftarrow
- let-binding
- arbitrary (co-)products

The β normalform of formulae in the lambda calculus has already been introduced in [Bar92]. There, a formula is turned into its β normalform by exhaustively β -reducing redexes of the form $(\lambda x : \sigma.\psi : \tau)(y : \sigma)$ to $\psi[y/x]$.

For CCSL we define the β -Normalform in a similar manner. Therefor assume a CCSL term t , then its β normalform $[t]^\beta$ can be computed by exhaustively applying the following rewrite rule throughout t

$$(\lambda x : \sigma.t')s \longrightarrow_\beta t'[s/x]$$

for s being of type σ . Thereby s is assumed not to contain free occurrences of variables bound by quantifiers or lambda abstraction in t' .

It is worth noting that such a normalform does exist for any CCSL formula. By knowing that CCSL underlies a strong type system, confluence of the above rewrite rule is guaranteed, and thus also uniqueness of β normalforms if they exist. The proof of existence of the β normalform of CCSL terms reduces in the well-known manner to termination of the term rewriting system induced by the above rule.

Assuming a term $[t]^\beta$ in beta normalform it is common to assume negation to occur only in front of atomic formulae. The negation normalform $[t]^\neg$ of $[t]^\beta$ can be computed by pushing the negations inwards according to the following schema

- $\neg\neg\phi \longrightarrow_{\neg} \phi$
- $\neg\top \longrightarrow_{\neg} \perp$ and $\neg\perp \longrightarrow_{\neg} \top$
- $\neg(\phi \vee \psi) \longrightarrow_{\neg} \neg\phi \wedge \neg\psi$ and $\neg(\phi \wedge \psi) \longrightarrow_{\neg} \neg\phi \vee \neg\psi$
- $\neg\forall s : \sigma.\psi \longrightarrow_{\neg} \exists s : \sigma.\neg\psi$ and $\neg\exists s : \sigma.\psi \longrightarrow_{\neg} \forall s : \sigma.\neg\psi$
- $\neg\Box^P\phi \longrightarrow_{\neg} \Diamond^P\neg\phi$ and $\neg\Diamond^P\phi \longrightarrow_{\neg} \Box^P\neg\phi$
- $\neg\mu F \longrightarrow_{\neg} \nu F[\neg Z/Z]$ and $\neg\nu F \longrightarrow_{\neg} \mu F[\neg Z/Z]$

It is an easy matter to see that the above rules induce a strongly terminating term rewrite system. Thereby it can be concluded that a negation normalform exists for each term (in β normalform). Furthermore, those rules do not give rise to critical overlaps jeopardizing confluence. Hence, we can also conclude uniqueness of negation normalforms.

Definition 26 (Syntactic Monotonicity Criterion) *Consider a coalgebraic class signature $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$ such that $T = [\![\sigma_M]\!]_{U_1, \dots, U_n}$ is the polynomial functor describing the types in Σ_M for the interpretation U_1, \dots, U_n of the type parameters. Let furthermore be $F = \lambda(Z : \mathbf{Self} \rightarrow \mathbf{Prop}).\lambda(x : \mathbf{Self}).\phi$ a predicate transformer with ϕ being a CCSL formula in negation normalform containing free variables Z and x . Then, ϕ satisfies the syntactic Monotonicity Criterion for Z iff*

1. ϕ does not contain any occurrence of Z
2. ϕ has the form Zx' for x' being a variable of type \mathbf{Self}
3. ϕ has the form $\phi_1 \vee \phi_2$ or $\phi_1 \wedge \phi_2$ and ϕ_1 and ϕ_2 satisfy the syntactic monotonicity criterion
4. ϕ has the form $\forall(v : \tau).\phi_1$ or $\exists(v : \tau).\phi_1$ and ϕ_1 satisfies the syntactic monotonicity criterion
5. ϕ has the form $\Box^P(\lambda(x : \mathbf{Self}).\phi_1)$ or $\Diamond^P(\lambda(x : \mathbf{Self}).\phi_1)$ for valid paths P not containing any occurrence of Z
6. ϕ has the form $\mu(\lambda(Z : \mathbf{Self}' \rightarrow \mathbf{Prop}).(\lambda(x : \mathbf{Self}).\phi_1))$ or $\nu(\lambda(Z : \mathbf{Self}' \rightarrow \mathbf{Prop}).(\lambda(x : \mathbf{Self}).\phi_1))$ and ϕ_1 satisfies the syntactic monotonicity criterion for Z and Z'

F is monotonic if ϕ satisfies the syntactic monotonicity criterion.

The latter proposition can be proved by induction over the structure of ϕ . The proof resembles the argument for the logic μFO that we have derived the criterion from. However, CCSL's notions for the single-step modal operators and modal fixed-point operators differs slightly from those in μFO . Therefore, I will carry out the proof in the following.

For the first base case assume ϕ does not contain any occurrence of Z then ϕ is constant and trivially follows $\phi[Z^1/Z] \Rightarrow \phi[Z^2/Z]$. For the second base case, when ϕ has the form Zx' for x' being a variable of type **Self**, $Z^1 \subseteq Z^2$ implies $Z^1x' \Rightarrow Z^2x'$.

For the induction assume that for a formula ϕ_i with $i \in \{1, 2\}$ it is known that $Z^1 \subseteq Z^2$ implies $\phi_i[Z^1/Z] \Rightarrow \phi_i[Z^2/Z]$. Obviously it follows that $Z^1 \subseteq Z^2$ implies both $\phi_1[Z^1/Z] \vee \phi_2[Z^1/Z] \Rightarrow \phi_1[Z^2/Z] \vee \phi_2[Z^2/Z]$ and $\phi_1[Z^1/Z] \wedge \phi_2[Z^1/Z] \Rightarrow \phi_1[Z^2/Z] \wedge \phi_2[Z^2/Z]$.

It furthermore easy to see that $Z^1 \subseteq Z^2$ implies $\forall(a : \tau).\phi_1[Z^1/Z] \Rightarrow \forall(a : \tau).\phi_1[Z^2/Z]$ by case distinction on whether a occurs in ϕ_1 or not. In the first case a , as a free variable in ϕ_1 , is implicitly universally quantified. In the second case quantifying over a does not change the semantics of the formula. A similar arguments holds for existential quantification as well.

As indicated at the beginning of this proof the notions for the modal operators is slightly different in CCSL from those in μFO in that \Box and \Diamond contain explicitly a variable for the next state like in $\Box^P(\lambda(x' : \mathbf{Self}).\phi_i)$. It is left to show that $\phi_1 \Rightarrow \phi_2$ implies $\Box^P(\lambda(x' : \mathbf{Self}).\phi_1) \Rightarrow \Box^P(\lambda(x' : \mathbf{Self}).\phi_2)$. This implication, however, follows immediately from the definition of \Box . The same follows for \Diamond .

The proof that $\phi_1 \Rightarrow \phi_2$ implies both $\mu(\lambda(Z' : \mathbf{Self} \rightarrow \mathbf{Prop}).(\lambda(x' : \mathbf{Self}).\phi_1)) \Rightarrow \mu(\lambda(Z' : \mathbf{Self} \rightarrow \mathbf{Prop}).(\lambda(x' : \mathbf{Self}).\phi_2))$ and $\nu(\lambda(Z' : \mathbf{Self} \rightarrow \mathbf{Prop}).(\lambda(x' : \mathbf{Self}).\phi_1)) \Rightarrow \nu(\lambda(Z' : \mathbf{Self} \rightarrow \mathbf{Prop}).(\lambda(x' : \mathbf{Self}).\phi_2))$ exploits that the least and greatest fixed-points can be computed by unfolding the predicate transformer. The exact definition of μ and ν by unfolding can be adopted from standard μ -calculus as given in [Sti92]. Therefrom we obtain for μ . $\mu(\lambda(Z' : \mathbf{Self} \rightarrow \mathbf{Prop}).(\lambda(x' : \mathbf{Self}).\phi_i)) = \perp \vee \phi_1[x'/x][\perp/Z'] \vee \phi_1[x'/x][\phi_1[x'/x][\perp/Z']/Z'] \vee \dots$. It is obvious that the obtained formula is monotonic if ϕ_1 satisfies the syntactic monotonicity criterion for Z and Z' .

We leave out the infinitary modal operators in this proof, because they can be represented by the modal fixed-point and the single-step modal operators without negation as we will see at the end of this chapter. Thereby the infinitary modal operators trivially preserve monotonicity.

As I have indicated previously, the syntactic monotonicity criterion is not complete in that sense, that there may be monotonic formula transformer classified as non-monotonic. As a simple example consider F with $FZ = \lambda(x : \mathbf{Self}).\neg Z(x) \vee Z(x)$. The left disjunct marks F as not monotonic, although it obviously is. The least and greatest fixed-point of F are both $\lambda(x : \mathbf{Self}).\top$. One may argue that formula transformer of that kind can be further simplified using rules from propositional logic. In general, simplifications of that kind involve semantic evaluation that we intend to avoid in this context.

7.2 Defining the modal Fixed-Point Operators

We define μ and ν for a predicate transformer $F : (\mathbf{Self} \rightarrow \mathbf{Prop}) \rightarrow (\mathbf{Self} \rightarrow \mathbf{Prop})$ on sets of states as follows.

Definition 27 (The modal Fixed Point Operators - extending Definitions 15 and 24)

Let Σ be a coalgebraic class signature and F a monotonic formula transformer defined over Σ , then the following operators determine the least and greatest fixed points of F , respectively.

- $\mu : ((\mathbf{Self} \rightarrow \mathbf{Prop}) \rightarrow (\mathbf{Self} \rightarrow \mathbf{Prop})) \rightarrow (\mathbf{Self} \rightarrow \mathbf{Prop})$
- $\nu : ((\mathbf{Self} \rightarrow \mathbf{Prop}) \rightarrow (\mathbf{Self} \rightarrow \mathbf{Prop})) \rightarrow (\mathbf{Self} \rightarrow \mathbf{Prop})$

The derivation rules for the modal fixed point operators are given subsequently to extend figure 5.2

$$\frac{\exists|\Gamma \vdash F : (\mathbf{Self} \rightarrow \mathbf{Prop}) \rightarrow (\mathbf{Self} \rightarrow \mathbf{Self})}{\exists|\Gamma \vdash \mu F : \mathbf{Self} \rightarrow \mathbf{Prop}} \text{ mu}$$

$$\frac{\exists|\Gamma \vdash F : (\mathbf{Self} \rightarrow \mathbf{Prop}) \rightarrow (\mathbf{Self} \rightarrow \mathbf{Prop})}{\exists|\Gamma \vdash \nu F : \mathbf{Self} \rightarrow \mathbf{Prop}} \text{ nu}$$

In both rules F is implicitly assumed to satisfy the syntactic or semantic monotonicity criterion.

Note that μ and ν take a predicate transformer over \mathbf{Self} and map it to a predicate on the same carrier \mathbf{Self} being the least or greatest fixed point of the predicate transformer.

For readability we write $\mu(Z : (\mathbf{Self} \rightarrow \mathbf{Prop})).\phi$ instead of $\mu(\lambda(Z : (\mathbf{Self} \rightarrow \mathbf{Prop})).\phi)$ and $\nu(Z : (\mathbf{Self} \rightarrow \mathbf{Prop})).\phi$ instead of $\nu(\lambda(Z : (\mathbf{Self} \rightarrow \mathbf{Prop})).\phi)$ for ϕ being a predicate over \mathbf{Self} with Z occurring freely. Furthermore, we consider ν to be defined by the abbreviation $\nu(Z : (\mathbf{Self} \rightarrow \mathbf{Prop})).F[Z] = \neg\mu(Z : (\mathbf{Self} \rightarrow \mathbf{Prop})).(\neg F[\neg Z/Z])$.

The semantics of μ is defined in terms of the respective set-theoretic least fixed-point operator lfp introduced in the preliminary section.

Definition 28 (The Semantics of μ - extending Definition 17) Let $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$ be a coalgebraic class signature and let $\mathcal{M} = \langle X, c, a \rangle$ be a model for Σ . We fix an interpretation X of \mathbf{Self} . Let $\bar{x} : \bar{\sigma}$ denote the tuple of arguments $x_1 : [\sigma_1], \dots, x_k : [\sigma_k]$. Then we define the semantics of μ as follows

$$[[\mu(\lambda(Z : \mathbf{Self} \rightarrow \mathbf{Prop}).F)]]_{U_1, \dots, U_n} = \lambda \bar{x} : \bar{\sigma}. \text{lfp}(\lambda Z : X \rightarrow 2. [[F]]_{U_1, \dots, U_n}(\bar{x}.Z))$$

if $\lambda Z. [[F]]_{U_1, \dots, U_n}(\bar{x}.Z)$ is monotonic.

7.3 Examples and Applications

The following examples show how the well-known temporal connectives can be represented by the single-step modal operators and the modal fixed point operators.

Example 29 (Temporal Connectives) *Consider predicates $\phi : (\mathbf{Self} \rightarrow \mathbf{Prop})$ and $\psi : (\mathbf{Self} \rightarrow \mathbf{Prop})$, then the following are abbreviations for the well-known temporal connectives from modal logics.*

- $AG.\phi = \nu(Z : (\mathbf{Self} \rightarrow \mathbf{Prop})).(\lambda(x : \mathbf{Self}).\phi(x) \wedge (\Box Z)(x))$ *Every run does only contain states satisfying ϕ .*
- $AF.\phi = \mu(Z : (\mathbf{Self} \rightarrow \mathbf{Prop})).(\lambda(x : \mathbf{Self}).\phi(x) \vee ((\Diamond \top)(x) \wedge \Box Z)(x))$ *In every finite run there is a state satisfying ϕ .*
- $EG.\phi = \nu(Z : (\mathbf{Self} \rightarrow \mathbf{Prop})).(\lambda(x : \mathbf{Self}).\phi(x) \wedge ((\Box \perp)(x) \vee (\Diamond Z)(x)))$ *There is run containing only states satisfying ϕ .*
- $EF.\phi = \mu(Z : (\mathbf{Self} \rightarrow \mathbf{Prop})).(\lambda(x : \mathbf{Self}).\phi(x) \vee (\Diamond Z)(x))$ *There is a run containing a state satisfying ϕ .*
- $\phi UNTIL \psi = \nu(Z : (\mathbf{Self} \rightarrow \mathbf{Prop})).(\lambda(x : \mathbf{Self}).\psi(x) \vee (\phi(x) \wedge (\Box Z)(x)))$ *ϕ does hold until a state satisfying ψ is reached.*
- $\phi BEFORE \psi = \mu(Z : (\mathbf{Self} \rightarrow \mathbf{Prop})).(\lambda(x : \mathbf{Self}).\phi(x) \vee (\neg\psi(x) \wedge (\Diamond \top)(x) \wedge (\Box Z)(x)))$ *ϕ is satisfied by a state before another state satisfying ψ is reached.*

In the previous Section 7.1 we have indicated that the infinitary modal operators can be represented by modal fixed-point and single-step modal operators. Subsequently, we discuss the details of this representation.

Example 30 (Representation of the infinitary Modal Operators) *The infinitary modal operators as given in [Rot00] can not be defined in a straightforward manner, since Rothe used method projection to select successor states. Then the infinitary modal operators are defined to be of the following form*

- $\blacksquare^M \phi : \mathbf{Self} \rightarrow \mathbf{Prop}$ *and*
- $\blacklozenge^M \phi : \mathbf{Self} \rightarrow \mathbf{Prop}$

for M being a set of methods in the regarded coalgebraic class signature and thus $M \subseteq \Sigma_M$. Thereby he restricted the class of method types to those allowing only one successor state in the return value.

However, it is not hard to see that each method projection of that kind can be represented by a path as defined above. We obtain infinitary modal operators of the form

- $\blacksquare^P \phi : \mathbf{Self} \rightarrow \mathbf{Prop}$ and
- $\blacklozenge^P \phi : \mathbf{Self} \rightarrow \mathbf{Prop}$

for P being a set of valid Paths through $[[\sigma_M]]_{U_1, \dots, U_n}$. For simplicities sake we will implicitly assume infinitary modal operators to be defined w.r.t. paths instead of methods. We obtain the following representation of \blacksquare

$$\blacksquare^P \phi \stackrel{def}{=} \nu(\lambda(Z : \mathbf{Self} \rightarrow \mathbf{Prop}).(\lambda(x : \mathbf{Self}).\phi(x) \wedge (\square^P Z)(x)))$$

and by a short derivation

$$\begin{aligned} \blacklozenge^P \phi &= \neg \blacksquare^P \neg \phi \\ &= \neg \nu(Z : \mathbf{Self} \rightarrow \mathbf{Prop}).(\lambda(x : \mathbf{Self}).\neg \phi(x) \wedge (\square^P Z)(x)) \\ &= \mu(Z : \mathbf{Self} \rightarrow \mathbf{Prop}).(\lambda(x : \mathbf{Self}).\phi(x) \vee \neg(\square^P \neg Z)(x)) \end{aligned}$$

we obtain for \blacklozenge

$$\blacklozenge^P \phi \stackrel{def}{=} \mu(\lambda(Z : \mathbf{Self} \rightarrow \mathbf{Prop}).(\lambda(x : \mathbf{Self}).\phi(x) \vee (\blacklozenge^P Z)(x)))$$

In the following I will give some examples for applying the modal operators \blacklozenge and \square from the previous section and the modal fixed-point operators to the specification of our scheduler example.

Example 31 (Scheduler - continueing example 19) *For the example properties we need the paths from example 23 on page 27. Furthermore, consider the following polynomial functor interpreting Σ_M of the signature of our Scheduler.*

$$T(X) = \begin{array}{cccc} \text{request} & \text{assign} & \text{finish} & \text{get_resource} \\ (C \rightarrow X) & \times ((C \times R) \rightarrow X) & \times ((C \times R) \rightarrow X) & \times (R + 1) \\ \hat{\kappa}_1 & \hat{\kappa}_2 & \hat{\kappa}_3 & \hat{\kappa}_4 \end{array}$$

such that the successor states for their respective methods are determined by the following paths

- *request* : $\hat{\kappa}_1(C \times \{*\})$
- *assign* : $\hat{\kappa}_2((C \times R) \times \{*\})$
- *finished* : $\hat{\kappa}_3((C \times R) \times \{*\})$
- *get_resource* does not change the state

The following liveness property denotes that any requesting client will finally be assigned a resources. To support readability we introduce the truth predicate $\top_{\mathbf{Self}}$ on the state space as an abbreviation $\top_{\mathbf{Self}} \stackrel{def}{=} \lambda(x : \mathbf{Self}).\top$; and similarly the empty predicate $\perp_{\mathbf{Self}} \stackrel{def}{=} \lambda(x : \mathbf{Self}).\perp$.

$$P_1 = \lambda(x : \mathbf{Self}). \forall(c : \mathit{Client}). \Box^{\kappa_1(c,*)}. P_1^{wba}$$

such that P_1^{wba} (“will be assigned”), describing that c will eventually be assigned a resource, is defined by

$$P_1^{wba} = \mu(Z : \mathbf{Self} \rightarrow \mathbf{Prop}). (\lambda(x' : \mathbf{Self}). \left(\begin{array}{c} P^{\text{definitely assigned}}_{\vee} \\ (\Diamond \top_{\mathbf{Self}})(x') \wedge (\Box Z)(x') \end{array} \right))$$

with $P_1^{\text{definitely assigned}}$, that the assignment definitely takes place, being defined by

$$P_1^{\text{definitely assigned}} = \left(\begin{array}{c} (\Diamond^{\hat{\kappa}_2}(\{c\} \times R) \times \{*\}) \top_{\mathbf{Self}}(x') \wedge \\ (\Box^{P_T - \hat{\kappa}_2}(\{c\} \times R) \times \{*\}) \perp_{\mathbf{Self}}(x') \end{array} \right)$$

As finiteness (or more accurately liveness) property as it becomes visible in the subterm P_1^{wba} can not be designed in the unextended logic of CCSL as defined in [Tew02a].

Liveness properties like in the previous example find their applications wherever a system must be ensured not to run into dead loops. However, a system satisfying such a property may eventually respond but after too many steps. The user may want to specify that a print server responds after 10 minutes, even if it rejects the request. However, even the newly introduced operators do not provide a smart way to state such kind of properties.

Another application of a finer grained liveness property is for semaphore systems. After a semaphore is checked for availability it should be assigned within at most two steps for instance. This can be done attaching single-step modal operators. This kind of sequential combination is also used in the following example.

Example 32 (Scheduler - continueing example 19) *The striking feature of the modal operators is that one can easily combine actions like in the following property.*

$$P_2 = \nu(Z : \mathbf{Self} \rightarrow \mathbf{Prop}). \left(\begin{array}{c} \Diamond^{\hat{\kappa}_2}((C \times R) \times \{*\}) \Box^{\hat{\kappa}_3}((C \times R) \times \{*\}) \lambda(x : \mathbf{Self}). \text{up?}(get_resource\ x) \wedge (Z\ x) \end{array} \right)$$

P_2 expresses that in a state one can assign a resource to a client. After this assignment and the reassignment of any resource a resource must be available and the game starts again.

Although this example is not of any practical use, it shows how naturally actions can be plugged together. Such a construction can obviously not be realized without the operators introduced in this thesis.

Even though the latter two examples were of purely academic nature they already showed how quickly formulae involving the single-step modal operators or the modal fixed-point operators can become incomprehensible.

Chapter 8

Related Work and Conclusions

In [Ven04] Venema describes an efficient way to compute the semantics of least and greatest fixed point formulae for coalgebras. Therefor he introduces an indexed family of logics $\mu\mathcal{L}^F$, and an algorithm translating each $\mu\mathcal{L}^F$ -formula into an automaton. The acceptance condition of such an automaton is determined by an infinite two-player graph game.

In Chapter 4 we have seen how a a class signature $\langle \Sigma_M, \Sigma_C \rangle$ determines a polynomial functor underlying a T -coalgebra that interpretes Σ_M . Venema exclusively considers R -standard functors, i.e. functors preserving inclusions and weak pullbacks. Note that the polynomial functors as introduced in the preliminary Chapter 2 preserve inclusions which can be proved by structural induction. Furthermore it has been indicated in [Tew02b] that polynomial functors preserve weak pullbacks like R -standard functors.

In the preliminaries we have seen how a functor T potentiates a T -coalgebra, i.e. a function mapping X to TX . Such a T -coalgebra is pointed if it comes equipped with an element x from the underlying set X acting in some sense as an initial element. In CCSL the algebraic part a of a model $\langle X, c, a \rangle$ interpreting a class signature as above determines several of those initial states.

As already mentioned at the beginning of this section, Venema introduced a family of logics $\mu\mathcal{L}^F$ to describe properties of F -coalgebras for R -standard functors F . Each such logic $\mu\mathcal{L}^F$ constitutes a proper sublogic of the CCSL logic extended by single-step modal operators and modal fixed-point operators in that it does not provide e.g. higher order constructions like lambda abstraction. Their semantics coincide with semantics of the according restriction of CCSL as reconstructable by comparing the interpretations in $\mu\mathcal{L}^F$ and CCSL.

Another work that I referred to through the previous chapters is the diploma thesis [Rot00] of Jan Rothe. He introduced the infinitary modal operators into CCSL. As I argued in chapter 6 these operators are useful for software specification but too coarse when being applied with the modal fixed-point operators. Rothe furthermore restricted the logic of CCSL by assuming monomorphic

types, only, and polynomial types not allowing **Self** to occur on the left hand side of an exponent. The latter restriction limits the user in specifying copy constructors. This special kind of method is, however, often used in object oriented programming.

Concluding this thesis we can remark that both the single-step modal operators and the modal fixed point operators add expressivity to the logic of CCSL. By using path selectors for the modal operators we obtained a finer granularity for selecting successor states than method projection offered. Furthermore, we have seen that the modal fixed-point operators allow the user not only to discriminate between finite and infinite runs, but also to reason over sequences of actions.

On the other hand I have to admit that fitting the modal operators into the higher order logic of CCSL came with the cost of an incomprehensibly complex syntax.

Bibliography

- [Bar92] Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures)*, Abramsky & Gabbay & Maibaum (Eds.), Clarendon, volume 2. a, 1992.
- [Bor94] Francis Borceux. *Handbook of categorical Algebra: Basic Category Theory*. Cambridge University Press, 1994.
- [Jac99] Bart Jacobs. *Categorical Logic and Type Theory*. Elsevier, 1999.
- [Lei94] Daniel Leivant. Higher order logic. pages 229–321. Oxford University Press, Inc., 1994.
- [Rot00] Jan Rothe. Modal logics for coalgebraic class specification. Master’s thesis, Inst. Theor. Informatik, TU Dresden, D-01062 Dresden, Germany, 2000.
- [Rut96] J. J.M.M. Rutten. Universal coalgebra: a theory of systems. Technical report, CWI (Centre for Mathematics and Computer Science), 1996.
- [Rut97] Bart Jacobs; Jan Rutten. A tutorial on (co)algebras and (co)induction, 1997.
- [Sti92] C. Stirling. Modal and temporal logics. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science: Background - Computational Structures (Volume 2)*, pages 477–563. Clarendon Press, Oxford, 1992.
- [Tew01] Jan Rothe; Bart Jacobs; Hendrik Tews. The coalgebraic class specification language cctl. 2001.
- [Tew02a] Hendrik Tews. The coalgebraic class specification language cctl - syntax and semantics -. Technical report, Univ. of Technologies Dresden, Dept. of CS, 2002.
- [Tew02b] Hendrik Tews. *Coalgebraic Methods for Object-Oriented Specification*. PhD thesis, 2002.
- [Ven04] Yde Venema. Automata and fixed point logic - a coalgebraic perspective. Technical report, Universiteit van Amsterdam, ILLC, 2004.