TECHNISCHE UNIVERSITÄT
DRESDEN

Fakultät Informatik

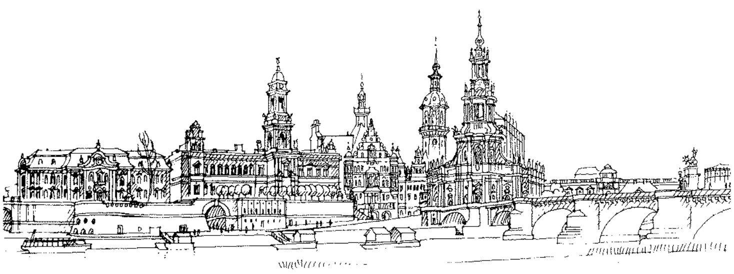
TUD–FI02–08–August 2002

Hendrik Tews

Institut für Theoretische Informatik

The Coalgebraic Class Specification Language CCSL
—Syntax and Semantics—

Technische Berichte
Technical Reports
ISSN 1430-211X



Technische Universität Dresden
Fakultät Informatik
D-01062 Dresden
Germany

URL: http://www.inf.tu-dresden.de/

The Coalgebraic Class Specification Language CCSL —Syntax and Semantics—

Hendrik Tews

5th August 2002

Abstract. This report describes the coalgebraic Class Specification Language CCSL, its syntax, its semantics, and the CCSL compiler that translates CCSL specifications into higher-order logic. The material in this report is mostly identical with Chapter 4 of my forthcoming PhD [Tew02b].

Acknowledgements. CCSL and the CCSL compiler have been designed and implemented in the LOOP project on formal methods of object-orientation. All members of the LOOP team contributed to CCSL, often substantially. Bart Jacobs and Jan Rothe read and corrected various draft versions of the material in this report.

Contents

List of Figures and Tables

1. Introduction

This report describes the Coalgebraic Class Specification Language CCSL, its syntax, its semantics, and the CCSL compiler that translates CCSL specifications into higher-order logic. The material in this report is mostly identical with Chapter 4 of my forthcoming PhD [Tew02b].

The most distinguishing feature of CCSL is the provided notion of coalgebraic specification. Further, CCSL does not force its users into a religious decision to adopt either the algebraic or the coalgebraic point of view. Instead CCSL encourages the combination of abstract data type specifications with coalgebraic specifications in an iterative way. Real world examples often involve both: abstract data types *and* behavioural aspects (or process types). Such examples can be mapped to CCSL in a very natural way. CCSL was first presented to public in [HHJT98], a recent reference is [RTJ01], and some more technical aspects are described in [Tew02a].

The specification language CCSL (together with some supporting tools) was developed in close cooperation with the people who are associated within the LOOP project on formal methods for object-oriented programming. LOOP stands for Logic of Object-Oriented Programming.[1] It is a project on formal methods for object-oriented languages. It started in 1997 as a joint project between the Katholieke Universiteit Nijmegen (University of Nijmegen) and the Technische Universität Dresden (Dresden University of Technology). Apart from myself the following people do or have been working within the LOOP project: Joachim van den Berg, Ulrich Hensel, Marieke Huisman, Bart Jacobs, Erik Poll, and Jan Rothe. There is a certain diversity in the research done in the LOOP project. The common underlying base is the use of coalgebras as a semantics for object orientation and the use of theorem proving support. Apart from the work that is described in the present thesis, the research in the LOOP project focuses on a formal semantics of the programming language Java and the verification of Java programs (see [Hui01]), especially for Java Card programs. Another topic is the design of JML, the Java Modelling Language (see [LLP+00]). JML is an extension of Java that allows one to specify the detailed design of Java classes and interfaces. The work on CCSL goes back to the beginning of the LOOP project. All LOOP project members have contributed to CCSL in one or the other way, often substantially.

CCSL is based on the observation of [Rei95] that coalgebras can give semantics to classes of object-oriented languages. Jacobs picked this idea up and developed it further in a series of publications, see [Jac95, Jac96, Jac97a, Jac97b]. Some important notions (for instance that of an *invariant*) and even parts of the syntax of CCSL can be traced back to this work. An important difference between this earlier work of Jacobs on coalgebraic specification and the work in the LOOP project is, that all work in the LOOP project is centred around *mechanical verification*. There are two reasons for the shift towards mechanical verification. First, software verification is intrinsically difficult because it involves a large amount of detail, especially many case distinctions. So to apply software verification to real programs written in a mainstream programming language (as opposed to academic examples written

[1]The LOOP project is on the world wide web, see URL http://www.cs.kun.nl/~bart/LOOP/.

in a clean academic programming language) requires tool support. With the right computer support, the person who carries out the verification can concentrate on the important (and difficult) parts, while the verification tool carries out simple computations and ensures accuracy and the correctness of the whole verification.

Secondly also for academic environments and pure science right tool support is important. It enables the scientist to test his or her results and to get inspiration from large examples. For instance the work on coalgebraic refinement, presented in [JT01], was inspired by a large case study on coalgebraic specification of lists.

The design goals of CCSL are:

1. to provide a notation for parametrised class specifications based on coalgebras;
2. to provide algebraic specifications of abstract data types based on initial algebras;
3. to use a familiar logic;
4. to restrict expressiveness only when absolutely necessary;
5. to provide theorem proving support.

Let me discuss design goal 5 first. The importance of theorem proving support has been explained before. In order to provide a theorem proving environment for CCSL there is the following alternative: On the one hand one can write a special purpose theorem prover. On the other hand one can develop a front end to existing theorem provers. The former variant sounds attractive but is (and was) far beyond the man power of the LOOP team. In the LOOP project we therefore chose the latter variant. It has the additional advantage that we can choose among the available theorem provers and thus profit from the work that has been spent into these tools.

The front end that connects CCSL with a theorem prover can be seen as a compiler that translates CCSL into its semantics in higher-order logic. There exists a prototype implementation that supports the two theorem provers PVS [ORR+96, ORSvH95] and ISABELLE/HOL in the new style ISAR syntax [NPW02b, Wen02] (but see also [NPW02a, Pau02]). I refer to this prototype as the *CCSL compiler* from now on.

To meet design goals 1 and 2 CCSL contains concrete syntax for algebraic and for coalgebraic signatures. The concrete syntax for coalgebraic signatures uses terminology from object-oriented programming, for instance coalgebraic operations are declared with the keyword `METHOD`. With coalgebraic signatures one cannot describe the construction of new objects. Therefore class signatures in CCSL contain a degenerated algebraic signature (describing the constructors) in addition to the coalgebraic signature.

For design goal 2 CCSL currently supports only abstract data type specifications in the sense of this report. That is, the abstract data types of CCSL do neither contain axioms nor equations. Both PVS and ISABELLE/HOL have extensions for the definition of abstract data types (without axioms). So it is straightforward to translate the abstract data types of CCSL into PVS and ISABELLE. To incorporate algebraic specifications into CCSL it would be necessary to define their semantics in higher-order logic. The problem here is that, although it is well known how to translate algebraic specifications into higher-order logic, this is quite a bit of work. Besides, such a translation has been done before (for instance for CASL in the common framework initiative [Mos97], see [Mos00]) and one cannot expect many new insights. It is very difficult to get this kind of work done in an academic environment.

A specification language comes always equipped with some kind of logic. A variety of different logics for coalgebras have been developed so far. One idea is that coalgebraic logic should be based on *coequations*, which are dualized equations. This approach is for instance pursued in [Cor98, Cîr99]. Other work proposes different modal logics, see for instance [Mos99, Kur00, Rößl00a, Hug01]. However, the work on modal logic for coalgebras is mostly driven by purely mathematical interests. The resulting logics are not well-suited for a specification language. The most modest approach for a coalgebraic logic considers coalgebraic signatures as special polymorphic signature and uses traditional first-order equational logic over these signatures (see for instance [Jac96, Kur98]). Such an equational logic is already sufficient for many examples. In the LOOP project we decided to use a higher-order equational logic to gain expressiveness.

Higher-order equational logic is certainly a well-known logic, as demanded by design goal number 3. It makes the semantics of CCSL specifications easy to understand. This way the user can devote his attention on the properties instead on how to express them.

With the choice of higher-order logic we deliberately neglected some proof theoretic issues. For instance there does not exist a complete derivation system for the logic of CCSL. In contrast, [Cor98], [Cîr99], and also [Röß00a] restrict their coalgebraic signatures and obtain a complete derivation system for their logics. As usual in interactive theorem proving, the lack of a completeness theorem has never been a problem in our case studies. In contrast, the additional expressivity of our notion of coalgebraic class signatures and of our logic turned out to be very useful. Similarly other famous negative results for higher-order logic, like the undecidability of unification, have never posed any problems.

The design goal number 4 is probably the most debatable one. Because of the expressiveness of CCSL an user can easily write inconsistent specifications. Further, it is possible to construct coalgebraic signatures that correspond to functors whose properties have not been investigated (yet). There are two arguments here. The first one is about correctness: Whatever the user writes in CCSL, the final working environment is either PVS or ISABELLE. Because the translation of CCSL uses almost no axioms[2] any inconsistency that passes the CCSL compiler is finally caught in the theorem prover. The bottom line here is that one can rest on the correctness of the theorem prover.

The second argument is that CCSL is a research tool that helps to explore the fascinating world of coalgebraic specification. If we exclude from CCSL everything that is not well understood then we cannot use it for future research.

Figure 1 depicts the working environment for coalgebraic specification when using CCSL. The CCSL compiler reads files containing coalgebraic specifications and produces output for either PVS or ISABELLE/HOL (depending on a command line switch). The produced output

[2]The exceptions are the axioms about the existence of loose and final models that facilitate aggregation. Subsection 8.3 gives guidelines on the use of the generated theories that ensure that these axioms cannot introduce inconsistencies.

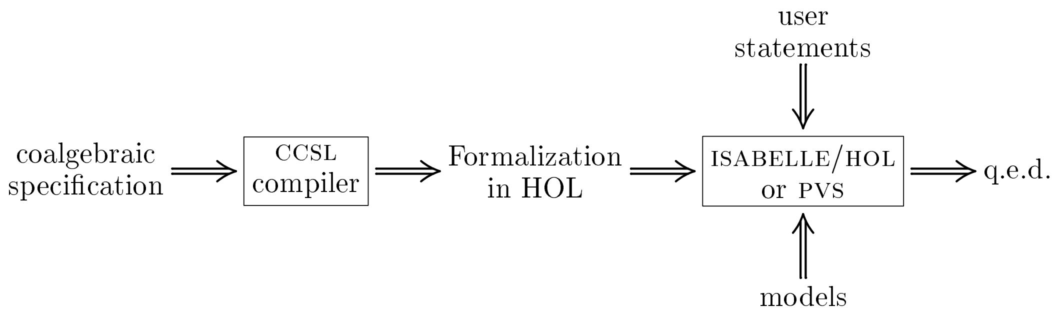


Figure 1: Working environment with CCSL

can be directly loaded into PVS or ISABELLE. In the following I refer to the chosen proof environment as the (target) theorem prover. The formulae one wants to prove and the models are usually formulated in the logic of the target theorem prover.

Internally the CCSL-compiler uses an abstract representation of higher-order logic, so that a third theorem prover could easily be supported by adding a new pretty-printing module that translates the abstract representation into the concrete syntax of the new theorem prover. For the work on Java and JML in the LOOP project a similar compiler has been developed [vdBJ01] that translates Java (respectively JML) into PVS or ISABELLE. Initially the compiler for Java and JML was derived from an early version of the CCSL compiler. At the moment both tools are separate programs that share parts of the internal data structures and a few modules (for instance the pretty printers for PVS and ISABELLE/HOL).

Some of the material of this report appeared partially already somewhere else. A simplified version of ground signatures and coalgebraic class specifications (without a proper treatment of variances) appeared originally in [RTJ01]. An even simpler version that might be called *first order coalgebraic class specification without binary methods* can be found in [Tew00]. The CCSL grammar is taken from the CCSL reference manual [Tew02a]. In comparison with these cited papers this report presents the syntax and the semantics of CCSL and its coalgebraic and type theoretic foundations together. Moreover, the material is presented here without the simplifications that were necessary because of the available space and the expected audience in [RTJ01] and [Tew00].

In describing the semantics of CCSL I am facing the following problem: The semantics of coalgebraic class specifications and that of abstract data type specifications are mutually dependent. Therefore I first describe *ground signatures* and their models. In the beginning it will be a bit unclear, where the items in the ground signature come from and how they get their semantics. After that I describe the semantics of class specifications and abstract data type specifications with respect to a given ground signature (and a model of it). In the end I close the circle with the discussion of iterated specifications: There I show how class specifications and abstract data type specifications extend the ground signature with new types and constants.

This report is structured as follows: Each section introduces one syntactic category or a specific problem of CCSL. The first section is on types. Section 3 is on variance checking. Then I define ground signatures and coalgebraic class signatures. Section 6 explains the higher-order logic of CCSL and Section 7 defines abstract data type specifications. Section 8 discusses iterated specifications. The following sections are less formal, Section 9 discusses the object-oriented features of CCSL and Section 10 combines a few minor topics that do not fit somewhere else. In the last section of this report I summarise and discuss related work. Most of the sections contain a subsection that describes the concrete CCSL syntax. For convenience the complete CCSL syntax is indexed in the subject index and collected in Appendix C. There are two additional appendices that present related material: Appendix A describes applications of CCSL, and Appendix B explains coalgebraic refinement.

2. The Type Theory of CCSL

In CCSL class specifications (and abstract data types) may depend on type parameters. For instance, a CCSL specification for (possibly infinite) sequences depends on one type parameter, the type of the elements of the sequences. When using the sequences in subsequent specifications this type parameter must be instantiated with a concrete type. For CCSL it is therefore necessary to develop a polymorphic type theory. A polymorphic type theory allows one to model parametric polymorphism in the sense of [CW85]. In such a type theory terms may depend on a finite set of type variables *and* a finite set of term variables. *(Term) judgements* are used to formally derive valid typings for terms. A term judgement consists of four parts, a type variable context, a term variable context, a term, and a type. A typical example is

$$\alpha : \mathsf{Type} \;\mid\; q : \mathsf{Seq}[\alpha] \;\vdash\; \mathsf{next}(q) : \mathbf{1} + \alpha \times \mathsf{Seq}[\alpha]$$

It states that, if the variable q has type $\mathsf{Seq}[\alpha]$ for an arbitrary type α, then the application $\mathsf{next}(q)$ has the depicted type. In the derivation system of a type theory one also has other kinds of judgements, for instance for deriving that a type expression is a valid type in the system. I introduce these different judgements when they are needed.

In the following I present the type theory for CCSL. It is a specialised version of the polymorphic type theory $\lambda\!\rightarrow$ (see Section 3 in [Bar92] or Section 8.1 in [Jac99]) enriched with product, coproduct, and exponent types and with the special types Self and Prop. The type Self represents the state space of classes and abstract data types. In Section 6 I describe a higher-order logic over the type theory of CCSL. The type Prop will then be the type of propositions. Instead of working with higher-order signatures as [Jac99] does, I prefer to formalise Prop as a special type.

In the type theory of CCSL *type constructors* will play an important role. A type constructor can be thought of as a function that acts on types. A typical example is the type constructor list that builds the type $\mathsf{list}[\tau]$ of (finite) lists over τ for any type τ. The number of arguments that a type constructor takes is called the *arity* of the type constructor. Type constructors of arity 0 (i.e., those that take no arguments) are *type constants* such as $\mathbb{N}$ for the natural numbers. Type constructors are the technical means to extend the specification environment. The technical details about this are in Section 8, but it is good to have a rough idea about what is going on.

Each CCSL specification consists of a finite list $\mathcal{S}_1, \ldots, \mathcal{S}_n$ of ground signature extensions (Section 4), coalgebraic class specifications (Sections 5 and 6), and abstract data type specifications (Section 7). Each of the $\mathcal{S}_i$ is relative to a ground signature Ω_i and every $\mathcal{S}_i$ can define type constructors, constants, and functions. The newly defined items are added to Ω_i to yield Ω_{i+1}. This way a specification $\mathcal{S}_i$ can refer to all specifications $\mathcal{S}_j$ with $j < i$.

In what follows type constructors are treated as syntactic entities that come along with a semantics. It might help to think of type constructors as resulting from a data type specification (such as finite lists or binary trees) or a process type specification (such as possibly infinite sequences) that already has been processed by the CCSL compiler.

2.1. Kinds

Kinds are used to distinguish types from type constructors and to count the type arguments of the type constructors. So kinds are natural numbers. (Other pure type systems, for instance λ_ω allow more complex kinds and also type variables of complex kinds, compare [Bar92] or [Jac99].) In judgements I use outlined lower-case letters like $\mathbb{k}$ for kinds. Ordinary types have kind 0. To improve readability I write $\sigma : \mathsf{Type}$ instead of $\sigma : 0$. Type constructors that take n arguments will have kind n. *Kind judgements* have the form

$$\vdash \mathbb{k} : \mathsf{Kind}$$

They state that $\mathbb{k}$ is a valid kind (i.e., a natural number).

2.2. Types

Types are build from type variables and type constructors including product types, coproduct (or sum) types, and exponent types. I use lowercase Greek variables like $\alpha, \beta, \ldots$ to denote type variables. A *type variable context* is a finite list of type variables. I assume that all type variables in a context are distinct. This can formally be ensured by using only type variables $\alpha_1, \alpha_2, \ldots$, but I would like to ignore these technicalities. In the type theory $\lambda{\rightarrow}$ type variables are place holders for types only (and not for arbitrary type constructors). To emphasise this I write type contexts as $\alpha_1 : \mathsf{Type},\ \alpha_2 : \mathsf{Type}, \ldots$ with the explicit kind Type. Arbitrary types are denoted with lower case Greek variables like τ, σ. I use *type judgements* to formally derive all types. A type judgement has the form

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

where Ξ is a type variable context, τ is a type expression containing only type variables from Ξ, and $\mathbb{k}$ is a valid kind. Such a type judgement states that τ is a type constructor of kind $\mathbb{k}$ (or an ordinary type, if $\mathbb{k} = \mathsf{Type}$).

Type constructors, which can be used to build composite types, are given as part of a ground signature (see Definition 4.1 on page 27 below). A type constructor C of kind (or arity) $\mathbb{k}$ for a valid kind $\mathbb{k}$ is given as a judgement

$$\vdash \mathsf{C} : \mathbb{k}$$

Type constructors of arity 0 will be called type constants. I assume a set $\mathcal{C}$ of type constructors in the following.

Definition 2.1 (Types) The types over a set of type constructors $\mathcal{C}$ are finitely generated as the least set including

- α for a type variable $\alpha : \mathsf{Type}$
- K for a type constant $\vdash \mathsf{K} : \mathsf{Type}$ in $\mathcal{C}$
- $\mathsf{C}[\tau_1, \ldots, \tau_{\mathbb{k}}]$ for a type constructor $\vdash \mathsf{C} : \mathbb{k}$ in $\mathcal{C}$ and types $\tau_1, \ldots, \tau_{\mathbb{k}}$
- Self, the special type that stands for the carrier set of class specifications and abstract data type specifications
- Prop, the type of propositions
- $\mathbf{1}$, the unit type and $\mathbf{0}$ the empty type
- the product $\tau \times \sigma$, the coproduct $\tau + \sigma$, and the exponent $\tau \Rightarrow \sigma$ for types τ and σ

Figure 2 contains a derivation system that allows one to derive a judgement $\Xi \vdash \tau : \mathsf{Type}$ precisely if τ is a type according to the preceding Definition with type variables Ξ.

In the following I assume that the product and the coproduct of types is associative (i.e., $(\tau_1 \times \tau_2) \times \tau_3 \cong \tau_1 \times (\tau_2 \times \tau_3)$ and $(\tau_1 + \tau_2) + \tau_3 \cong \tau_1 + (\tau_2 + \tau_3)$). I assume further the isomorphisms $\mathbf{1} \Rightarrow \tau \;\cong\; \tau$ and $\tau \times \mathbf{1} \;\cong\; \tau$. The semantics of types (Definition 3.5 on page 22) will ensure that the corresponding interpretations are isomorphic collections of sets. The exponent $\Rightarrow$ is assumed to associate to the right, that is $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \;=\; \tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3)$. I omit parenthesis in the following when they do not contribute to readability.

2.3. Type Expressions in CCSL

CCSL allows all types according to Definition 2.1. However there are the following points to note.

- Kinds do not appear in the concrete syntax of CCSL, the CCSL type checker ensures that all type constructors get the right number of arguments.
- Type variables appear as normal identifiers, which have been declared as type parameters.
- There are the two keywords `CARRIER` and `SELF` that represent the special type Self. The keyword `CARRIER` represents Self in abstract data type specifications (see Section 7), inside class specifications (Section 5) one has to use `SELF`.

kinds

$$\frac{}{\vdash \mathsf{Type} : \mathsf{Kind}} \qquad \frac{\vdash \mathbb{k} : \mathsf{Kind}}{\vdash \mathbb{k}+1 : \mathsf{Kind}}$$

type variable **Self** **Prop**

$$\frac{}{\Xi \vdash \alpha : \mathsf{Type}}\ \alpha \in \Xi \qquad \frac{}{\Xi \vdash \mathsf{Self} : \mathsf{Type}} \qquad \frac{}{\Xi \vdash \mathsf{Prop} : \mathsf{Type}}$$

type constructor

$$\frac{\vdash \mathbb{k} : \mathsf{Kind} \quad \Xi \vdash \sigma_1 : \mathsf{Type} \quad \cdots \quad \Xi \vdash \sigma_{\mathbb{k}} : \mathsf{Type}}{\Xi \vdash \mathsf{C}[\sigma_1, \ldots, \sigma_{\mathbb{k}}] : \mathsf{Type}} \quad \text{for } \mathsf{C} : \mathbb{k} \text{ in } \mathcal{C}$$

product type

$$\frac{}{\Xi \vdash \mathbf{1} : \mathsf{Type}} \qquad \frac{\Xi \vdash \tau : \mathsf{Type} \quad \Xi \vdash \sigma : \mathsf{Type}}{\Xi \vdash \tau \times \sigma : \mathsf{Type}}$$

coproduct type

$$\frac{}{\Xi \vdash \mathbf{0} : \mathsf{Type}} \qquad \frac{\Xi \vdash \tau : \mathsf{Type} \quad \Xi \vdash \sigma : \mathsf{Type}}{\Xi \vdash \tau + \sigma : \mathsf{Type}}$$

exponent type

$$\frac{\Xi \vdash \tau : \mathsf{Type} \quad \Xi \vdash \sigma : \mathsf{Type}}{\Xi \vdash \tau \Rightarrow \sigma : \mathsf{Type}}$$

The following two rules are not necessary to build all possible types, but sometimes it is convenient to use them. They can be derived by induction on the structure of derivations.

type context weakening **type substitution**

$$\frac{\Xi \vdash \tau : \mathsf{Type}}{\Xi, \alpha : \mathsf{Type} \vdash \tau : \mathsf{Type}}\ \alpha \notin \Xi \qquad \frac{\Xi, \alpha : \mathsf{Type} \vdash \tau : \mathsf{Type} \quad \Xi \vdash \sigma : \mathsf{Type}}{\Xi \vdash \tau[\sigma/\alpha] : \mathsf{Type}}$$

Figure 2: Derivation system for well-formed types over a set of type constructors $\mathcal{C}$.

- CCSL allows n-ary product types $\sigma_1 \times \cdots \times \sigma_n$. Further, the product of types $\sigma_1 \times \cdots \times \sigma_n$ is written with brackets `[`$\sigma_1, \ldots, \sigma_n$`]` like in PVS.

- The binary coproduct and the unit type are formalised as abstract data types, which are defined in the CCSL prelude (see Section 10.8). So there is no concrete syntax for unit and coproduct.

- The empty type is not available for the ISABELLE back end of CCSL. For the PVS back end the empty type is declared in the prelude.

- The type Prop is called bool, it is built-in into the CCSL compiler.

The grammar of CCSL is given in a BNF–like notation. Brackets [...] denote optional components, braces {| ...|} denote arbitrary repetition (including zero times), and parenthesis (...) denote grouping. Terminals are set in `UPPERCASE TYPEWRITER`, non–terminals in *lowercase slanted*. The terminal symbols for parenthesis and brackets are written as <u>(</u>, <u>)</u>, <u>[</u>, and <u>]</u>. For convenience all keywords and nonterminals of the CCSL grammar can be found in the subject index.

The concrete syntax for type expressions in CCSL is given by the following BNF rules.

type	::=	`SELF`
	|	`CARRIER`
	|	`BOOL`
	|	<u>[</u> *type* {| , *type* |} `->` *type* <u>]</u>
	|	<u>[</u> *type* {| , *type* |} <u>]</u>
	|	*qualifiedid*
	|	*identifier argumentlist*
argumentlist	::=	<u>[</u> *type* {| , *type* |} <u>]</u>

The form `[`σ_1`, ... ,`σ_n` -> `τ`]` is shorthand for `[[`σ_1`, ... ,`σ_n`] -> `τ`]`. Qualified identifiers are explained in Subsection 10.6 below (on page 102). In type expressions a qualified identifier can be either a simple *identifier* (referring to a type variable or a type constant), or an instantiated ground signature name with a type *identifier* declared in that ground signature.

3. Variance Checking

This section describes the algorithm that computes variances for type variables and for Self. I follow the ideas from [Sch97] but generalise Schroeder's variances such that I get information about the deepest nesting level at which a type variable (or Self) occurs positively and negatively.

Variances are mainly important for the CCSL typechecker. To get a semantics, the type expressions from a signature are translated to functors. Depending on the variance of Self one gets a polynomial functor, an extended polynomial functor, or a higher-order polynomial functor (see Proposition 3.8 on page 26). The class of models of the signature has very different properties depending on the functor (compare Chapter 3 of [Tew02b]). Another important point is that in abstract data-type definitions only type expressions with a certain variance are allowed (see Definition 7.1 on page 68). The reason for this restriction is that there is no initial semantics for arbitrary signatures [Gun92]. Finally, as a minor point, the CCSL compiler generates simpler output in the common case that a type variable does not occur with mixed variance (compare Proposition 3.6 and Subsection 3.4 on page 23ff).

In the following I first try to explain variances and the algorithm to compute them on an intuitive level. A precise definition follows in the first subsection, the development there is a bit technical, but there is nothing deep behind. To put it a bit sloppy, the algorithm to compute variances only counts parenthesis.

Informally speaking the *variance* of a type variable (or of Self) tells us if the type variable occurs on the left hand side of $\Rightarrow$, on its right hand side, or on both sides. Let me anticipate some bits of Definition 3.5 (interpretation of types) to explain this problem in greater detail. Consider type expressions over a set of type constructors that contains only one constant type, so $\mathcal{C} = \{\mathbb{N} : \mathsf{Type}\}$. If we ignore Self and type variables for a moment we can assign to every type τ a set $[\![\tau]\!]$ as its interpretation. We use $\mathbb{N}$, the set of natural numbers as interpretation of the type constant $\mathbb{N}$ and set $[\![\tau \odot \sigma]\!] = [\![\tau]\!] \odot [\![\sigma]\!]$ for $\odot \in \{\times, +, \Rightarrow\}$. (Here $\times, +$, and $\Rightarrow$ denotes the bicartesian closed structure on **Set**, that is, the cartesian product, the disjoint union, and the function space, respectively.)

Assume now that τ contains a type variable: $\alpha : \mathsf{Type} \vdash \tau : \mathsf{Type}$. Then its interpretation is a mapping that assigns to each set U, which we choose as an interpretation for α, the set $[\![\tau]\!]_U$. If for instance $\tau_1 = \alpha \times \mathbb{N}$ then $[\![\tau_1]\!]_U = U \times \mathbb{N}$. So the semantics of a type that contains type variables is an (set–) indexed collection of sets, where the indices are the interpretation of the type variables. Consider now two sets U and V as possible interpretations of α. A function $h : U \longrightarrow V$ gives in a canonical way rise to a function $[\![\tau_1]\!]_U \longrightarrow [\![\tau_1]\!]_V$. For τ_1 this function is given by $\lambda x : U \times \mathbb{N} \,.\, (h(\pi_1\, x), \pi_2\, x)$. If, for a second example, $\tau_2 = \mathbb{N} \Rightarrow \alpha$ then $[\![\tau_2]\!]_U = \mathbb{N} \Rightarrow U = \{f \mid f : \mathbb{N} \longrightarrow U\}$ and the function $[\![\tau_2]\!]_U \longrightarrow [\![\tau_2]\!]_V$ is given by $\lambda f : \mathbb{N} \longrightarrow U \,.\, h \circ f$.

Complications start if the type variable α occurs on the left hand side of $\Rightarrow$: Consider the type $\tau_3 = \alpha \Rightarrow \mathbb{N}$. This time a function $h : U \longrightarrow V$ induces a function $[\![\tau_3]\!]_V \longrightarrow [\![\tau_3]\!]_U$ *in the opposite direction!* It is given by $\lambda f : V \longrightarrow \mathbb{N} \,.\, f \circ h$. In this case, where the induced function goes into the opposite direction, one says that the type variable α occurs in τ with *negative variance* or alternatively one says that α occurs in τ at a *negative position*. A type variable occurs with *positive variance* (at a *positive position*) if the induced function keeps the direction, as in the preceding paragraph. If the type variable occurs with negative variance within a type expression that occurs itself at a negative position, then the type variable has positive variance again. Consider $\tau_4 = (\alpha \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}$. As interpretation we have

$$[\![\tau_4]\!]_U \quad = \quad \{f \mid f \text{ is a function that maps functions } U \longrightarrow \mathbb{N} \text{ to elements of } \mathbb{N}\}$$

With a function $h : U \longrightarrow V$ we can build the function

$$\lambda f : (U \Rightarrow \mathbb{N}) \longrightarrow \mathbb{N} \,.\, \big(\lambda g : V \longrightarrow \mathbb{N} \,.\, f(g \circ h)\big) \quad : \quad [\![\tau_4]\!]_U \longrightarrow [\![\tau_4]\!]_V$$

To distinguish the types τ_1 and τ_2 from τ_4 one says that α occurs in τ_1 and τ_2 *strictly positively*.

A type variable can also occur several times (with different variances) in one type expression. Consider $\tau_5 = \alpha \Rightarrow (\alpha \times \mathbb{N})$. To get a function $[\![\tau_5]\!]_U \longrightarrow [\![\tau_5]\!]_V$ we need now *two functions* $h^+ : U \longrightarrow V$ and $h^- : V \longrightarrow U$. The induced function on the interpretation of τ is

$$\lambda f : U \longrightarrow U \times \mathbb{N} \ . \ \big[\lambda v : V \ . \ \big(\ h^+(\pi_1(f(h^- \ v))), \ \pi_2(f(h^- \ v)) \ \big)\big]$$

A type variable that occurs with both positive and negative variance is said to have *mixed variance*.

In general the semantics of types is given by functors. The special type Self serves as a place holder for the arguments of the functor. So for the type $\sigma_1 = \mathsf{Self} \times \mathbb{N}$ we get as semantics the functor $T_1(X) = X \times \mathbb{N}$. And for $\sigma_2 = (\mathsf{Self} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}$ we get the functor $T_2(X) = (X \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}$. In both types σ_1 and σ_2 the type Self occurs with positive variance. However T_1 is a polynomial functor whereas T_2 is a higher-order polynomial functor. Because polynomial functors and higher-order polynomial functors have different properties, the CCSL compiler must generate different output, depending on whether a signature corresponds to a polynomial functor or a higher-order polynomial functor. So it is not only important if a type variable (or Self) occurs positively or negatively, it is also important at which maximum nesting level a type variable occurs. Therefore I use pairs (u_-, u_+) of natural numbers to denote variances. The first component u_- denotes the maximum nesting level at which the type variable occurs at a negative position. The second component u_+ denotes the maximum nesting level for positive occurrences. In the next subsection I formalise these variances as a special algebra and give an algorithm that computes the variances of the type variables and of Self in a type expression. In the following I describe informally how variances can be computed.

To compute the variances of a type expression τ one has to annotate every subexpression of τ with a natural number in the following way: The whole type is annotated with 0, walk now recursively down the structure of τ and increase the current number every time you pass over to the left hand side of a $\Rightarrow$. Keep the number constant if you pass over $\times$ or $+$ or if you stay on the right hand side of $\Rightarrow$. Let me write the annotations as subscripts to parenthesis, which enclose the subexpressions. Then I get for τ_4

$$(((\alpha)_2 \Rightarrow (\mathbb{N})_1)_1 \Rightarrow (\mathbb{N})_0)_0$$

Observe that subexpressions at positive positions get even numbers and subexpression at negative positions get odd numbers. This is because the variance is toggled from positive to negative and vice versa on the left hand side of $\Rightarrow$. To get the variance for the type variable α, pick out the maximum annotation of α for all negative occurrences (i.e., the maximal odd number for α) and the maximum annotation for all positive occurrences of α (i.e., the maximal even number). Use ? if the type variable does not occur with the corresponding variance. So we get that α has variance $(?, 2)$ in τ_4. For a more complex example consider the type expression

$$\Big[\big[\big((\alpha)_3 \Rightarrow (\mathbb{N})_2\big)_2 \Rightarrow (\alpha)_1\big]_1 \Rightarrow \big[\big((\alpha)_2 \Rightarrow (\mathbb{N})_1\big)_1 \Rightarrow (\alpha)_0\big]_0\Big]_0$$

Here α has the variance $(3, 2)$.

For the computation of variances also nonconstant type constructors are important. For the discussion of variances one can think of a type constructor as a macro that can be expanded into a type expression. For example let

$$\mathsf{C}[\alpha, \beta] \quad \stackrel{\text{def}}{=} \quad ((\alpha \Rightarrow \beta) \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}$$

Now we can form the type $\tau_5 = \mathsf{C}[\mathsf{Self}, \alpha] \Rightarrow \mathbb{N}$. After expanding C we can compute the variances and get that Self occurs in τ_5 with variance $(?, 4)$ and α with $(3, ?)$. So for the first argument of C the variance is toggled and the nesting level is increased by three. To denote this I say that the first argument of C has variance $(3, ?)$. Similarly, the second argument of C has variance $(?, 2)$. A type constructor can also copy its arguments into a positive and a negative position, for instance:

$$\mathsf{C}'[\alpha] \quad \stackrel{\text{def}}{=} \quad \alpha \Rightarrow \alpha$$

Therefore the variance of the only argument of C' is $(1, 0)$

For the formal treatment of type constructors I assume that, from now on, type constructors are given with variance annotation by a sequent

$$\vdash \mathsf{C} :: [(3, ?); (?, 2)]$$

Formally the variance annotation is a finite list of variances. The length of the list equals the arity of the type constructor and the elements of the list stand for the variances of its arguments.

The preceding algorithm to compute variances works only well if all variances of the type constructor are either of form $(?, u_+)$ or of form $(u_-, ?)$. For such a type constructor one adds either u_+ or u_- to the current number. So in the type expression

$$(\mathsf{C}[(\alpha)_4, (\alpha)_3])_1 \Rightarrow (\mathbb{N})_0)_0$$

α has variance $(3, 4)$. The treatment of arbitrary variances for type constructors does not really fit in this simplified annotation algorithm. For arbitrary type expressions it is best to use the algorithm from the next subsection.

I prefer to formalise the product, the coproduct, and the exponent of types and the type Prop by giving extra rules for them. Alternatively one can consider them as type constructors with the following variance:

$$\begin{array}{lcl}
\vdash & \mathsf{Prop} & :: [] \\
\vdash & \mathbf{1} & :: [] \\
\vdash & \mathbf{0} & :: [] \\
\vdash & \times & :: [(?, 0); (?, 0)] \\
\vdash & + & :: [(?, 0); (?, 0)] \\
\vdash & \Rightarrow & :: [(1, ?); (?, 0)]
\end{array}$$

3.1. Formalising Variances

To formalise variance checking I need the natural numbers enriched with an additional element ?, which is a zero for addition.[3] So set $\mathbb{N}^? \stackrel{\text{def}}{=} \mathbb{N} \cup \{?\}$ and extend addition to $\mathbb{N}^?$ with $? + n = n + ? = ?$ for all $n \in \mathbb{N}^?$. I further extend the order $<$ and make ? the least element: $\forall i \in \mathbb{N} \,.\, ? < i$. Now I can use max with the extended natural numbers, for instance $\mathsf{max}(?, n) = \mathsf{max}(n, ?) = n$ for all $n \in \mathbb{N}^?$. Although we saw in the last section that variances are pairs of an odd and an even number it is technically easier to define them as pairs of $\mathbb{N}^?$.

Definition 3.1 The *variance algebra* is the triple $(V, \cdot, \vee)$ such that

- $V = \mathbb{N}^? \times \mathbb{N}^?$ is the set of variances, where ? abbreviates $(?, ?) \in V$,
- $\cdot : V \times V \longrightarrow V$ is the *substitution operation* defined by

$$(u_-, u_+) \cdot (v_-, v_+) \quad \stackrel{\text{def}}{=} \quad \big(\mathsf{max}(u_- + v_+, u_+ + v_-),\ \mathsf{max}(u_- + v_-, u_+ + v_+)\big)$$

- and $\vee : V \times V \longrightarrow V$ is the *join operation* given by

$$(u_-, u_+) \vee (v_-, v_+) \quad \stackrel{\text{def}}{=} \quad \big(\mathsf{max}(u_-, v_-),\ \mathsf{max}(u_+, v_+)\big)$$

The set of well-formed variances $\overline{V} \subseteq V$ is given by

$$\overline{V} \quad \stackrel{\text{def}}{=} \quad \{(u_-, u_+) \mid (u_- = ? \text{ or } u_- \text{ is odd}) \wedge (u_+ = ? \text{ or } u_+ \text{ is even})\}$$

In the following I assume that $\cdot$ binds tighter than $\vee$, so $u_1 \cdot u_2 \vee u_3 = (u_1 \cdot u_2) \vee u_3$.

Lemma 3.2

1. *$(\overline{V}, \cdot, \vee)$ is a subalgebra of the variance algebra, that is, both the substitution and the join operation restrict to $\overline{V} \times \overline{V} \longrightarrow \overline{V}$.*
2. *$(V, \cdot)$ forms a commutative monoid with zero element ?.*
3. *$(V, \vee)$ forms a commutative monoid with identity ?.*
4. *The substitution operation $\cdot$ distributes over $\vee$, that is $u \cdot (v \vee w) = (u \cdot v) \vee (u \cdot w)$*

Proof The proofs are straightforward computations, they have all been formalised in PVS. □

[3] Recall that the number zero is a one (i.e., a neutral element) for addition.

The preceding definition of variances generalises the variances in [Sch97]. There, Schroeder uses the finite set $\{+, -, *, ?\}$ as variances, denoting positive, negative, mixed, and unknown variance, respectively. The substitution and join operations are given by

$\cdot$	$?$	$+$	$-$	$*$
$?$	$?$	$?$	$?$	$?$
$+$	$?$	$+$	$-$	$*$
$-$	$?$	$-$	$+$	$*$
$*$	$?$	$*$	$*$	$*$

$\vee$	$?$	$+$	$-$	$*$
$?$	$?$	$+$	$-$	$*$
$+$	$+$	$+$	$*$	$*$
$-$	$-$	$*$	$-$	$*$
$*$	$*$	$*$	$*$	$*$

There is an algebra morphism from my variances to Schroeder's variances. It sends $(?, ?)$ to $?$, $(?, u_+)$ to $+$, $(u_-, ?)$ to $-$, and all other elements of V to $*$.

Next I present the variance checking algorithm. For a type judgement $\Xi \vdash \tau : \mathsf{Type}$ the variance checking algorithm assigns to every type variable $\alpha \in \Xi$ and to Self a well-formed variance from $\overline{V}$. I give the algorithm by annotating type judgements and the derivation system for types from Figure 2. Type judgements have now the form

$$\alpha_1 :: v_1, \ldots, \alpha_n :: v_n,\ \mathsf{Self} :: v \quad \vdash \quad \tau : \mathsf{Type}$$

for $v, v_1, \ldots, v_n \in \overline{V}$. It is the formal statement that τ is a type where the type variables α_i have variance v_i and Self has variance v in τ. Note that in the above judgement Self does not belong to the type variable context. It is just that I did not find a better way to incorporate the variance of Self into judgements. However, the notation is not completely misleading: Instead of making Self a special type (as I preferred in Definition 2.1) one could formalise Self as a distinct type variable. Indeed, the variance algorithm that follows treats Self exactly like a type variable.

In the following I assume that the type constructors in $\mathcal{C}$ are given with variance annotations as judgements

$$\vdash \mathsf{C} :: [v_1, \ldots, v_{\Bbbk}]$$

where $[v_1, \ldots, v_{\Bbbk}]$ is a finite list of length $\Bbbk$ over $\overline{V}$ and $\Bbbk$ is the arity of C. Type constants are given as $\vdash \mathsf{K} :: []$.

Consider the most basic well-formed type $\Xi \vdash \alpha : \mathsf{Type}$. The type variable α occurs with variance $(?, 0)$ and all other type variables from Ξ and Self do not occur (so they have variance $(?, ?)$). So the new rule for type variables is obviously

type variable

$$\frac{}{\alpha_1 :: ?, \ldots, \alpha_{i-1} :: ?,\ \alpha_i :: (?, 0),\ \alpha_{i+1} :: ?, \ldots, \alpha_n :: ?,\ \mathsf{Self} :: ? \ \vdash \ \alpha_i : \mathsf{Type}}$$

The rule for type constructors is the most difficult one, so let me postpone it for a moment. The other obvious rules are those for Self, $\mathbf{1}$, and $\mathbf{0}$. In the following rules I abbreviate an arbitrary type variable context $\alpha_1 :: u_1, \ldots, \alpha_n :: u_n$ by writing $\alpha_i :: u_i$.

Self

$$\frac{}{\alpha_i :: ?,\ \mathsf{Self} :: (?, 0) \ \vdash \ \mathsf{Self} : \mathsf{Type}}$$

unit type

$$\frac{}{\alpha_i :: ?,\ \mathsf{Self} :: ? \ \vdash \ \mathbf{1} : \mathsf{Type}}$$

empty type

$$\frac{}{\alpha_i :: ?,\ \mathsf{Self} :: ?\ \vdash\ \mathbf{0} : \mathsf{Type}}$$

Prop

$$\frac{}{\alpha_i :: ?,\ \mathsf{Self} :: ?\ \vdash\ \mathsf{Prop} : \mathsf{Type}}$$

The product and the coproduct do not change the variances. We only have to keep in mind, that in $\sigma \times \tau$ every type variable might occur in both types σ and τ, so we have to join the variances.

product

$$\frac{\alpha_i :: u_i,\ \mathsf{Self} :: u\ \vdash\ \tau : \mathsf{Type} \qquad \alpha_i :: v_i,\ \mathsf{Self} :: v\ \vdash\ \sigma : \mathsf{Type}}{\alpha_i :: (u_i \vee v_i),\ \mathsf{Self} :: (u \vee v)\ \vdash\ \tau \times \sigma : \mathsf{Type}}$$

coproduct

$$\frac{\alpha_i :: u_i,\ \mathsf{Self} :: u\ \vdash\ \tau : \mathsf{Type} \qquad \alpha_i :: v_i,\ \mathsf{Self} :: v\ \vdash\ \sigma : \mathsf{Type}}{\alpha_i :: (u_i \vee v_i),\ \mathsf{Self} :: (u \vee v)\ \vdash\ \tau + \sigma : \mathsf{Type}}$$

For the exponent $\sigma \Rightarrow \tau$ we also have to join the variances for each type variable. Thereby we have to keep in mind that for all type variables in σ the variances flip around and the nesting level is increased by one. Formally this is done by applying the substitution operation with $(1, ?)$. Note that $(1, ?) \cdot (u_-, u_+) = (u_+ + 1, u_- + 1)$.

exponent type

$$\frac{\alpha_i :: u_i,\ \mathsf{Self} :: u\ \vdash\ \sigma : \mathsf{Type} \qquad \alpha_i :: v_i,\ \mathsf{Self} :: v\ \vdash\ \tau : \mathsf{Type}}{\alpha_i :: ((1,?) \cdot u_i \vee v_i),\ \mathsf{Self} :: ((1,?) \cdot u \vee v)\ \vdash\ \sigma \Rightarrow \tau : \mathsf{Type}}$$

The rule for the type constructor is a generalised version of the rule for the exponent. Assume a type constructor $\mathsf{C} :: [u_1, \ldots, u_{\mathbb{k}}]$ in $\mathcal{C}$. Before joining the variances from all the types σ_j we have to apply the substitution operation with the variances u_j.

type constructor

$$\frac{\alpha_i :: v_i^1,\ \mathsf{Self} :: v^1 \vdash \sigma_1 : \mathsf{Type} \quad \cdots \quad \alpha_i :: v_i^{\mathbb{k}},\ \mathsf{Self} :: v^{\mathbb{k}} \vdash \sigma_{\mathbb{k}} : \mathsf{Type}}{\alpha_i :: \overline{v_i},\ \mathsf{Self} :: \overline{v}\ \vdash\ \mathsf{C}[\sigma_1, \ldots, \sigma_{\mathbb{k}}] : \mathsf{Type}}$$

where

$$\begin{aligned} \overline{v_i} &= u_1 \cdot v_i^1\ \vee\ \cdots\ \vee\ u_{\mathbb{k}} \cdot v_i^{\mathbb{k}} \\ \overline{v} &= u_1 \cdot v^1\ \vee\ \cdots\ \vee\ u_{\mathbb{k}} \cdot v^{\mathbb{k}} \end{aligned}$$

For completeness I also show the rules for weakening and substitution. Both rules are derivable.

type context weakening

$$\frac{\alpha_1 :: u_1, \ldots, \alpha_n :: u_n,\ \mathsf{Self} :: u\ \vdash\ \tau : \mathsf{Type}}{\alpha_1 :: u_1, \ldots, \alpha_n :: u_n,\ \alpha_{n+1} :: ?,\ \mathsf{Self} :: u\ \vdash\ \tau : \mathsf{Type}}$$

type substitution

$$\frac{\begin{array}{r}\alpha_1 :: u_1, \ldots, \alpha_n :: u_n, \alpha_{n+1} :: u_{n+1},\ \mathsf{Self} :: u \ \vdash\ \tau : \mathsf{Type} \\ \alpha_1 :: v_1, \ldots, \alpha_n :: v_n,\ \mathsf{Self} :: v \ \vdash\ \sigma : \mathsf{Type}\end{array}}{\alpha_1 :: \overline{u_1}, \ldots, \alpha_n :: \overline{u_n},\ \mathsf{Self} :: \overline{u} \ \vdash\ \tau[\sigma/\alpha_{n+1}] : \mathsf{Type}}$$

where

$$\begin{array}{rcl}\overline{u_i} & = & (u_{n+1} \cdot v_i) \vee u_i \\ \overline{u} & = & (u_{n+1} \cdot v) \vee u\end{array}$$

Now it is possible to give a simple and precise definition for the terms positive, negative and mixed variance.

Definition 3.3 Let τ be a type such that $\Gamma, \alpha :: (v_-, v_+),\ \mathsf{Self} :: (u_-, u_+) \vdash \tau : \mathsf{Type}$ is derivable. The type variable α occurs in τ with

- *strictly positive variance* if $v_- = ?$ and $v_+ \leq 0$
- *positive variance* if $v_- = ?$
- *negative variance* if $v_+ = ?$
- *mixed variance* if $v_- \neq ?$ and $v_+ \neq ?$

Similarly for Self and (u_-, u_+).

3.2. Variance Checking in CCSL

For CCSL it is more useful to have an algorithm that works top–down (instead of bottom–up like a derivation system). Let x be Self or a type variable. Then $\mathcal{V}_x(\tau)$ denotes the variance of x in τ. It is defined by induction on the structure of τ, see Figure 3.

Proposition 3.4 *The function $\mathcal{V}_x$ computes the variances as defined by the derivation system. More precisely, let τ be a type that contains the type variables $\alpha_1, \ldots, \alpha_n$. Then one can derive the following judgement:*

$$\alpha_i :: \mathcal{V}_{\alpha_i}(\tau),\ \mathsf{Self} :: \mathcal{V}_{\mathsf{Self}}(\tau) \quad \vdash \quad \tau : \mathsf{Type}$$

Further $\mathcal{V}_\beta(\tau) = (?, ?)$ if $\beta \notin \{\alpha_1, \ldots, \alpha_n\}$.

Proof By induction on the structure of types. The induction steps are immediate. □

The function $\mathcal{V}_x$ can be further optimised into a tail-recursive function by storing the variance of the type variable or of Self in a reference cell. This reference cell will be initialised with $?$, the neutral element for $\vee$. Then the equations from Figure 3 can be transformed into tail recursive form because $\cdot$ distributes over $\vee$ (Lemma 3.2 (4)). This tail-recursive variant of the equations in Figure 3 is used in the CCSL-compiler.

$$
\begin{array}{rcll}
\mathcal{V}_x(x) & = & (?, 0) & \\
\mathcal{V}_x(\alpha) & = & (?, ?) & \text{for } x \neq \alpha \\
\mathcal{V}_x(\mathbf{0}) & = & (?, ?) & \\
\mathcal{V}_x(\mathbf{1}) & = & (?, ?) & \\
\mathcal{V}_x(\mathsf{Prop}) & = & (?, ?) & \\
\mathcal{V}_x(\mathsf{Self}) & = & (?, ?) & \text{for } x \neq \mathsf{Self} \\
\mathcal{V}_x(\sigma_1 \times \sigma_2) & = & \mathcal{V}_x(\sigma_1) \vee \mathcal{V}_x(\sigma_2) & \\
\mathcal{V}_x(\sigma_1 + \sigma_2) & = & \mathcal{V}_x(\sigma_1) \vee \mathcal{V}_x(\sigma_2) & \\
\mathcal{V}_x(\sigma_1 \Rightarrow \sigma_2) & = & (1, ?) \cdot \mathcal{V}_x(\sigma_1) \;\vee\; \mathcal{V}_x(\sigma_2) & \\
\mathcal{V}_x(\mathsf{C}[\sigma_1, \ldots, \sigma_{\Bbbk}]) & = & u_1 \cdot \mathcal{V}_x(\sigma_1) \vee \cdots \vee u_{\Bbbk} \cdot \mathcal{V}_x(\sigma_{\Bbbk}) & \text{for } \vdash \mathsf{C} :: [u_1, \ldots, u_{\Bbbk}]
\end{array}
$$

Figure 3: A top down algorithm for computing the variance $\mathcal{V}_x(\tau)$ of type τ with respect to x

The variances presented here are not sufficient to distinguish extended cartesian functors from extended polynomial functors. The CCSL compiler uses an additional check to determine if a type corresponds to an extended cartesian functor.

Variances appear in CCSL specifications as annotations to type parameters. They restrict the variance of the annotated type parameter. Their concrete syntax is as follows.

$$
\begin{array}{lcl}
\textit{variance} & ::= & \texttt{POS} \\
 & | & \texttt{NEG} \\
 & | & \texttt{MIXED} \\
 & | & \underline{(}\ \textit{numberorquestion}\ \underline{,}\ \textit{numberorquestion}\ \underline{)} \\
\textit{numberorquestion} & ::= & \texttt{?} \\
 & | & \textit{number}
\end{array}
$$

In addition to the variances of Definition 3.1 the CCSL compiler recognises the keywords `POS`, `NEG` and `MIXED` as variances. These latter three variances denote an infinite nesting level and must be used for type constructors of ground signatures. The compiler extends the join and substitution of variances in the obvious way, for instance $\texttt{POS} \vee (?, 2) = \texttt{POS}$, $\texttt{POS} \vee (3, ?) = \texttt{MIXED}$, and $\texttt{POS} \cdot (3, ?) = \texttt{NEG}$. Variances given as a pair of numbers (or question marks) must be proper variances.

3.3. Semantics of Types

The discussion on negative variances at the beginning of this section showed that for the semantics of types some notions (for instance the `map`–combinator) differ with respect to the variance of type variables. Here I take the general approach and develop a semantics for types under the assumption that all type variables and Self occur with mixed variance. For type variables that occur only with positive or negative variance considerable simplifications are possible, see the lemmas below and the next subsection. For the semantics of types I thereby deliberately deviate from the CCSL compiler at the advantage of a clearer presentation.

In the standard case every type $\Xi \vdash \tau : \mathsf{Type}$ gives rise to an indexed collection of functors

$$\left(\llbracket\tau\rrbracket_{U_1^-,U_1^+,U_2^-,U_2^+,\ldots,U_n^-,U_n^+} \quad : \quad \mathbf{Set}^{\mathrm{op}} \times \mathbf{Set} \longrightarrow \mathbf{Set}\right)_{U_i^-,U_i^+ \in |\mathbf{Set}|}$$

where n is the number of type variables in Ξ. The indices $U_1^-, \ldots, U_n^+$ are sets for the interpretation of the type variables. The set U_i^- is used for the negative occurrences of α_i and U_i^+ for the positive ones. The arguments of the functor itself are used for the negative and positive occurrences of Self, respectively. In the following I abbreviate the list of indices as $U_i^{-/+}$ if there is no danger of confusion.

The standard case, to which I refer in the previous paragraph, is the case where for every type constructor of arity $\Bbbk$ there is an interpretation functor taking $2\Bbbk$ arguments. The arguments are doubled to separate them into positive and negative occurrences. The following definition deals with the standard case only, I discuss some abnormalities below.

Definition 3.5 (Interpretation of Types) Let $\mathcal{C}$ be a set of type constructors.

1. Let $\vdash \mathsf{C} :: [v_1, \ldots, v_\Bbbk]$ be a type constructor of arity $\Bbbk$ and let $(-)^\Bbbk$ denote the $\Bbbk$–fold product. An *interpretation* of C is a functor

$$\llbracket\mathsf{C}\rrbracket : (\mathbf{Set}^{\mathrm{op}} \times \mathbf{Set})^\Bbbk \longrightarrow \mathbf{Set}$$

   if, for arbitrary sets $V, V', U_1, \ldots, U_{2n}$, it has the following property

   - if the i-th argument of C has positive variance then $\llbracket\mathsf{C}\rrbracket$ is constant in its $(2i-1)$-th argument (which interprets the negative occurrences of the i-th argument):

$$\begin{aligned}\llbracket\mathsf{C}\rrbracket(U_1, \ldots, U_{2i-2}, V, U_{2i}, \ldots, U_{2n}) \quad &= \\ &\llbracket\mathsf{C}\rrbracket(U_1, \ldots, U_{2i-2}, V', U_{2i}, \ldots, U_{2n})\end{aligned}$$

   - if the i-th argument of C has negative variance then $\llbracket\mathsf{C}\rrbracket$ is constant in its $2i$-th argument (interpreting the positive occurrences of the i-th argument).
   - if the i-th argument of C has unknown variance then $\llbracket\mathsf{C}\rrbracket$ is constant in both its $(2i-1)$-th and its $2i$-th argument.

2. Let $\alpha_1, \ldots, \alpha_n \vdash \tau : \mathsf{Type}$ be a type with type constructors from $\mathcal{C}$. Assume that for every $\mathsf{C} \in \mathcal{C}$ we have an interpretation $\llbracket\mathsf{C}\rrbracket$. The interpretation of τ is defined by induction on the structure of types.

$$\begin{aligned}\llbracket\alpha_i\rrbracket_{U_1^-,\ldots,U_n^+}(Y, X) \quad &= \quad U_i^+ \\ \llbracket\mathsf{Self}\rrbracket_{U_1^-,\ldots,U_n^+}(Y, X) \quad &= \quad X\end{aligned}$$

$$
\begin{array}{rcl}
[\![\mathbf{1}]\!]_{U_1^-,\ldots,U_n^+}(Y,X) & = & \mathbf{1} \quad = \{*\} \\
[\![\mathbf{0}]\!]_{U_1^-,\ldots,U_n^+}(Y,X) & = & \mathbf{0} \quad = \emptyset \\
[\![\mathsf{Prop}]\!]_{U_1^-,\ldots,U_n^+}(Y,X) & = & \mathsf{bool} \quad = \{\bot,\top\} \\
[\![\sigma_1+\sigma_2]\!]_{U_1^-,\ldots,U_n^+}(Y,X) & = & [\![\sigma_1]\!]_{U_1^-,\ldots,U_n^+}(Y,X) \quad + \quad [\![\sigma_2]\!]_{U_1^-,\ldots,U_n^+}(Y,X) \\
[\![\sigma_1\times\sigma_2]\!]_{U_1^-,\ldots,U_n^+}(Y,X) & = & [\![\sigma_1]\!]_{U_1^-,\ldots,U_n^+}(Y,X) \quad \times \quad [\![\sigma_2]\!]_{U_1^-,\ldots,U_n^+}(Y,X) \\
[\![\sigma_1\Rightarrow\sigma_2]\!]_{U_1^-,\ldots,U_n^+}(Y,X) & = & [\![\sigma_1]\!]_{U_1^+,U_1^-,U_2^+,U_2^-,\ldots,U_n^+,U_n^-}(X,Y) \quad \Rightarrow \\
& & \qquad [\![\sigma_2]\!]_{U_1^-,U_1^+,U_2^-,U_2^+,\ldots,U_n^-,U_n^+}(Y,X) \\
[\![\mathsf{C}[\sigma_1,\ldots,\sigma_k]]\!]_{U_1^-,\ldots,U_n^+}(Y,X) & = & [\![\mathsf{C}]\!]\Big([\![\sigma_1]\!]_{U_1^+,U_1^-,U_2^+,U_2^-,\ldots,U_n^+,U_n^-}(X,Y), \\
& & \quad [\![\sigma_1]\!]_{U_1^-,U_1^+,U_2^-,U_2^+,\ldots,U_n^-,U_n^+}(Y,X), \\
& & \quad \vdots \\
& & \quad [\![\sigma_n]\!]_{U_1^+,U_1^-,U_2^+,U_2^-,\ldots,U_n^+,U_n^-}(X,Y), \\
& & \quad [\![\sigma_n]\!]_{U_1^-,U_1^+,U_2^-,U_2^+,\ldots,U_n^-,U_n^+}(Y,X)\Big)
\end{array}
$$

The morphism part is defined in the obvious way (by replacing Y and X with suitable functions f^- and f^+).

Observe how the indices and the arguments for positive and negative occurrences are flipped around on the left hand side of the exponent and in the arguments for the functor $[\![\mathsf{C}]\!]$.

The interpretation of a type τ containing n type variables can be extended (in the obvious way) to a functor taking $2n+2$ arguments. Its morphism part is denoted with $[\![\tau]\!]_{\overline{g}}(f^-,f^+)$ for a suitable list of functions $\overline{g}=g_1^-,g_1^+,\ldots,g_n^-,g_n^+$.

Under certain circumstances (occurring in conjunction with iterated specifications, see Section 8) for some type constructor C only the object mapping of an interpretation functor might be available (i.e., there is no morphism part for $[\![\mathsf{C}]\!]$). In this case the interpretation $[\![\tau]\!]$ degrades to an indexed collection of mappings $|\mathbf{Set}|\times|\mathbf{Set}|\longrightarrow|\mathbf{Set}|$.

In even more obscure cases the interpretation $[\![\mathsf{C}]\!](\cdots U_i^-,U_i^+\cdots)$ of a type constructor is only defined if the respective arguments for positive and negative occurrences are equal, that is if $U_i^-=U_i^+$ for all i. In this case the interpretation $[\![\tau]\!]_{\cdots U_i^-,U_i^+\cdots}(Y,X)$ is only defined for $Y=X$ and $U_i^-=U_i^+$.

If Self occurs in τ with positive variance then the first argument is ignored for every functor in the collection ($[\![\tau]\!]$). In this case the interpretation functors factor through $\pi_2:\mathbf{Set}^{\mathrm{op}}\times\mathbf{Set}\longrightarrow\mathbf{Set}$. Similarly the second argument is ignored if Self occurs in τ with negative variance. Some of the indices $U_i^{-/+}$ are ignored if not all type variables occur with mixed variance in τ.

Proposition 3.6 *Let $\alpha_1,\ldots,\alpha_n\vdash\tau:\mathsf{Type}$ be a type as before. Let V and V' be arbitrary sets.*

- *If* Self *occurs in τ with positive variance, then*

$$[\![\tau]\!]_{U_1^-,\ldots,U_n^+}(V,X) \quad = \quad [\![\tau]\!]_{U_1^-,\ldots,U_n^+}(V',X)$$

- *If* Self *occurs in τ with negative variance, we have*

$$[\![\tau]\!]_{U_1^-,\ldots,U_n^+}(Y,V) \quad = \quad [\![\tau]\!]_{U_1^-,\ldots,U_n^+}(Y,V')$$

- *Assume the type variable α_i has positive variance in τ. Then*

$$[\![\tau]\!]_{U_1^-,\ldots,V,U_i^+,\ldots,U_n^+}(Y,X) \quad = \quad [\![\tau]\!]_{U_1^-,\ldots,V',U_i^+,\ldots,U_n^+}(Y,X)$$

- *And if α_i has negative variance then*

$$[\![\tau]\!]_{U_1^-,\ldots,U_i^-,V,\ldots,U_n^+}(Y,X) \quad = \quad [\![\tau]\!]_{U_1^-,\ldots,U_i^-,V',\ldots,U_n^+}(Y,X)$$

Proof By induction on the structure of types. □

For a type σ that does not contain Self every functor in the interpretation $[\![\sigma]\!]$ is a constant functor (i.e., the result does not depend on the arguments). Therefore I write for the interpretation of such types $[\![\tau]\!]_{U_1^-,\ldots,U_n^+}$ instead of $[\![\tau]\!]_{U_1^-,\ldots,U_n^+}(Y,X)$. The indexing with the interpretations for the type variables looks complicated but is in fact a rather simple idea. After one got used to this concept the complicated notation distracts from the interesting points. In following sections I will therefore sometimes drop these indexes and simply write $[\![\tau]\!]$ for a fixed interpretation of the type variables and of Self.

3.4. Separation of Variances

In this subsection I explain how the CCSL compiler deals with type variables with mixed variance. It is possible to skip this subsection and return later, if questions about this issue remain open.

Ideally the CCSL compiler should implement the semantics of types of Definition 3.5 literally. It should do a variance analysis on the input and separate all type variables into their positive and negative occurrences. However, there are the following problems with such a rigorous approach: First, very often there is no type variable with mixed variance in a specification. For this common case Proposition 3.6 shows that many of the indices (that interpret a particular variance of a type variable) are superfluous. These superfluous items would probably confuse the users of CCSL. Almost certainly PVS would get confused. The second problem is that it is not possible to separate variances in the semantics of CCSL's logic (described in Section 6).

For these reasons the CCSL compiler generally uses only one interpretation for any type variable. If the type variable occurs only positively or only negatively it is otherwise correctly handled according to Definition 3.5. For a type expression that contains a type variable α_i

with mixed variance the morphism part of the semantics stays undefined. Further, for the object part of the semantics the compiler assumes that the arguments for positive and negative occurrences of α_i are equal, that is that $U_i^+ = U_i^-$. If a type variable with mixed variance poses problems then the compiler issues a warning. For data type specifications and for class specifications without assertions the user can easily separate himself the type variable with mixed variance into a positive and a negative one.

The special type Self is always handled in a correct way (even if it occurs with mixed variance). This ensures that the CCSL compiler generates the correct notion of coalgebra morphism for class signatures as long as for all type constructors the morphism part of their semantics is defined.

Predicate and relation lifting (defined in Subsection 5.1 below) are treated differently. For type variables that occur only positively or only negatively the compiler uses one parameter predicate for predicate lifting. So for those type variables there is no separation. However, a type variable that has mixed variance is separated into its positive and its negative positions. For such a type variable the compiler introduces two predicates in the generated code. This guarantees that the definitions for predicate lifting and invariant are correctly generated for all class specifications. Relation lifting is treated analogously. Therefore the generated notions of relation lifting and bisimulation are correctly generated as long as the greatest bisimulation does exists.

3.5. Classification of Types

Types are classified according to the variance of Self. Via the interpretation of types there is a correspondence with the classification of functors into polynomial, extended polynomial, and higher-order polynomial functors.

Definition 3.7 Let $\mathcal{C}$ be a set of type constructors with variance annotations and let τ be a type over $\mathcal{C}$.

1. τ is a *constant type* if $\mathcal{V}_{\mathsf{Self}}(\tau) = ?$.

2. τ is a *polynomial type* if Self occurs only at strictly covariant positions in τ, that is if $\mathcal{V}_{\mathsf{Self}}(\tau) = (?, u_+)$ for $u_+ \leq 0$.

3. τ is an *extended polynomial type* if $\mathcal{V}_{\mathsf{Self}}(\tau) = (u_-, u_+)$ for $u_- \leq 1$ and $u_+ \leq 0$.

4. τ is a *higher-order polynomial type*, if it is not extended polynomial.

5. τ is a *constructor type* in case $\tau = \sigma \Rightarrow \mathsf{Self}$ and σ is a polynomial type.

6. τ is a *constant constructor type* if $\tau = \sigma \Rightarrow \mathsf{Self}$ is a constructor type and if additionally σ is a constant type.

7. τ is a *method type* if $\tau = (\mathsf{Self} \times \sigma) \Rightarrow \rho$.

8. A method type $\tau = (\mathsf{Self} \times \sigma) \Rightarrow \rho$ is a *polynomial/extended-polynomial/higher-order polynomial method type* if $\sigma \Rightarrow \rho$ is a polynomial/extended-polynomial/higher-order polynomial type.

Note that via the isomorphism $\tau \times \mathbf{1} \cong \tau$ and the associativity of $\times$ also $\mathsf{Self} \Rightarrow \rho$ and $\mathsf{Self} \times \sigma_1 \times \cdots \times \sigma_n \Rightarrow \rho$ are method types for arbitrary $\sigma, \sigma_1, \ldots, \sigma_n, \rho$. Via the isomorphism $\mathbf{1} \Rightarrow \tau \cong \tau$ the type Self is a constant constructor type.

With the formalisation of variances it is now possible to define the term *binary method* precisely. A method of type $\mathsf{Self} \times \sigma \Rightarrow \rho$ is called a binary method, if Self occurs with negative variance in $\sigma \Rightarrow \rho$. Note that in this case $\mathsf{Self} \times \sigma \Rightarrow \rho$ cannot be a polynomial method type. However not every method of extended-polynomial or higher-order polynomial type is also a binary method. For CCSL the classification of methods into binary and unary methods is not really important. Here only the classification of Definition 3.7 is significant.

The use of the attributes polynomial, extended polynomial and higher-order to classify types is justified by the following proposition.

Proposition 3.8 *Assume that $\mathcal{C}$ contains only type constants and let $\alpha_1, \ldots, \alpha_n \vdash \tau : \mathsf{Type}$ be an arbitrary type over $\mathcal{C}$. Let $F = [\![\tau]\!]_{U_1^-, \ldots, U_n^+}$ be the interpretation functor for fixed sets $U_1^-, \ldots, U_n^+$. The type τ is a*

$$\left\{\begin{array}{l} \textit{constant} \\ \textit{polynomial} \\ \textit{extended-polynomial} \\ \textit{higher-order} \end{array}\right\} \textit{type precisely if } F \textit{ is a } \left\{\begin{array}{l} \textit{constant} \\ \textit{polynomial} \\ \textit{extended-polynomial} \\ \textit{higher-order} \end{array}\right\} \textit{functor.}$$

Proof By induction on the structure of τ. □

The restriction in the preceding proposition that τ does not contain any nonconstant type constructor is rather severe. The weak requirements for the interpretation of nonconstant type constructors do not allow one to derive anything in general. For the CCSL compiler the situation is slightly better: The compiler can keep track of type constructors that stem from a processed class or data type specification. Types that contain such type constructors give rise to *data functors* in the sense of [Hen99], but see also [HJ97, Röß99]. I discuss this issue in Section 8 on iterated specifications.

4. Ground Signatures

In this section I define (polymorphic) *ground signatures*. Ground signatures are used to declare types, functions, and constants that are available in the specification environment. For instance one usually expects that the natural numbers $\mathbb{N}$ with addition and multiplication are available. In CCSL ground signatures also serve a second purpose: They make iterated specifications (Section 8) possible.

The logic that I define in Section 6 places a few restriction on ground signatures and their models. For the semantics of *behavioural equality* and of modal operators every type constructor (of arity greater than zero) in the ground signature must be equipped with two special constants: predicate and relation lifting. This requirement is captured with the notion of *proper ground signatures.*

Definition 4.1 (Ground Signature)

- A ground signature Ω consists of
  - a set $|\Omega|$ of type constructors with variance annotations,
  - an indexed set (Ω_σ) of constant symbols for each constant type σ over $|\Omega|$. A constant symbol $f \in \Omega_\sigma$ is given as a (term) judgement $\Xi \mid \emptyset \vdash f : \sigma$ where σ is a constant type such that $\Xi \vdash \sigma : \mathsf{Type}$ is derivable.
- A ground signature is called *plain* if the set $|\Omega|$ contains only type constants (i.e., type constructors of arity zero).
- A ground signature is called *proper* if the set of constant symbols contains at least the following two symbols for every type constructor $\mathsf{C} \in |\Omega|$ of arity $\Bbbk$

$$\begin{aligned}\alpha_1, \dots, \alpha_\Bbbk \mid \emptyset \vdash \mathsf{Pred}_\mathsf{C} : {} & (\alpha_1 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_1 \Rightarrow \mathsf{Prop}) \Rightarrow \\ & (\alpha_2 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_2 \Rightarrow \mathsf{Prop}) \Rightarrow \cdots \Rightarrow \\ & (\alpha_\Bbbk \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_\Bbbk \Rightarrow \mathsf{Prop}) \Rightarrow \mathsf{C}[\alpha_1, \dots, \alpha_\Bbbk] \Rightarrow \mathsf{Prop}\end{aligned}$$

$$\begin{aligned}\alpha_1, \dots, \alpha_\Bbbk,\ \beta_1, \dots \beta_\Bbbk \mid \emptyset \vdash \mathsf{Rel}_\mathsf{C} : {} & (\alpha_1 \times \beta_1 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_1 \times \beta_1 \Rightarrow \mathsf{Prop}) \Rightarrow \\ & (\alpha_2 \times \beta_2 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_2 \times \beta_2 \Rightarrow \mathsf{Prop}) \Rightarrow \cdots \Rightarrow (\alpha_\Bbbk \times \beta_\Bbbk \Rightarrow \mathsf{Prop}) \Rightarrow \\ & (\alpha_\Bbbk \times \beta_\Bbbk \Rightarrow \mathsf{Prop}) \Rightarrow \mathsf{C}[\alpha_1, \dots, \alpha_\Bbbk] \times \mathsf{C}[\beta_1, \dots, \beta_\Bbbk] \Rightarrow \mathsf{Prop}\end{aligned}$$

  The constant Pred_C is the *predicate lifting* of C and Rel_C is its *relation lifting.* Note that both take $2\Bbbk$ arguments. This is necessary to separate co– and contravariant occurrences.

Example 4.2 The CCSL compiler starts with an empty ground signature. Before the user file is read the CCSL *prelude* is processed (see also Subsection 10.8). This prelude is a valid CCSL string, which is hard wired into the compiler. So the user file is opened with a (proper) ground signature Ω_P that contains the declarations from the prelude.

The signature Ω_P contains the following type constructors[4]

$$\begin{aligned} \vdash \mathsf{list} &: [(?, 0)] \\ \vdash \mathsf{Lift} &: [(?, 0)] \\ \vdash \mathsf{Coproduct} &: [(?, 0); (?, 0)] \\ \vdash \mathsf{Unit} &: [] \\ \vdash \mathsf{EmptyType} &: [] \end{aligned}$$

[4]The type constructor EmptyType is not defined for the ISABELLE back end.

The intended semantics (which is ensured by the CCSL compiler in cooperation with the target theorem prover) is that list constructs the finite lists over a given type, $\mathsf{Lift}[\alpha]$ is an abbreviation for $\alpha + \mathbf{1}$ and $\mathsf{Coproduct}[\alpha, \beta] = \alpha + \beta$. The type constructors Unit and $\mathsf{EmptyType}$ give the two special types $\mathbf{1}$ and $\mathbf{0}$, respectively. The type constructor Lift is used in CCSL to model partial functions. The coproduct is in the prelude because there is no special syntax for the coproduct of types in CCSL.

Besides the predicate and relation lifting of the three type constructors the ground signature Ω_P contains the following constants:[5]

$$
\begin{aligned}
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{null} : \mathsf{list}[\alpha] \\
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{cons} : \alpha \times \mathsf{list}[\alpha] \Rightarrow \mathsf{list}[\alpha] \\
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{null?} : \mathsf{list}[\alpha] \Rightarrow \mathsf{Prop} \\
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{cons?} : \mathsf{list}[\alpha] \Rightarrow \mathsf{Prop} \\[1ex]
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{bot} : \mathsf{Lift}[\alpha] \\
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{up} : \alpha \Rightarrow \mathsf{Lift}[\alpha] \\
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{bot?} : \mathsf{Lift}[\alpha] \Rightarrow \mathsf{Prop} \\
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{up?} : \mathsf{Lift}[\alpha] \Rightarrow \mathsf{Prop} \\[1ex]
\alpha : \mathsf{Type}, \beta : \mathsf{Type} \mid \emptyset &\vdash \mathsf{in1} : \alpha \Rightarrow \mathsf{Coproduct}[\alpha, \beta] \\
\alpha : \mathsf{Type}, \beta : \mathsf{Type} \mid \emptyset &\vdash \mathsf{in2} : \beta \Rightarrow \mathsf{Coproduct}[\alpha, \beta] \\
\alpha : \mathsf{Type}, \beta : \mathsf{Type} \mid \emptyset &\vdash \mathsf{in1?} : \mathsf{Coproduct}[\alpha, \beta] \Rightarrow \mathsf{Prop} \\
\alpha : \mathsf{Type}, \beta : \mathsf{Type} \mid \emptyset &\vdash \mathsf{in2?} : \mathsf{Coproduct}[\alpha, \beta] \Rightarrow \mathsf{Prop} \\[1ex]
\emptyset \mid \emptyset &\vdash \mathsf{unit} : \mathsf{Unit} \\[1ex]
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{empty\_fun} : \mathsf{EmptyType} \Rightarrow \alpha
\end{aligned}
$$

The constants null, cons, unit, up, $\mathsf{in1}$, $\mathsf{in2}$, and unit are the expected constructors and injections. The other constants are recogniser predicates. The predicate $\mathsf{cons?}$, for instance, is true for a list l if $l = \mathsf{cons}(a, l')$ for some a and l'. Similarly for the other recognisers.

In applications it is nice to have also accessor functions, like $\mathsf{car} : \mathsf{List}[\alpha] \rightharpoonup \alpha$, which delivers the head for nonempty lists. Accessors are usually partial functions (depicted as partial arrow $\rightharpoonup$). The type theory that I developed here deals (for simplicity) only with total functions. So formally car cannot be incorporated as a constant of type $\mathsf{List}[\alpha] \Rightarrow \alpha$ without leading to inconsistencies. However, the CCSL compiler is a bit more relaxed and declares also the following accessors:

$$
\begin{aligned}
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{car} : \mathsf{list}[\alpha] \Rightarrow \alpha \\
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{cdr} : \mathsf{list}[\alpha] \Rightarrow \mathsf{list}[\alpha]
\end{aligned}
$$

[5]In PVS identifiers can contain question marks. The same applies to CCSL. When generating output for ISABELLE the names for the recognisers are is_null, is_cons, and so on. Further for ISABELLE the native ISABELLE constructor names Nil and Cons are used for lists.

$$\alpha : \mathsf{Type} \mid \emptyset \vdash \mathsf{down} : \mathsf{Lift}[\alpha] \Rightarrow \alpha$$

$$\alpha : \mathsf{Type}, \beta : \mathsf{Type} \mid \emptyset \vdash \mathsf{out1} : \mathsf{Coproduct}[\alpha, \beta] \Rightarrow \alpha$$
$$\alpha : \mathsf{Type}, \beta : \mathsf{Type} \mid \emptyset \vdash \mathsf{out2} : \mathsf{Coproduct}[\alpha, \beta] \Rightarrow \beta$$

The CCSL type checker treats accessors (erroneously) as total functions. There are no consistency problems for the following reasons: PVS has a type theory with predicate subtypes. There the accessor car is a total function, which has the subtype of nonempty lists as domain. The CCSL compiler uses this correctly typed function as semantics of car. The theorem prover ISABELLE/HOL has no predicate subtypes but its semantics is based in the HOL tradition [GM93] on an universe of nonempty sets. Consequently ISABELLE/HOL does not allow for empty types and there is the special constant arbitrary that inhabits every type.[6] If the CCSL compiler generates output for ISABELLE/HOL then as semantics for car it takes a function that returns arbitrary for the empty list. ■

A *model* of the ground signature contains functors, (polymorphic) functions and constants that can be used to interpret the syntactic symbols in the ground signature. For proper models of proper ground signatures I require two basic properties for the interpretation of predicate and relation lifting. Namely that predicate lifting commutes with truth and that relation lifting commutes with equality.

Definition 4.3 (Model of Ground Signature)

- Let Ω be a ground signature. A model of it consists of
  - the object part of an interpretation functor $[\![\mathsf{C}]\!]$ for every type constructor $\mathsf{C} \in |\Omega|$,
  - an indexed family of functions or constants

    $$[\![f]\!]_{U_1, U_2, \ldots U_n} : [\![\sigma]\!]_{U_1, U_1, U_2, U_2, \ldots, U_n, U_n}$$

    for every constant symbol $\alpha_1, \ldots, \alpha_n \mid \emptyset \vdash f : \sigma$ in Ω.
- A model of a proper ground signature is called *proper* if all the interpretations $[\![\mathsf{C}]\!]$ are functors and if additionally the following condition is satisfied: For all type constructors $\mathsf{C} \in |\Omega|$ of arity $\Bbbk$ it holds that

$$[\![\mathsf{Pred}_{\mathsf{C}}]\!]_{U_1, \ldots, U_\Bbbk}(\top_{U_1}, \top_{U_1}, \ldots, \top_{U_\Bbbk}, \top_{U_\Bbbk}) = \top_{[\![\mathsf{C}]\!](U_1, U_1, \ldots, U_\Bbbk, U_\Bbbk)}$$
$$[\![\mathsf{Rel}_{\mathsf{C}}]\!]_{U_1, \ldots, U_\Bbbk, U_1, \ldots, U_\Bbbk}(\mathrm{Eq}(U_1), \mathrm{Eq}(U_1), \ldots, \mathrm{Eq}(U_\Bbbk), \mathrm{Eq}(U_\Bbbk)) = \mathrm{Eq}([\![\mathsf{C}]\!](U_1, U_1, \ldots, U_\Bbbk, U_\Bbbk))$$

The two conditions on proper models of ground signatures ensure that also in the presence of type constructors predicate lifting commutes with truth and relation lifting with

[6] The constant arbitrary is neither in [NPW02a] nor in [NPW02b] described. See the file `src/HOL/HOL.thy` in the ISABELLE distribution.

equality (see Lemma 5.9 on page 39 below). In turn this implies that truth is an invariant and equality is a bisimulation (see Proposition 5.11 on page 40). On type constants K these two conditions have the following effect: $\mathsf{Pred}_{\mathsf{K}} = \top_{[\![\mathsf{K}]\!]}$ and $\mathsf{Rel}_{\mathsf{K}} = \mathrm{Eq}([\![\mathsf{K}]\!])$. This matches the treatment of constants in predicate and relation lifting for higher-order polynomial functors.

Example 4.4 The model for Ω_P maps list to the functor that yields the initial algebra for the functor $F^U_{\mathsf{list}}(X) = X \times U + \mathbf{1}$ for every argument U. For Lift, $\mathsf{Coproduct}$, Unit, $\mathsf{EmptyType}$ and the constant symbols it takes the obvious constructions. The predicate and relation lifting for $\mathsf{Coproduct}$ is given by $+_{\mathrm{P}}$ and $+_{\mathrm{R}}$, respectively. Also the liftings for Lift are obvious: It is $\mathsf{Pred}_{\mathsf{Lift}}(P) = P +_{\mathrm{P}} \top_{\mathbf{1}}$ and $\mathsf{Rel}_{\mathsf{Lift}}(R) = R +_{\mathrm{R}} \mathrm{Eq}(\mathbf{1})$. The liftings for list are defined by induction and follow the general description in Section 8 below (I ignore the contravariant arguments):

$$
\begin{aligned}
[\![\mathsf{Pred}_{\mathsf{list}}]\!](P)(\mathsf{null}) &= \top \\
[\![\mathsf{Pred}_{\mathsf{list}}]\!](P)(\mathsf{cons}(a,l)) &= P(a) \;\wedge\; [\![\mathsf{Pred}_{\mathsf{list}}]\!](P)(l)
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathsf{Rel}_{\mathsf{list}}]\!](R)(\mathsf{null},\mathsf{null}) &= \top \\
[\![\mathsf{Rel}_{\mathsf{list}}]\!](R)(\mathsf{cons}(a,l),\mathsf{null}) &= \bot \\
[\![\mathsf{Rel}_{\mathsf{list}}]\!](R)(\mathsf{null},\mathsf{cons}(a,l)) &= \bot \\
[\![\mathsf{Rel}_{\mathsf{list}}]\!](R)(\mathsf{cons}(a,l),\mathsf{cons}(a',l')) &= R(a,a') \;\wedge\; [\![\mathsf{Rel}_{\mathsf{list}}]\!](R)(l,l') \qquad \blacksquare
\end{aligned}
$$

The ground signature describes types and operations that are available in the environment. So usually a model for it is provided (automatically) by the environment. The CCSL compiler for instance assumes that all symbols in the ground signature are defined in the target theorem prover. Therefore it treats symbols from the ground signature literally: their semantics is the same symbol again. In the following sections I assume a proper ground signature Ω and a proper model $\mathcal{M}_\Omega$ of it. Types that appear will be types over $|\Omega|$ and their interpretation will be the interpretation with respect to $\mathcal{M}_\Omega$.

4.1. Ground Signatures in CCSL

Ground signatures in CCSL are actually ground signature extensions. As explained before the CCSL compiler keeps a current ground signature while parsing the source file. A ground signature declaration extends this current ground signature with type constructors and constants. These items must be defined either in the ground signature itself or in the target theorem prover. The concrete grammar is as follows.

groundsignature	::=	`BEGIN` *identifier* [*parameterlist*] `:` `GROUNDSIGNATURE` ⦃ *importing* ⦄ ⦃ *signaturesection* ⦄ `END` *identifier*
parameterlist	::=	<u>[</u> *parameters* ⦃ `,` *parameters* ⦄ <u>]</u>
parameters	::=	*identifier* ⦃ `,` *identifier* ⦄ `:` [*variance*] `TYPE`

A Ground signature (extension) starts with the keyword `BEGIN`, followed by the name of the ground signature, an optional (global) type parameter list and the keyword `GROUNDSIGNATURE`. The type parameters build a type variable context for all declarations in the ground signature. In CCSL it is necessary to declare type variables as type parameters because there is no special syntax to distinguish type variables from other identifiers.

Any type parameter can get a variance annotation. The variance annotation is compulsory for all type parameters if the ground signature declares a type constructor without giving its definition.

Importing clauses are explained in Subsection 10.4. For ground signatures they are necessary, if the items that are declared in the ground signature require extra theories to be loaded in the target theorem prover.

The body of a ground signature contains an arbitrary number of sections, declaring or defining type constructors and (possibly) polymorphic constants or functions.

signaturesection	::=	*typedef*
	|	*signaturesymbolsection* [`;`]
typedef	::=	`TYPE` *identifier* [*parameterlist*] [`=` *type*]
signaturesymbolsection	::=	`CONSTANT` *termdef* ⦃ `;` *termdef* ⦄
termdef	::=	*idorinfix* [*parameterlist*] `:` *type* [*formula*]
idorinfix	::=	(*infix_operator*)
	|	*identifier*

Each item in a ground signature can declare additional (local) type parameters in a separate parameter list. The type variable context of an item is given by the concatenation of the global parameter list with the local parameter list of that item. The local type parameters are syntactic sugar. They are convenient if only a few items in one ground signature require additional type parameters.

Type constructors are introduced with the keyword `TYPE`. The arity of the new type constructor is defined as the number of the declared (global and local) type parameters. If the optional type expression is present, then the type constructor is defined in CCSL. In this case the CCSL compiler derives the variances annotations, the predicate and relation lifting, and the morphism component of (the semantics of) the type constructor.

If the type constructor is not defined (the optional type expression is left out) then all type parameter must have variance annotations to allow the compiler to derive the variance of the type constructor. For such declarations the compiler assumes that the type constructor and its liftings are defined in the target theorem prover as functions of an appropriate type. These functions are assumed to fulfil the conditions for proper models of ground signatures. Their names are derived from the name of the type constructor by appending the suffixes "Pred", "Rel", and "Map".

Constants and functions are introduced with the keyword `CONSTANT`. One can also declare infix operators, see Subsection 10.5 (on page 101) for the details. The constants can be

```
Begin SetSig [ U : Neg Type ] : GroundSignature
  Type set = [U -> bool]
  Constant
    empty : set[U]
    empty = Lambda(t : U) : false;

    intersect : [set[U], set[U] -> set[U]];
    (+)       : [set[U], set[U] -> set[U]];

    ( * ) [V : Type] : [set[U], set[V] -> set[[U,V]]]
    (S * R)(u,v) = ( S u And R v );
End SetSig
```

Figure 4: A rudimentary ground signature extension for power sets

defined in CCSL by providing a definition in higher-order logic in the syntax formulae (see Subsection 6.4). If a formula is present, then it must be an equation and the left hand side of the equation must be the constant to be defined, possibly applied to some variables.

Figure 4 contains as example a rudimentary ground signature extension for the (contravariant) powerset type constructor. In addition to the type constructor it declares a constant for the empty set, a function for intersection, the infix operator $+$ for union, and the operator $*$ for the cartesian product. For demonstration purposes I defined some of the constants.

There is also a more lightweight syntax for declaring single types and constants, see Subsection 10.7 on anonymous ground signatures (on page 102 below).

5. Coalgebraic Class Signatures

This section introduces the structural aspects of coalgebraic specification: signatures and signature models. The following definitions follow very closely what is implemented in the CCSL compiler. The definitions are optimised for practicability — and not for succinctness. Recall from Definition 3.7 that method types are types of the form $\mathsf{Self} \times \tau \Rightarrow \tau'$ and constant constructor types have the form $\sigma \Rightarrow \mathsf{Self}$, where Self must not occur in σ.

Definition 5.1 (Coalgebraic Class Signature) Assume a set of type constructors $\mathcal{C}$ (possibly stemming from a ground signature). A *coalgebraic class signature* is a pair $\langle \Sigma_M, \Sigma_C \rangle$ where Σ_M is a finite set of method declarations $m_i : \tau_i$, for method types τ_i, and Σ_C is a finite set of constructor declarations $c_i : \sigma_i$ for constant constructor types σ_i. The set of type variables occurring in the τ_i and the σ_i are the *type parameters* of the signature.

Example 5.2 (Queue Signature) Consider a *first–in–first–out* (*FIFO*) queue. It supports two operations, one for enqueueing elements (put) and one for removing elements from the head (top). Removing the first element from a queue is a partial operation, which fails if the queue is empty. Therefore the signature Σ_{Queue} contains the following method two declarations

$$\begin{aligned} \mathsf{put} &: \mathsf{Self} \times \alpha \longrightarrow \mathsf{Self} \\ \mathsf{top} &: \mathsf{Self} \longrightarrow \mathsf{Lift}[\alpha \times \mathsf{Self}] \end{aligned}$$

Additionally, there is the constructor declaration

$$\mathsf{new} \;:\; \mathsf{Self}$$

For any element x of Self we have either $\mathsf{top}(x) = \mathsf{bot}$ (signalling an empty queue) or $\mathsf{top}(x) = \mathsf{up}(a, x')$, where a is the first element of the queue and x' is the successor state of x with a removed. Instead of the simple constructor new, one could also use a constructor $\mathsf{new\_from\_list} : \mathsf{list}[\alpha] \to \mathsf{Self}$ that takes the elements of a list to initialise the queue.

This example of queues is the running example of this and the following section. The example has been fully worked out in CCSL and PVS.[7] ■

In the following I need the term of a *subsignature*. Subsignatures will be used for inheritance, for the visibility modifiers `PUBLIC` and `PRIVATE`, and for the modal operators. The following definition might be a bit surprising on first sight, because it completely neglects constructor declarations. I motivate this decision in the general discussion about inheritance in Subsection 9.1 (on page 90) below.

Definition 5.3 (Subsignature) Assume a ground signature Ω and let $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$ be a coalgebraic class signature. A class signature $\Sigma' = \langle \Sigma'_M, \Sigma'_C \rangle$ is a *subsignature* of Σ, denoted by $\Sigma' \leq \Sigma$, if $\Sigma'_M \subseteq \Sigma_M$.

Coalgebraic class signatures are classified according to the method types they contain. If a signature contains a higher-order (respectively extended polynomial) method type, then I refer to it as a signature with a higher-order method (with an extended polynomial method, respectively).

To define the semantics of class signatures it is necessary to extract type information from the signature. Assume a coalgebraic class signature Σ in the following. Let $m : \tau \in \Sigma_M$ be a method declaration, so that τ is a method type. Define the operation $\mathcal{T}_M$ as follows

$$\mathcal{T}_M(m) \quad = \quad \begin{cases} \rho & \text{if} \quad \tau = \;\mathsf{Self} \Rightarrow \rho \\ (\sigma_1 \times \cdots \times \sigma_n) \Rightarrow \rho & \text{if} \quad \tau = \;(\mathsf{Self} \times \sigma_1 \times \cdots \times \sigma_n) \Rightarrow \rho \end{cases}$$

And if $c : \tau \in \Sigma_C$ is a constructor declaration (for a constructor type τ) set

$$\mathcal{T}_C(c) \quad = \quad \begin{cases} \sigma & \text{if} \quad \tau = \sigma \Rightarrow \mathsf{Self} \\ \mathbf{1} & \text{if} \quad \tau = \mathsf{Self} \end{cases}$$

[7] The complete sources are available in the material distributed with my PhD, seehttp://wwwtcs.inf.tu-dresden.de/ tews/PhD/.

Let $m_1, \ldots, m_k$ be the method declarations of Σ. The combined method type of Σ, denoted by τ_Σ, is defined as

$$\tau_\Sigma \quad = \quad \mathcal{T}_M(m_1) \times \cdots \times \mathcal{T}_M(m_k)$$

If Σ_M is empty then $\tau_\Sigma = \mathbf{1}$. The combined constructor type for Σ, denoted by σ_Σ, is defined by

$$\sigma_\Sigma \quad = \quad \mathcal{T}_C(c_1) + \cdots + \mathcal{T}_C(c_l)$$

under the assumption, that $\Sigma_C = \{c_1, \ldots, c_l\}$. If Σ_C is empty then $\sigma_\Sigma = \mathbf{0}$.

Note that σ_Σ is always a constant type. For τ_Σ we have that

$$\tau_\Sigma \text{ is a} \left\{ \begin{array}{l} \text{polynomial} \\ \text{extended-polynomial} \\ \text{higher-order} \end{array} \right\} \text{type if } \Sigma \text{ contains} \left\{ \begin{array}{l} \text{only polynomial methods} \\ \text{no higher-order methods} \\ \text{a higher-order method.} \end{array} \right.$$

Only practical considerations are responsible for allowing several method and constructor declarations in one class signature. Any class signature Σ is equivalent to a signature Σ' that contains exactly one method declaration of type $\mathsf{Self} \Rightarrow \tau_\Sigma$ and one constructor declaration of type $\sigma_\Sigma \Rightarrow \mathsf{Self}$. So one could equivalently define the term coalgebraic class signature as a pair $\langle \tau, \sigma \rangle$ of an arbitrary type τ and a constant type σ. However, in applications it is nice to have different names for different operations.

From Proposition 3.8 we can deduce that $[\![\tau_\Sigma]\!]$ is a polynomial functor if Σ contains only polynomial method declarations and the set of type constructors $\mathcal{C}$ contains only type constants. Similarly for extended polynomial functors and higher-order polynomial functors.

However, it is much more interesting to build class signatures which use type constructors of arity greater than zero in their method declarations, like in the example of queues. Let C be such a type constructor and let Σ be a coalgebraic class signature that makes use of C. In this case the functor $[\![\tau_\Sigma]\!]$ depends in a nontrivial way on the semantics $[\![\mathsf{C}]\!]$ of the type constructor C (which comes along with a model of the ground signature). Because there is no restriction on $[\![\mathsf{C}]\!]$ one cannot say much about the properties of $[\![\tau_\Sigma]\!]$. In a typical application of CCSL all (nonconstant) type constructors stem from abstract data type specifications (to be dealt with in Section 7) and from coalgebraic class specifications. In this case $[\![\tau_\Sigma]\!]$ is a data functor in the sense of [Hen99, Röß00b], provided a technical condition on the variances of type parameters is fulfilled; see Section 8.

For a fixed interpretation of the type parameters a model of a coalgebraic class signature Σ is a triple consisting of the state space of the model, a coalgebra for the functor $[\![\tau_\Sigma]\!]$ that interprets the method declarations, and an algebra for the functor $[\![\sigma_\Sigma]\!]$ that is used for the constructor declarations. A complete model is then a collection of such triples indexed by the interpretations for the type parameters.

Definition 5.4 (Model of Class Signature) Let Σ be a coalgebraic class signature with n type parameters $\alpha_1, \ldots, \alpha_n$. A *model* for Σ consists of an indexed collection of triples $\big(\langle X, c, a\rangle_{U_1,\ldots,U_n}\big)_{U_i \in |\mathbf{Set}|}$ where, for each interpretation $U_1, \ldots, U_n$ of the type parameters, X is a set (the state space), c is a coalgebra, and a is an algebra as in

$$[\![\sigma_\Sigma]\!]_{U_1,U_1,U_2,U_2,\ldots,U_n,U_n} \xrightarrow{\quad a \quad} X \xrightarrow{\quad c \quad} [\![\tau_\Sigma]\!]_{U_1,U_1,U_2,U_2,\ldots,U_n,U_n}(X, X)$$

Remark 5.5 The preceding definition does not distinguish between co– and contravariant occurrences of the type variables and of Self. Therefore, a proper model $\mathcal{M}_\Omega$ of the ground signature is not strictly necessary here. It is sufficient if $\mathcal{M}_\Omega$ defines $\llbracket \mathsf{C} \rrbracket$ for those argument vectors whose respective co– and contravariant positions are equal.

In case $\mathcal{M}_\Omega$ is proper one can form the following category of signature models for every interpretation $U_1, \ldots, U_n$ of the type parameters: Objects are triples $\langle X, c, a\rangle$ and $\langle Y, d, b\rangle$. Morphisms are $\llbracket \tau_\Sigma \rrbracket$ coalgebra morphism that commute with the constructors:

$$
\begin{tikzcd}
 & X \arrow[r, "c"] \arrow[dd, "f"] & \llbracket\tau_\Sigma\rrbracket(X,X) \arrow[dr, "\llbracket\tau_\Sigma\rrbracket(X,f)"] & \\
\llbracket\sigma_\Sigma\rrbracket \arrow[ur, "a"] \arrow[dr, "b"'] & & & \llbracket\tau_\Sigma\rrbracket(X,Y) \\
 & Y \arrow[r, "d"] & \llbracket\tau_\Sigma\rrbracket(Y,Y) \arrow[ur, "\llbracket\tau_\Sigma\rrbracket(f,Y)"'] &
\end{tikzcd}
$$

Instead of the left triangle one could require that the constructor algebras are behaviourally equivalent, that is, that[8] $\forall u \in \llbracket\sigma_\Sigma\rrbracket \,.\, (a\, u) \;{}_c\!\underline{\leftrightarrow}_d\; (b\, u)$.

Example 5.6 (Model for Queue) The signature Σ_{Queue} from Example 5.2 has one type parameter α. A model for this signature consists of a set X_U for every set U, a coalgebra $c_U : X_U \longrightarrow (U \Rightarrow X_U) \times \mathsf{Lift}[U \times X_U]$ and an algebra $a_U : \mathbf{1} \longrightarrow X_U$. To describe such a model let $\mathbb{N}^+$ be the natural numbers including infinity ∞ and take

$$X_U \quad = \quad \{(n, f) \mid n \in \mathbb{N}^+ \wedge f : {<}n \longrightarrow U\}$$

where ${<}n \;=\; \{i \mid i < n\}$ is the initial segment of $\mathbb{N}^+$ below n.[9] So a state in X_U is a pair $\langle n, f\rangle$, consisting of the number n of elements in the queue and a function f that gives the elements in the queue for arguments less than n. I set

$$
\begin{array}{rcl}
c(n,f) & = & \left\{\begin{array}{ll}
\big(\lambda u : U \,.\, (1,\ \lambda i \,.\, u),\quad \mathsf{bot}\big) & \text{if } n = 0 \\
\big(\lambda u : U \,.\, (n, f),\quad \mathsf{up}(f(0),\ \ (\infty,\ \lambda i \,.\, f(i+1)))\big) & \text{if } n = \infty \\
\big(\lambda u : U \,.\, (n+1,\ \lambda i \,.\ \text{if } i = n \text{ then } u \text{ else } f(i)), & \multirow{2}{*}{otherwise} \\
\quad \mathsf{up}(f(0),\ (n-1, \lambda n \,.\, f(n+1)))\big) &
\end{array}\right. \\
\mathsf{new} & = & (0, f_\emptyset)
\end{array}
$$

where $f_\emptyset$ is the empty function $\emptyset \longrightarrow U$. It is easy to see, that the coalgebra c obeys the dependent typing of X_U.[10] Note that at this stage there is nothing that restricts the behaviour of these methods: There exist models of the Queue signature that contain only infinite queues and there are also models that do not resemble FIFO queues at all. ■

[8] By anticipating Definition 5.7 this condition is equivalent with $\mathrm{Rel}(\llbracket\sigma_\Sigma \Rightarrow \mathsf{Self}\rrbracket)({}_c\!\underline{\leftrightarrow}_d)(a, b)$.

[9] Note that one cannot use $\mathbb{N}^+ \times (\mathbb{N} \Rightarrow U)$ for X_U, because then it is impossible to define new for $U = \emptyset$.

[10] The corresponding proofs have all been done in PVS. This was a quite difficult task for PVS, it revealed six bugs, see problem reports number 483–486 on http://pvs.csl.sri.com/cgi-bin/pvs/pvs-bug-list/.

There exist class signatures which *do not* have a model. This is because I explicitly allow the empty set as interpretation for type parameters. An example is a class signature $\Sigma_\emptyset$ with one method declaration $m : \mathsf{Self} \Rightarrow \alpha$ and one constructor declaration $c : \mathbf{1} \Rightarrow \mathsf{Self}$. There is no set X with two functions $\mathbf{1} \longrightarrow X \longrightarrow \emptyset$, so there is no model of $\Sigma_\emptyset$ if the type parameter α is interpreted by the empty set. With slight changes the signature $\Sigma_\emptyset$ can be made consistent: either the method declaration is changed to $m : \mathsf{Self} \Rightarrow (\alpha + \mathbf{1})$ or the constructor declaration is changed to $c : \alpha \Rightarrow \mathsf{Self}$.

There are mainly two possibilities to ensure that every signature has a model. First one could restrict the interpretation of the type parameters to nonempty sets. This restriction takes effect if one uses CCSL together with ISABELLE/HOL, because there are no empty types in ISABELLE. A second possibility is to require that all constructors are parametrised by a tuple of all type parameters. This requirement could be combined with an emptiness analysis of the method types.

In the last part of this subsection I explain how a model of a class signature Σ gives rise to an interpretation of the method declarations of Σ and all its subsignatures.

Let $\mathcal{M} = \langle X, c, a \rangle$ be a model of an arbitrary class signature Σ for a fixed interpretation of its type parameters. For every method declaration $m : \tau$ there is a projection

$$\pi_m \ : \ [\![\tau_\Sigma]\!](X, X) = [\![\cdots \times \mathcal{T}_M(m) \times \cdots]\!](X, X) \ \longrightarrow \ [\![\mathcal{T}_M(m)]\!]\,(X, X)$$

which extends to a natural transformation $\tau_m : [\![\tau_\Sigma]\!] \Longrightarrow [\![\mathcal{T}_M(m)]\!]$. Similarly for every constructor declaration $e : \tau$ there is an injection

$$\kappa_e \ : \ [\![\mathcal{T}_C(e)]\!] \longrightarrow [\![\cdots + \mathcal{T}_C(e) + \cdots]\!] = [\![\sigma_\Sigma]\!]$$

extending to a natural transformation $\kappa_e : [\![\mathcal{T}_C(e)]\!] \Longrightarrow [\![\sigma_\Sigma]\!]$.

Via these projections and injections the model $\mathcal{M}$ gives rise to an interpretation of the method and constructor declarations. Let $m : \mathsf{Self} \times \sigma_1 \times \cdots \times \sigma_n \Rightarrow \rho$ be a method declaration in Σ_M and $e : \sigma \Rightarrow \mathsf{Self}$ be a constructor declaration in Σ_C. Then

$$\begin{array}{rcl} [\![m]\!]^{\mathcal{M}} & = & \lambda x : X, p_1 : [\![\sigma_1]\!], \ldots, p_n : [\![\sigma_n]\!] \,.\, (\pi_m(c\,x))(p_1, \ldots, p_n) \\ [\![e]\!]^{\mathcal{M}} & = & a \circ \kappa_e \end{array}$$

Assume now a subsignature Σ' of Σ and let $\Sigma'_M = \{m_1, \ldots, m_n\}$. The natural transformation

$$\langle \pi_{m_1}, \ldots, \pi_{m_n} \rangle \ : \ [\![\tau_\Sigma]\!] \ \Longrightarrow \ [\![\tau_{\Sigma'}]\!]$$

defined by component wise pairing is called the *subsignature projection* and denoted by $\pi_{\Sigma'}$ (where Σ is left implicit). It is easy to show that $\pi_{\Sigma'}$ gives rise to a functor that maps $[\![\tau_\Sigma]\!]$ coalgebras to $[\![\tau_{\Sigma'}]\!]$ coalgebras. Its object part is given by post composition; for any coalgebra $c : X \longrightarrow [\![\tau_\Sigma]\!]$ there is the following coalgebra for Σ'

$$\pi_{\Sigma'} \circ c \ = \ \langle \pi_{m_1}, \ldots, \pi_{m_n} \rangle \circ c \ : \ X \longrightarrow [\![\tau_{\Sigma'}]\!]$$

This way a model $\mathcal{M}$ for Σ provides an interpretation for the method declarations of all subsignatures of Σ.

5.1. Invariants and Bisimulations

This subsection defines the notions of invariant and bisimulation for CCSL class signatures. The definitions rely on predicate and relation lifting, see [HJ98] and Chapter 3 of [Tew02b]. For CCSL it is necessary to extend predicate and relation lifting for type variables and nonconstant type constructors. Let me explain how this works for predicate lifting. Let τ be a type with type variables $\alpha_1, \ldots, \alpha_n$. Fix an interpretation $U_1, \ldots, U_n$ such that U_i is used for the positive and the negative occurrences of α_i in τ. For each type variable the predicate lifting $\mathrm{Pred}(\llbracket\tau\rrbracket)$ gets two additional parameter predicates $P_i^-, P_i^+ \subseteq U_i$. These predicates are used for the negative and positive occurrences of α_i in τ, respectively.[11] For the type constructors that occur in τ one simply uses the predicate lifting that is supplied by the model of the ground signature.

Definition 5.7 (Predicate and Relation Lifting) Let $\alpha_1, \ldots, \alpha_n \vdash \tau : \mathsf{Type}$ be a type over an arbitrary proper ground signature Ω. Fix a model $\mathcal{M}$ of Ω, an interpretation $U_1, \ldots, U_n$ for the type variables, and an interpretation X for Self.

1. The *predicate lifting* of the interpretation of τ (with respect to $\mathcal{M}$), denoted by $\mathrm{Pred}(\llbracket\tau\rrbracket)$, is an operation that takes $2n+2$ predicates as arguments (two predicates for each type variable and two for Self) and yields a predicate on $\llbracket\tau\rrbracket_{U_1,U_1,\ldots,U_n,U_n}(X,X)$. Let $\overline{P} = P_1^-, P_1^+, \ldots, P_{n+1}^-, P_{n+1}^+$ be a tuple of predicates such that $P_i^-, P_i^+ \subseteq U_i$ for $i \leq n$ and $P_{n+1}^-, P_{n+1}^+ \subseteq X$. Let $\overline{P^-} = P_1^+, P_1^-, \ldots, P_i^+, P_i^-, \ldots, P_{n+1}^+, P_{n+1}^-$ denote the tuple in which the predicates are pairwise swapped (to exchange the predicates for positive and negative occurrences). The predicate lifting $\mathrm{Pred}(\llbracket\tau\rrbracket)$ is an extension of the predicate lifting for higher-order polynomial functors and is defined by induction on the structure of the interpretation functor $\llbracket\tau\rrbracket$.

$$
\begin{array}{rcl}
\mathrm{Pred}(\llbracket\alpha_i\rrbracket)(\overline{P}) & = & P_i^+ \\
\mathrm{Pred}(\llbracket\mathsf{Self}\rrbracket)(\overline{P}) & = & P_{n+1}^+ \\
\mathrm{Pred}(\llbracket\mathsf{Prop}\rrbracket)(\overline{P}) & = & \top_{\mathsf{bool}} \quad = \quad \{\top, \bot\} \\
\mathrm{Pred}(\llbracket\mathbf{1}\rrbracket)(\overline{P}) & = & \top_{\mathbf{1}} \quad = \quad \{*\} \\
\mathrm{Pred}(\llbracket\mathbf{0}\rrbracket)(\overline{P}) & = & \top_{\mathbf{0}} \quad = \quad \emptyset \\
\mathrm{Pred}(\llbracket\sigma + \tau\rrbracket)(\overline{P}) & = & \mathrm{Pred}(\llbracket\sigma\rrbracket)(\overline{P}) \ +_{\mathrm{P}} \mathrm{Pred}(\llbracket\tau\rrbracket)(\overline{P}) \\
& = & \big\{(\kappa_1\, x) \mid \mathrm{Pred}(\llbracket\sigma\rrbracket)(\overline{P})(x)\big\} \ \cup \\
& & \quad \big\{(\kappa_2\, y) \mid \mathrm{Pred}(\llbracket\tau\rrbracket)(\overline{P})(y)\big\} \\
\mathrm{Pred}(\llbracket\sigma \times \tau\rrbracket)(\overline{P}) & = & \mathrm{Pred}(\llbracket\sigma\rrbracket)(\overline{P}) \ \times_{\mathrm{P}} \mathrm{Pred}(\llbracket\tau\rrbracket)(\overline{P}) \\
& = & \big\{(x, y) \mid \mathrm{Pred}(\llbracket\sigma\rrbracket)(\overline{P})(x) \ \wedge \ \mathrm{Pred}(\llbracket\tau\rrbracket)(\overline{P})(y)\big\}
\end{array}
$$

[11] One could generalise predicate lifting (and also relation lifting) to take argument predicates $P_i^- \subseteq U_i^-$, $P_i^+ \subseteq U_i^+$, where U_i^- interprets the negative occurrences of α_i and U_i^+ the positive ones. However, predicate lifting is only used within one model where $U_i^- = U_i^+$.

$$\begin{aligned}
\mathrm{Pred}(\llbracket \sigma \Rightarrow \tau \rrbracket)(\overline{P}) &= \mathrm{Pred}(\llbracket \sigma \rrbracket)(\overline{P^-}) \Rightarrow_{\mathrm{P}} \mathrm{Pred}(\llbracket \tau \rrbracket)(\overline{P}) \\
&= \big\{ f \mid \forall x \in \llbracket \sigma \rrbracket \,.\, \mathrm{Pred}(\llbracket \sigma \rrbracket)(\overline{P^-})(x) \\
&\qquad\qquad \text{implies } \mathrm{Pred}(\llbracket \tau \rrbracket)(\overline{P})(f\,x) \big\} \\
\mathrm{Pred}(\llbracket \mathsf{C}[\sigma_1, \ldots, \sigma_{\Bbbk}] \rrbracket)(\overline{P}) &= \llbracket \mathsf{Pred}_{\mathsf{C}} \rrbracket_{A_1,\ldots,A_n} \big(\mathrm{Pred}(\llbracket \sigma_1 \rrbracket)(\overline{P^-}), \mathrm{Pred}(\llbracket \sigma_1 \rrbracket)(\overline{P}), \\
&\qquad \vdots \\
&\qquad \mathrm{Pred}(\llbracket \sigma_{\Bbbk} \rrbracket)(\overline{P^-}), \mathrm{Pred}(\llbracket \sigma_{\Bbbk} \rrbracket)(\overline{P}) \big)
\end{aligned}$$

where, in the case for the constructor, $A_i = \llbracket \sigma_i \rrbracket_{\overline{U}}(X, X)$.

2. Fix now a second interpretation $V_1, \ldots, V_n, Y$ for the type variables α_i and for Self and let $\llbracket \tau \rrbracket_{\overline{V}}$ and $\llbracket \tau \rrbracket_{\overline{U}}$ denote the interpretation of τ with respect to the V_i and U_i, respectively. Let $\overline{R} = R_1^-, R_1^+, \ldots, R_{n+1}^-, R_{n+1}^+$ be a tuple of relations such that $R_i^-, R_i^+ \subseteq U_i \times V_i$ and $R_{n+1}^-, R_{n+1}^+ \subseteq X \times Y$. The *relation lifting* of τ (with respect to the ground signature model $\mathcal{M}$), denoted by $\mathrm{Rel}(\llbracket \tau \rrbracket)$, is an operation that maps the tuple $\overline{R}$ to a relation on $\llbracket \tau \rrbracket_{\overline{U}}(X, X) \times \llbracket \tau \rrbracket_{\overline{V}}(Y, Y)$. It is defined as an extension of the relation lifting for higher-order polynomial functors by induction on the structure of the interpretation of τ:

$$\begin{aligned}
\mathrm{Rel}(\llbracket \alpha_i \rrbracket)(\overline{R}) &= R_i^+ \\
\mathrm{Rel}(\llbracket \mathsf{Self} \rrbracket)(\overline{R}) &= R_{n+1}^+ \\
\mathrm{Rel}(\llbracket \mathsf{Prop} \rrbracket)(\overline{R}) &= \mathrm{Eq}(\mathsf{bool}) \qquad = \{(a, b) \mid a = b\} \\
\mathrm{Rel}(\llbracket \mathbf{1} \rrbracket)(\overline{R}) &= \mathrm{Eq}(\mathbf{1}) \qquad = \{(*, *)\} \\
\mathrm{Rel}(\llbracket \mathbf{0} \rrbracket)(\overline{R}) &= \mathrm{Eq}(\mathbf{0}) \qquad = \emptyset \\
\mathrm{Rel}(\llbracket \sigma + \tau \rrbracket)(\overline{R}) &= \mathrm{Rel}(\llbracket \sigma \rrbracket)(\overline{R}) +_{\mathrm{R}} \mathrm{Rel}(\llbracket \tau \rrbracket)(\overline{R}) \\
&= \big\{ (\kappa_1\, x_1, \kappa_1\, y_1) \mid \mathrm{Rel}(\llbracket \sigma \rrbracket)(\overline{R})(x_1, y_1) \big\} \;\cup \\
&\qquad \big\{ (\kappa_2\, x_2, \kappa_2\, y_2) \mid \mathrm{Rel}(\llbracket \tau \rrbracket)(\overline{R})(x_2, y_2) \big\} \\
\mathrm{Rel}(\llbracket \sigma \times \tau \rrbracket)(\overline{R}) &= \mathrm{Rel}(\llbracket \sigma \rrbracket)(\overline{R}) \times_{\mathrm{R}} \mathrm{Rel}(\llbracket \tau \rrbracket)(\overline{R}) \\
&= \big\{ ((x_1, x_2), (y_1, y_2)) \mid \\
&\qquad \mathrm{Rel}(\llbracket \sigma \rrbracket)(\overline{R})(x_1, y_1) \;\wedge\; \mathrm{Rel}(\llbracket \tau \rrbracket)(\overline{R})(x_2, y_2) \big\} \\
\mathrm{Rel}(\llbracket \sigma \Rightarrow \tau \rrbracket)(\overline{R}) &= \mathrm{Rel}(\llbracket \sigma \rrbracket)(\overline{R^-}) \Rightarrow_{\mathrm{R}} \mathrm{Rel}(\llbracket \tau \rrbracket)(\overline{R}) \\
&= \big\{ (g, h) \mid \forall x \in \llbracket \sigma \rrbracket_{\overline{U}}(X, X),\; y \in \llbracket \sigma \rrbracket_{\overline{V}}(Y, Y) \,. \\
&\qquad \mathrm{Rel}(\llbracket \sigma \rrbracket)(\overline{R^-})(x, y) \text{ implies } \mathrm{Rel}(\llbracket \tau \rrbracket)(\overline{R})(g\,x, h\,y) \big\} \\
\mathrm{Rel}(\llbracket \mathsf{C}[\sigma_1, \ldots, \sigma_{\Bbbk}] \rrbracket)(\overline{R}) &= \llbracket \mathsf{Rel}_{\mathsf{C}} \rrbracket_{\overline{A}, \overline{B}} \big(\mathrm{Rel}(\llbracket \sigma_1 \rrbracket)(\overline{R^-}), \mathrm{Rel}(\llbracket \sigma_1 \rrbracket)(\overline{R}), \\
&\qquad \vdots \\
&\qquad \mathrm{Rel}(\llbracket \sigma_{\Bbbk} \rrbracket)(\overline{R^-}), \mathrm{Rel}(\llbracket \sigma_{\Bbbk} \rrbracket)(\overline{R}) \big)
\end{aligned}$$

where, in the case for the constructor, $\overline{A}$ stands for the list $[\![\sigma_i]\!]_{\overline{U}}(X, X)$ and $\overline{B}$ for $[\![\sigma_i]\!]_{\overline{V}}(Y, Y)$.

Remark 5.8 The liftings of the preceding definition are sometimes referred to as *full liftings* in contrast to the lifting described in [HJ98] and Chapter 3 of [Tew02b] that neglect type variables. The structure for type variables is not needed for the definition of bisimulation and invariant in this subsection, but it will be needed to give semantics to iterated specifications in Section 8.

In this presentation I prefer to consider predicate and relation lifting (and also bisimulations and invariants) completely as semantic notions. One could equivalently define predicate and relation lifting for types as expressions in the logic of CCSL.

The requirement of a proper ground signature in the preceding definition cannot be dropped. If for one type constructor C from Ω its predicate lifting $\mathsf{Pred}_{\mathsf{C}}$ (respectively its relation lifting $\mathsf{Rel}_{\mathsf{C}}$) is not available, then predicate lifting (relation lifting) for types over Ω cannot be defined.

It is possible to adopt the results about predicate and relation lifting of [Tew02b] to the full liftings of the preceding definition. For most of the results one has to assume that the liftings of the involved type constructors have appropriate properties. For the commutation of the liftings with truth and equality the required properties are built-in into the notion of a proper model of a ground signature.

Lemma 5.9 *Let $\mathcal{M}$ be a proper model of a proper ground signature Ω. Then predicate lifting and relation lifting (with respect to $\mathcal{M}$) commute with truth and equality, respectively:*

$$\begin{array}{rcl}\mathrm{Pred}([\![\tau]\!])(\top_{U_1}, \top_{U_1}, \ldots, \top_n, \top_n, \top_X, \top_X) & = & \top_{[\![\tau]\!]} \\ \mathrm{Rel}([\![\tau]\!])(\mathrm{Eq}(U_1), \mathrm{Eq}(U_1), \ldots, \mathrm{Eq}(U_n), \mathrm{Eq}(U_n), \mathrm{Eq}(X), \mathrm{Eq}(X)) & = & \mathrm{Eq}([\![\tau]\!])\end{array}$$

where τ is an arbitrary type over Ω with n type variables and $U_1, \ldots, U_n, X$ is a (fixed) interpretation of the type variables and of Self.

Proof By induction on the structure of τ. □

The notions of bisimulation and invariant are defined using predicate and relation lifting. Bisimulations in CCSL are a generalisation of Hermida/Jacobs bisimulations from Definition 3.3.3 in [Tew02b]. However, the invariants in CCSL are the strong invariants from Subsection 3.4.6 of [Tew02b].

Technically a bisimulation should relate two models, so a bisimulation should be a family of relations indexed by the interpretation of the type parameters. This generality is rarely needed: When working with bisimulations one is usually in a context where the interpretation of the type parameters is fixed. Therefore I prefer to define the notions of bisimulation and invariant only for a fixed interpretation of the type parameters.

Definition 5.10 (Bisimulation & Invariant) Let Σ be a coalgebraic signature with n type parameters over a proper ground signature Ω. Assume that $\mathcal{M} = \langle X, c, a\rangle$ and $\mathcal{M}' = \langle Y, d, b\rangle$ are models of Σ for a fixed interpretation U_i of the type parameters.

1. A predicate $P \subseteq X$ is an *invariant* for $\mathcal{M}$ if for all $x \in X$

$$P(x) \quad \text{implies} \quad \mathrm{Pred}([\![\tau_\Sigma]\!])(\top_{U_1}, \top_{U_1}, \top_{U_2}, \top_{U_2}, \ldots, \top_X, P)(c(x))$$

2. A predicate $P \subseteq X$ *holds initially* in $\mathcal{M}$ if

$$\mathrm{Pred}([\![\sigma_\Sigma \Rightarrow X]\!])(\top_{U_1}, \top_{U_1}, \top_{U_2}, \top_{U_2}, \ldots, \top_X, P)(a)$$

3. A relation $R \subseteq X \times Y$ is a *bisimulation* for $\mathcal{M}$ and $\mathcal{M}'$ if for all $x \in X, y \in Y$

$$R(x,y) \quad \text{implies} \quad \mathrm{Rel}([\![\tau_\Sigma]\!])(\mathrm{Eq}(U_1), \mathrm{Eq}(U_1), \mathrm{Eq}(U_2), \mathrm{Eq}(U_2), \ldots, R, R)(c(x), d(y))$$

Note that the notions of bisimulation and invariant are only defined for proper ground signatures. The union of all bisimulations on one model $\mathcal{M}$ for a fixed interpretation of the type parameters is denoted by $\underline{\leftrightarrow}_{\mathcal{M}}$. The fact whether the relation $\underline{\leftrightarrow}_{\mathcal{M}}$ is again a bisimulation depends both on the class signature Σ *and* on the model of the ground signature Ω. If, for instance, the class signature contains only polynomial methods and the ground signature is plain, then bisimilarity $\underline{\leftrightarrow}_{\mathcal{M}}$ is a bisimulation for all proper models of Ω. I discuss the case of a non-plain ground signature in Section 8.

For proper models of proper ground signatures it is possible to infer some trivial properties for bisimulations and invariants.

Proposition 5.11 *Let $\mathcal{M}_\Omega$ be a proper model of a proper ground signature Ω. Then for any model $\mathcal{M}_\Sigma = \langle X, c, a\rangle$ of an arbitrary coalgebraic signature Σ the truth predicate $\top_X$ is an invariant for $\mathcal{M}_\Sigma$ and the equality relation* $\mathrm{Eq}(X)$ *is a bisimulation for $\mathcal{M}_\Sigma$.*

Proof Apply Lemma 5.9. □

5.2. Class Signatures in CCSL

In CCSL class signatures are part of class specifications. The main difference to Definition 5.1 is, that in CCSL type parameters must be declared in advance. An identifier which is neither in the ground signature nor declared as a type parameter yields an error. Figure 5 shows the signature of the queues from Example 5.2 in CCSL. The grammar for class specifications is as follows:

```
Begin Queue[ A : Type ] : ClassSpec
  Method
    put : [Self, A] -> Self;
    top : Self -> Lift[[A, Self]];

  Constructor
    new : Self;
End Queue
```

Figure 5: The queue signature in CCSL syntax

classspec	::=	BEGIN *identifier* [*parameterlist*] : [FINAL] CLASSSPEC
		{| *importing* |} {| *classsection* |}
		END *identifier*
classsection	::=	*inheritsection*
	|	[*visibility*] *attributesection* [;]
	|	[*visibility*] *methodsection* [;]
	|	*definitionsection*
	|	*classconstructorsection* [;]
	|	*assertionsection*
	|	*creationsection*
	|	*theoremsection*
	|	*requestsection* [;]
visibility	::=	PUBLIC
	|	PRIVATE

Every specification in CCSL starts with the keyword `BEGIN` followed by the name of the specification and the type parameters in brackets. Variance annotations for type parameters are treated like type constraints. If they are present the compiler compares them with the internally computed variances. The compiler reports an error if the variance annotations are too restrictive (i.e., giving a `POS` annotation for a type parameter that has mixed variance). To facilitate aggregation the CCSL compiler generates an axiomatic model for every class specification. The model is either the final one, or an arbitrarily chosen loose one, depending on whether the keyword `FINAL` is present.

A class specification can start with importing clauses (see Subsection 10.4 on page 99 below). Importing clauses in class specifications are only needed under special circumstances.

The body of a specification consists of a number of sections. The attribute section, the method section, and the section of class constructors constitute the class signature. The definition section defines definitional extensions. For the inherit section see Subsection 9.1. The assertion section, the creation section, and the theorem section are explained in Section 6.

The assertion section and the creation section contain the axioms of the specification. The theorem section has no influence on the semantics of the specification. It allows the user to exploit the CCSL compiler for translating formulae (which are hopefully provable theorems) into the logic of the target theorem prover. Finally, the request section is there to request the generation of relation liftings for particular types, see Subsection 10.3.

The attribute section provides a form of syntactic sugar to ease the modelling of the state of the objects as a record. Formally an attribute declaration is a method declaration with the additional requirement that the type is of the form $\mathsf{Self} \times \sigma \Rightarrow \tau$ where σ and τ are constant types. For every attribute declaration $\mathsf{a} : \mathsf{Self} \times \sigma \Rightarrow \tau$ the compiler adds a method declaration $\mathsf{set\_a} : \mathsf{Self} \times \sigma \times \tau \Rightarrow \mathsf{Self}$ to the signature. The intention is that $\mathsf{set\_a}$ is the update operation that can be used to change the value of the attribute a. Further, the CCSL compiler generates a number of assertions that describe the behaviour of the update method, see Section 6.4.

The modifiers `PUBLIC` and `PRIVATE` classify the methods and attributes into two disjoint sets. If no modifier is present then the methods or attributes are `PUBLIC` by default. The intention is that private methods should not be visible from outside of the class. However, to enforce this one would need existential types (compare [MP88, AC96]), which are not present in the theorem provers PVS and ISABELLE. The CCSL compiler uses the public/private classification to derive two signatures from every specification. The first one contains all methods and attributes, the second one is the subsignature containing only the public attributes and methods. All relevant definitions and lemmas are generated twice, first for the full signature, then for public subsignature. This way the user can prove that two models are bisimilar with respect to the public interface, thus ignoring the private methods and attributes. This can be used, for instance, to prove special refinements [JT01].

The grammar for the sections of attributes, methods, constructors, and definitions is as follows.

attributesection	::=	`ATTRIBUTE` *member* ⦃ ; *member* ⦄
methodsection	::=	`METHOD` *member* ⦃ ; *member* ⦄
member	::=	*identifier* : *type* `->` *type*
definitionsection	::=	`DEFINING` *member formula* ; ⦃ *member formula* ; ⦄
classconstructorsection	::=	`CONSTRUCTOR` *classconstructor* ⦃ ; *classconstructor* ⦄
classconstructor	::=	*identifier* : *type*
	|	*identifier* : *type* `->` *type*

Each of these sections contains a list of attributes, methods, constructors, or definitions. The CCSL compiler checks that the declared attributes and methods have method types according to Definition 3.7. The constructors must have constant constructor types. The identifiers declared as attributes and methods (together with inherited attributes and methods) form the set of method declarations Σ_M and the class constructor declarations form the set Σ_C.

In the definition section one can define additional methods in terms of other methods

```
QueueInterface[Self : Type , A : Type] : Theory
Begin
  Importing Lift[[A , Self]]

  QueueSignature : Type =
    [# put : [[Self , A] -> Self],
       top : [Self -> Lift[[A , Self]]]
    #]
  QueueConstructors : Type = [#     new : Self     #]
End QueueInterface
```

Figure 6: PVS translation of the queue signature

(and attributes). The defining formula must be an equation according to the syntax described in Subsection 6.4. One can use the full power of CCSL's logic with the following exceptions: Modal operators of the current class and behavioural equality on a type that contains Self are not allowed in definitional extensions. The reason for this restriction is that the CCSL compiler outputs definitions at a position at which these notions are not yet defined for the current class.

For the semantics of the class signatures the CCSL compiler deviates slightly from Definition 5.4 in two points. First, the functors that give the semantics of types are not present in the output. The CCSL compiler uses type expressions in the logic of PVS or ISABELLE instead. The arguments of the functors become additional theory parameters or type variables.

The second point is that a model of a signature consists of two labelled records of functions (instead of a coalgebra/algebra pair)[12]. The first record contains for every method declaration $m_i : \mathsf{Self} \times \sigma \Rightarrow \rho$ a function $[\![\mathsf{Self} \times \sigma \Rightarrow \rho]\!](X, X)$ where X is an additional type parameter that works as a place holder for the state space. Similarly the second labelled record contains a function (or constant) for every constructor declaration. With this different notion of model, the CCSL compiler cannot use the definitions of this section for morphisms, bisimulation, and invariants literally. Instead it uses suitable modifications.

As an illustration I show some parts of the PVS material that the CCSL compiler generates for the queue signature. The material is taken from the file `Queue_basic.pvs`, which was obtained by running the CCSL compiler on the queue signature in Figure 5.

The first theory formalises the queue signature, it is shown in Figure 6. The name of the theory is QueueInterface, it imports the data type Lift from the (translated) prelude, and declares the method signature and the constructor signature of queue as a labelled records. (The theories that are generated by the CCSL compiler and their contents are described in Subsection 10.1 on page 97 below.)

[12]An earlier version of the compiler implemented Definition 5.4 exactly. This often lead to complications with automatic proof strategies.

The interface theory is used in the following way. Inhabitants of the type QueueSignature correspond to queue coalgebras. Assume that c has this type (i.e., c is a queue signature model), then one can write put(c)($\cdots$) in PVS to get the interpretation of the put method with respect to c.[13]

After the signature the compiler outputs theories for predicate lifting and invariants. These theories are a bit more difficult to understand, because the predicate lifting is generated method wise. This way it can be reused for the (method wise) modal operators of CCSL's logic (they are handled in Subsection 6.2 on page 54 below). Therefore I prefer to show the output that is generated for relation lifting and bisimulations.

Predicate and relation lifting is built into the CCSL compiler. It can generate expressions corresponding directly to $\mathrm{Rel}([\![\tau]\!])$. After some experience with the first versions of the CCSL compiler we learned that one usually needs $(c \times d)^* \,\mathrm{Rel}([\![\tau]\!])$ (for two coalgebras c and d). Therefore the current version of the compiler mingles method invocation with relation lifting. The result is not so pleasant from a theoretical point of view, but much easier to use in practice. For the queue signature the PVS theory QueueBisimilarity contains the following.

```
c1 : Var QueueSignature[Self1 , A]
c2 : Var QueueSignature[Self2 , A]

Queue_Rel(c1 , c2) :
  [[[Self1 , Self2] -> bool] -> [[Self1 , Self2] -> bool]] =
  Lambda (R: [[Self1 , Self2] -> bool]) :   Lambda (x1: Self1 , x2: Self2) :
      (Forall (a1: A) : R(put(c1)(x1 , a1) , put(c2)(x2 , a1)))      And
       (Cases top(c1)(x1) OF
          bot   : bot?(top(c2)(x2)),
          up(p0): up?(top(c2)(x2)) And
                  Proj_1(p0) = Proj_1(down(top(c2)(x2))) And
                  R(Proj_2(p0) , Proj_2(down(top(c2)(x2))))
        Endcases )
```

The first two lines declare two queue coalgebras.[14] The coalgebra c1 runs on state space Self1 and c2 on Self2. Then Queue_Rel is defined as a function on Self1 $\times$ Self2 $\Rightarrow$ bool. There are several optimisations built-in into the CCSL compiler that simplify the generated output. For the definition of bisimulation the parameter relations are instantiated with equality. This leads to formulae of the form $\forall a, b : \tau \,.\, a = b \supset \cdots$, which can be simplified into $\forall a : \tau \,.\, (\cdots)[a/b]$. With optimisations turned off the fourth line of the definition of Queue_Rel would look as follows

```
Forall(a1 : A, a2 : A) : a1 = a2   Implies   R(put(c1)(x1, a1), put(c2)(x2, a2))
```

[13] In PVS record selection can be written like function application, so put(c) denotes the put field of the record c.

[14] Variable declarations are syntactic sugar in PVS. The two declarations save the lambda abstractions in the definition of Queue_Rel for the arguments c1 and c2.

Another optimisation that is performed by the compiler is the inlining of liftings for nonrecursive data types and class types. In the queue example the type constructor Lift is defined as a nonrecursive abstract data type. Therefore the compiler outputs a case expression instead of applying the relation lifting for Lift.

After the relation lifting the compiler outputs a recogniser on queue bisimulations:

```
bisimulation?(c1 , c2) : [[[Self1 , Self2] -> bool] -> bool] =
  Lambda (R: [[Self1 , Self2] -> bool]) :   Forall (x1: Self1 , x2: Self2) :
      R(x1 , x2)    Implies   Queue_Rel(c1 , c2)(R)(x1 , x2)
```

The predicate bisimulation? holds for a relation R if and only if R is a queue bisimulation. With it, bisimilarity is defined as follows.

```
bisim?(c1 , c2) : [[Self1 , Self2] -> bool] =
  Lambda (x1: Self1 , x2: Self2) :    Exists (R: [[Self1 , Self2] -> bool]) :
      bisimulation?(c1 , c2)(R)    And    R(x1 , x2)
```

The generation of definitions for bisimulation and invariant for coalgebraic signatures is only one part of a translation into higher order logic. An equally well important task is the generation of lemmas that capture standard results. For instance, for signatures corresponding to polynomial functors the compiler generates lemmas stating that bisimilarity is an equivalence relation. To achieve this, bisimulation and bisimilarity is first defined for one coalgebra (the following material is from the theories QueueBisimilarityEquivalence and QueueBisimilarityEqRewrite):

```
c : Var QueueSignature[Self , A]
bisimulation?(c) : [[[Self , Self] -> bool] -> bool] = bisimulation?(c , c);
bisim?(c) : [[Self , Self] -> bool] = bisim?(c , c) ;
```

Then, the statement that equality is a queue bisimulation reads as follows:

```
eq_bisim : Lemma bisimulation?(c)(Lambda(x1: Self , x2: Self) : x1 = x2)
```

For reflexivity and symmetry of bisimilarity the compiler generates

```
bisim_refl : Lemma Forall (x: Self) : bisim?(c)(x , x)
bisim_sym : Lemma Forall (x1: Self , x2: Self) :
    bisim?(c)(x1 , x2)   Implies   bisim?(c)(x2 , x1)
```

In reasoning with bisimulations one often needs lemmas that express that a method delivers the same (or bisimilar) results when invoked for bisimilar states. The CCSL compiler generates one such lemma for each method, here I show only the one for the method put.

```
bisim_put : Lemma Forall (x1: Self , x2: Self , a1: A) :
  bisim?(c)(x1 , x2)   Implies   bisim?(c)(put(c)(x1 , a1) , put(c)(x2 , a1))
```

Lemmas like bisim_put seem to be trivial, because they follow immediately from the definition of bisimulation. However, these little lemmas are extremely useful in applications, their generation is one of the great benefits of the CCSL compiler.

Ideally one would like that for all generated lemmas the CCSL compiler outputs proofs in the format of the target theorem prover. However, it is very difficult to generate proofs that work properly for all possible signatures. The current compiler version generates only a few proofs.

6. Assertions and Creation Conditions

The previous section discussed the structural aspect of coalgebraic specification. In this section I turn to the logical aspects. The signature of FIFO queues in Example 5.2 contains nothing to actually restrict the class of models to those that can be considered as FIFO queues. And, indeed, also (*last–in–first–out*) stacks give rise to models of Σ_{Queue}. In this section I define a logic that allows one to express properties of methods and constructors from a coalgebraic class signature. A signature together with a set of logical formulae is called a specification. The models of the specification are those models of the signature that fulfil the formulae in a suitable sense.

In the following I present the logic of CCSL. This is an entirely standard higher-order logic over a polymorphic signature with two extensions. The extensions are behavioural equality and (infinitary) method-wise modal operators. The first subsection presents the higher-order logic with behavioural equality over a coalgebraic class signature. The second subsection defines infinitary modal operators for coalgebras. Subsection 6.3 is on coalgebraic class specifications and Subsection 6.4 explains the syntax of CCSL.

6.1. Higher-order Logic

The striking property of higher-order logic is that formulae are terms of the special type Prop. Thereby it is possible to quantify over subsets of individuals and also over predicates. Terms may contain *(term) variables*, which are declared to have a certain type in the *term variable context*. All types can contain type variables drawn from a type variable context. Formally a term variable context (over a type variable context Ξ) is a finite list of distinct variable declarations $x : \tau$ such that $\Xi \vdash \tau : \mathsf{Type}$ is derivable.[15] Terms are formed from variables, constructions like tuples or case analysis, and logical connectives. A term is given by a *term judgement*

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

Here Ξ is a type variable context, Γ is a term variable context and t is a well-typed term of type τ according to the rules below. All free (term) variables of t must be declared in Γ and all type variables that occur in t, τ, and Γ must be declared in Ξ.

[15] The condition that a context contains no variable twice can be enforced at the expense of a more complicated derivation system, see for instance Section 2.1 in [Jac99].

The following definition describes the terms and formulae over a coalgebraic class signature. Modal operators are added in Definition 6.7 (on page 56) below.

Definition 6.1 (Terms and Formulae) Let Σ be a coalgebraic class signature over a proper ground signature Ω. The set of terms over Σ, denoted with $\mathit{Terms}(\Sigma)$, is the least set containing:

- $x : \tau$ for a variable x of type τ
- $* : \mathbf{1}$ the only inhabitant of $\mathbf{1}$
- $\bot : \mathsf{Prop}$, $\top : \mathsf{Prop}$ the boolean constants false and true
- $f : \sigma$ for constants $f \in \Omega_\sigma$
- $m : \mathsf{Self} \times \sigma \Rightarrow \rho$ for all method declarations $m \in \Sigma_M$
- $c : \sigma \Rightarrow \mathsf{Self}$ for all constructor declarations $c \in \Sigma_C$
- $(t_1, t_2) : \sigma \times \tau$, the tuple for terms $t_1 : \sigma$ and $t_2 : \tau$
- $\pi_1\, t : \sigma$ and $\pi_2\, t : \tau$, the projections for a term $t : \sigma \times \tau$
- $\kappa_1\, s : \sigma + \tau$ and $\kappa_2\, t : \sigma + \tau$, the injections for terms $s : \sigma$ and $t : \tau$
- $\mathsf{cases}\ t\ \mathsf{of}\ \kappa_1\, x : r,\ \kappa_2\, y : s \ : \ \tau$, the case analyses for terms $t : \sigma_1 + \sigma_2$, $r : \tau$, and $s : \tau$. The term r contains the variable x free and the term s contains y free. The term t gets bound to either x or y, depending on the result of the evaluation. In the complete case expression the variables x and y are bound.
- $\mathsf{if}\ r\ \mathsf{then}\ s\ \mathsf{else}\ t \ : \ \tau$, the conditional for a term $r : \mathsf{Prop}$ and two terms s and t of the same type τ
- $\lambda x : \sigma \,.\, t \ : \ \sigma \Rightarrow \tau$, lambda abstraction for a variable $x : \sigma$ and a term $t : \tau$.
- $t_1\, t_2 : \tau$, application for two terms $t_1 : \sigma \Rightarrow \tau$ and $t_2 : \sigma$
- $t_1 = t_2 : \mathsf{Prop}$ equality for two terms of the same type τ
- $t_1 \sim t_2 : \mathsf{Prop}$, behavioural equality for two terms of the same type τ
- $\neg t : \mathsf{Prop}$, the negation for a term $t : \mathsf{Prop}$,
- $t_1 \wedge t_2 : \mathsf{Prop}$ and $t_1 \vee t_2 : \mathsf{Prop}$, the conjunction and the disjunction for terms $t_1 : \mathsf{Prop}$ and $t_2 : \mathsf{Prop}$
- $\forall x : \tau \,.\, t : \mathsf{Prop}$, universal quantification for a variable $x : \tau$ and a term $t : \mathsf{Prop}$

A derivation system for term judgements for well-typed terms is in the Figures 7 and 8. As abbreviations I define

- $t_1 \supset t_2 \stackrel{\text{def}}{=} \neg t_1 \vee t_2$, implication
- $t_1 \supset\subset t_2 \stackrel{\text{def}}{=} (t_1 \supset t_2) \wedge (t_2 \supset t_1)$, logical equivalence
- $\mathsf{let}\ x : \tau = t_1\ \mathsf{in}\ t_2 \stackrel{\text{def}}{=} (\lambda x : \tau \,.\, t_2)\, t_1$, let bindings
- $\exists x : \tau \,.\, t \stackrel{\text{def}}{=} \neg \forall x : \tau \,.\, \neg t$, existential quantification

Terms of type Prop are called formulae and denoted with Greek letters like φ, ψ. The formulae over Σ are denoted with $\mathit{Form}(\Sigma)$.

The only non-standard term in the preceding definition is behavioural equality. Its semantics is given by the relation lifting of bisimilarity. For instance for two terms t_1 and t_2 of type $\alpha \times \mathsf{Self}$ the equation $t_1 \sim t_2$ holds if and only if $\pi_1\, t_1 = \pi_1\, t_2$ and $(\pi_2\, t_1) \underline{\leftrightarrow} (\pi_2\, t_2)$. For class signatures over non-proper ground signatures one has to restrict the terms to those which do not contain behavioural equality $\sim$.

Example 6.2 In Example 5.2 I described the Queue signature. Here I show two formulae that separate *FIFO* queues from other models of the Queue–signature. The first property is about empty queues. A queue q is considered empty if the top methods fails on it (i.e., if $\mathsf{top}(q) = \mathsf{bot}$).

$$F_{\mathsf{empty}}(q) \quad \stackrel{\text{def}}{=} \quad \Big[\quad \mathsf{top}(q) = \mathsf{bot} \ \supset\ \forall a : \alpha \,.\, \mathsf{top}(\mathsf{put}(q, a)) \sim \mathsf{up}(a, q) \quad\Big]$$

So if the queue is empty then $\mathsf{top}(\mathsf{put}(q, a))$ should always be successful (i.e., it never equals bot) and return a pair (b, q') where $a = b$ and q' is an empty queue again. To be precise F_{empty} is a term

$$\alpha : \mathsf{Type} \ \mid\ q : \mathsf{Self} \ \vdash\ F_{\mathsf{empty}}(q) \ : \ \mathsf{Prop}$$

The second property (over the same contexts) is about nonempty queues.

$$F_{\mathsf{filled}}(q) \quad \stackrel{\text{def}}{=} \quad \left[\begin{array}{l} \forall a_1 : \alpha, q' : \mathsf{Self} \,.\, \mathsf{top}(q) \sim \mathsf{up}(a_1, q') \ \supset \\ \qquad\qquad\qquad \forall a_2 : \alpha \,.\, \mathsf{top}(\mathsf{put}(q, a_2)) \ \sim\ \mathsf{up}(a_1, \mathsf{put}(q', a_2)) \end{array}\right]$$

This says that, if q is nonempty, then the two operations of adding an element (at the end) and of removing the first element are interchangeable. ■

The semantics of the logic is completely standard. Let $\Xi \mid \Gamma \vdash t : \tau$ be a term. Fix an interpretation $U_1, \ldots, U_n$ for the type variables in Ξ and a set X as interpretation of Self. You can think of the term t as a function that maps any values that you assign to the variables in Γ to a value in τ. Consequently the semantics of t for a fixed interpretation of the type variables and of Self is a function

$$[\![\sigma_1]\!] \times \cdots \times [\![\sigma_k]\!] \longrightarrow [\![\tau]\!]$$

ground terms

$$\frac{\Xi \vdash \sigma : \mathsf{Type}}{\Xi \mid \Gamma \vdash x : \sigma}\ x : \sigma \in \Gamma \qquad \frac{\Xi \vdash \sigma : \mathsf{Type}}{\Xi \mid \Gamma \vdash f : \sigma}\ f \in \Omega_\sigma$$

$$\frac{\Xi \vdash \tau : \mathsf{Type}}{\Xi \mid \Gamma \vdash m : \tau}\ m : \tau \in \Sigma_M \qquad \frac{\Xi \vdash \tau : \mathsf{Type}}{\Xi \mid \Gamma \vdash c : \tau}\ c : \tau \in \Sigma_C$$

$$\frac{}{\Xi \mid \Gamma \vdash * : \mathbf{1}} \qquad \frac{}{\Xi \mid \Gamma \vdash \bot : \mathsf{Prop}} \qquad \frac{}{\Xi \mid \Gamma \vdash \top : \mathsf{Prop}}$$

tuples

$$\frac{\Xi \mid \Gamma \vdash s : \sigma \quad \Xi \mid \Gamma \vdash t : \tau}{\Xi \mid \Gamma \vdash (s, t) : \sigma \times \tau} \qquad \frac{\Xi \mid \Gamma \vdash t : \sigma \times \tau}{\Xi \mid \Gamma \vdash \pi_1\, t : \sigma} \qquad \frac{\Xi \mid \Gamma \vdash t : \sigma \times \tau}{\Xi \mid \Gamma \vdash \pi_2\, t : \tau}$$

variants

$$\frac{\Xi \mid \Gamma \vdash s : \sigma \quad \Xi \vdash \tau : \mathsf{Type}}{\Xi \mid \Gamma \vdash \kappa_1\, s : \sigma + \tau} \qquad \frac{\Xi \vdash \sigma : \mathsf{Type} \quad \Xi \mid \Gamma \vdash t : \tau}{\Xi \mid \Gamma \vdash \kappa_2\, t : \sigma + \tau}$$

$$\frac{\Xi \mid \Gamma \vdash t : \sigma_1 + \sigma_2 \quad \Xi \mid \Gamma, x : \sigma_1 \vdash r : \tau \quad \Xi \mid \Gamma, y : \sigma_2 \vdash s : \tau}{\Xi \mid \Gamma \vdash \mathsf{cases}\ t\ \mathsf{of}\ \kappa_1\, x : r,\ \kappa_2\, y : s \quad : \ \tau}\ x \notin \Gamma, y \notin \Gamma$$

conditional

$$\frac{\Xi \mid \Gamma \vdash r : \mathsf{Prop} \quad \Xi \mid \Gamma \vdash s : \tau \quad \Xi \mid \Gamma \vdash t : \tau}{\Xi \mid \Gamma \vdash \mathsf{if}\ r\ \mathsf{then}\ s\ \mathsf{else}\ t \ : \ \tau}$$

abstraction & application

$$\frac{\Xi \vdash \sigma : \mathsf{Type} \quad \Xi \mid \Gamma, x : \sigma \vdash t : \tau}{\Xi \mid \Gamma \vdash \lambda x : \sigma\,.\, t : \sigma \Rightarrow \tau}\ x \notin \Gamma \qquad \frac{\Xi \mid \Gamma \vdash t : \sigma \Rightarrow \tau \quad \Xi \mid \Gamma \vdash s : \sigma}{\Xi \mid \Gamma \vdash t\, s : \tau}$$

Figure 7: Derivation system for the terms over a coalgebraic class signature Σ and a ground signature Ω, Part I

equality

$$\frac{\Xi \mid \Gamma \vdash s : \tau \quad \Xi \mid \Gamma \vdash t : \tau}{\Xi \mid \Gamma \vdash s = t : \mathsf{Prop}} \qquad \frac{\Xi \mid \Gamma \vdash s : \tau \quad \Xi \mid \Gamma \vdash t : \tau}{\Xi \mid \Gamma \vdash s \sim t : \mathsf{Prop}}$$

conjunction & disjunction

$$\frac{\Xi \mid \Gamma \vdash s : \mathsf{Prop} \quad \Xi \mid \Gamma \vdash t : \mathsf{Prop}}{\Xi \mid \Gamma \vdash s \wedge t : \mathsf{Prop}} \qquad \frac{\Xi \mid \Gamma \vdash s : \mathsf{Prop} \quad \Xi \mid \Gamma \vdash t : \mathsf{Prop}}{\Xi \mid \Gamma \vdash s \vee t : \mathsf{Prop}}$$

negation

$$\frac{\Xi \mid \Gamma \vdash t : \mathsf{Prop}}{\Xi \mid \Gamma \vdash \neg t : \mathsf{Prop}}$$

universal quantification

$$\frac{\Xi \vdash \tau : \mathsf{Type} \quad \Xi \mid \Gamma, x : \tau \vdash t : \mathsf{Prop}}{\Xi \mid \Gamma \vdash \forall x : \tau \,.\, t : \mathsf{Prop}} \; x \notin \Gamma$$

The following rules can be derived.

weakening

$$\frac{\Xi \mid \Gamma \vdash t : \tau}{\Xi, \alpha : \mathsf{Type} \mid \Gamma \vdash t : \tau} \; \alpha \notin \Xi \qquad \frac{\Xi \vdash \sigma : \mathsf{Type} \quad \Xi \mid \Gamma \vdash t : \tau}{\Xi \mid \Gamma, x : \sigma \vdash t : \tau} \; x \notin \Gamma$$

type substitution

$$\frac{\Xi \vdash \sigma : \mathsf{Type} \quad \Xi, \alpha : \mathsf{Type} \mid \Gamma \vdash t : \tau}{\Xi \mid \Gamma[\sigma/\alpha] \vdash t[\sigma/\alpha] : \tau[\sigma/\alpha]} \; \alpha \notin \Xi$$

term substitution

$$\frac{\Xi \mid \Gamma \vdash s : \sigma \quad \Xi \mid \Gamma, x : \sigma \vdash t : \tau}{\Xi \mid \Gamma \vdash t[s/x] : \tau} \; x \notin \Gamma$$

Figure 8: Derivation system for the terms over a coalgebraic class signature Σ and a ground signature Ω, Part II

where I assume that $\Gamma = x_1 : \sigma_1, \ldots, x_k : \sigma_k$. The complete semantics for t is an indexed collection of such functions:

$$\left(\begin{array}{l}[\![\sigma_1]\!]_{U_1,U_1,\ldots,U_n,U_n}(X,X) \times \cdots \times [\![\sigma_k]\!]_{U_1,U_1,\ldots,U_n,U_n}(X,X) \\ \qquad\qquad\qquad\qquad \longrightarrow [\![\tau]\!]_{U_1,U_1,\ldots,U_n,U_n}(X,X)\end{array}\right)_{X,U_1,\ldots,U_n \in |\mathbf{Set}|}$$

If $t : \tau$ is a formula then $[\![\tau]\!]$ is the set of booleans and the interpretation function returns true for exactly those elements of $[\![\sigma_i]\!]$ that fulfil t. So one can equivalently consider the semantics of a formula $x : \sigma \vdash \varphi : \mathsf{Prop}$ as a (collection of) predicate(s) $[\![\varphi]\!] \subseteq [\![\sigma]\!]$.

Definition 6.3 (Semantics) Let Σ be a coalgebraic class signature over a proper ground signature Ω and assume a model $\mathcal{M}_\Omega$ of Ω and a model $\mathcal{M}_\Sigma = \langle X, c, a\rangle$ of Σ. The interpretation of a term $\alpha_1, \ldots, \alpha_n \mid x_1 : \sigma_1, \ldots, x_k : \sigma_k \vdash t : \tau$ with respect to $\mathcal{M}_\Omega$ and $\mathcal{M}_\Sigma$ is denoted by $[\![t]\!]^{\mathcal{M}_\Omega,\mathcal{M}_\Sigma}$, where I omit the superscripts if they are clear from the context. The interpretation $[\![t]\!]$ is defined by induction on the structure of terms. Fix an interpretation $U_1, \ldots, U_n$ of the type variables α_i and and an interpretation X of Self. Let $\overline{x} : \overline{\sigma}$ denote the tuple of arguments $x_1 : [\![\sigma_1]\!], \ldots, x_k : [\![\sigma_k]\!]$.

$$\begin{array}{rcll}
[\![x_i]\!] &=& \pi_i & \\
[\![f]\!] &=& [\![f]\!] & \text{for } f \in \Omega_\sigma \\
[\![m : \mathsf{Self} \times \sigma' \Rightarrow \tau']\!] &=& \lambda \overline{x} : \overline{\sigma} \,.\, \big(\lambda x : X, p : [\![\sigma']\!] \,.\, \pi_m(c\,x)(p)\big) & \\
[\![c]\!] &=& \lambda \overline{x} : \overline{\sigma} \,.\, a \circ \kappa_c & \\
[\![*]\!] &=& \lambda \overline{x} : \overline{\sigma} \,.\, * & \\
[\![\bot]\!] &=& \lambda \overline{x} : \overline{\sigma} \,.\, \bot & \\
[\![\top]\!] &=& \lambda \overline{x} : \overline{\sigma} \,.\, \top & \\
[\![(t_1, t_2)]\!] &=& \langle [\![t_1]\!], [\![t_2]\!] \rangle & \\
[\![\pi_1\, t]\!] &=& \pi_1 \circ [\![t]\!] & \\
[\![\pi_2\, t]\!] &=& \pi_2 \circ [\![t]\!] & \\
[\![\kappa_1\, s]\!] &=& \kappa_1 \circ [\![t]\!] & \\
[\![\kappa_2\, t]\!] &=& \kappa_2 \circ [\![t]\!] & \\
[\![\mathsf{cases}\ t\ \mathsf{of}\ \kappa_1\, x : r,\ \kappa_2\, y : s]\!] &=& \lambda \overline{x} : \overline{\sigma} \,.\, \left\{ \begin{array}{lll} [\![r]\!](\overline{x}, z_1) & \text{if} & [\![t]\!]\,\overline{x} = \kappa_1 z_1 \\ [\![s]\!](\overline{x}, z_2) & \text{if} & [\![t]\!]\,\overline{x} = \kappa_2 z_2 \end{array}\right. & \\
[\![\mathsf{if}\ r\ \mathsf{then}\ s\ \mathsf{else}\ t]\!] &=& \lambda \overline{x} : \overline{\sigma} \,.\, \left\{ \begin{array}{lll} [\![s]\!]\overline{x} & \text{if} & [\![r]\!]\overline{x} = \top \\ [\![t]\!]\overline{x} & \text{if} & [\![r]\!]\overline{x} = \bot \end{array}\right. & \\
[\![\lambda x : \rho \,.\, t]\!] &=& \lambda \overline{x} : \overline{\sigma} \,.\, \big(\lambda y : [\![\rho]\!] \,.\, [\![t]\!](\overline{x}, y)\big) & \\
[\![t_1\, t_2]\!] &=& \lambda \overline{x} : \overline{\sigma} \,.\, [\![t_1]\!](\overline{x})\ ([\![t_2]\!]\,\overline{x}) & \\
[\![t_1 = t_2]\!] &=& \lambda \overline{x} : \overline{\sigma} \,.\, [\![t_1]\!]\overline{x}\ =\ [\![t_2]\!]\overline{x} & \\
[\![t_1 \sim t_2]\!] &=& \lambda \overline{x} : \overline{\sigma} \,.\, \mathrm{Rel}([\![\rho]\!])(\underline{\leftrightarrow}_{\mathcal{M}_\Sigma})([\![t_1]\!]\,\overline{x},\ [\![t_2]\!]\,\overline{x}) & \\
&&& \text{(for } t_1 \text{ and } t_2 \text{ of type } \rho)
\end{array}$$

$$
\begin{array}{rcl}
[\![\neg t]\!] & = & \lambda \overline{x} : \overline{\sigma} \,.\, \neg [\![t]\!] \overline{x} \\
[\![t_1 \wedge t_2]\!] & = & \lambda \overline{x} : \overline{\sigma} \,.\, [\![t_1]\!] \overline{x} \;\wedge\; [\![t_2]\!] \overline{x} \\
[\![t_1 \vee t_2]\!] & = & \lambda \overline{x} : \overline{\sigma} \,.\, [\![t_1]\!] \overline{x} \;\vee\; [\![t_2]\!] \overline{x} \\
[\![\forall x : \tau \,.\, t]\!] & = & \lambda \overline{x} : \overline{\sigma} \,.\, \begin{cases} \top & \text{if } \;[\![t]\!](\overline{x}, y) = \top \text{ for all } y \in [\![\tau]\!] \\ \bot & \text{otherwise} \end{cases}
\end{array}
$$

In the following I elaborate on the expressiveness of the logic of CCSL.

Proposition 6.4 *The logic of* CCSL *is complete with respect to bisimilarity. More precisely, for every closed term $t \in Term(\Sigma)$ of type* Self *over a signature Σ, there exists a formula $x : \mathsf{Self} \vdash F(x) : \mathsf{Prop}$ with the following property. For an arbitrary model $\mathcal{M} = \langle X, c, a\rangle$ of Σ and a state $x \in X$ one has $[\![F]\!](x) = \top$ if and only if there is a bisimulation on $\mathcal{M}$ relating x and $[\![t]\!]^{\mathcal{M}}$.*

Proof (Sketch) The definition of relation lifting and bisimulation can be directly expressed in the logic of CCSL. Thus there is a formula $R : \mathsf{Self} \times \mathsf{Self} \Rightarrow \mathsf{Prop} \vdash \mathsf{bisim}(R) : \mathsf{Prop}$ which is true, precisely if R is interpreted with a bisimulation. One can take as F

$$\lambda x : \mathsf{Self} \,.\; \exists R : \mathsf{Self} \times \mathsf{Self} \Rightarrow \mathsf{Prop} \,.\; \mathsf{bisim}(R) \;\wedge\; R(t, x) \qquad \square$$

The logic of CCSL has equality on all types, including Self. Therefore it is easy to construct a formula that evaluates to different values for bisimilar states. For coalgebraic specification it is often desirable to restrict the expressiveness of the logic, such that it cannot distinguish bisimilar states, or, in other words, is sound with respect to bisimilarity. This restricted expressiveness is for instance necessary for some results in Section 8 and for the result about behavioural refinement in [JT01]. In the following I characterise a fragment of the logic of CCSL that is sound with respect to bisimilarity. The higher-order aspects make it necessary to consider terms in general.

Definition 6.5 (Behavioural invariance) Let $\mathcal{M}$ be a model of a proper ground signature Ω and let Σ be a coalgebraic class signature over Ω. Let $\Xi \mid \Gamma \vdash t : \tau$ be a term over Σ with $\Gamma = a_1 : \sigma_1, \ldots, a_k : \sigma_k$. The term t is *invariant with respect to behavioural equality for $\mathcal{M}$* (or more succinctly *behaviourally invariant for $\mathcal{M}$*) if for all models $\mathcal{A} = \big(\langle X, c, a\rangle_{\overline{U}}\big)$ and $\mathcal{B} = \big(\langle Y, d, b\rangle_{\overline{U}}\big)$ and all interpretations $\overline{U} = U_1, \ldots, U_n$ of the type parameters of Σ the following condition is fulfilled. Let $x_i \in [\![\sigma_i]\!]_{\overline{U}}(X, X)$ and $y_i \in [\![\sigma_i]\!]_{\overline{U}}(Y, Y)$ be two interpretations of the variables a_i and let $R \subseteq X \times Y$ be a bisimulation for c and d. If

$$\mathrm{Rel}\big([\![\sigma_i]\!]\big)\;\big(\mathrm{Eq}(U_1), \ldots, \mathrm{Eq}(U_n), R, R\big)\;(x_i, y_i)$$

holds for all i, then it also holds that

$$\mathrm{Rel}\big([\![\tau]\!]\big)\;\big(\mathrm{Eq}(U_1), \ldots, \mathrm{Eq}(U_n), R, R\big)\;\big(\;[\![t]\!]^{\mathcal{M},A}(x_1, \ldots, x_k),\;[\![t]\!]^{\mathcal{M},B}(y_1, \ldots, y_k)\;\big)$$

This definition is carefully formulated to apply to arbitrary signatures, for which a greatest bisimulation might not exist. Of course, if bisimilarity as greatest bisimulation does exists, then behavioural invariance with respect to bisimilarity implies behavioural invariance with respect to any other bisimulation. It is easy to give some sufficient syntactical criteria for behavioural invariance of terms.

Proposition 6.6 *Let Σ be a coalgebraic class signature. The following basic terms are behaviourally invariant:*

- *variables $x : \tau$*
- *the constants $\bot, \top : \mathsf{Prop}$, and $* : \mathbf{1}$*
- *methods from Σ*

The following constructions preserve behavioural invariance:

- *pairing (t_1, t_2): that is if t_1 and t_2 are behavioural invariant then so is (t_1, t_2),*
- *projections $\pi_{1/2}\, t : \sigma$, injections $\kappa_{1/2}\, t$, and case analyses* $\mathsf{cases}\ t\ \mathsf{of}\ \kappa_1\, x : t_1,\ \kappa_2\, y : t_2\ : \tau$
- *the conditional* $\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3$,
- *lambda abstraction $\lambda x : \sigma\, .\, t$ and application $t_1\, t_2$*
- *negation $\neg t$, conjunction $t_1 \wedge t_2$, and disjunction $t_1 \vee t_2$*

For proper models of the ground signature:

- *universal quantification over constants $\forall x : \tau\, .\, t$, where τ is a constant type (i.e., $\mathcal{V}_{\mathsf{Self}}(\tau) = ?$)*

If additionally bisimulations are closed under composition and if the greatest bisimulation $\underline{\leftrightarrow}$ does exist:

- *behavioural equality $t_1 \sim t_2$*

Proof I abbreviate the longish $\mathrm{Rel}(\llbracket\tau\rrbracket)(\mathrm{Eq}(U_1), \ldots, \mathrm{Eq}(U_n), R, R)$ as $\mathrm{Rel}(\tau)(\cdots)$ and use $\overline{x} = x_1, \ldots, x_k$ and $\overline{y} = y_1, \ldots, y_k$.

- $\mathrm{Rel}(\tau)(\cdots)(\llbracket x\rrbracket^A, \llbracket x\rrbracket^B)$ follows from the definition.
- The basic terms $*$, $\bot$, and $\top$ are behavioural invariant because Definition 5.7 (2), uses equality for Prop and $\mathbf{1}$.
- Let $t = m$ be a method, then $\mathrm{Rel}(\tau)(\cdots)(\llbracket m\rrbracket^{\mathcal{A}}, \llbracket m\rrbracket^{\mathcal{B}})$ follows from the definition of bisimulation.

- If t is a pair, a projection, an injection, or a case analyses, then the conclusion follows directly from the definition of relation lifting.

- For the conditional $t = \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3$ assume that t_1, t_2, and t_3 are behavioural invariant. Then $[\![t_1]\!]^{\mathcal{A}}\,\overline{x} = [\![t_1]\!]^{\mathcal{B}}\,\overline{y}$ and the conclusion follows by the induction hypothesis on t_2 and t_3.

- For $t = \lambda x : \sigma \,.\, t_1$ we have to show that for all $x \in [\![\sigma]\!]^{\mathcal{A}}$ and $y \in [\![\sigma]\!]^{\mathcal{B}}$ with $\mathrm{Rel}(\sigma)(\cdots)(x, y)$ also $\mathrm{Rel}(\tau)(\cdots)([\![t_1]\!]^{\mathcal{A}}(\overline{x}, x), [\![t_1]\!]^{\mathcal{B}}(\overline{y}, y))$. This fact follows directly from the behavioural invariance of t_1.

  The case $t = t_1\, t_2$ follows directly from the behavioural invariance of t_1 and t_2.

- If t is one of the propositional connectives, then the conclusion follows again from the fact that the relation lifting for type Prop is given by equality.

- If $t = \forall a : \sigma \,.\, t'$ then we have $[\![\sigma]\!]^{\mathcal{A}} = [\![\sigma]\!]^{\mathcal{B}}$ because σ is a constant type. Additionally Lemma 5.9 yields $\mathrm{Rel}(\sigma)(\cdots) = \mathrm{Eq}([\![\sigma]\!])$ for a proper model of the ground signature. So the assumption about the behavioural invariance of t_1 implies for every $a \in [\![\sigma]\!]$ that $[\![t]\!]^{\mathcal{A}}(\overline{x}, a) = [\![t]\!]^{\mathcal{B}}(\overline{y}, a)$.

- For $t = \ t_1 \sim t_2$ assume that t_1 and t_2 are behavioural invariant. With composition of bisimulations $\mathrm{Rel}(\tau)(\cdots)([\![t_1]\!]^{\mathcal{A}}, [\![t_2]\!]^{\mathcal{A}})$ holds precisely if $\mathrm{Rel}(\tau)(\cdots)([\![t_1]\!]^{\mathcal{B}}, [\![t_2]\!]^{\mathcal{B}})$ holds. Hence $[\![t]\!]^{\mathcal{A}} = [\![t]\!]^{\mathcal{B}}$. $\square$

The notion of behavioural invariance and the preceding proposition are interesting for class signatures for which coalgebra morphisms are functional bisimulations. In this case behavioural invariance implies stability under coalgebra morphisms. I exploit this fact in Proposition 6.18 and in Subsection 8.2.

Note that the preceding proposition does neither include constants from the ground signature nor constructors from class specifications. Monomorphic constants (i.e., those over the empty type variable context) are behaviourally invariant if the relation lifting for their type coincides with equality (which holds for proper models of proper ground signatures). Polymorphic constants might or might not be behaviourally invariant, depending on the model of the ground signature. With the `-pedantic` switch (see Subsection 10.9 on page 103 below) the CCSL compiler recognises polymorphic constants as behavioural invariant if they are instantiated with a constant type. This relaxed policy rests on an argument similar to that in the preceding proof for the item of universal quantification and on the fact that the `-pedantic` switch implies a proper ground signature with a proper model.

6.2. Infinitary Modal Operators

This subsection describes joint work with Bart Jacobs and Jan Rothe. Following the observations that, firstly, modal logic [Gol92] is the logic for describing dynamic systems and that, secondly, coalgebras are the mathematical structures that capture dynamic systems one has to expect a close relationship between modal logic and coalgebras. Currently modal logic is used in the field of coalgebras mainly in two different ways. On the one hand modal logic is used as a tool to investigate the theory of coalgebras. On the other hand modal logic enriches coalgebraic specification.

In the former line of work [Mos99] describes characterising (modal) formulae for the state space of a coalgebra. Rößiger uses modal logic to construct final coalgebras for data functors [Röß00a] (but see also [Röß00b]). Modal logic also plays an important role in the search for the coalgebraic analogy of Birkhoffs theorem [Kur00, Hug01, Gol01]. The modal logics of Rößiger, Moss, and Goldblatt are all sound and complete with respect to bisimilarity (i.e., bisimilar states fulfil the same set of formulae and for any two non-bisimilar states there is a formula that distinguishes both). The logics of Kurz and Hughes have this property when restricted to one colour. However, all these logics have often been designed towards a certain theorem. Without deprecating all this work, one notices that these logics are not very practical for expressing interesting properties of coalgebras. For instance in the queue example, I consider the following property F_{finite} as interesting: A queue fulfils F_{finite} if the successive application of the method top eventually yields an empty queue (i.e., top returns eventually bot). To express F_{finite} in the framework of [Mos99] or [Röß00a] one needs infinitary conjunctions or an infinite set of formulae.

The second line of research that connects modal logic and coalgebra tries to enrich coalgebraic specification with modalities to express certain properties more succinctly. [Jac97b] shows that the infinitary[16] modality always can get its semantics via greatest invariants (contained in some predicate). [Rot00] picks this idea up and describes method wise infinitary modal operators for CCSL (see also Section 4 in [RTJ01]). This section describes Rothes method wise modal operators in the formal context of coalgebraic class signatures over a polymorphic type theory.

In the following I consider infinitary versions of the two modal operators $\Box$ and $\Diamond$. For a (syntactic) predicate P the modality $\Box P$ (always P or henceforth P) holds for a state x of a coalgebra c, if P holds for x and all successor states of x, which can be reached via c. So $\Box P$ is the safety property that assures that the bad event $\neg P$ never happens. The modality $\Diamond$ (eventually) is the dual of $\Box$, that is $\Diamond P = \neg\Box\neg P$. The formula $\Diamond P$ holds for those states x that have at least one successor state that makes P true. Therefore one can view $\Diamond P$ as the liveness property that holds if the good thing P does eventually happen.[17]

The semantics of the modalities is given by the greatest invariant (contained in some predicate), compare Section 2.6.6 and Section 3.4.6 in [Tew02b]. Because CCSL uses strong invariants the greatest invariant exists for all class signatures over a proper plain ground signature (see Proposition 3.4.25 in [Tew02b]). For non-plain ground signatures I assume that the predicate lifting of all type constructors C is monotone in its positive positions:

$$P_i \subseteq Q_i, \qquad \text{implies} \qquad \mathsf{Pred}_{\mathsf{C}}(\top, P_1, \ldots, \top, P_n) \subseteq \mathsf{Pred}_{\mathsf{C}}(\top, Q_1, \ldots, \top, Q_n) \qquad (1)$$

[16] Infinitary means here, that the modal operator applies to all following successor states and not only to the next state. Thus is satisfies the schema 4 [Gol92] and, equivalently, the underlying Kripke structure is transitive.

[17] Note that in general $\Diamond P$ is not a liveness property according to the rigorous definition of [AS85]: $\Diamond P$ is not dense in the obvious topology. However $\Box P$ is closed and therefore a safety property in the sense of Alpern and Schneider.

The interaction of the modalities with the higher-order logic of Definition 6.1 is a bit tricky, so let me discuss the type of $\Box$ before I present the definition (the following explanation applies to $\Diamond$ in the same way). From the preceding paragraph it is clear that the expression, to which $\Box$ is applied to, must be a predicate on the state space of the coalgebra. Therefore it must be of type $\mathsf{Self} \Rightarrow \mathsf{bool}$. Assume that P is of type Prop and has a free variable $x : \mathsf{Self}$, then we can form the expression $\Box(\lambda x : \mathsf{Self}\,.\,P)$. This expression is again a predicate on the state space, so we have $\Box(\lambda x : \mathsf{Self}\,.\,P) : \mathsf{Self} \Rightarrow \mathsf{Prop}$.

Note that, different to traditional modal logic, the predicate P can contain additional free variables. In this case both $\Box(\lambda x : \mathsf{Self}\,.\,\forall a : \tau\,.\,P)$ and $\forall a : \tau\,.\,\Box(\lambda x : \mathsf{Self}\,.\,P)$ are possible.

When working with class signatures, one often wants to express that only the subset $\{m_1, \ldots, m_n\}$ of all available methods retains a safety property P. Thereby one explicitly allows that a method $m_0 \notin \{m_1, \ldots, m_n\}$ yields a successor state that violates P. This cannot be expressed with the $\Box$ operator described so far. Similarly, for liveness properties one might want to ensure that a state fulfilling P can be reached by only using a subset of all available methods. One example for this is the property F_{finite} from before, where the empty queue is reached by only applying the method top.

The solution of this problem is to annotate the modal operators with sets of method identifiers, like in $\Box^{\{m_1,m_2\}}(\lambda x : \mathsf{Self}\,.\,P)$. The annotation restricts the number of successor states that are considered: The formula $\Box^M(\lambda x : \mathsf{Self}\,.\,P)$ holds for y if P holds for all successor states that can be reached with methods in M. The value of P on a successor state that was obtained via a method $m \notin M$ does not play any role. Similarly for $\Diamond$.

Definition 6.7 (Modal Operators) Let $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$ be a coalgebraic class signature over a proper ground signature. The set of terms over Σ contains in addition to Definition 6.1

- $\Box^M P : \mathsf{Self} \Rightarrow \mathsf{Prop}$ for a set of method (identifiers) $M \subseteq \Sigma_M$ and a predicate $P : \mathsf{Self} \Rightarrow \mathsf{Prop}$.

The derivation rule is in Figure 9. There are the following abbreviations

- $\Diamond^M P \stackrel{\text{def}}{=} \lambda y : \mathsf{Self}\,.\,\neg\; \Box^M(\lambda x : \mathsf{Self}\,.\,\neg P\,x)(y)$
- $\Box P \stackrel{\text{def}}{=} \Box^{\Sigma_M} P$
- $\Diamond P \stackrel{\text{def}}{=} \Diamond^{\Sigma_M} P$

Remark 6.8 There are several alternatives to introduce modal operators into the higher-order logic of CCSL. The preceding definition seems to be the most appropriate compromise for getting the succinctness of modal logics without cluttering CCSL too much. The following alternatives have been discarded.

- Rößiger and Jacobs define in [Röß00a] and [Jac00] the notion of paths by induction on the structure of the signature functor. One path denotes precisely one possible

always

$$\frac{\Xi \mid \Gamma \vdash P : \mathsf{Self} \Rightarrow \mathsf{Prop}}{\Xi \mid \Gamma \vdash \Box^M P : \mathsf{Self} \Rightarrow \mathsf{Prop}}\ M \subseteq \Sigma_M$$

eventually

$$\frac{\Xi \mid \Gamma \vdash P : \mathsf{Self} \Rightarrow \mathsf{Prop}}{\Xi \mid \Gamma \vdash \Diamond^M P : \mathsf{Self} \Rightarrow \mathsf{Prop}}\ M \subseteq \Sigma_M$$

Figure 9: Derivation rules for the modal operators over a class signature $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$.

way to extract a successor state. Path-wise modal operators allow one to distinguish between several successor states that are obtained from one method application. As an example consider a method (or a coalgebra) $m : \mathsf{Self} \;\Rightarrow\; (\mathsf{Self} \times \mathsf{Self}) + \mathsf{Self}$. For this method there are three paths $\kappa_1 \cdot \pi_1$, $\kappa_1 \cdot \pi_2$, and κ_2. If $m\,x = \kappa_1(y_1, y_2)$ then the path $\kappa_1 \cdot \pi_2$ denotes y_2 and in this case the path κ_2 does *not* denote a successor state.

Path wise modal operators are finer than method wise ones. However, for CCSL the granularity of methods seems appropriate.

- A modal μ–calculus [Sti92] for coalgebras would be even more flexible than path wise modal operators. This remains future work, a first discussion is in [Jac02].
- The preceding definition introduces $\Box(\lambda x : \mathsf{Self}\,.\,P)$ as a special term. Alternatively one can assume a (higher-order) function $\Box^M : (\mathsf{Self} \Rightarrow \mathsf{Prop}) \Rightarrow (\mathsf{Self} \Rightarrow \mathsf{Prop})$.
- The operator $\Box$ acts as a variable binder. In Definition 6.7 this is captured by requiring that P is of type $\mathsf{Self} \Rightarrow \mathsf{bool}$ (the binding is done by lambda abstraction). In a first-order version the bound variable must be explicit, as in the following (where I neglect the annotation with methods):

$$\frac{\Gamma, x : \mathsf{Self} \;\vdash\; P : \mathsf{Prop}}{\Gamma, y : \mathsf{Self} \;\vdash\; (\Box_x P)\; y : \mathsf{Prop}}$$

Here y is a fresh variable and the x in P is bound by $\Box_x$. The substitution rule is as follows:

$$((\Box_x\; P)\; t)\; [s/z] \quad = \quad \begin{cases} (\Box_x\; P)\; (t[s/z]) & \text{for } x = z \\ (\Box_x\; P[s/z])\; (t[s/z]) & \text{for } x \neq z \end{cases}$$

Definition 6.9 (Semantics of $\Box$) Let $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$ be a coalgebraic class signature over a proper ground signature Ω and let $\mathcal{A} = \langle X, c, a \rangle$ and $\mathcal{M}_\Omega$ be models for Σ and Ω, respectively. Assume that $\mathcal{M}_\Omega$ satisfies the monotonicity requirement of Equation 1. Consider the term $\Xi \mid \Gamma \vdash \Box^M P$ with contexts $\Xi = \alpha_1, \ldots, \alpha_n$ and $\Gamma = x_1 : \sigma_1, \ldots, x_k : \sigma_k$. By definition the set of method annotations M forms a subsignature $\Sigma' = \langle M, \emptyset \rangle \leq \Sigma$. Now, fix an interpretation $U_1, \ldots, U_n$ of the type variables and an interpretation X of Self. Let $\overline{x} : \overline{\sigma}$ denote the tuple $x_1 : [\![\sigma_1]\!]_{U_1,\ldots,U_n}(X, X)$, $\ldots$, $x_k : [\![\sigma_k]\!]_{U_1,\ldots,U_n}(X, X)$ for the interpretation of the term variables $x_1, \ldots, x_k$. Then

$$[\![\Box^M P]\!]^{\mathcal{A}} \quad = \quad \lambda \overline{x} : \overline{\sigma}\,.\, \underline{\left([\![P]\!]^{\mathcal{A}}\,(\overline{x})\right)}_{\pi_M \circ c}$$

where $\underline{(-)}_{\pi_M \circ c}$ denotes the greatest invariant with respect to the induced coalgebra

$$\pi_{\Sigma'} \circ c : X \longrightarrow [\![\tau_{\Sigma'}]\!](X, X)$$

Example 6.10 A queue contains a finite number of elements if the top method eventually returns bot. This property can now be formalised as

$$F_{\mathsf{finite}}(q) \quad \stackrel{\text{def}}{=} \quad (\Diamond^{\mathsf{top}}\ \lambda x : \mathsf{Self}\,.\,\mathsf{top}(x) = \mathsf{bot})\,(q)$$

as negation we obtain

$$F_{\mathsf{infinite}}(q) \quad \stackrel{\text{def}}{=} \quad (\Box^{\mathsf{top}}\ \lambda x : \mathsf{Self}\,.\,\neg\ \mathsf{top}(x) = \mathsf{bot})\,(q)$$

For another example recall the formulae F_{empty} and F_{filled} from Example 6.2 that capture the behaviour of FIFO queues. Define a formula that describes finite FIFO queues as

$$F_{\mathrm{FFIFO}}(x) \quad \stackrel{\text{def}}{=} \quad F_{\mathsf{empty}}(x) \wedge F_{\mathsf{filled}}(x) \wedge F_{\mathsf{finite}}(x)$$

Let q be an arbitrary finite FIFO queue. Starting from any other finite FIFO queue p one can construct a queue p' such that p' and q are bisimilar. This is expressed with

$$F_{\mathrm{FFIFO}}(p) \wedge F_{\mathrm{FFIFO}}(q) \quad \supset \quad (\Diamond\ \lambda p' : \mathsf{Self}\,.\,p' \sim q)(p) \tag{2}$$

with method wise modal operators it can be slightly strengthened to

$$F_{\mathrm{FFIFO}}(p) \wedge F_{\mathrm{FFIFO}}(q) \quad \supset \quad (\Diamond^{\mathsf{top}}\ \Diamond^{\mathsf{put}}\ \lambda p' : \mathsf{Self}\,.\,p' \sim q)(p) \tag{3}$$

which expresses that the successor state p' that is equivalent to q can be reached by first emptying p and then filling it. The preceding two statements can be put as theorems into the queue specification, see Figure 11 (on page 65 below). Although the proof of 2 and 3 is intuitively very simple it requires a fair amount of work to prove the two statements in PVS. The proof distributed with the sources of the queue example requires 54 utility lemmas that have been proved with about 700 PVS proof commands. ■

In the remainder I show a few general results about the modal operators. I first investigate behavioural invariance, extending Proposition 6.6.

Proposition 6.11 *Let Σ be a coalgebraic class signature over a plain ground signature that contains only polynomial methods. If the term $\Xi \mid \Gamma \vdash P$ is invariant with respect to behavioural equality, then so is $\Xi \mid \Gamma \vdash \Box^M P$.*

The preceding proposition does not hold if Σ contains a binary method. It is easy to construct an example that shows this. The problem here is that CCSL uses strong invariants and that Proposition 3.4.11 in [Tew02b] fails for strong invariants.

Proof Under the assumptions of the proposition Σ corresponds to a polynomial functor. Consider two models $\mathcal{A} = \langle X, c, a \rangle$ and $\mathcal{B} = \langle Y, d, b \rangle$ of Σ and let R be a bisimulation for c and d that relates $x \in X$ and $y \in Y$. Assume $x \in [\![\Box^M P]\!]^{\mathcal{A}}$, it remains to show that also $y \in [\![\Box^M P]\!]^{\mathcal{B}}$. Note that R is also a bisimulation for $\pi_M \circ c$ and $\pi_M \circ d$. Consider now the predicate $Q = \coprod_{\pi_2} (R \wedge \pi_1^* [\![\Box^M P]\!]^{\mathcal{A}}) = \{y \mid \exists x \,.\, x \, R \, y \wedge x \in [\![\Box^M P]\!]^{\mathcal{A}}\}$. By Proposition 2.6.17 and 2.6.15 in [Tew02b] Q is an invariant for $\pi_M \circ d$. And because P is behaviourally invariant, we have $Q \subseteq [\![P]\!]^{\mathcal{B}}$. $\Box$

Proposition 6.12 ([Rot00]) *The method wise-modal operators $\Box^M$ fulfil the* S4 *rules. For arbitrary predicates $\Xi \mid \Gamma \vdash P : \mathsf{Self} \Rightarrow \mathsf{bool}$ and $\Xi \mid \Gamma \vdash Q : \mathsf{Self} \Rightarrow \mathsf{bool}$ with $x \notin \Gamma$ we have*

$$\begin{array}{rlcl} K: & \forall x : \mathsf{Self}\,. \;\; \big(\Box^M \lambda x : \mathsf{Self}\,.\,(P\,x \supset Q\,x)\big)(x) & \supset & (\Box^M P)(x) \supset (\Box^M Q)(x) \\ T: & \forall x : \mathsf{Self}\,. \;\; (\Box^M P)(x) & \supset & P(x) \\ 4: & \forall x : \mathsf{Self}\,. \;\; (\Box^M P)(x) & \supset & (\Box^M \Box^M P)(x) \end{array}$$

As usual in modal logic one can get theorems for $\Diamond^M$ by dualization, for instance, the dualized version of T is

$$\forall x : \mathsf{Self}\,. \quad P(x) \quad \supset \quad (\Diamond^M P)(x)$$

Proof The proof follows from basic properties of greatest invariants, for instance $[\![\Box^M P]\!]$ is an invariant, therefore $[\![\Box^M \Box^M P]\!] = [\![\Box^M P]\!]$, which implies 4. $\Box$

Proposition 6.13 *Let $\Xi \mid \Gamma, y : \tau \vdash P : \mathsf{Self} \Rightarrow \mathsf{bool}$ be a predicate, which possibly contains y freely. Then for all $x : \mathsf{Self}$ it holds that*

1. $\forall y : \tau \,.\, (\Box^M P)(x) \quad \supset\!\subset \quad \big(\Box^M \lambda x' : \mathsf{Self}\,.\,\forall y : \tau \,.\, P(x')\big)(x)$
2. $\exists y : \tau \,.\, (\Box^M P)(x) \quad \supset \quad \big(\Box^M \lambda x' : \mathsf{Self}\,.\,\exists y : \tau \,.\, P(x')\big)(x)$

Proof For 1 it suffices to prove that $Q(x) \stackrel{\text{def}}{=} \forall y : \tau \,.\, (\Box^M P)(x)$ is the greatest invariant implying $\forall y : \tau \,.\, P(x)$. Clearly, the predicate Q is an invariant that implies $\forall y : \tau \,.\, P(x)$. Now assume an invariant Q' such that $Q'(x)$ implies $\forall y : \tau \,.\, P(x)$. Then $Q'(x)$ implies also $P(x)$ for an arbitrary but fixed y. By definition $\Box^M P(x)$ is the greatest invariant implying $P(x)$ for any fixed y, therefore $Q'(x)$ implies also $\Box^M P(x)$ for any fixed y. Because the y was chosen arbitrarily, $Q'(x)$ implies $\forall y : \tau \,.\, \Box^M P(x)$, so Q is indeed the greatest invariant.

For 2 it suffices to show that $\exists y : \tau \,.\, (\Box^M P)$ is an invariant that implies $\exists y : \tau \,.\, P$. Both is obvious. $\Box$

6.3. Coalgebraic Class Specifications

A specification is a signature whose class of models is restricted with a set of axioms.

Definition 6.14 (Coalgebraic Class Specification) Let Σ be a coalgebraic class signature with type parameters $\Xi = \alpha_1, \ldots, \alpha_n$.

1. A formula φ is a Σ *method assertion*, if $\Xi \mid x : \mathsf{Self} \vdash \varphi$ and if φ contains no constructor from Σ_C.

2. A formula ψ is a Σ *constructor assertion* if $\Xi \mid \emptyset \vdash \psi$.

3. A *coalgebraic class specification* is a triple $\langle \Sigma, \mathcal{A}_M, \mathcal{A}_C \rangle$ where Σ is a coalgebraic class signature, $\mathcal{A}_M$ is a finite set of Σ method assertions, and $\mathcal{A}_C$ is a finite set of Σ constructor assertions.

Example 6.15 In a class specification for queues it makes sense to demand that the queue constructor new returns an empty queue. Therefore I set

$$\mathsf{Queue} = \langle \Sigma_{\mathsf{Queue}}, \{F_{\mathsf{empty}}, F_{\mathsf{filled}}\}, \{F_{\mathsf{new}}\} \rangle$$

where

$$F_{\mathsf{new}} \quad = \quad [\quad \mathsf{top}(\mathsf{new}) \;=\; \mathsf{bot} \quad]$$

specifies that new delivers the empty queue. ■

The notion of a *subspecification* is needed below. The restriction to method assertions is implied by the restriction to method declarations in subsignatures and will be explained in Subsection 9.1 (on page 90) below.

Definition 6.16 (Subspecification) A class specification $\mathcal{S}' = \langle \Sigma', \mathcal{A}'_M, \mathcal{A}'_C \rangle$ is a *subspecification* of $\mathcal{S} = \langle \Sigma, \mathcal{A}_M, \mathcal{A}_C \rangle$, denoted as $\mathcal{S}' \leq \mathcal{S}$ if $\Sigma' \leq \Sigma$ and $\mathcal{A}'_M \subseteq \mathcal{A}_M$.

Definition 6.17 (Semantics of Class Specifications) Let $\langle \Sigma, \mathcal{A}_M, \mathcal{A}_C \rangle$ be a coalgebraic class specification. A *model* of this class specification is a model $\mathcal{M}$ of Σ such that for all interpretations U_i of the type variables the following holds.

- For all $x \in X$ all method assertions hold: $[\![\varphi]\!]^{\mathcal{M}}\, x = \top$ for all $\varphi \in \mathcal{A}_M$.

- All constructor assertions are fulfilled: $[\![\psi]\!]^{\mathcal{M}} = \top$ for $\psi \in \mathcal{A}_C$.

Example 5.6 has been carefully constructed, it actually is a model of the Queue–specification from Example 6.15. A class specification is *consistent* if it has at least one model with a nonempty state space. Note that for a consistent class specification the models form always a proper class.

Assume a subspecification $\mathcal{S}'$ of $\mathcal{S}$ (involving $\Sigma' \leq \Sigma$) and a model $\mathcal{M} = \langle X, c, a \rangle$ of $\mathcal{S}$. The coalgebra c fulfils all assertions of $\mathcal{S}$, so it obviously fulfils the assertions of $\mathcal{S}'$. Therefore also $\pi_{\Sigma'} \circ c$ fulfils all assertions of $\mathcal{S}'$.

The following standard result describes under which condition one can obtain a final coalgebra satisfying the method assertions of a specification. A first version of it appeared in [Jac96].

Proposition 6.18 *Let $\mathcal{S} = \langle \Sigma, \mathcal{A}_M, \mathcal{A}_C \rangle$ be a consistent coalgebraic class specification over a plain ground signature which contains only polynomial methods. Let τ_Σ be its combined method type. If all method assertions and constructor assertions of $\mathcal{S}$ are invariant with respect to behavioural equality, then there exists a model $\mathcal{M} = \langle X, c, a \rangle$ of $\mathcal{S}$ such that c is the final τ_Σ coalgebra satisfying the method assertions $\mathcal{A}_M$.*

Proof Under the assumptions the semantics of τ_Σ is a polynomial functor. For polynomial functors coalgebra morphisms are bisimulations, therefore every τ_Σ coalgebra morphism preserves the validity of the method and constructor assertions. Let $z : Z \longrightarrow [\![\tau_\Sigma]\!](Z)$ be the final τ_Σ coalgebra (which always exists, see [Rut00, KM00]). Let X be the greatest invariant contained in the interpretation of the method assertions on Z. By Proposition 2.6.8 in [Tew02b] there is an induced coalgebra structure on X, this gives c. It remains to construct the constructor algebra for X. Note that X is nonempty because it must contain the image of the state space of the assumed model. Therefore one can set $a = \,! \circ a'$, where a' is a constructor algebra of an arbitrary model. $\square$

6.4. Class Specifications in CCSL

The concrete syntax of the higher-order logic for CCSL is in Figure 10. CCSL follows quite closely the concrete syntax of PVS. Especially the ASCII representations of the logical notation and the projections is the same as in PVS. There are the following points to note with respect to Figure 10:

- CCSL has concrete syntax for expressions that are syntactic sugar with respect to Definition 6.1, for instance quantification and abstraction over several variables simultaneously, the `LET` construct, function update with `WITH`, or the keyword `IFF`.

- The keywords `ALWAYS` and `EVENTUALLY` give the modal operators from the preceding subsection. The correspondence between the symbolic notation and the concrete grammar of CCSL is as follows:

$$\begin{array}{lcl} \Box^M\ (\lambda x : \mathsf{Self}\,.\, P) & \equiv & \texttt{ALWAYS LAMBDA(}\ x : \texttt{SELF)}\ .\ P\ \texttt{FOR}\ \{\ M\ \} \\ \Diamond^M\ (\lambda x : \mathsf{Self}\,.\, P) & \equiv & \texttt{EVENTUALLY LAMBDA(}\ x : \texttt{SELF)}\ .\ P\ \texttt{FOR}\ \{\ M\ \} \end{array}$$

  The optional identifier with an argument list before the method list can be used to access a modal operator of a different class specification. If it is omitted it defaults to the enclosing class specification.

- The syntax for `CASES` provides terms for the abstract data types. Abstract data types are in Section 7. I define the semantics of the case construct in Subsection 8.1. The case construct in *Terms*(Σ) corresponds to case construct in CCSL for the abstract data type Coproduct, which is defined in the prelude (see Example 4.2 and Subsection 10.8).

formula	::=	`FORALL` (*vardecl* ⦃ , *vardecl* ⦄) (: | .) *formula*
	|	`EXISTS` (*vardecl* ⦃ , *vardecl* ⦄) (: | .) *formula*
	|	`LAMBDA` (*vardecl* ⦃ , *vardecl* ⦄) (: | .) *formula*
	|	`LET` *binding* ⦃ (; | ,) *binding* ⦄ [; | ,] `IN` *formula*
	|	*formula* `IFF` *formula*
	|	*formula* `IMPLIES` *formula*
	|	*formula* `OR` *formula*
	|	*formula* `AND` *formula*
	|	`IF` *formula* `THEN` *formula* `ELSE` *formula*
	|	`NOT` *formula*
	|	*formula* *infix_operator* *formula*
	|	`ALWAYS` *formula* `FOR`
		[*identifier* [*argumentlist*] : :] *methodlist*
	|	`EVENTUALLY` *formula* `FOR`
		[*identifier* [*argumentlist*] : :] *methodlist*
	|	`CASES` *formula* `OF` *caselist* [; | ,] `ENDCASES`
	|	*formula* `WITH` [*update* ⦃ , *update* ⦄]
	|	*formula* . *qualifiedid*
	|	*formula* *formula*
	|	`TRUE`
	|	`FALSE`
	|	`PROJ_N`
	|	*number*
	|	*qualifiedid*
	|	(*formula* : *type*)
	|	(*formula* ⦃ , *formula* ⦄)
vardecl	::=	*identifier* ⦃ , *identifier* ⦄ : *type*
methodlist	::=	{ *identifier* ⦃ , *identifier* ⦄ }
binding	::=	*identifier* [: *type*] = *formula*
caselist	::=	*pattern* : *formula* ⦃ (; | ,) *pattern* : *formula* ⦄
pattern	::=	*identifier* [(*identifier* ⦃ , *identifier* ⦄)]
update	::=	*formula* := *formula*

Figure 10: CCSL Syntax for expressions and formulae

- CCSL allows object-oriented syntax for method calls: One can write $x.m(-)$ instead of $m(x, -)$ to give the specifications a bit more object-oriented look and feel.
- The projections are `PROJ_N`, where `N` stands for a natural number.
- Infix operators are sequences of special characters like `*`, the details are in Subsection 10.5.
- CCSL tries to be relaxed about the use of delimiters. For instance, variable binders like `FORALL` can be separated from the following formula by either a colon or a dot (thus comforting users of PVS *and* ISABELLE). The last delimiter in a list of cases or let bindings is optional.
- The precedence for the constructions in Figure 10 increases from the top to the bottom. So conjunction (`AND`) binds stronger then disjunction (`OR`). For instance the expression f_1 `OR` $f_2\ f_3$. m is parsed as f_1 `OR` $((f_2\ f_3)$. m).
- The CCSL compiler defines four special class members that make the notions of invariants, bisimulations, coalgebra morphisms (in the form of recognisers), and coinduction (in the form of the coreduce combinator) available in the logic of CCSL. Their names are as follows:

concept	identifier for class ⟨`class`⟩	relevant definition
invariant	⟨`class`⟩`_class_invariant?`	Definition 5.10 (1)
bisimulation	⟨`class`⟩`_class_bisimulation?`	Definition 5.10 (3)
morphism	⟨`class`⟩`_class_morphism?`	Definition 3.2.2 of [Tew02b]
coinduction	`coreduce`	Item Coreduce on page 82

  The types of these identifiers depend on the method declarations that are present in the signature, see Subsection 8.1 for the details. The identifier for coreduce does not depend on the class name, so one usually has to use a qualified identifier for it (see Subsection 10.6 on page 102). Figure 11 shows as an example how to express in CCSL that the predicate F_{finite} from Example 6.10 is an invariant for queues.

  Apart from `coreduce` (whose semantics also depends on the method assertions) these identifiers are visible in method and constructor assertions. (Technically the compiler makes a ground signature extension just before processing method and constructor assertions.)

- The current compiler version supports only one kind of immediate constants: natural numbers. Their type can be changed via the `-nattype` command line switch, see Subsection 10.9.

The syntax for class specifications was already given in Subsection 5.2. Recall that the sections for attributes, methods and constructors contributed to the signature. The section for assertions contains method assertions in the sense of Definition 6.14, the section for creation conditions contains constructor assertions. The theorem section gives one the opportunity to state theorems in the logic of CCSL that are believed to hold for all models. From the point of view of the CCSL compiler the theorem section contains arbitrary formulae (without influence on the semantics of the class specification) that should get translated into the logic of the target theorem prover. The syntax of these sections is as follows:

assertionsection	::=	`ASSERTION` {| *importing* |} [*assertionselfvar*] {| *freevarlist* |} *namedformula* {| *namedformula* |}
assertionselfvar	::=	`SELFVAR` *identifier* `:` `SELF`
freevarlist	::=	`VAR` *vardecl* {| `;` *vardecl* |}
creationsection	::=	`CREATION` {| *importing* |} {| *freevarlist* |} *namedformula* {| *namedformula* |}
theoremsection	::=	`THEOREM` {| *importing* |} {| *freevarlist* |} *namedformula* {| *namedformula* |}
namedformula	::=	*identifier* `:` *formula* `;`

All three sections can contain an arbitrary number of (named) formulae. The importings are explained in Section 10.4. Every assertion section can contain a `SELFVAR` declaration for the free variable that can occur in method assertions. Variables declared with the keyword `VAR` constitute a context for all formulae of the affected section. This is syntactic sugar: The CCSL compiler universally quantifies all the variables declared with `VAR` on the outermost level.

The complete queue specification in CCSL syntax is in Figure 11. Besides the three assertions $F_{\mathsf{empty}}, F_{\mathsf{filled}}$, and F_{new} from Example 6.15 it contains also two theorems. The first one corresponds to Equation 3 and the second one states that the predicate F_{finite} from Example 6.10 is an invariant.[18] The CCSL compiler translates the two theorems into lemmas in a separate PVS file. The utility lemmas that are necessary for the two theorems are in the PVS theory QueueModal, to make them available I add an appropriate importing clause.

In the remainder of this subsection I show how the CCSL compiler translates the two queue assertions into PVS. Further below I discuss how the compiler treats attribute declarations and how the associated update assertions look like.

During type checking the CCSL compiler records that the queue assertions use behavioural equality on the type $\mathsf{Lift}[A \times \mathsf{Self}]$. Therefore it generates the theory

[18] The distributed sources contain also a theorem that corresponds to Equation 2. Unfortunately it does not fit into the figure.

```
Begin Queue[ A : Type ] : ClassSpec
  Method
    put : [Self, A] -> Self;
    top : Self -> Lift[[A,Self]];

  Constructor
    new : Self;

  Assertion Selfvar x : Self
    q_empty :  x.top ~ bot      Implies
        Forall(a : A) . x.put(a).top ~ up(a,x);

    q_filled :    Forall(a1 : A, y : Self) . x.top ~ up(a1, y)      Implies
        Forall(a2 : A) . x.put(a2).top ~ up(a1, y.put(a2));

  Creation
    q_new : new.top ~ bot;

  Theorem
    Importing QueueModal[Self, A]

    strong_reachable :  Forall(p, q : Self) :
      Let finite? : [Self -> bool] = Lambda(q : Self) :
              (Eventually Lambda(x : Self) : x.top = bot For {top}) q
      IN
        finite? p And finite? q    Implies
          (Eventually
               (Eventually Lambda(r : Self) : r ~ q For {put})
           For {top}
          ) p;

    finite_invariant :
      Let finite? : [Self -> bool] = Lambda(q : Self) :
              (Eventually Lambda(x : Self) : x.top = bot For {top}) q
      IN
        Queue_class_invariant?(put, top)(finite?) ;
End Queue
```

Figure 11: The queue specification in CCSL syntax

QueueReqObsEq, which contains the following lifting of bisimilarity to the type Lift[$A \times$ Self].

```
c : Var QueueSignature[Self, A]

ObsEq_Lift_A_Self(c) : [[Lift[[A, Self]], Lift[[A, Self]]] -> bool] =
    Lambda(l1 : Lift[[A, Self]], l2 : Lift[[A, Self]]) :
        Cases l1 OF
            bot      : bot?(l2),
            up(p0) : up?(l2)    And    PROJ_1(p0) = Proj_1(down(l2))    And
                         bisim?(c)(PROJ_2(p0), PROJ_2(down(l2)))
        Endcases
```

The method assertions are translated into predicates on queue coalgebras. For the method assertion q_empty the compiler generates the predicate q_empty?. The following material is taken from the theory QueueSemantics.

```
q_empty?(c) : [Self -> bool] = Lambda (x : Self) :
    ObsEq_Lift_A_Self(c)(top(c)(x), bot)       Implies
    Forall(a : A) : ObsEq_Lift_A_Self(c)(top(c)(put(c)(x, a)), up(a, x))
```

Observe that the compiler inserts coalgebras to make the methods dependent on a model. The assertion q_filled is translated into PVS in the same way. All such translated method assertions are combined into one ⟨class⟩Assert? predicate, which holds precisely on those coalgebras that fulfil all method assertions.

```
QueueAssert?(c) : bool =
    Forall(x : Self) : q_empty?(c)(x)    And   q_filled?(c)(x)
```

In a similar way the constructor assertions are translated into predicates on the constructor algebra. Like in the queue example the constructor assertions can contain methods. Therefore the predicates for the constructor assertions depend on an interpretation of the methods. In the PVS translation this is captured with an additional argument c:

```
q_new?(c) : [QueueConstructors[Self, A] -> bool] =
    Lambda(z : QueueConstructors[Self, A]) :
        ObsEq_Lift_A_Self(c)(top(c)(new(z)), bot)

QueueCreate?(c) : [QueueConstructors[Self, A] -> bool] =
    Lambda(z : QueueConstructors[Self, A]) : q_new?(c)(z)
```

Finally the predicates ⟨class⟩Assert and ⟨class⟩Create are combined into an recogniser of queue models:

```
QueueModel?(c : QueueSignature[Self, A], z : QueueConstructors[Self, A]) : bool=
    QueueAssert?(c)    And    QueueCreate?(c)(z)
```

The construction of a model of the queue specification in the target theorem prover consists of a prove of a theorem of the form

```
model : Proposition QueueModel?[QueueState, A](Queue_c, Queue_constr)
```

where QueueState is the type of the state space of the model and A is a type parameter. The records Queue_c and Queue_constr contain (the user defined) interpretation of the queue signature.

As I said before an important task of the CCSL compiler is the generation of lemmas. For every method and constructor assertion the compiler generates one lemma. The lemma for the assertion q_empty looks as follows:

```
q_empty : Lemma
    Forall(c : QueueSignature[Self, A], x : Self) : QueueAssert?(c)     Implies
        ObsEq_Lift_A_Self(c)(top(c)(x), bot)     Implies
            Forall(a : A) : ObsEq_Lift_A_Self(c)(top(c)(put(c)(x, a)), up(a, x)
```

Let me now discuss attribute declarations and their associated assertions. If the signature of the class contains attributes, then the compiler generates not only additional method declarations. It also generates additional assertions that describe the behaviour of the attributes with respect to the generated update methods. Assume for an example a specification with the following attribute declarations (where U and V are type parameters):

```
Attribute
  a1 : Self -> Bool;
  a2 : [Self, U] -> V;
```

As update methods the compiler generates the following two method declarations.

```
  set_a1 : [Self, Bool] -> Self;
  set_a2 : [Self, U, V] -> Self;
```

For each combination of attribute and update method there is an *update assertion* generated that describes if and how the attribute changes. In CCSL syntax these assertions would look as follows.

```
Assertion SelfVar x : Self
  a1_set_a1 : Forall (y : Bool) : a1(set_a1(x, y)) = y;

  a1_set_a2 : Forall (u : U, v : V) : a1(set_a2(x, u, v)) = a1(x);

  a2_set_a1 : Forall (y : Bool, u : U) : a2(set_a1(x, y), u) = a2(x, u);

  a2_set_a2 : Forall (u1 : U, u2 : U, v : V) : a2(set_a2(x, u1, v), u2) =
                          IF u1 = u2 Then v Else a2(x, u2);
```

7. Abstract Data Types

Abstract data types are widely accepted as the right formalism to specify finitely generated data structures such as lists or trees. Many functional programming languages (e.g., SML [MTH91], OCAML [LDG+01], and Haskell [ABB+99, HPF92]) allow the definition of abstract data types. The logic of the theorem provers PVS and ISABELLE/HOL has been extended with means to specify abstract data types [OS93, BW99].

Although it is possible, it does not make much sense to model abstract data types with behavioural types. Therefore CCSL contains the possibility to specify abstract data types as initial algebras. This way, the decision whether to choose an algebraic or a coalgebraic approach to model a given type is left to the user. Further, it is possible to mix abstract data type specifications with coalgebraic class specifications, this leads to iterated specifications, see Section 8.

For reasons that have been described in the introduction the CCSL compiler accepts currently only abstract data type specifications without axioms. As a consequence also this report is restricted to abstract data types. There is no problem with general algebraic specifications. The extension of CCSL with general algebraic specifications is one of the points that remain to be done in the future.

Definition 7.1 (ADT) Assume a ground signature Ω. An *abstract data type specification* is a finite set Σ of constructor declarations $c_i : \sigma_i$ where σ is a constructor type. The type variables occurring in the σ_i are the *type parameters* of the abstract data type specification.

Recall from Definition 3.7 (on page 25) that a constructor type is a type expression $\sigma \Rightarrow \mathsf{Self}$ such that Self occurs in σ only strictly positive. This restriction in the above definition is necessary, because initial algebras exist only for certain functors [Gun92, OS93, BW99].

The semantics of abstract data type specifications is given by a collection of initial algebras.

Definition 7.2 Let Σ be an abstract data type specification with n type parameters and k constructors $c_1, \ldots, c_k$. Let $\sigma_\Sigma = \mathcal{T}_C(c_1) + \cdots + \mathcal{T}_C(c_k)$ denote the combined constructor type of Σ. A *model* for Σ is an indexed collection of pairs $\big(\langle X, a\rangle_{U_1,\ldots U_n}\big)_{U_i \in |\mathbf{Set}|}$ such that for each interpretation $U_1, \ldots, U_n$ of the type variables

$$[\![\sigma_\Sigma]\!]_{U_1,U_1,U_2,U_2,\ldots,U_n,U_n}(X, X) \xrightarrow{\quad a \quad} X$$

is an initial algebra.

7.1. CCSL Syntax for Abstract Data Types

The concrete syntax for abstract data type specifications is similar to that of class specifications. The keyword `ADT` indicates an abstract data type.

adtspec ::= `BEGIN` *identifier* [*parameterlist*] `:` `ADT`

Begin tree[A, B : **Type**] : **Adt**
  Constructor
    leaf : B –> **Carrier**;
    node : [**Carrier**, A, **Carrier**] –> **Carrier**
End tree

Figure 12: The abstract data type of binary trees in CCSL

		⦃ *adtsection* ⦄
		`END` *identifier*
adtsection	::=	*adtconstructorlist* [;]
adtconstructorlist	::=	`CONSTRUCTOR` *adtconstructor* ⦃ ; *adtconstructor* ⦄
adtconstructor	::=	*identifier* [*adtaccessors*] : *type*
	|	*identifier* [*adtaccessors*] : *type* `->` *type*
adtaccessors	::=	(*identifier* ⦃ , *identifier* ⦄)

The accessors are syntactic sugar, so let me ignore them for a moment. The set of declared adt-constructors constitute an abstract data type specification. The compiler checks that the types are constructor types. Recall, that for this presentation I assume only one special type Self, whereas the CCSL compiler has two keywords for it, `SELF` and `CARRIER`. In abstract data type specification one has to use the latter one.

From every constructor the compiler derives a recogniser predicate by appending a question mark (for PVS) or prepending the prefix is_ (for ISABELLE). A recogniser holds for an element of the abstract data type if this element was built with the corresponding constructor.

The optional accessors declare (partial) accessor function. If accessors are given then their number must match the number of arguments of the constructor and their names must be unique. Accessors allow one to decompose an element of the abstract data type and extract the arguments of the constructor with which this element was built. Because accessors are partial functions, they cannot be formalised in the setting of this report. Nevertheless the CCSL compiler allows them. The compiler together with the semantics of the theorem provers PVS and ISABELLE/HOL ensure a correct treatment of these partial functions (I discussed this issue already in Example 4.2).

Figure 12 shows as an example the abstract data type of binary trees in CCSL syntax.

The compiler does not generate the semantics for abstract data types. It rather outputs an abstract data type declaration in the syntax of the target theorem prover. Both ISABELLE/HOL and PVS use an initial semantics for their abstract data types. In ISABELLE this is implemented as a conservative extension[19] [BW99]; PVS uses an axiomatic ap-

[19]Provided the `quick_and_dirty` flag is set to false.

proach [OS93].

The simple mapping of CCSL data type definitions to the target theorem prover has one serious drawback: Inside an abstract data type all type constructors stemming from a class specification may only be instantiated with constant types. This is because both PVS and ISABELLE place restrictions on the types that may be used in a nested recursion (on the type level) with an abstract data type definition. In principle, nested recursion with (some) behavioural types could be allowed, see the following Section 8.

The theorem provers PVS and ISABELLE give different support for their versions of abstract data types: Recognisers, accessors, and the map combinator are not provided by the data type package of ISABELLE/HOL. More importantly, some notions, which are needed when abstract data types occur inside coalgebraic class specifications, are not supported in the needed generality or are not supported at all. For instance neither PVS nor ISABELLE defines relation lifting for abstract data types. PVS generates for every abstract data type the combinators every and map. For an abstract data type in which all type variables occur at strictly positive position the combinator every coincides with predicate lifting and the combinator map with the morphism component of the semantics of the data type. If a type variable occurs not strictly positive then PVS does not generate map. The combinator every is still generated but disregards all type variables occurring not only strictly positive. So in this case every cannot be used for predicate lifting.

The CCSL compiler works hard to blur the differences between data type definitions in PVS and ISABELLE. It also fixes some of their shortcomings. For ISABELLE the compiler generates definitions for recogniser predicates, and accessor functions. For both PVS and ISABELLE the compiler generates predicate and relation lifting for the abstract data type as described in [Hen99] and in the following Section.

In the remainder of this section I show what the CCSL compiler generates for the data type of trees of Figure 12. For a diversion I show this time what is generated for ISABELLE/HOL.

As first the data type tree is defined:

```
datatype ('A, 'B) tree =
    leaf "'B"
  | node "('A, 'B) tree"   "'A"   "('A, 'B) tree"
```

ISABELLE/HOL data type declarations have an SML like syntax. Different constructors are separated with a vertical bar and the argument types follow the name of the constructor. Identifiers that start with a tick like 'A are free type variables. Instantiations of type constructors are written in a postfix form. The ISABELLE type expression ('A, 'B) tree corresponds to tree[A, B] in PVS. ISABELLE requires the user to enclose all special syntax from the object logic in double quotes. The CCSL compiler behaves conservatively and puts double quotes around all critical entities.

For the definition of functions over data types the ISABELLE/HOL documentation advocates the **primrec** feature. However, I found that functions defined with **primrec** are quite slow to type check. More importantly, **primec** is quite difficult to use with nested data types.[20] I therefore decided to base all definitions on a self-defined reduce combinator. This has the additional advantage that more modules in the CCSL compiler get independent of the target theorem prover (in PVS the reduce combinator is provided by the system). The disadvantage is that for ISABELLE the CCSL compiler has to derive the reduce combinator from the internal recursion combinator of ISABELLE. In the tree example this looks as follows:

```
constdefs reduce_tree :: "('B => 'Result) =>
                ('Result => 'A => 'Result => 'Result) =>
                ('A, 'B) tree => 'Result"
  "reduce_tree leaf_fun node_fun   ==   %(t :: ('A, 'B) tree) .
     tree_rec ( %(b1 :: 'B) .  leaf_fun b1)
              ( %(x1 :: ('A, 'B) tree)  (a1 :: 'A)   (x2 :: ('A, 'B) tree)
                  (x3 :: 'Result)   (x4 :: 'Result) . node_fun x3 a1 x4) t"
```

In ISABELLE a constant definition starts with the keyword **constdefs**, followed by a type annotation (on the first three lines) and a string that contains a meta equality (==). The higher-order function reduce_tree takes as argument a tree algebra on 'Result (consisting of two functions, one for the constructor leaf and one for node). It returns the unique tree–algebra morphism originating in the initial tree algebra (compare Item reduce in Subsection 8.1 on page 75). The definition of reduce_tree uses ISABELLE's internal recursion operator tree_rec. This internal combinator could be paraphrased as strong reduce combinator for the unfolded data type. The percent sign % is the ASCII version of λ in ISABELLE.

The reduce combinator gives the induction proof principle (sometimes known as primitive recursion) for trees. It allows one to define recursive functions on trees without doing recursion. For instance to count the number of nodes in a tree t one can use the following expression:

```
reduce_tree (% (b :: 'B) . 0)   (% (l :: nat) (a :: 'A) (r :: % nat) . l + r +1)  t
```

The data type package of ISABELLE/HOL does neither provide accessors nor recogniser predicates. The CCSL compiler generates code to define them. For instance:

```
constdefs leaf_acc :: "('A, 'B) tree => 'B"
  "leaf_acc t == case t of
       leaf b => b
     | node p0 a p1 => arbitrary"
```

In case one applies leaf_acc to a node one gets arbitrary as result, where arbitrary is special constant that inhabits every type.[21]

[20] The problem is that one cannot pass the function being defined by the current **primrec** into an already defined (higher-order) function. Therefore, for nested data types, one cannot use the map combinator of the nested data type.

[21] Recall that the semantics of ISABELLE/HOL is based on an universe of nonempty sets. So arbitrary can be obtained by applying the Axiom of Choice to every type.

The recognisers can be easily defined with reduce, as example I show the recogniser for leafs:

```
constdefs is_leaf :: "('A, 'B) tree => bool"
  "is_leaf == % (t :: ('A, 'B) tree) .
     reduce_tree (% (b :: 'B) . True)
                 (% (x1 :: bool) (a :: 'A) (x2 :: bool) . False) t"
```

There are three more definitions that are generated for every abstract data type: The map combinator, predicate lifting and relation lifting. Here I show only the map combinator and predicate lifting. Relation lifting for abstract data types is quite difficult to understand. It is explained in Item 3 of Remark 8.1 (on page 78f).

```
constdefs treeMap :: "('A1 => 'A2) => ('B1 => 'B2) =>
                        ('A1, 'B1) tree => ('A2, 'B2) tree"
  "treeMap f g == % (t :: ('A1, 'B1) tree) .
     reduce_tree (% (b :: 'B1) . leaf (g b))
                 (% (p0 :: ('A2, 'B2) tree) (a :: 'A1)
                 (p1 :: ('A2, 'B2) tree) . node p0 (f a) p1) t"
```

The map combinator takes two functions as arguments, one for transforming the labels in the leafs and one for the labels of the nodes and applies both functions recursively in the whole tree.

```
constdefs
  Everytree :: "('A => bool) => ('B => bool) => ('A, 'B) tree => bool"
  "Everytree P Q ==
       reduce_tree (% (b :: 'B) . Q b)
                   (% (x1 :: bool) (a :: 'A)  (x2 :: bool) .
                          (x1 = True) & (P a) & (x2 = True))"
```

Predicate lifting takes two argument predicates, one on the type parameter A and one on B. It applies these predicates in the whole tree and returns true if all labels are in the supplied predicates (the ampersand & denotes conjunction in ISABELLE).

8. Iterated Specifications

In analogy with the *iterated data types* of [Hen99] and [HJ97] I use the informal term *iterated specification* to describe a situation in which a specification $\mathcal{S}_i$ depends on a specification $\mathcal{S}_j$. Both involved specifications $\mathcal{S}_i$ and $\mathcal{S}_j$ can be either coalgebraic class specifications or abstract data type specifications. In a typical example the dependence on $\mathcal{S}_j$ comes from the signature of $\mathcal{S}_i$, which may involve a type constructor $\mathsf{C}_{\mathcal{S}_j}$ whose semantics is a distinguished model of $\mathcal{S}_j$. However, it can also be the case that only some assertion of $\mathcal{S}_i$ uses a constructor

```
Begin list[ T : Type ] : Adt
  Constructor
    null : Carrier;
    cons( car, cdr ) : [T, Carrier] -> Carrier
End list

Begin InfTreeFin[ T : Type ] : ClassSpec
  Method
    branch : Self -> List[[T, Self]];
End InfTreeFin
```

Figure 13: Trees of finite width and (possibly) infinite depth in CCSL (from [Hen99])

of $\mathcal{S}_j$. Note that the use of *iterated* refers to iteration of different induction and coinduction principles that come into play in the described situation. Iterated does *not* refer to a mutual recursion of the specifications $\mathcal{S}_i$ and $\mathcal{S}_j$. In fact in CCSL the dependency relation between specifications is always a strict order.

An example of an iterated data type (that is an iterated specification without assertions) is the behavioural data type of trees of (possibly) infinite depth and arbitrary but finite width. The example appears originally in [Hen99]. Figure 13 shows it in CCSL syntax. The crucial point here is, that the class specification InfTreeFin involves the type constructor List. This type constructor and its semantics is defined by the first specification List.

The functors that capture signatures of iterated specifications are usually called *data functors*. Data functors have been studied in [Jay96, CS92] and also in [HJ97, Hen99, Röß00a, Röß00b]. The work of Cockett and Spencer led to the categorical programming language CHARITY [CF92]. In a sense CCSL can be viewed as the specification language for CHARITY. The work of Hensel and Jacobs describes definition and proof principles for iterated data types under the assumption that suitable initial algebras and final coalgebras do exist in the base category. Rößiger proves that initial algebras and final coalgebras exist in **Set** for all (covariant) data functors. These latter two results are plugged together in this section.

The iterated specifications of CCSL are more general than the iterated data types that have been considered by Hensel, Jacobs, and Rößiger. Up to my knowledge, there are a number of open questions related to the semantics of iterated specifications. Therefore, a complete treatment of iterated specifications is beyond the scope of the present report. The approach taken here is very pragmatic: Until better solutions are available, CCSL uses predicate lifting and relation lifting for abstract data types and behavioural types as described in [HJ97, Hen99]. Because of the greater generality of iterated specifications in CCSL, the various liftings are not always well defined (for instance in case a class contains binary methods). In case they are not defined certain restrictions are imposed on iterated specifications (via improper ground signatures).

The first subsection describes the technicalities. The material is taken from [Hen99] and adopted to the setting of this report. I can only give the definitions here, for the rationale behind them I refer the reader to [HJ97] or [Hen99]. The second subsection characterises those iterated specifications which have a well-defined semantics. The third subsection gives guidelines on how to ensure consistency for iterated specifications.

8.1. Semantics of Iterated Specifications

The technical means to allow type checking and semantics for iterated specifications are ground signatures. A CCSL specification consists of a finite list of entities $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n$, standing one after each other in one file. Each of the $\mathcal{S}_i$ is either a ground signature extension, a class specification, or an abstract data type specification. For each of the $\mathcal{S}_i$ there is a ground signature Ω_i and a model $\mathcal{M}_i$ of it, which are both not explicit in the CCSL source. The first pair $\langle \Omega_0, \mathcal{M}_0 \rangle$ consists of the empty ground signature (i.e., $|\Omega_0| = \emptyset$ and $\Omega_{0\sigma} = \emptyset$) and the empty model. Each of the $\mathcal{S}_i$ can define type constructors and constants. These items are added[22] to Ω_i and $\mathcal{M}_i$ to yield Ω_{i+1} and $\mathcal{M}_{i+1}$. Then Ω_{i+1} is used to type check $\mathcal{S}_{i+1}$ and $\mathcal{M}_{i+1}$ is used for the semantics of $\mathcal{S}_{i+1}$. This way a specification has access to (or can use) all the specifications and all the ground signature extensions that appear before it.

In the following I consider an arbitrary $\mathcal{S}$ from the finite list of $\mathcal{S}_i$ with associated ground signature Ω and a model $\mathcal{M}$ of Ω. For the three possibilities ($\mathcal{S}$ is a ground signature, an abstract data type, or a class specification) I describe which type constructors and constants are defined by $\mathcal{S}$ and what semantics they have. For some of the items a semantics can only be defined if the model $\mathcal{M}$ is proper and/or additional conditions hold. For these items I take the following approach: I first describe their semantics. If this is well defined, then the corresponding item is defined by $\mathcal{S}$ and added to the ground signature. The item stays undefined (and is not added to the ground signature) otherwise. This way it can happen that the ground signature Ω' (or its model) that is build from Ω and $\mathcal{S}$ is not proper, despite the fact that Ω (or its model) is proper. This has the described consequences for subsequent specifications.

The CCSL compiler deviates slightly from what I describe in the following. It generally uses only one interpretation for any given type variable, see Subsection 3.4.

8.1.1. Ground Signature Extensions

Let $\mathcal{S}$ be a ground signature extension. To state the obvious, $\mathcal{S}$ defines all items which are declared in $\mathcal{S}$. The semantics is taken from the environment or from the CCSL source as described in Subsection 4.

8.1.2. Abstract Data Type Specifications

Let $\mathcal{S}$ be an abstract data type specification with $\Bbbk$ type parameters $\alpha_1, \ldots, \alpha_\Bbbk$ and n constructor declarations $c_1 : \sigma_1, \ldots, c_n : \sigma_n$. Recall from page 34 that the combined constructor

[22] I disregard name clashes here. In the CCSL compiler later defined items hide earlier ones with the same name (there is no overloading).

type of $\mathcal{S}$ is defined as $\sigma_{\mathcal{S}} = \mathcal{T}_C(c_1) + \cdots + \mathcal{T}_C(c_n)$. Recall also that the special type Self, which functions in the σ_i as place holder for the data type being defined, occurs in all the σ_i only strictly covariantly. Therefore I drop the contravariant argument, when considering the semantics of $\sigma_{\mathcal{S}}$. For data type specifications the variance of the type parameters is defined as

$$\mathcal{V}_x(\mathcal{S}) \quad \stackrel{\mathrm{def}}{=} \quad \mathcal{V}_x(\sigma_{\mathcal{S}})$$

The following list describes what items are defined by the specification $\mathcal{S}$. In the description I use $\overline{U}$ to denote an arbitrary interpretation of the type variables $\overline{\alpha} = \alpha_1, \ldots, \alpha_{\Bbbk}$. For the interpretation of types positive and negative occurrences of the type variables are interpreted with different sets. However, for terms every type variable is interpreted by only one set. So depending on the context $\overline{U}$ denotes either $2\Bbbk$ sets $U_1^-, U_1^+, \ldots, U_{\Bbbk}^-, U_{\Bbbk}^+$ or only $\Bbbk$ sets $U_1, \ldots, U_{\Bbbk}$. Further I set $F_{\overline{U}}^{\mathcal{S}}(X) \stackrel{\mathrm{def}}{=} [\![\sigma_{\mathcal{S}}]\!]_{\overline{U}}(X)$.

Type Constructor $\mathsf{C}_{\mathcal{S}}$ Let $\delta_{\overline{U}} : F_{\overline{U}}^{\mathcal{S}}(X_{\overline{U}}) \longrightarrow X_{\overline{U}}$ denote the initial $F_{\overline{U}}^{\mathcal{S}}$ algebra. If $\delta_{\overline{U}}$ exists for all interpretations $\overline{U}$, then the specification defines the type constructor $\mathsf{C}_{\mathcal{S}}$ of arity $\Bbbk$:

$$\vdash \mathsf{C}_{\mathcal{S}} \;::\; [\, \mathcal{V}_{\alpha_1}(\mathcal{S}); \ldots ; \mathcal{V}_{\alpha_{\Bbbk}}(\mathcal{S})]$$

Its semantics is the carrier of the initial algebra:

$$[\![\mathsf{C}_{\mathcal{S}}]\!](\overline{U}) \quad = \quad X_{\overline{U}}$$

The action on morphisms can be defined via reduce, see below. None of the following items is defined, if $\delta_{\overline{U}}$ does not exist for all $\overline{U}$.

Constructors For every constructor declaration $c_i : \sigma_i$ the specification $\mathcal{S}$ defines a constant

$$\overline{\alpha} \;\mid\; \emptyset \;\vdash\; \mathsf{c}_i \;:\; \sigma_i[\, \mathsf{C}_{\mathcal{S}}[\overline{\alpha}] \;/\; \mathsf{Self}\,]$$

where $[\, \mathsf{C}_{\mathcal{S}}[\overline{\alpha}] \;/\; \mathsf{Self}\,]$ denotes the substitution of $\mathsf{C}_{\mathcal{S}}[\overline{\alpha}]$ for Self in σ_i. The semantics is

$$[\![\mathsf{c}_i]\!]_{\overline{U}} \quad = \quad \delta_{\overline{U}} \circ \kappa_i$$

where κ_i is the interpretation injection belonging to c_i (compare page 36).

Reduce The (higher-order) function reduce creates the unique algebra morphism out of the initial $[\![\sigma_{\mathcal{S}}]\!]$ algebra.

$$\overline{\alpha}, \beta \;\mid\; \emptyset \;\vdash\; \mathsf{reduce}_{\mathcal{S}} \;:\; (\sigma_1[\,\beta \;/\; \mathsf{Self}\,] \times \cdots \times \sigma_n[\,\beta \;/\; \mathsf{Self}\,]) \;\Rightarrow\; \mathsf{C}_{\mathcal{S}}[\overline{\alpha}] \;\Rightarrow\; \beta$$

Let V be the interpretation of β and fix a list of functions $\overline{f} = f_1 : [\![\sigma_1]\!]_{\overline{U}}(V), \ldots, f_n : [\![\sigma_n]\!]_{\overline{U}}(V)$. Note that the copairing $[f_1, \ldots, f_n]$ is then a function with codomain V.

Now $[\![\mathsf{reduce}]\!]_{\overline{U},V}(\overline{f})$ is defined as the unique function that makes the following diagram commute.

$$\begin{tikzcd}
F^{\mathcal{S}}_{\overline{U}}(X_{\overline{U}}) \arrow[rr, "\delta_{\overline{U}}"] \arrow[d, dashed, "F^{\mathcal{S}}_{\overline{U}}([\![\mathsf{reduce}_{\mathcal{S}}]\!]_{\overline{U},V}(\overline{f}))"'] & & X_{\overline{U}} \arrow[d, dashed, "{[\![\mathsf{reduce}_{\mathcal{S}}]\!]_{\overline{U},V}(\overline{f})}"] \\
F^{\mathcal{S}}_{\overline{U}}(V) \arrow[rr, "{[f_1, \ldots, f_n]}"] & & V
\end{tikzcd}$$

Map The map combinator gives the action of the functor $[\![\mathsf{C}_{\mathcal{S}}]\!]$ on morphisms. Recall from Section 4 that for a proper model $\mathcal{M}$ of the ground signature Ω the interpretation of the σ_i can be considered as a functor taking $2\Bbbk + 1$ arguments[23] Its action on morphisms is denoted with $[\![\sigma_i]\!]_{\overline{g}}(f)$ for suitable functions $\overline{g}$ and f. For each constructor c_i set $\widehat{\sigma}_i \overset{\text{def}}{=} \mathcal{T}_C(c_i)$. Let $\overline{V} = V_1^-, V_1^+, \ldots, V_{\Bbbk}^-, V_{\Bbbk}^+$ be another interpretation of the type parameters $\alpha_1, \ldots, \alpha_{\Bbbk}$ and let $\overline{g} = g_1^-, g_1^+, \ldots, g_{\Bbbk}^-, g_{\Bbbk}^+$ be a list of functions such that

$$\begin{array}{ll} g_1^- : V_1^- \longrightarrow U_1^- & \\ g_1^+ : U_1^+ \longrightarrow V_1^+ & \end{array} \quad \cdots \quad \begin{array}{ll} g_{\Bbbk}^- : V_{\Bbbk}^- \longrightarrow U_{\Bbbk}^- \\ g_{\Bbbk}^+ : U_{\Bbbk}^+ \longrightarrow V_{\Bbbk}^+ \end{array}$$

Then, for proper models $\mathcal{M}$ of the ground signature, the semantics of $\mathsf{C}_{\mathcal{S}}$ is extended to a functor via

$$[\![\mathsf{C}_{\mathcal{S}}]\!](\overline{g}) \quad = \quad [\![\mathsf{reduce}_{\mathcal{S}}]\!]_{\overline{U},X_{\overline{V}}}\left([\![c_1]\!]_{\overline{V}} \circ [\![\widehat{\sigma}_1]\!]_{\overline{g}}(\mathrm{id}_{X_{\overline{V}}}),\ \ldots,\ [\![c_n]\!]_{\overline{V}} \circ [\![\widehat{\sigma}_n]\!]_{\overline{g}}(\mathrm{id}_{X_{\overline{V}}})\right)$$

Case Distinction For each constructor c_i let $\widehat{\sigma}_i \overset{\text{def}}{=} \mathcal{T}_C(c_i)[\,\mathsf{C}_{\mathcal{S}}[\overline{\alpha}]\ /\ \mathsf{Self}\,]$. The specification $\mathcal{S}$ defines case distinction as

$$\overline{\alpha}, \beta \mid \emptyset \vdash \mathsf{case}_{\mathcal{S}} \ :\ \big((\widehat{\sigma}_1 \Rightarrow \beta) \times \cdots \times (\widehat{\sigma}_n \Rightarrow \beta)\big) \ \Rightarrow\ \mathsf{C}_{\mathcal{S}}[\overline{\alpha}] \ \Rightarrow\ \beta$$

Let V be an interpretation for β, $f_1, \ldots, f_n$ be a suitable vector of functions, and $x \in X_{\overline{U}}$. Then

$$[\![\mathsf{case}_{\mathcal{S}}]\!]_{\overline{U},V}(f_1, \ldots, f_n)(x) \quad = \quad \begin{cases} & \vdots \\ f_i(y) & \text{if } \exists y \in [\![\widehat{\sigma}_i]\!]_{\overline{U}}\ .\ x = \delta_{\overline{U}}(\kappa_i\, y) \\ & \vdots \end{cases}$$

Recognisers For each constructor declaration $c_i : \sigma_i$ the specification $\mathcal{S}$ defines the recogniser

$$\overline{\alpha} \mid \emptyset \vdash \mathsf{c?}_i \ :\ \mathsf{C}_{\mathcal{S}}[\overline{\alpha}] \Rightarrow \mathsf{Prop}$$

[23] The contravariant argument for Self is ignored here.

The semantics is

$$[\![\mathsf{c?}_i]\!]_{\overline{U}}(x) \quad = \quad [\![\mathsf{case}_{\mathcal{S}}]\!]_{\overline{U},\mathsf{bool}}(f_1, \ldots, f_n)(x)$$

where $f_i = \lambda x \,.\, \top$ and $f_j = \lambda x \,.\, \bot$ for $j \neq i$.

Predicate Lifting $\mathsf{Pred}_{\mathsf{C}_{\mathcal{S}}}$ For predicate lifting consider the following operator for a fixed list of $2\Bbbk$ parameter predicates $\overline{P}$.[24]

$$Q \subseteq X_{\overline{U}} \longmapsto \coprod_{\delta_{\overline{U}}} \mathrm{Pred}([\![\sigma_{\mathcal{S}}]\!])(\overline{P}, Q, Q) \tag{4}$$

If this operator has a least fixed point, then the specification $\mathcal{S}$ defines predicate lifting as

$$\begin{aligned}\overline{\alpha} \mid \emptyset \vdash \mathsf{Pred}_{\mathsf{C}_{\mathcal{S}}} \;:\; & (\alpha_1 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_1 \Rightarrow \mathsf{Prop}) \Rightarrow \\ & (\alpha_2 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_2 \Rightarrow \mathsf{Prop}) \Rightarrow \cdots \Rightarrow \\ & (\alpha_\Bbbk \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_\Bbbk \Rightarrow \mathsf{Prop}) \Rightarrow \mathsf{C}_{\mathcal{S}}[\alpha_1, \ldots, \alpha_\Bbbk] \Rightarrow \mathsf{Prop}\end{aligned}$$

Its semantics is the least fixed point of (4).

Relation Lifting $\mathsf{Rel}_{\mathsf{C}_{\mathcal{S}}}$ For relation lifting fix $2\Bbbk$ parameter relations (such that $R_i^+ \subseteq U_i^+ \times V_i^+$ and $R_i^- \subseteq U_i^- \times V_i^-$) and consider the following operator.

$$S \subseteq X_{\overline{U}} \times X_{\overline{V}} \longmapsto \coprod_{\delta_{\overline{U}} \times \delta_{\overline{V}}} \mathrm{Rel}([\![\sigma_{\mathcal{S}}]\!])(\overline{R}, S, S) \tag{5}$$

If this has a least fixed point, then the specification $\mathcal{S}$ defines relation lifting as

$$\begin{aligned}\overline{\alpha}, \overline{\beta} \mid \emptyset \vdash \mathsf{Rel}_{\mathsf{C}_{\mathcal{S}}} \;:\; & (\alpha_1 \times \beta_1 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_1 \times \beta_1 \Rightarrow \mathsf{Prop}) \Rightarrow \\ & (\alpha_2 \times \beta_2 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_2 \times \beta_2 \Rightarrow \mathsf{Prop}) \Rightarrow \cdots \Rightarrow (\alpha_\Bbbk \times \beta_\Bbbk \Rightarrow \mathsf{Prop}) \Rightarrow \\ & (\alpha_\Bbbk \times \beta_\Bbbk \Rightarrow \mathsf{Prop}) \Rightarrow \mathsf{C}_{\mathcal{S}}[\alpha_1, \ldots, \alpha_\Bbbk] \times \mathsf{C}_{\mathcal{S}}[\beta_1, \ldots, \beta_\Bbbk] \Rightarrow \mathsf{Prop}\end{aligned}$$

The semantics of $\mathsf{Rel}_{\mathsf{C}_{\mathcal{S}}}$ is the least fixed point of (5).

Remark 8.1

1. In addition to the items above, the CCSL compiler also defines accessor functions. Accessor functions cannot be correctly typed in the type theory of this report (see the remarks on this issue in Example 4.2 on page 28). For a constructor $c : \sigma^1 \times \cdots \times \sigma^m \Rightarrow \mathsf{Self}$ that takes m arguments there are m (partial) accessor functions

$$\overline{\alpha} \mid \emptyset \vdash \mathsf{acc}_c^j : \mathsf{C}_{\mathcal{S}}[\overline{\alpha}] \;\Rightarrow\; \sigma^j$$

[24][Hen99] defines this operator as $Q \subseteq X_{\overline{U}} \longmapsto (\delta_{\overline{U}}{}^{-1})^* \,\mathrm{Pred}(\cdots)$. However, the equation $(f^{-1})^* = \coprod_f$ holds in **Pred** for isomorphisms f.

They get semantics either as a partial function or as a dependently typed function via

$$[\![\mathsf{acc}_c^j]\!]_{\overline{U}}(x) \quad = \quad \begin{cases} y_j & \text{if } \exists y_1, \ldots, y_m \,.\, \delta_{\overline{U}}(\kappa_c(y_1, \ldots, y_m)) = x \\ \text{undefined} & \text{otherwise} \end{cases}$$

where κ_c is the interpretation injection for the constructor c.

2. The semantics of case can be defined via reduce. Consider the following diagram:

$$\begin{tikzcd}
F^{\mathcal{S}}_{\overline{U}}(X_{\overline{U}}) \arrow[rr, dashed, "F^{\mathcal{S}}_{\overline{U}}(\mathsf{precase})"] \arrow[dd, "\delta_{\overline{U}}"'] & & F^{\mathcal{S}}_{\overline{U}}(X_{\overline{U}} \times V) \arrow[d, "F^{\mathcal{S}}_{\overline{U}}(\pi_1)"] & \\
& & F^{\mathcal{S}}_{\overline{U}}(X_{\overline{U}}) \arrow[d, "\langle \delta_{\overline{U}}, [f_1, \ldots, f_n] \rangle"] & \\
X_{\overline{U}} \arrow[rr, dashed, "\mathsf{precase}"] & & X_{\overline{U}} \times V \arrow[r, "\pi_2"] & V
\end{tikzcd}$$

By exploiting the arguments $f_1, \ldots, f_n$ one can define an $F^{\mathcal{S}}_{\overline{U}}$ algebra on $X_{\overline{U}} \times V$ (on the right hand side in the preceding diagram). Initiality of $\delta_{\overline{U}}$ defines the unique function $\mathsf{precase}$, which makes the preceding diagram commute. Then one can define $[\![\mathsf{case}_{\mathcal{S}}]\!]_{\overline{U},V}(f_1, \ldots, f_n) = \pi_2 \circ \mathsf{precase}$.

3. The predicate lifting for the abstract data type $\mathcal{S}$ is defined if the (full) predicate lifting for $[\![\sigma_{\mathcal{S}}]\!]$ is defined. This is for instance the case, if the ground signature Ω and its model $\mathcal{M}$ are proper. For the relation lifting an analogous statement holds. Both, the predicate lifting and the relation lifting, are computable functions.[25] The definition that I gave on the preceding pages does not describe an algorithm because I use a fixed point construction. The theorem provers PVS and ISABELLE/HOL admit such definitions. However, in proofs it is often easier to work with definitions that describe a terminating algorithm (because then one can turn the definition into a terminating rewrite system). For this reason the CCSL compiler uses the following —equivalent— definitions for predicate and relation lifting of abstract data types.

   For predicate lifting the CCSL compiler outputs the following (I take the liberty to omit the technical noise of the $\overline{U}$ and of $[\![-]\!]$):

$$\mathsf{Pred}_{\mathsf{C}_{\mathcal{S}}}(\overline{P}) \quad = \quad \mathsf{reduce}_{\mathcal{S}}\big(\, \mathrm{Pred}([\![\sigma_1]\!])\ (\overline{P}, \mathsf{tt}, \mathsf{tt}), \ldots, \mathrm{Pred}([\![\sigma_n]\!])\ (\overline{P}, \mathsf{tt}, \mathsf{tt})\ \big)$$

   Here $\mathsf{tt} \subseteq \mathsf{bool} \stackrel{\text{def}}{=} \{\top\}$ is the predicate that holds only for true.

[25] A function is computable if there exists a algorithm (in the form of a Turing machine for instance) that can compute the function.

The relation lifting for abstract data types is slightly more complicated to define. The idea is as follows: From an element $u \in X_{\overline{U}}$ one computes a function $f \in X_{\overline{V}} \Rightarrow \mathsf{bool}$ such that for $v \in X_{\overline{V}}$ one has $f\,v = \top$ if and only if $[\![\mathsf{Rel}_{\mathsf{C}_\mathcal{S}}]\!]_{\overline{U},\overline{V}}(\overline{R})(u, v)$. As definition mechanism for functions with domain $X_{\overline{U}}$ there is only reduce available, therefore one needs an algebra acting on $X_{\overline{V}} \Rightarrow \mathsf{bool}$. This algebra is defined with a particular instantiation of the operator for relation lifting. Recall from Definition 5.7 (2) that relation lifting is a function of the following type

$$\mathrm{Rel}(\tau)(\overline{R}, S) \quad : \quad [\![\tau]\!]_{\overline{U}}(Y) \times [\![\tau]\!]_{\overline{V}}(X) \longrightarrow \mathsf{bool} \qquad (*)$$

where $\overline{R}$ is a list of parameter relations $R_i \subseteq U_i \times V_i$ and $S \subseteq Y \times X$ (I ignore the contravariant argument relation for Self). Define now

$$\begin{array}{lcl} \mathrm{Rel}'(\tau)(\overline{R}) & : & [\![\tau]\!]_{\overline{U}}(X \Rightarrow \mathsf{bool}) \times [\![\tau]\!]_{\overline{V}}(X) \longrightarrow \mathsf{bool} \\ \mathrm{Rel}'(\tau)(\overline{R}) & \stackrel{\mathrm{def}}{=} & \mathrm{Rel}(\tau)(\overline{R},\ \lambda f \in X \Rightarrow \mathsf{bool},\ y \in X\ .\ f\,y) \end{array}$$

That is, in $(*)$ one takes $Y = X \Rightarrow \mathsf{bool}$ and uses an S that performs function application. Now, for any constructor $c_i : \sigma_i \Rightarrow \mathsf{Self}$, there is the following function

$$\begin{array}{lcl} \mathrm{Rel}_{c_i}(\overline{R}) & : & [\![\sigma_i]\!]_{\overline{U}}(X_{\overline{V}} \Rightarrow \mathsf{bool}) \longrightarrow X_{\overline{V}} \Rightarrow \mathsf{bool} \\ \mathrm{Rel}_{c_i}(\overline{R}) & \stackrel{\mathrm{def}}{=} & \lambda f : [\![\sigma_i]\!]_{\overline{U}}(X_{\overline{V}} \Rightarrow \mathsf{bool})\ .\ \lambda y : X_{\overline{V}}\ . \\ & & \qquad [\![\mathsf{c?}_i]\!]_{\overline{V}}(y) \ \wedge\ \mathrm{Rel}'([\![\sigma_i]\!])(\overline{R})(f, \delta_{\overline{V}}^{-1}\, y) \end{array}$$

Here $\wedge$ should be evaluated in a non-strict way: if the recogniser $\mathsf{c?}_i$ returns false then the result is false. Otherwise the inverse of the algebra $\delta_{\overline{V}}$ delivers something in $[\![\sigma_i]\!]_{\overline{V}}(X_{\overline{V}})$, as required by Rel′. Note that the Rel_{c_i} form an $\mathcal{S}$ algebra on $X_{\overline{V}} \Rightarrow \mathsf{bool}$ and can therefore be passed as argument to reduce:

$$[\![\mathsf{Rel}_{\mathsf{C}_\mathcal{S}}]\!]_{\overline{U},\overline{V}}(\overline{R})(u, v) \quad = \quad [\![\mathsf{reduce}_\mathcal{S}]\!]_{\overline{U}, X_V \Rightarrow \mathsf{bool}}\big(\ \cdots \mathrm{Rel}_{c_i}(\overline{R}) \cdots\ \big)\ u\ v$$

This definition yields the least fixed point of 5.

As an example for this mind twisting definition I show in Figure 15 what the CCSL compiler generates as relation lifting for the abstract data type of lists from Figure 14. The relation lifting is called RelEvery there and instead of the inverted list algebra the CCSL compiler uses the two accessors car and cdr. All instantiations are given in square brackets after the identifier.

Example 8.2 This example shows the items that are defined by the abstract data type specification of lists from the CCSL prelude. For convenience Figure 14 repeats the CCSL source code. It defines the type constructor

$$\vdash \mathsf{list}\ :\ [(?, 0)]$$

```
Begin list[ A : Type ] : ADT
  Constructor
    null : Carrier;
    cons( car, cdr ) : [A, Carrier] -> Carrier
End list
```

Figure 14: The data type of lists from the CCSL prelude

```
RelEvery(R: [[U , V] -> bool])  :  [[list[U] , list[V]] -> bool] =
 Lambda (u: list[U] , v: list[V]) :
  reduce[U, [list[V] -> bool]]
    (null?[V] ,
     Lambda (x: U , y: [list[V] -> bool]) :  Lambda (l: list[V]) :
       cons?[V](l) And R(x , car[V](l)) And y(cdr[V](l)))
    (u)(v)
```

Figure 15: The relation lifting for lists, generated by the CCSL compiler

Its semantics is the functor $\mathsf{list} : \mathbf{Set} \longrightarrow \mathbf{Set}$ that maps every set A to the initial list algebra $[\mathsf{nil}_A, \mathsf{cons}_A] : \mathbf{1} + A \times A^* \longrightarrow A^*$, where A^* is the set of finite words over A. As before I ignore the argument for the negative occurrences of A.

For a function $f : X \longrightarrow Y$ the action of the functor $\mathsf{list}(f) : X^* \longrightarrow Y^*$ (i.e., the map combinator for lists) is defined as

$$\begin{aligned} \mathsf{list}(f)(\mathsf{nil}_X) &= \mathsf{nil}_Y \\ \mathsf{list}(f)(\mathsf{cons}_X(x, l)) &= \mathsf{cons}(f\,x, \mathsf{list}(f)(l)) \end{aligned}$$

In the context of lists, reduce is sometimes called foldright. For a constant $y \in Y$ and a function $g : X \times Y \longrightarrow Y$ it is defined as

$$\begin{aligned} \mathsf{reduce}(y, g)(\mathsf{nil}_X) &= y \\ \mathsf{reduce}(y, g)(\mathsf{cons}_X(x, l)) &= g(x, \mathsf{reduce}(y, g)(l)) \end{aligned}$$

For a parameter predicate $P \subseteq X$ the predicate lifting $\mathsf{Pred}_{\mathsf{list}}(P) \subseteq X^*$ is

$$\begin{aligned} \mathsf{Pred}_{\mathsf{list}}(P)(\mathsf{nil}_X) &= \top \\ \mathsf{Pred}_{\mathsf{list}}(P)(\mathsf{cons}_X(x, l)) &= P\,x \;\wedge\; \mathsf{Pred}_{\mathsf{list}}(P)(l) \end{aligned}$$

For relation lifting one has to stare for a while at Figure 15 to see that it is equivalent with the following characterisation (for $R \subseteq U \times V$)

$$\mathsf{Rel}_{\mathsf{list}}(R)(l_1, l_2) \quad = \quad \begin{cases} \top & \text{if } l_1 = \mathsf{nil}_U \ \wedge \ l_2 = \mathsf{nil}_V \\ R(u,v) \ \wedge \ \mathsf{Rel}_{\mathsf{list}}(R)(l_1', l_2') & \text{if } l_1 = \mathsf{cons}_U(u, l_1') \\ & \quad \wedge \ l_2 = \mathsf{cons}_V(v, l_2') \\ \bot & \text{otherwise} \end{cases}$$ ■

8.1.3. Coalgebraic Class Specifications

I turn now to the description of the items that class specifications contribute to the current ground signature.

Let $\mathcal{S}$ be a class specification over the ground signature Ω with a model $\mathcal{M}$ of Ω. Assume that the signature of $\mathcal{S}$ contains $\Bbbk$ type parameters $\overline{\alpha} = \alpha_1, \dots, \alpha_\Bbbk$, n method declarations $m_i : \tau_i$, and m constructor declarations $c_j : \sigma_j$. Subsection 5 defines (on page 34) the combined method type of $\mathcal{S}$ as $\tau_{\mathcal{S}} = \mathcal{T}_M(\sigma_1) \times \cdots \times \mathcal{T}_M(\sigma_n)$ and the combined constructor type $\sigma_{\mathcal{S}} = \mathcal{T}_C(\sigma_1) + \cdots + \mathcal{T}_C(\sigma_m)$. The variance of the type parameters and of Self in $\mathcal{S}$ is defined as

$$\mathcal{V}_x(\mathcal{S}) \quad \stackrel{\text{def}}{=} \quad \mathcal{V}_x(\tau_{\mathcal{S}})$$

What items are defined by $\mathcal{S}$ in the following depends (among other things) on whether $\mathcal{S}$ is processed with final or loose semantics. Loose semantics is the default, final semantics can be chosen with the keyword `FINAL`, see Subsection 5.2. Like on the preceding pages I use $\overline{U}$ to denote an arbitrary interpretation of the type variables $\overline{\alpha}$ and also $F^{\mathcal{S}}_{\overline{U}}(Y, X) = [\![\tau_{\mathcal{S}}]\!]_{\overline{U}}(Y, X)$.

Type Constructor $\mathsf{C}_{\mathcal{S}}$ The specification $\mathcal{S}$ defines the type constructor $\mathsf{C}_{\mathcal{S}}$ of arity $\Bbbk$:

$$\vdash \mathsf{C}_{\mathcal{S}} \ :: \ [\ \mathcal{V}_{\alpha_1}(\mathcal{S}); \dots ; \mathcal{V}_{\alpha_\Bbbk}(\mathcal{S})]$$

The semantics of $\mathsf{C}_{\mathcal{S}}$ should be a functor taking $2\Bbbk$ arguments (compare Definition 3.5 on page 22). However, in many cases this functor is not fully defined. The morphism component of $[\![\mathsf{C}_{\mathcal{S}}]\!]$ is only defined under the following two conditions: The specification $\mathcal{S}$ must request final semantics and $\mathcal{S}$ must not contain any assertions. If the specification $\mathcal{S}$ does contain assertions then the object component of $[\![\mathsf{C}_{\mathcal{S}}]\!]$ is only defined if the respective arguments for positive and negative occurrences are equal, that is if $U_i^- = U_i^+$, regardless whether final or loose semantics is used. In the following I simply state the definitions without repeating these side conditions again.

For final semantics let $\varepsilon_{\overline{U}} : X_{\overline{U}} \longrightarrow F^{\mathcal{S}}_{\overline{U}}(X_{\overline{U}}, X_{\overline{U}})$ denote the final $F^{\mathcal{S}}_{\overline{U}}$ coalgebra satisfying the assertions of $\mathcal{S}$.[26] For every possible $\overline{U}$ choose $\delta_{\overline{U}}$ such that $(\langle X_{\overline{U}}, \varepsilon_{\overline{U}}, \delta_{\overline{U}} \rangle)_{\overline{U}}$ is a model of $\mathcal{S}$.

For loose semantics choose an arbitrary model $(\langle X_{\overline{U}}, \varepsilon_{\overline{U}}, \delta_{\overline{U}} \rangle)_{\overline{U}}$ of $\mathcal{S}$.

Then

$$[\![\mathsf{C}_{\mathcal{S}}]\!](\overline{U}) \quad = \quad X_{\overline{U}}$$

[26] Note that such $\varepsilon_{\overline{U}}$ might exist, even in case where there is no final coalgebra for $F^{\mathcal{S}}_{\overline{U}}$.

If the specification $\mathcal{S}$ contains no assertions and if non of the α_i has mixed variance, then for final semantics the mapping $[\![\mathsf{C}_{\mathcal{S}}]\!]$ is extended to a functor, see Item Map below.

Methods For each method declaration $m_i : \tau_i$ the specification $\mathcal{S}$ defines a symbol

$$\overline{\alpha} \mid \emptyset \vdash \mathsf{m}_i : \tau_i[\,\mathsf{C}_{\mathcal{S}}[\overline{\alpha}] \,/\, \mathsf{Self}\,]$$

Note that τ_i is a method type, so it can be decomposed into $\tau_i = (\mathsf{Self} \times \tau_i') \Rightarrow \tau_i''$. The semantics of m_i is (as defined in Definition 6.3):

$$[\![\mathsf{m}_i]\!]_{\overline{U}} \quad = \quad \lambda x : X_{\overline{U}}, p : [\![\tau_i']\!]_{\overline{U}}(X_{\overline{U}}, X_{\overline{U}}) \,.\, \pi_i(\varepsilon_{\overline{U}}(x))\,(p)$$

Here π_i is the interpretation projection belonging to the method m_i (see page 36).

Constructors For each constructor declaration $c_j : \sigma_j$ the specification $\mathcal{S}$ defines a symbol

$$\overline{\alpha} \mid \emptyset \vdash \mathsf{c}_j : \sigma_j[\,\mathsf{C}_{\mathcal{S}}[\overline{\alpha}] \,/\, \mathsf{Self}\,]$$

By definition σ_j is a constant constructor type, so $\sigma_j = \sigma_j' \Rightarrow \mathsf{Self}$. Now

$$[\![\mathsf{c}_j]\!]_{\overline{U}} \quad = \quad \delta_{\overline{U}} \circ \kappa_j$$

Coreduce If $\mathcal{S}$ is processed with final semantics then there is a (higher-order) function $\mathsf{coreduce}$ (sometimes also called unfold) that creates the unique morphism into the final coalgebra.

$$\overline{\alpha}, \beta \mid \emptyset \vdash \mathsf{coreduce}_{\mathcal{S}} : (\tau_1[\,\beta \,/\, \mathsf{Self}\,] \times \cdots \times \tau_n[\,\beta \,/\, \mathsf{Self}\,]) \Rightarrow \beta \Rightarrow \mathsf{C}_{\mathcal{S}}[\overline{\alpha}]$$

For the semantics fix an interpretation V of β and let $\overline{f} = f_1, \ldots, f_n$ be a list of functions such that $f_i : [\![\tau_i]\!]_{\overline{U}}(V, V)$. With some shuffling one can transform the f_i into a $F^{\mathcal{S}}_{\overline{U}}$ coalgebra on state space V, denoted with $\langle \overline{f} \rangle$. The semantic of $\mathsf{coreduce}_{\mathcal{S}}$ is only defined for those f_i for which $\langle \overline{f} \rangle$ fulfils the method assertions of $\mathcal{S}$. If this is the case then $[\![\mathsf{coreduce}]\!]_{\overline{U},V}(f_1, \ldots, f_n)$ is the unique function that lets the following diagram commute.

$$\begin{tikzcd}
V \arrow[rr, dashed, "{[\![\mathsf{coreduce}_{\mathcal{S}}]\!]_{\overline{U},V}(\overline{f})}"] \arrow[d, "\langle\overline{f}\rangle"'] & & X_{\overline{U}} \arrow[d, "\varepsilon_{\overline{U}}"] \\
F^{\mathcal{S}}_{\overline{U}}(V, V) \arrow[dr, dashed, "{F^{\mathcal{S}}_{\overline{U}}(V, \mathsf{coreduce}_{\mathcal{S}}]\!]_{\overline{U},V}(\overline{f}))}"'] & & F^{\mathcal{S}}_{\overline{U}}(X, X) \arrow[dl, dashed, "{F^{\mathcal{S}}_{\overline{U}}([\![\mathsf{coreduce}_{\mathcal{S}}]\!]_{\overline{U},V}(\overline{f}), X)}"] \\
& F^{\mathcal{S}}_{\overline{U}}(V, X) &
\end{tikzcd}$$

Map For final semantics the morphism part of $[\![\mathsf{C}_{\mathcal{S}}]\!]$ is defined under the following conditions: First, the model $\mathcal{M}$ of the ground signature must be proper. Second $\mathcal{S}$ must not contain any assertions.

If these conditions are met, the interpretation of the method types τ_i can be regarded as a functor taking $2\Bbbk + 2$ arguments, whose morphism part is denoted with $[\![\tau_i]\!]_{\overline{g}}(f^-, f^+)$. The morphism part of $[\![\mathsf{C}_{\mathcal{S}}]\!]$ can now be defined via $\mathsf{coreduce}$: Fix a second interpretations for the type parameters $\overline{V} = V_1^-, V_1^+, \ldots, V_{\Bbbk}^-, V_{\Bbbk}^+$ and assume a vector of functions

$$\begin{array}{lll} g_1^- : U_1^- \longrightarrow V_1^- & & g_{\Bbbk}^- : U_{\Bbbk}^- \longrightarrow V_{\Bbbk}^- \\ & \cdots & \\ g_1^+ : V_1^+ \longrightarrow U_1^+ & & g_{\Bbbk}^+ : U_{\Bbbk}^+ \longrightarrow V_{\Bbbk}^+ \end{array}$$

and set $\overline{g} = g_1^-, g_1^+, \ldots, g_{\Bbbk}^-, g_{\Bbbk}^+$. Then

$$[\![\mathsf{C}_{\mathcal{S}}]\!](\overline{g}) \quad = \quad [\![\mathsf{coreduce}_{\mathcal{S}}]\!]_{\overline{U}, X_{\overline{V}}}\big(\cdots\ [\![\tau_i]\!]_{\overline{g}}(\mathrm{id}_{X_{\overline{V}}}, \mathrm{id}_{X_{\overline{V}}})\ \big([\![\mathsf{m}_i]\!]_{\overline{V}}\big)\ \cdots\big)$$

Invariant Recogniser The invariant recogniser for $\mathcal{S}$ is a functional that takes a signature model of $\mathcal{S}$ and a predicate on the state space of that signature model as arguments. It returns true if the predicate is an invariant (for the signature model) according to Definition 5.10 (1). The invariant recogniser is defined whenever the ground signature Ω is proper. Its type is as follows.

$$\begin{aligned} \overline{\alpha}, \beta \mid \emptyset \vdash \mathsf{invariant}_{\mathcal{S}} : (\tau_1[\,\beta \,/\, \mathsf{Self}\,] \times \cdots \times \tau_n[\,\beta \,/\, \mathsf{Self}\,]) \Rightarrow \\ (\beta \Rightarrow \mathsf{Prop}) \Rightarrow \mathsf{Prop} \end{aligned}$$

If $\langle$`class`$\rangle$ is the name of $\mathcal{S}$ in the CCSL source code then the CCSL compiler generates the identifier $\langle$`class`$\rangle$`_class_invariant?` for the invariant recogniser.

Bisimulation Recogniser The bisimulation recogniser takes two signature models and a relation as arguments. It returns true if the relation is a bisimulation according to Definition 5.10 (3). The type of the bisimulation recogniser is

$$\begin{aligned} \overline{\alpha}, \beta, \gamma \mid \emptyset \vdash \mathsf{bisimulation}_{\mathcal{S}} : (\tau_1[\,\beta \,/\, \mathsf{Self}\,] \times \cdots \times \tau_n[\,\beta \,/\, \mathsf{Self}\,]) \Rightarrow \\ (\tau_1[\,\gamma \,/\, \mathsf{Self}\,] \times \cdots \times \tau_n[\,\gamma \,/\, \mathsf{Self}\,]) \Rightarrow (\beta \times \gamma \Rightarrow \mathsf{Prop}) \Rightarrow \mathsf{Prop} \end{aligned}$$

The compiler uses $\langle$`class`$\rangle$`_class_bisimulation?` as identifier for $\mathsf{bisimulation}_{\mathcal{S}}$.

Morphism Recogniser The morphism recogniser returns true for functions that are coalgebra morphisms. Its type is

$$\begin{aligned} \overline{\alpha}, \beta, \gamma \mid \emptyset \vdash \mathsf{morphism}_{\mathcal{S}} : (\tau_1[\,\beta \,/\, \mathsf{Self}\,] \times \cdots \times \tau_n[\,\beta \,/\, \mathsf{Self}\,]) \Rightarrow \\ (\tau_1[\,\gamma \,/\, \mathsf{Self}\,] \times \cdots \times \tau_n[\,\gamma \,/\, \mathsf{Self}\,]) \Rightarrow (\beta \Rightarrow \gamma) \Rightarrow \mathsf{Prop} \end{aligned}$$

The CCSL compiler generates the name $\langle$`class`$\rangle$`_class_morphism?` for it.

The CCSL compiler treats the three recognisers for invariants, bisimulations, and morphisms special. They are added to the ground signature after processing the signature of $\mathcal{S}$ such that one can use these recognisers in method and constructor assertions.

Predicate Lifting $\mathsf{Pred}_{\mathsf{C}_\mathcal{S}}$ For predicate lifting consider the following operator

$$Q \subseteq X_{\overline{U}} \longmapsto \varepsilon^*_{\overline{U}} \; \mathrm{Pred}([\![\tau_\mathcal{S}]\!])(\overline{P}, \top_{X_{\overline{U}}}, Q) \tag{6}$$

where $\overline{P} = P_1^-, P_1^+, \ldots, P_{\Bbbk}^-, P_{\Bbbk}^+$ are $2\Bbbk$ parameter predicates. If (6) has a greatest fixed point for all parameter predicates then the specification $\mathcal{S}$ defines the constant for predicate lifting as

$$\begin{aligned} \overline{\alpha} \mid \emptyset \vdash \mathsf{Pred}_{\mathsf{C}_\mathcal{S}} \; : \; & (\alpha_1 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_1 \Rightarrow \mathsf{Prop}) \Rightarrow \\ & (\alpha_2 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_2 \Rightarrow \mathsf{Prop}) \Rightarrow \cdots \Rightarrow \\ & (\alpha_{\Bbbk} \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_{\Bbbk} \Rightarrow \mathsf{Prop}) \Rightarrow \mathsf{C}_\mathcal{S}[\overline{\alpha}] \Rightarrow \mathsf{Prop} \end{aligned}$$

The semantics of $\mathsf{Pred}_{\mathsf{C}_\mathcal{S}}$ is the greatest fixed point of (6).

Relation Lifting $\mathsf{Rel}_{\mathsf{C}_\mathcal{S}}$ For relation lifting consider the following operator for a suitable list of parameter relations $\overline{R}$ (with $R_i^+ \subseteq U_i^+ \times V_i^+$ and $R_i^- \subseteq U_i^- \times V_i^-$):

$$S \subseteq X_{\overline{U}} \times X_{\overline{V}} \longmapsto (\varepsilon^*_{\overline{U}} \times \varepsilon^*_{\overline{V}}) \; \mathrm{Rel}([\![\tau_\mathcal{S}]\!])(\overline{R}, S, S) \tag{7}$$

If it has a greatest fixed point then there is a constant for relation lifting of the following type

$$\begin{aligned} \overline{\alpha}, \overline{\beta} \mid \emptyset \vdash \mathsf{Rel}_{\mathsf{C}_\mathcal{S}} \; : \; & (\alpha_1 \times \beta_1 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_1 \times \beta_1 \Rightarrow \mathsf{Prop}) \Rightarrow \\ & (\alpha_2 \times \beta_2 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_2 \times \beta_2 \Rightarrow \mathsf{Prop}) \Rightarrow \cdots \Rightarrow (\alpha_{\Bbbk} \times \beta_{\Bbbk} \Rightarrow \mathsf{Prop}) \Rightarrow \\ & (\alpha_{\Bbbk} \times \beta_{\Bbbk} \Rightarrow \mathsf{Prop}) \Rightarrow \mathsf{C}_\mathcal{S}[\overline{\alpha}] \times \mathsf{C}_\mathcal{S}[\overline{\beta}] \Rightarrow \mathsf{Prop} \end{aligned}$$

Its semantics is the greatest fixed point of (7).

In general, predicate and relation lifting for class specifications is only semi decidable.[27] So it is impossible to give an algorithmic description for the liftings of class specification. The CCSL compiler outputs definitions that rely on a the Knaster/Tarski characterisation of fixed points in complete lattices [Tar55].

Example 8.3 The queue example is not well-suited for illustration here because its type parameter occurs with mixed variance, so for the queue specification not all items are fully

[27] A predicate P is semi decidable if there exists an algorithm with the following properties. If the algorithm gets $x \in P$ as input then it terminates with result „yes". On an input $x \notin P$ it terminates with „no" or runs forever, see [U. 97]. In particular the characteristic function of a semi decidable predicate cannot be computable.

```
Begin Sequence[ A : Type ] : ClassSpec
  Method
    next : Self -> Lift[[A,Self]];
End Sequence
```

Figure 16: Possibly infinite queues in CCSL

defined. Let me therefore consider possibly infinite sequences. Its rather short CCSL version is in Figure 16.

The sequence signature contains one type parameter occurring strictly covariant, therefore

$$\vdash \mathsf{Sequence} :: [(?, 0)]$$

The final model for sequences is described in Subsection 2.6.7 in [Tew02b]. The interpretation for the type constructor Sequence is defined for all interpretations U^-, U^+ of the type parameter A. However, since the type parameter A has positive variance, the argument U^- is ignored:

$$\begin{aligned} [\![\mathsf{Sequence}]\!](U^-, U^+) &= \mathsf{Seq}[U^+] \\ &= \{f : \mathbb{N} \longrightarrow \mathsf{Lift}[U^+] \mid \forall n \in \mathbb{N} \,.\, f(n) = \mathsf{bot} \;\text{ implies }\; \forall m > n \,.\, f(m) = \mathsf{bot}\} \end{aligned}$$

where bot is the constant associated with the abstract data type Lift, see Example 4.2 (on page 27). In the following I silently ignore the argument for the negative occurrences of the type parameter A.

The interpretation of coreduce is given by the unique function ! into the final sequence coalgebra. Let g be a function $Y \longrightarrow U \times Y + \mathbf{1}$, then $h = [\![\mathsf{coreduce}]\!]_{U,Y}(g) : Y \longrightarrow \mathsf{Seq}[U]$ is the unique function for which the following equation holds:

$$[\![\mathsf{next}]\!]_U(h\,y) = \begin{cases} \mathsf{bot} & \text{if } g\,y = \mathsf{bot} \\ \mathsf{up}(u,\; h\,y') & \text{if } g\,y = \mathsf{up}(u, y') \end{cases}$$

(As an alternative to the definition of coreduce for a concrete representation of the final model one can use the preceding equation with a lazy evaluation scheme as the definition of coreduce. This is in fact what the experimental programming language Charity [CF92] does.)

Let me turn to the morphism part of $[\![\mathsf{Sequence}]\!]$ now. Assume a function $g : V \longrightarrow U$. Then $[\![\mathsf{Sequence}]\!](g)$ is a function $\mathsf{Seq}[V] \longrightarrow \mathsf{Seq}[U]$, which is defined as

$$[\![\mathsf{Sequence}]\!](g)\; f\; n = \begin{cases} \mathsf{bot} & \text{if } f\,n = \mathsf{bot} \\ \mathsf{up}(g\,v) & \text{if } f\,n = \mathsf{up}\,v \end{cases}$$

The predicate lifting $\mathsf{Pred}_{\mathsf{Sequence}}(P)$, for a parameter predicate $P \subseteq U$, is the greatest predicate Q with the following property

$$Q(f) \quad \text{if and only if} \quad \begin{cases} [\![\mathsf{next}]\!]_U(f) = \bot & \text{or} \\ [\![\mathsf{next}]\!]_U(f) = \mathsf{up}(u, f') \;\wedge\; P(u) \;\wedge\; Q(f') \end{cases}$$

for all $f \in \mathsf{Seq}[U]$. It is easy to see that

$$\mathsf{Pred}_{\mathsf{Sequence}}(P)(f) \qquad \text{if and only if} \qquad \forall n \,.\, f\,n = \mathsf{up}(u) \;\; \text{implies} \;\; P\,u$$

The relation lifting $\mathsf{Rel}_{\mathsf{Sequence}}(R)$, for a parameter relation $R \subseteq U \times V$, is the greatest relation $S \subseteq \mathsf{Seq}[U] \times \mathsf{Seq}[V]$ such that

$$S(f,g) \quad \text{if and only if} \quad \begin{cases} [\![\mathsf{next}]\!]_U(f) = \bot \;\wedge\; [\![\mathsf{next}]\!]_V(g) = \bot & \text{or} \\ [\![\mathsf{next}]\!]_U(f) = \mathsf{up}(u, f') \;\wedge\; [\![\mathsf{next}]\!]_V(g) = \mathsf{up}(v, g') \\ \qquad\qquad \wedge\; R(u,v) \wedge S(f',g') \end{cases}$$

Again, for the concrete final model, it is easy to see that

$$\mathsf{Rel}_{\mathsf{Sequence}}(R)(f,g) \quad \text{if and only if} \quad \forall n \,.\, \begin{cases} f\,n = g\,n = \bot & \text{or} \\ f\,n = \mathsf{up}(u) \;\wedge\; g\,n = \mathsf{up}(v) \;\wedge\; R(u,v) \end{cases} \qquad \blacksquare$$

8.2. Iterated Specifications for Polynomial Functors

The general case of iterated specifications is not completely understood yet. For instance, it is unclear how to get a functorial semantics of the queue specification from Example 6.15. Further, the case where iterated specifications contain class specifications with binary methods has not been investigated. Only the case of polynomial functors has been investigated in [HJ97, Hen99, Röß00b]. The result is the following theorem.

Theorem 8.4 *Let $\langle \mathcal{S}_i, \Omega_i, \mathcal{M}_i \rangle_{i \leq n}$ be a finite list of triples, where each $\mathcal{S}_i$ is either a ground signature extension, a coalgebraic class specification, or an abstract data type specification, and where the Ω_i and the $\mathcal{M}_i$ are constructed as described on the preceding pages. Assume that all the $\mathcal{S}_i$ comply with the following conditions:*

- *If $\mathcal{S}_i$ is a ground signature extension, then $\mathcal{S}_i$ and its model are proper. Further, all type constructors of $\mathcal{S}_i$ are type constants (i.e., have arity zero).*
- *If $\mathcal{S}_i$ is a coalgebraic class specification, then*
  - *all its type parameters have strictly positive variance,*
  - *all its method types are polynomial,*
  - *it specifies final semantics via the keyword* `FINAL`*,*
  - *if $\mathcal{S}_i$ contains assertions then all $\mathcal{S}_j$ with $j > i$ use the type constructor $\mathsf{C}_{\mathcal{S}_i}$ only with constant arguments,*
  - *all method assertions and all constructor assertions of $\mathcal{S}_i$ are invariant with respect to behavioural equality,*
  - *$\mathcal{S}_i$ is consistent.*
- *If $\mathcal{S}_i$ is an abstract data type specification, then all its type parameters have strictly positive variance.*

If these conditions hold then all Ω_i and all $\mathcal{M}_i$ are proper with one exception: The morphism component of some class specifications might be undefined. In particular, there exist initial models for all data type specifications and final models for all class specifications among the $\mathcal{S}_i$.

Further the following two technical conditions are fulfilled for all type constructors C: The interpretation of the relation lifting Rel_C is fibred (over the interpretation of C) and it commutes with equality.

Before I can tackle the proof I have to generalise a few results from Chapter 3 of [Tew02b]. For the following two lemmas let $\mathcal{M}$ be a proper model of a proper ground signature Ω such that all relation liftings in $\mathcal{M}$ are fibred and commute with equality.

Lemma 8.5 *Let τ be a polynomial type over Ω. Then the relation lifting of τ is fibred and commutes with equality.*

Proof By induction on the structure of types. □

Lemma 8.6 *Let τ be a polynomial type over Ω. Then τ coalgebra morphisms are functional bisimulations.*

Proof Follows from fibredness of the relation lifting of τ. □

Proof (of Theorem 8.4) The proof goes by induction on i and coalesces Proposition 6.18 of this report with the results of Chapter 4 of [Hen99] and Chapter 6 of [Röß00b]. For $i = 0$ the proposition holds trivially, because Ω_0 is the empty ground signature and $\mathcal{M}_0$ the empty model. In the induction step there are three possibilities:

- If $\mathcal{S}_i$ is a ground signature extension, then the assumptions on ground signatures guarantees that Ω_i and $\mathcal{M}_i$ are proper. The relation lifting of the type constants in $\mathcal{S}_i$ fulfils the technical conditions trivially, because the relation lifting for constants takes no arguments.

- Let $\mathcal{S}_i$ be a coalgebraic class specification over signature Σ with combined method type τ. All type constructors in Ω_{i-1} are either constants, least fixed points (stemming from abstract data type specifications), or greatest fixed points (stemming from coalgebraic class specifications). Therefore the semantics of τ is a data functor in the sense of Rößiger and Hensel. Rößiger's Lemma 6.2.7 gives the final τ coalgebra as a set of labelled elementary trees.

  If $\mathcal{S}_i$ does not contains any assertions then its semantics is fully defined (including the morphism component).

  In case $\mathcal{S}_i$ does contain assertions then, by Lemma 8.6, τ coalgebra morphisms preserve the validity of the method assertions of $\mathcal{S}_i$ and we can construct the final model of $\mathcal{S}_i$ as in Proposition 6.18. This gives the semantics of any constant instantiation of $\mathcal{S}_i$

in subsequent specifications. The morphism component of the semantics of $\mathcal{S}_i$ is not used and stays undefined.

The proof of this case is finished with Hensel's results: His Theorem 4.8 proves the existence of predicate and relation lifting, Proposition 4.9 shows that relation lifting is fibred, and Lemma 4.22 that it commutes with equality.

- Let $\mathcal{S}_i$ be an algebraic class specification with combined constructor type σ. Again the semantics of σ is a data functor and the initial σ algebra exists by Rößiger's Lemma 6.2.6. Then Hensel's Theorem 4.8 shows that predicate and relation lifting for $\mathcal{S}_i$ exist, Proposition 4.9 shows that relation lifting is fibred, and by Lemma 4.18 it commutes with equality. □

In this subsection I combined results from [HJ97] and [Röß00b] to characterise the fragment of CCSL for which (at the time of writing) the semantics is well defined. The CCSL compiler provides the `-pedantic` switch (see Subsection 10.9 on page 103) for checking whether a specification lies within the well defined fragment of CCSL.

Note that even the simple queue specification from Figure 11 does not fulfil the assumptions of the preceding theorem because it has a type parameter with mixed variance. This shows that there is still a need for more general results on the existence of initial algebras and final coalgebras.

8.3. Using CCSL consistently

The preceding theorem 8.4 proves that the semantics of CCSL is well defined for polynomial functors and their iterations. As long as one stays in this fragment one can only introduce inconsistencies by writing an inconsistent specification. Interesting examples lead often beyond the assumptions of Theorem 8.4: Already the queue specification contains a contravariant type variable and does therefore not fit into the preceding theorem.

To cope with the general situation, the CCSL compiler is very carefully constructed such that a few guide lines suffice to ensure consistency. For instance, when the CCSL compiler generates the relation lifting of a class specification it uses a greatest fixed point construction in the target theorem prover. This way one has to prove in the theorem prover that the greatest fixed point does indeed exists before one can use the relation lifting. Further the compiler does only generate those items of the semantics that are well defined. Assume for example a class specification $\mathcal{S}$ that depends on a class specification $\mathcal{S}'$, where $\mathcal{S}'$ contains a type parameter with mixed variance. In this case the compiler does not generate the definition of coalgebra morphism for the signature of $\mathcal{S}$.

The only dangerous point is the semantics of the type constructors for coalgebraic class specifications. It does not make sense to use Rößiger's construction in conjunction with Proposition 6.18 to built the final model of a class specification in the target theorem prover. Rößiger's construction is far too complicated for this purpose. Therefore, for the semantics of class specifications, the compiler generates a new type and a few axioms. This can lead to inconsistencies, if the class specification has no model (for loose semantics) or if it does not have a final model (for final semantics).

Therefore the golden rule for using CCSL consistently is

> If S is a class specification processed with loose semantics: Do not proceed until you have constructed a model of S in the target theorem prover.
>
> If S is processed with final semantics, then do not proceed until you have constructed the final model of S.

If this does not give enough security, then one can use the compiler switch `-pedantic`. It causes the compiler to accept only those source files that fulfil the assumptions of Theorem 8.4 (see Subsection 10.9 for the user interface of the CCSL compiler).

9. CCSL and Object Orientation

In this section I investigate the relation of CCSL to the concept object orientation. Chapter 2 of [Mey97] lists 29 criteria of object orientation. Depending on personal preferences and the interpretation of these criteria one can argue whether CCSL fulfils more criteria than, for instance, Java. The main difference is that CCSL is a specification language. So some of the criteria make no sense at all for CCSL. Consider for instance *dynamic binding* (often called late binding). When a method is called for a specific object, then the method body corresponding to the actual type of the object (in contrast to the static type of the identifier that refers to the object) should be executed. Method declarations in CCSL have no method bodies. Further it is unclear what execution should mean for CCSL. So the term dynamic binding does not apply to CCSL (and not to specification languages in general).

Based on this illustration I consider the question whether CCSL is object oriented as irrelevant. The term object-orientation does apply to software construction systems, it does not apply to a single specification language. So CCSL *is not* an object-oriented specification language. However, I claim that CCSL is a specification language *for* object-oriented programming. A CCSL specification is organised as a series of class and abstract data type specifications. Each class specification contains a number of method declarations, whose behaviour is specified together. This perfectly matches the view of object-oriented programming, where software is organised in classes. However, object orientation consists of more than just the concept of classes.

In the past CCSL has sometimes been criticised for the lack of a particular object-oriented feature. It would indeed be possible to make CCSL more object-oriented in the sense of providing additional syntax for a specific object-oriented feature and including it into the semantics. However, even for key features of object orientation there is no consensus among the object-oriented community on how to do it right. Witness for this are programming languages in the field, for instance C++, Java and Eiffel, which are quite different in their view on object orientation. An attempt to make CCSL more object oriented would necessarily specialise CCSL from a general specification language for object orientation to a specification language for a specific programming language. While it is an interesting challenge in its own to design, for instance, a specification language for Java, the aim of this work was to create a specification language that is independent of a specific programming language. As a result syntax and semantic of CCSL are relatively simple.

In this section I discuss some design choices that have been made for CCSL and compare it with the choices of the programming languages OCAML, Eiffel, C++, and Java. The information about these languages has been taken from [LDG+01, Mey92, Mey97, Str97] and [GJS96], respectively. This section is necessarily more informal in style. The arguments in favour of or against a particular decision are often of similar strength, the decision depends then on personal preferences.

The following subsection shed light on the relation of CCSL with inheritance (Subsection 9.1), subtyping (Subsection 9.2), multiple inheritance (9.3), and overriding and dynamic binding (9.4).

9.1. Inheritance

Inheritance allows one to derive an implementation for a class heir from a class parent without actually copying the source code of parent. In this case the class heir *inherits* from class parent. Equivalently one says that heir is a *descendent* of parent or that parent is an *ancestor* of heir. Inheritance is a key concept of object orientation. For CCSL inheritance is important in two ways. First, it would be nice if a specification for both classes heir and parent has a similar structure. It should consist of a class specification $\mathcal{S}_P$ for the class parent and a class specification $\mathcal{S}_H$ for the class heir such that $\mathcal{S}_H$ is derived from $\mathcal{S}_P$. Second, it is desirable that this derivation at the specification level does not involve textual copying of $\mathcal{S}_P$.

The first point is an abstract property of the involved specifications, it is covered by the notion of subspecification (Definition 6.16). The second point is a syntactic feature of CCSL that is independent from the notion of coalgebraic specification.

Object-oriented programming languages differ with respect to whether inheritance propagates constructors or not. Let us consider two classes heir and parent in different programming languages. Let parent be an ancestor of heir, let parent contain a constructor c, and assume that heir does not override or redefine c. In C++ the constructor c is inherited by heir and if it is called to initialise an object of heir the additional instance variables of heir stay uninitialised, possibly containing random data. In Java and in OCAML the situation is similar, but the problem with the random data does not occur. The Java runtime system performs an initialisation of all instance variables with default values for their respective type. In OCAML variables can only be introduced by a let–binding, which at the same time provides an initialisation for the variable. This applies also to record fields and instance variables.

On a conceptual level constructors are used to initialise newly created objects and to establish invariants that are necessary for the correct behaviour of the methods. An inherited constructor can achieve this only in very special cases. Similar considerations lead in [Mey97] to the following design decision: All features (in Eiffel methods are called features) of a class are inherited. The *creation features* (the Eiffel term for constructors) of a parent class loose their special status during inheritance. So it is not possible to use a creation feature of the parent class for initialisation (unless the derived class declares it as creation feature again). For CCSL I adopt the same point of view. Therefore the definitions of subsignature and subspecification (compare Definitions 5.3 and 6.16) neglect constructor declarations and creation conditions.

The concrete syntax of CCSL contains an inheritance clause with which it is possible to build a subsignature hierarchy without copying. The syntax is as follows.

inheritsection	::=	`INHERIT FROM` *ancestor* {| , *ancestor* |}
ancestor	::=	*identifier* [*argumentlist*] [`RENAMING` *renaming* {| `AND` *renaming* |}]
renaming	::=	*identifier* `AS` *identifier*

An inheritance clause has the following effect: First the type parameters of the ancestor are instantiated with the provided type expressions. Then the instantiated attribute and method declarations are added (disjointly) to the current class together with all method assertions. If an attribute or method identifier of a parent class occurs already in the heir, then it is renamed automatically. The user can rename attributes and methods himself with `RENAMING`'s in order to prevent unintended name clashes. The CCSL compiler takes care that, if a renaming occurs, the inherited method assertions refer to the inherited methods.

9.2. Subtyping

A second key concept of object orientation is that one can pass an object o into an environment which expects only a subset of the methods that are available for o. This idea is formalised by enriching the type system with a *subtype relation*. Intuitively a type σ is a *subtype* of τ (alternatively τ is a *supertype* of σ), denoted by $\sigma \leq \tau$, if it is safe to pass an inhabitant of σ to a function that has domain τ. Type systems for object orientation are usually equipped with a subtype relation, see for instance [PT94, CW85, AC96, Cas97]

A language has *implicit subtyping* if the programmer is not required to insert a type conversion when he passes an object to an environment that expects a supertype. The languages C++, Eiffel, and Java do have implicit subtyping. In OCAML the types for objects are modelled with parametric polymorphism involving an anonymous type variable (often called the row variable). One of the consequences is that in OCAML one has to use *explicit subtyping*. This means that the programmer has to insert a type conversion into the OCAML source code at each point where an object is passed into a function that expects an object of a different class. One can argue that explicit subtyping has not much to do with subtyping, because a type conversion that converts objects of a subtype σ to a supertype τ can be seen as a function $\sigma \longrightarrow \tau$, so that no subtype relation is required at all. One can also consider implicit subtyping as an additional feature that is provided by the compiler, which automatically inserts a type conversion at every point where types do not match. Indeed, such behaviour is specified for Java (compare §5 in [GJS96]).

CCSL has a semantics in set theory. There, types are represented by sets and implicit subtyping is provided by the subset relation. However, the subset relation is far to restrictive, for instance $M \times N \not\subseteq M$ in general, so CCSL and its semantics cannot provide implicit subtyping.

The subtype relation is often confused with the inheritance relation. In [CHC90] it is shown that both relations are independent. The programming language OCAML adopts this point of view: It is an easy exercise to write three independent OCAML programs, each containing two classes a and b, that have the following properties. In the first program b inherits from a but a is a subtype of b. In the second program b inherits from a and a and b are not related by subtyping. Finally in the third program b is a subtype of a but neither a inherits from b nor b inherits from a.

In practice software is structured in an inheritance hierarchy and it is often desirable to identify subtyping and inheritance as much as possible. Moreover, understanding a subtype relation is a difficult challenge, whereas understanding an inheritance relation is relatively simple. Therefore many languages identify subtyping and inheritance at the price of loosing (static) type safety. Examples are C++, Java, and Eiffel.

In an object-oriented programming language a class usually gives rise to a type, the type of objects of that class. As a consequence all objects that belong to one class have an uniform structure. In CCSL there is the notion of class specifications and of models of that class specification. One class specification can have different models. The state space of two such models can have different structure. Consider for instance the model of the queue signature in Example 5.6. There I used pairs of natural numbers and functions as state space. There are models of this signature in which the state space is a set of functions, in other models it is a set of lists. There is no uniform type for the state space of all models of one class. So for a function that models explicit or implicit subtyping it is not clear what codomain this function should have. For this reason it does not make sense to require that CCSL models implicit or explicit subtyping. The user of CCSL who constructs the models has the choice: He can build the models in a way such that the state spaces are in a subset relation. Alternatively he can provide conversion functions that model explicit subtyping.

Certain conversion functions are always provided through the structural properties of coalgebraic class specifications. Consider a model $\mathcal{M} = \langle X, c, a \rangle$ of a specification $\mathcal{S}$. The subsignature projection $\pi_{\Sigma'}$ that belongs to a subspecification $\mathcal{S}' \leq \mathcal{S}$ yields a coalgebra $\pi_{\Sigma'} \circ c$ that fulfils the assertions of $\mathcal{S}'$, so in a sense, it converts objects that fulfil the specification $\mathcal{S}$ into objects that fulfil the method assertions of $\mathcal{S}'$. Note that it might be impossible to find an algebra a' such that $\langle X, \pi_{\Sigma'} \circ c, a' \rangle$ is a model of $\mathcal{S}'$. This happens because the definition of subspecification and of subsignature put no constraints on constructors and creation conditions. [Jac96] suggests to use final semantics, then $\mathsf{coreduce}_{\mathcal{S}'}(\pi_{\Sigma'} \circ c)$ maps objects in X to the canonical model of $\mathcal{S}'$.

9.3. Multiple Inheritance

The programming languages Eiffel, C++, and OCAML allow *multiple inheritance*. And so does CCSL. Multiple inheritance means that a given class can inherit from multiple ancestors. In particular it is possible that a given ancestor is inherited twice or more times via different paths. This is called *repeated inheritance*. The question is, if an object of the heir should contain the instance variables (and the methods) of a repeated ancestor once (the repeated ancestor is shared) or several times (the repeated ancestor is not shared). There is no general answer, because there exist examples where sharing has advantages over non-sharing and vice versa. In OCAML common ancestors are never shared (the last copy hides and overrides all previous ones). In C++ the user has the choice via declaring the ancestor class as virtual or not. Eiffel solves the problem via its resolving of name clashes: Features that have the same name are shared (if certain consistency requirements are met), features that have different names are not shared. Java has the simplest answer to this question: it supports only single inheritance.

In CCSL the question is a bit more difficult, because usually the ancestors are parametric in some type variables. So in order to decide whether a given class is inherited twice it is necessary to have an equality relation on types. On the one hand, if CCSL would allow sharing of common ancestors one would need an appropriate equality relation on types. Besides that additional syntax would be necessary to let the user decide whether he or she wants sharing in a particular case or not. On the other hand even if in CCSL repeatedly inherited classes are never shared, an user can easily enforce sharing by an assertion

$$\langle \text{path to first copy} \rangle(x) \quad = \quad \langle \text{path to second copy} \rangle(x)$$

where $\langle$path to ...$\rangle$ is a suitable combination of subsignature projections.

Under these considerations it seems best to opt for the second alternative: If a class specification is repeatedly inherited in CCSL its method declarations and assertions are included multiple times. Name clashes (one identifier refers ambiguously to more than one declaration) cannot occur, because the semantics of the inherit section is defined via disjoint union. The CCSL compiler automatically renames declarations if otherwise a name clash would occur.

9.4. Overriding and Dynamic Binding

Overriding describes the technique to give a new definition for a method that is inherited from an ancestor class. In [Mey97] Meyer distinguishes *dynamic* and *static binding*. The term *late binding* is a synonym for dynamic binding.

Dynamic binding means that for overridden methods the executed method body is chosen according to the dynamic type of the object. For static binding the type of the variable that holds the object determines the method body that will be executed. Eiffel, Java, and OCAML offer only dynamic binding. In C++ the user has the choice: dynamic binding takes effect if the method is declared as virtual in the parent class and if the object is handled via a reference or a pointer. Otherwise static binding is used.

The first (and more important) question is, how one can model programs in CCSL that exploit dynamic binding. And, secondly, although Meyer considers static binding as "gravest possible crime in object-oriented technology"[28] it is interesting if one can model static binding at the same time. The general answer is that CCSL can model both static and dynamic

[28] [Mey92], page 345

binding in different ways. In the following I will show several examples to demonstrate how this can be done. All these examples are the result of a long discussion with Bart Jacobs about static and dynamic binding in CCSL.

Consider the following OCAML fragment.

```
class parent = object
  method m = 0
end

class heir = object
  inherit parent
  method m = 1
end
```

Class parent contains one method m that returns the integer 0 on every invocation. Class heir inherits from parent and overrides m to return 1. Consider now the method invocation o#m where o is a variable of type parent.[29] What can we derive about the result of o#m? Certainly not that the result is 0, because in a run of the program an instance of heir could have been assigned to o. Assuming that there are no other subtypes of class heir we *can* derive that the result is less than 2. To be more precise we need information about the dynamic type of the object that is held by o.

How can a specification for the classes parent and heir look like? It should be possible to reason not only about complete programs but also about program fragments. Thus it would be inadequate to assume that one can derive the dynamic type of every object for every method call in the verification environment. Let $\mathcal{S}_{\mathsf{parent}}$ denote the specification for class parent. There are at least two points of view: First, the *monotone* approach considers the specification $\mathcal{S}_{\mathsf{parent}}$ as a specification of the objects of parent and all its descendents. Second, in the *nonmonotone* approach $\mathcal{S}_{\mathsf{parent}}$ is a specification for the objects of class parent only. Objects of heir do not need to fulfil the method assertions of $\mathcal{S}_{\mathsf{parent}}$.

The monotone approach is consistent with Eiffel. There the class invariants, the pre- and the postconditions are a specific conjunction of the corresponding properties of the ancestor classes.[30] The monotone approach is also preferable from a logical point of view. The nonmonotone approach takes the point of view of a programmer who expects an assertion $\mathsf{o.m} = 0$ in $\mathcal{S}_{\mathsf{parent}}$ because this fits best with the source code of parent. Both approaches are consistent with the semantics of CCSL as described so far. However to force either one, one had to introduce technical complications into the semantics of CCSL. At the point of writing it is not clear if the monotone approach is superior to the nonmonotone or vice versa. It seems

[29] OCAML uses o#m instead of o.m to syntactically distinguish method invocation from record selection.

[30] In Eiffel classes can contain logical properties formulated in a special propositional logic. Via a compiler switch the user can enable their evaluation at runtime. If one of the properties is violated it yields an exception (similar to the assert directive in C++). One can specify class invariants (properties that are checked whenever the control flow enters or leaves a feature of that class), preconditions (properties about the arguments of a feature), and postconditions (properties about the return value of a feature).

```
Begin SParent : ClassSpec
  Method
    m : Self -> nat;
  Assertion Selfvar x : Self
    p1 : x.m ≤  1;
End SParent

Begin SHeir : Classspec
  Inherit From SParent
  Assertion Selfvar x : Self
    h1 : x.m = 1;
End SHeir
```

Figure 17: The monotone approach to model dynamic binding.

that the decision, which approach to prefer, depends very much on the concrete verification problem. Moreover both approaches are equivalent in the sense that if the semantics of CCSL would be monotone, then it would be possible to write specification that mimic nonmonotone semantics and vice versa. As a conclusion from these various considerations it seems best to leave the semantics of CCSL as simple as possible and let the user decide. In the following I describe how one can model the monotone and the nonmonotone style of specification in CCSL.

In the monotone style one adds assertions to further restrict the inherited methods. The resulting specifications are in Figure 17. The specifications SParent and SHeir in Figure 17 are both consistent. The only problem that remains is that, in case we know that an object of type parent is assigned to a variable o then, we cannot derive that o#m = 0. To fix this it is necessary to incorporate the notion of dynamic type into the specification. The simplest way to do this is to assume that the ground signature contains a type of sufficient cardinality that models the dynamic types of the objects. For simplicity I use the natural numbers here, and let 0 be the dynamic type of parent and 1 be the dynamic type of heir. The modified specification that takes dynamic type information into account is in Figure 18.

Now we can derive o#m = 0 provided we have information about the dynamic type of o. The subsignature projection is linked to dynamic binding because it does not change the value of dynamic_type. In terms of Java it is a widening reference conversion (§5.1.4 in [GJS96]).

The specification in Figure 18 does not model static binding. One can easily fix this by adding a method declarations static_parent : Self ⇒ Self with an additional assertion dynamic_type(static_parent(x)) $= 0$, where x is a variable of type Self.

The modelling of the nonmonotone approach follows ideas from the Java branch in the LOOP project, see [HJ00]. An example specification for the nonmonotone approach is in

```
Begin SParent : ClassSpec
  Method
    dynamic_type : Self -> nat;
    m : Self -> nat;
  Assertion Selfvar x : Self
    p1 : x.m ≤  1;
    p2 : dynamic_type(x) = 0 Implies x.m = 0;

  Constructor
    new_parent : Self;
  Creation
    p3 : dynamic_type(new_parent) = 0
End SParent

Begin SHeir : ClassSpec
  Inherit From SParent
  Assertion Selfvar x : Self
    h1 : dynamic_type(x) = 1 Implies x.m = 1;

  Constructor
    new_heir : Self;
  Creation
    h2 : dynamic_type(new_heir) = 1
End SHeir
```

Figure 18: The monotone approach to model dynamic binding taking dynamic type information into account.

Figure 19. Note that the specification SHeir is consistent because during inheritance the method declaration m of SParent is renamed, say to parent_m, and the inherited assertion p_1 refers to parent_m. The important thing to note is that now the subsignature projection corresponds to static binding. The problem in this approach lies in the modelling of dynamic binding.

Let me fix some notation to explain how this works. Let $\mathcal{S}_P = \langle \Sigma_P, \{\mathsf{p}_1\}, \emptyset\rangle$ be the coalgebraic specification for SParent and let $\mathcal{S}_H = \langle \Sigma_H, \{\mathsf{p}_1', \mathsf{h}_1\}, \emptyset\rangle$ be the specification for SHeir. A model for $\mathcal{S}_H$ consists of a state space X together with a coalgebra $c : X \longrightarrow \mathbb{N} \times \mathbb{N}$ where $\pi_1 \circ c$ interprets the method parent_m and $\pi_2 \circ c$ interprets m. The subsignature projection maps c to $\pi_1 \circ c$, which is a model for $\mathcal{S}_P$. The trick in getting dynamic binding to work lays in a suitable rearrangement of the methods in the coalgebra. In this simple example we see that $\pi_2 \circ c$ is a (signature) model for Σ_P in which the interpretation of m provably equals 1. Note that $\pi_2 \circ c$ does not fulfil the assertion p_1. With sophisticated rearrangements

```
Begin SParent : ClassSpec
  Method
    m : Self -> nat;
  Assertion Selfvar x : Self
    p1 : x.m = 0;
End SParent

Begin SHeir : ClassSpec
  Inherit From SParent
  Method
    m : Self -> nat;
  Assertion Selfvar x : Self
    h1 : x.m = 1;
End SHeir
```

Figure 19: Example for the nonmonotone approach to model dynamic binding.

one can model upcasts that bind some methods statically and some dynamically. This can be used to model C++, where dynamic binding applies only to methods that are declared as virtual. In [HJ00] this technique is used to model the widening reference conversion of Java. The special feature of Java is that this conversion uses dynamic binding for the methods and static binding for the fields (instance variables).

10. Miscellaneous

This section explains those parts of CCSL that do not fit into one of the previous sections. The first subsection describes the structure of input files and what the compiler generates. The following subsections are on the include directive, on lifting requests, importings, infix operators, (qualified) identifiers, anonymous ground signatures, the prelude, the user interface of the CCSL compiler, and the implementation and internal structure of the current implementation.

10.1. Input and Output Files

A complete CCSL specification consists of a sequence of ground signature extensions, class specifications, and abstract data type specifications. Such a sequence can be spread over several files by using the include directive (see the following Subsection). In the following grammar rules the meta symbol *file* stands for a complete CCSL specification (and not for the contents of exactly one file).

file	::=	⦃ *declaration* ⦄ `EOF`
declaration	::=	*classspec*
	|	*adtspec*
	|	*groundsignature*
	|	*typedef*
	|	*groundtermdef*

The meta symbol *typedef* is also allowed outside of ground signatures. Together with the meta symbol *groundtermdef* it provides a lightweight syntax for ground signature extensions, see 10.7 below.

The theorem provers ISABELLE and PVS organise their input material in *theories*. Each theory can depend on a number of other theories and contains axioms, type and constant declarations, and proof goals. For PVS it is important that all the material in one theory depends on all type parameters of the theory. Therefore PVS theories tend to be rather short. For PVS one file can contain several theories. In ISABELLE there is no problem with the type parameters. However, an ISABELLE file might contain only one theory.

In this setting the CCSL compiler behaves as follows. For each specification or ground signature ⟨`spec`⟩ in the input file it generates a number of different *internal* theories. What theories are precisely generated depends on the properties of the input and on the target theorem prover. For PVS the compiler dumps one internal theory into one PVS theory. All theories that belong to the specification ⟨`spec`⟩ are put into one file ⟨`spec`⟩`_basic.pvs`. For ISABELLE the compiler combines all internal theories into one ISABELLE theory ⟨`spec`⟩`_basic` and prints it into the file ⟨`spec`⟩`_basic.thy`.

If a class specification contains a theorem section then the formulas there are translated into a theory ⟨`spec`⟩`Theorem` and written into a separate file.

For a ground signature ⟨`gsig`⟩ that defines types or constants version 2.2 of the CCSL compiler generates one theory ⟨`gsig`⟩`Definition` and possibly several theories ⟨`gsig`⟩`Definition`n, where n stands for a generated sequence number. The reason for separating the material of one ground signature into several theories lays in the different treatment of type parameters in PVS and CCSL. The compiler generates no output for ground signatures that contain only declarations (i.e., no defining equations).

For an abstract data type specification ⟨`adt`⟩ the CCSL compiler can generate the following theories.

name of theory	contents
⟨`adt`⟩	data type declaration
⟨`adt`⟩`Util`	reduce, accessors, recognisers
⟨`adt`⟩`Map`	map combinator
⟨`adt`⟩`Every`	(full) predicate lifting
⟨`adt`⟩`RelLift`	(full) relation lifting

The theory ⟨`adt`⟩`Util` is only generated for ISABELLE. The theories for the map combinator and for the liftings are generated if these notions exist for the adt in question and if the target theorem prover does not provide them.

The theories that can be generated for a class specification are displayed in the Tables 20 and 21. Again, these theories are only generated, if there contents is defined for the actual class specification. For instance for class specification for which loose semantics is requested the theories for the final model and for the map combinator are not generated.

The CCSL compiler generates a fair amount of theorems. Unfortunately, it generates only very few proofs. For PVS this is no problem. For ISABELLE the compiler generates sorry[31] proofs.

10.2. Include Directive

The CCSL compiler supports a C-preprocessor like include directive:

include ::= `#include` "*string*"

The string must be the name of a file, which is literally substituted for the include directive. The include directive is handled by the lexer, it can appear at any place in the input.

10.3. Lifting Requests

During type checking the CCSL compiler determines all uses of behavioural equality, derives the types and generates appropriate liftings. However, sometimes an user wishes to use behavioural equality for types that do not occur in the specification. The CCSL compiler supports these users via lifting requests. A lifting request consists of a name and a type expression. The compiler generates the relation lifting for this type and adds a declaration with the given name in the generated files.

requestsection	::=	`REQUEST` *request* {| ; *request* |}
request	::=	*identifier* : *type*

10.4. Importings

In PVS and in ISABELLE there must be a strict hierarchy between all theories. One can only use the identifiers that are declared in the current theory or in one of the theories on which the current theory depends.

Therefore it is necessary that the CCSL compiler generates the right dependencies between the theories in its generated output. During parsing the CCSL compiler collects all type constructors that are used in each specification and generates the right dependencies. For ground signatures that declare nonstandard material it is necessary to inform the CCSL

[31] The ISAR command sorry does a fake proof pretending to solve the pending proof goal without further ado [Wen02].

name of theory	contents
⟨class⟩Interface	signature declaration
⟨class⟩Method_Id	tags for method wise modal operators
⟨class⟩MethodPredicateLifting	(method wise) predicate lifting, method-wise invariants
⟨class⟩MethodInvariantRewrite	utility lemmas for invariants
⟨class⟩MethodInvariantInherit	link methodwise invariants with super classes
⟨class⟩Box	(method wise) modal operators
⟨class⟩BoxInherit	link modal operators with super classes
⟨class⟩Bisimilarity	relation lifting and bisimulations
⟨class⟩BisimilarityRewrite	utility lemmas for bisimulations
⟨class⟩PublicBisimilarityRewrite	utility lemmas for bisimulations with respect to the public subsignature
⟨class⟩BisimilarityEquivalence	bisimulation on one model, bisimilarity
⟨class⟩BisimilarityEqRewrite	utility lemmas for bisimilarity
⟨class⟩PublicBisimilarityEqRewrite	utility lemmas for bisimilarity with respect to the public subsignature
⟨class⟩ReqObsEq	additional liftings
⟨class⟩Morphism	definition of ⟨class⟩–coalgebra morphisms
⟨class⟩MorphismRewrite	utility lemmas for morphisms
⟨class⟩Semantics	semantics of the specification
⟨class⟩Basic	utility lemmas for assertions and creation conditions
⟨class⟩FullInvariant	full predicate lifting and every combinator
⟨class⟩FullBisimulation	full relation lifting and relevery combinator
⟨class⟩Finality	properties of the final ⟨class⟩–coalgebra, coreduce
⟨class⟩FinalityBisim	Bisimilarity on the final model
⟨class⟩Final	axiomatic final model
⟨class⟩FinalProp	axiom for final model

Table 20: Generated theories for a class specification ⟨class⟩, Part I

name of theory	contents
⟨`class`⟩`MapStruct`	coalgebra structure for map combinator
⟨`class`⟩`Map`	map combinator
⟨`class`⟩`Loose`	axiomatic loose model
⟨`class`⟩	top level import theory for ⟨`class`⟩
⟨`class`⟩`Theorem`	translated theorem section

Table 21: Generated theories for a class specification ⟨`class`⟩, Part II

compiler where the material in the ground signature is defined. For special applications it is sometimes necessary to adapt the automatically inferred dependency relation. All this is done via the importing clause (in PVS the dependency between theories is given by importing statements). Importings can occur at the beginning of ground signatures or class specifications, or in a section for assertions or creation conditions.

importing ::= `IMPORTING` *identifier* [*argumentlist*]

For PVS it is sometimes preferable to instantiate parametric theories in importing statements. Therefore it is possible to provide an argument list in the CCSL importing clause. For ISABELLE the arguments are suppressed.

10.5. Infix Operators

CCSL permits the declaration of infix operators in ground signatures to allow expressions like $3 + 4$ in assertions. The infix operators of CCSL are very similar to the ones of OCAML [LDG+01] and use the same implementation technique. Infix operators can be several characters long. They are sequences of the following characters

`! $ & * + - . / \ : < = > ? @ ^ | ~ #`

where the first character is one of

`$ & * + - / \ < = > @ ^ | ~ #`

Infix operators are grouped into precedence levels according to their first characters. Associativity is fixed and depends also on the first characters. Operators starting with `**` have the highest precedence. These operators are right associative. On the next precedence level are the operators which have `*`, `/` or `\` as first character, followed by those with `+` or `-`, followed by the operators starting with `@`, `^`, or `#`. All these operators are left associative. On the least precedence level are the operators starting with `=`, `~`, `<`, `>`, `|`, `&`, or `$`. They are non-associative.

Infix operators must be declared as functions taking two arguments, so their type must have a structure either like $(\tau \times \sigma) \Rightarrow \rho$ or like $\tau \Rightarrow \sigma \Rightarrow \rho$. If an infix operator is surrounded by a pair of parenthesis it becomes a (prefix) function symbol. In the declaration in the ground signature the parenthesis are also required.

Two infix operators are predefined: `=` for equality and $\sim$ for behavioural equality.

10.6. Identifiers and Qualified Identifiers

Identifiers in CCSL are sequences of letters, digits, the underscore, and the question mark. Identifiers must begin with a letter. The list of reserved words is in the Appendix C on page 124.

Let me use the term *specification* in this subsection to denote a class specification, a ground signature, or an abstract data type specification that occurs in the CCSL input. Any specification defines certain items for the specifications that follow, as explained in Section 8. One can use a *qualified identifier* to refer to one of these items, even if the identifier is hidden by another declaration. Qualified identifiers can occur at the expression level in assertions (denoting constants or functions) or in type expressions (denoting types from a ground signature). Their syntax is as follows.

qualifiedid	::=	*idorinfix*
	|	*identifier* [*argumentlist*] : : *idorinfix*
idorinfix	::=	(*infix_operator*)
	|	*identifier*

The meta symbol *idorinfix* (whose definition is repeated here for convenience) stands for an unqualified identifier, which may be an infix operator in parenthesis. A qualified identifier consists of a specification identifier, an optional argument list, and an unqualified identifier. If the specification declares type parameters the argument list must be present.

10.7. Anonymous Ground Signatures

It is possible to define or declare type constructors and constants outside of ground signatures with the keywords `TYPE` and `CONSTANT`. The concrete syntax is the same as inside ground signatures. For convenience I repeat the relevant meta symbols from the grammar:

typedef	::=	`TYPE` *identifier* [*parameterlist*] [`=` *type*]
groundtermdef	::=	`CONSTANT` *termdef* [`;`]
termdef	::=	*idorinfix* [*parameterlist*] `:` *type* [*formula*]

The CCSL compiler combines any sequence of such declarations into an anonymous ground signature.

10.8. The Prelude

Before processing the actual input file the CCSL compiler parses a string that is hard wired into the compiler: *the CCSL prelude*. The prelude extends the ground signature to contain some basic types and constants. The prelude of the compiler version 2.2 is displayed in Figure 22. The ground signatures EmptySig and EmptyFunSig belong only to the prelude, if the target theorem prover is PVS. For ISABELLE the data type of lists has constructors

Nil and Cons to match ISABELLE's definition. The compiler is clever enough to avoid the repetition of the list data type in the target theorem prover.

10.9. User Interface

The CCSL compiler is a command line tool in the Unix tradition. Besides command line switches it expects its source files on the command line and outputs into files in the current directory (unless option `-d` is present). Here is a selection of the command line switches for version 2.2 (for a complete listing see the reference manual [Tew02a] or the manual page).

`-fixedpointlib path` Set the location of the PVS fixed point library. The `path` of the fixed point library appears in the generated output. It must point to the correct location, otherwise type checking in PVS fails. The default path is set during installation.

`-d dir` Place all generated files in directory `dir`

`-pvs` Set the target theorem prover to PVS. This is the default.

`-isa` Set the target theorem prover to ISABELLE/HOL in the syntax of new style ISAR theories [Wen02]. Of a sequence of `-pvs` and `-isa` options the last one takes effect.

`-nattype type` Set the type name of natural numbers to `type`. Defaults to `nat`. More precisely, the type checker uses type `type` for all natural number constants (consisting only of digits) in the source. This option is necessary to prevent type checking errors if you use natural number constants in combination with a type different from `nat` (for instance `int`).

Note that `type` must be a valid type at each occurrence of a natural number constant in the source. So you probably need to add a ground type declaration for `type` at the start of the specification.

`-batch` Generate a batch processing file. The precise behaviour depends on the output mode. For PVS the compiler generates a file pvs-batch.el containing Emacs lisp code. For ISABELLE the file is called ROOT.ML and contains SML code.

`-class spec` Only generate output for specification `spec`. Repeat this option to get output for several classes.

`-dependent-assertions` Normally the semantics of an assertion is a predicate on the state space that is independent from all other assertions. With this option each assertions has the preceding assertion as assumption. This does not change the semantics of a class specification. However, it makes it possible to discharge type-check conditions (TCC's) with the help of previous assertions.

`-pedantic` Enforce all assumptions of Theorem 8.4 except the consistency requirement for class specifications. To ensure invariance with respect to behavioural equality the compiler performs a syntactic check according to the Propositions 6.6 and 6.11. This check is relaxed in the following two cases:

```
Begin EmptySig : GroundSignature
   Importing EmptyTypeDef
   Type EmptyType
End EmptySig

Begin EmptyFunSig [A : Type]: GroundSignature
   Importing EmptyFun[A]
   Constant
      empty_fun : [EmptyType -> A];
End EmptyFunSig

Begin list[ X : Type ] : Adt
   Constructor
      null : Carrier;
      cons( car, cdr ) : [X, Carrier] -> Carrier
End list

Begin Lift[ X : Type ] : Adt
   Constructor
      bot : Carrier;
      up( down ) : X -> Carrier
End Lift

Begin Coproduct[ X : Type, Y : Type ] : Adt
   Constructor
      in1(out1) : X -> Carrier;
      in2(out2) : Y -> Carrier;
End Coproduct

Begin Unit : Adt
   Constructor
      unit : Carrier;
End Unit
```

Figure 22: The CCSL prelude

- Polymorphic constants are recognised as behaviourally invariant if they are instantiated with a constant type.
- Constructors of a class specification are allowed in the creation assertions of that class specification. This rests on the construction in the proof of Proposition 6.18.

`-expert` Turn on expert mode. This turns a number of errors into warnings. As a result the compiler might generate inconsistent output.

`-path` Generate output for inductive characterisation of invariants. This is currently experimental and does not work for all class signatures.

`-no-opt` Turn off formula optimisation. Normally the CCSL compiler performs several optimisations before printing formulae and expressions. (The compiler uses simple equivalences for optimisation like $\top \wedge p = p$ but assumes also that all ground signature extensions are proper.)

`-no-inline-lifting` Turn off inlining of liftings of non-recursive classes and of abstract data types. A non-recursive class (abstract data type) is one whose signature corresponds to a constant functor. The predicate and relation lifting for a such classes is a conjunction; and for non-recursive abstract data types it is a case distinction. Normally the CCSL compiler uses these liftings directly. With the option `-no-inline-lifting` it uses the appropriate combinators instead.

`-output-prelude` Print the prelude to stdout. See also Subsection 10.8.

`--help` Print usage information.

`-c` Act as filter. Print all generated output to `stdout`.

`-prf` Proofs only. Do not generate any theories, only generate proofs. Useful for proof testing.

`-prooftest name` Proof testing. Do not generate any output. Only print the proof for lemma `name` to `stdout`.

`-v` Verbose. Print some messages about compilation progress.

`-D number` Set debugging flags to `number`. If several options `-D` are given the result is xor of all numbers. The compiler recognises the following flags:

- `1` Verbose. Equivalent to `-v`.
- `2` Debug messages for the lexer.
- `4` Debug messages for the parser.
- `8` Debug messages for resolution pass.
- `16` Debug messages for inheritance pass.

`32`	Debug messages for type checking.
`64`	Dump internal syntax tree to `stderr` after type checking.
`128`	Dump symbol table to stderr on unknown identifier.
`256`	Apply debugging level also when processing the prelude.
`512`	Debug messages for type unification.
`1024`	Print assertions, creation conditions, and theorems to stderr after parsing. Useful for problems with operator precedence.
`2048`	Debug messages for variance pass.
`4096`	Debug messages for feature pass.
`8192`	Debug messages for pedantic checks.

10.10. Implementation

The CCSL compiler is implemented in the programming language OCAML [LDG+01] using standard compiler construction techniques (see for instance [ASU86]). It is organised in several passes. Version 2.2 consists of about 40.000 lines in about 100 files. OCAML is a strongly typed functional programming languages similar to SML [MTH91]. It contains extensions for an imperative programming style (references, while loops) and also for an object-oriented style.

OCAML was chosen for the following reasons: The transformation of CCSL into higher-order logic requires a lot of symbolic manipulations. This can most easily be programmed with abstract data types and pattern matching. OCAML integrates well with a standard Unix environment and with Emacs. The OCAML distribution contains, besides the compiler, OCAML versions of Lex and Yacc, the replay debugger that allows one to run programs *backwards*, and an extensive library. The compiler itself is small and produces fast code. One of the disadvantages of OCAML is that the object-oriented constructs integrate only purely with the functional part of the programming language. It is for instance not possible to define a set of mutually recursive types such that some of the types are classes and the other are abstract data types. To define such a set of types one has to use the special construction of object types, which makes the whole code very complex. A second problem with OCAML is that it does not allow one to specialise the result type of methods during inheritance.

Some of the intermediate data structures of the CCSL compiler are defined as classes and some as variant types. The three important classes are called `iface`, `member`, and `theory_body`. The internal representations of abstract data type specifications, class specifications, and ground signature extensions are derived from the class `iface` by inheritance. The classes for attributes, methods, constructors, and ground signature constants are derived from the class `member`. The class `theory_body` is the data structure to capture the output that the CCSL compiler generates. There is one specific class that inherits from `theory_body` for every theory in the output.

The variant types formalise the internal representation of types, formulae and expressions.[32] All these types are mutually dependent, for instance formulae are expressions (of boolean type), expressions contain types, type constructors that occur in types can stem from class specifications, and, finally, a class specification contains formulae. This dependency suggests to define all involved types in one mutual recursion. However this is not feasible for several reasons. The most important is the absence of method specialisation: The class `iface` contains a method `get_members` that returns a list of `member`'s. Using inheritance one wants to define the class `ccsl_iface` that overrides `get_methods` such that it returns now a list of `ccsl_member`'s. However, the OCAML type checker requires that the overriding method has the same type as the overridden method.

One solution to this problem (which has been adopted in the CCSL compiler) is to write a class pre_iface that is polymorphic in a type variable α (possibly constraining α to be a subtype of `member`). The method `get_methods` in pre_iface returns a list of α's. The desired iface class can be obtained by instantiating α with `member`. By inheritance one can derive a polymorphic class `ccsl_pre_iface`. The class ccsl_iface is obtained from ccsl_pre_iface by instantiating it with ccsl_member. Now the method get_members in ccsl_iface has the desired type. Note that in OCAML the class `ccsl_iface` *is not* a subtype of class `iface`.

For the CCSL compiler we introduced several type parameters to break the (formal) dependency between the type definitions. All the variant types are polymorphic in two type variables, one is instantiated with the internal type for class specifications, the other is instantiated with the internal representation of methods (and attributes). The class `pre_iface` takes three type parameters. The first will be instantiated with an instance of class `member`, the second with an instance of `theory_body`, and the third type variable is instantiated with the class `iface` itself. For instance the file `ccsl_classtype.ml` contains the following code.

```
class type ccsl_iface_type
  = [ccsl_member_type,
      (ccsl_iface_type, ccsl_member_type) ccsl_pre_theory_body_type,
      ccsl_iface_type] ccsl_pre_iface_type

and ccsl_member_type =
    [ccsl_iface_type, ccsl_member_type] ccsl_pre_member_type
```

This code fragment defines suitable instantiations of `iface` and `member` as abbreviations `ccsl_iface_type` and `ccsl_member_type`, respectively. The instantiations are in brackets. The second argument for `ccsl_pre_iface_type` must be instantiated as well. For some obscure reason one has to group these instantiations with parenthesis (instead of brackets). Note that both abbreviations are (mutually) recursive. This is no problem in OCAML.

The CCSL compiler consists of the following passes

[32] Although higher-order logic does not distinguish between formulae and expression, it is conceptually easier to use different types for formulae and expressions internally.

Lexing & Parsing The lexer and the parser are generated by ocamllex and ocamlyacc, respectively. Theses are the OCAML variants of LEX and YACC. Keywords are recognised with a hash table that sits in between the lexerer and the parser. The contents of this keyword hash table is generated from the YACC source by a home grown tool, which has been inspired by GPERF [Sch90].

The parser resolves all type identifiers. Identifiers for variables, methods and constructors are resolved later.

The result of the parser is an abstract syntax tree. All following passes work on this syntax tree and add information by destructive updates. This syntax tree contains information about source code locations such that later passes can generate exact error messages.

Update Methods Scan class signatures for attributes and generate update methods.

Inheritance Resolve inheritance in class specifications. Lookup ancestors, instantiate them, perform method renaming, check for name clashes, and update the symbol table of the heir. When this pass is completed all inherited methods can be found via the symbol table of the heir.

Update Assertions Generate update assertions. Take inherited attributes and inherited update methods into account.

Variance Compute variances for all type parameters. Classify interface functors for data types and classes according to the hierarchy of functors in Chapter 3 of [Tew02b].

Features Depending on the type constructors (and their instantiation) that are used in class and data type signatures this pass determines which parts of the general semantics described in Section 8 are defined for every specification.

Special Class Members Definition of the special class member `coreduce`, and the recognizers for invariants (⟨`class`⟩`_class_invariant?`),
bisimulations (⟨`class`⟩`_class_bisimulation?`),
and morphisms (⟨`class`⟩`_class_morphism?`).

Resolution Resolution of variables, methods and constructors.

Type Checking Type check all assertions. The type checker is based on unification of types. It temporarily inserts (internal) free type variables into the abstract syntax tree. Their solutions are determined with unification.

Theory Generation Generate all theories in an internal version of higher-order logic.

Pretty Printing Dump the generated theories in the syntax of the target theorem prover.

This description shows clearly that the design of the CCSL compiler is not optimised for efficiency. However, efficiency has never been a problem: An input file of a few hundred lines is processed in less than a second.

11. Summary

This report described the coalgebraic class specification language CCSL. The unique feature of CCSL is the combination of abstract data type specification with coalgebraic class specifications that enables the iteration of (algebraic) abstract data types and (coalgebraic) behavioural types. The semantics of CCSL is based on algebras *and* on coalgebras. Other distinctive features of CCSL are the higher-order equational logic used in the class specifications and the method-wise modal operators.

This report is mostly identical to Chapter 4 of my forthcoming PhD [Tew02b].

Comparison with other Specification Environments

OBJ is a family of similar algebraic specification languages based on order sorted algebra. Members of the family are for instance OBJ3 [GM96], CafeOBJ [DF98], and BOBJ. The latter two include support for hidden algebras and reasoning about hidden congruences, so it is possible to specify behavioural types in CafeOBJ and in BOBJ. All members of the OBJ family contain parametrised modules and provide module expressions for the manipulation of modules. OBJ inherits the restrictions of algebraic specification. Most notably, signatures can only contain operations of the form $S_1 \times \cdots \times S_n \longrightarrow S_0$, where all the S_i are primitive sorts. Structured input and output types, as they occur naturally in CCSL, are impossible in OBJ.

The common framework initiative [Mos97] aims at a common framework for algebraic specification and development. It integrates many of the diverging extensions of algebraic specification. The initiative includes the design of the Common Algebraic Specification Language, CASL[33] [Mos00]. The logic of CASL is a first-order logic with equality, partial functions, subsorting, and sort generation constraints. There are various sublanguages of CASL, for instance for conditional equational logic or horn-clause logic. However, CASL does only contain algebraic signatures, so structured input and output types are not possible. At the moment CASL does not support behavioural types.

DisCo [Kel97] is a specification method for reactive systems, it is developed at Tampere University of Technology, Finland. DisCo is based on the temporal logic of actions (TLA) [Lam94]. In the DisCo specification language one can specify object systems and their transitional behaviour. Objects have a state chart like hierarchical structure but may not contain methods. Methods are specified outside of the objects as actions. DisCo specifications can be animated or translated into PVS. In PVS one can do refinement proofs, but it appears that crucial parts of the refinement proof cannot be formalised in PVS, because the translation of DisCo into PVS is incomplete [Kel97]. There is no notion of abstract data type in the DisCo specification language.

The Unified Modelling Language UML is a graphical notion mainly developed for software design. However, in combination with the Object Constraint Logic OCL one can use UML class diagrams to specify properties of a software system. Such specifications can be

[33] The CASL language of the common framework initiative has nothing in common with the Custom Attack Simulation Language (CASL) [VEK00, Sec98].

equivalently described in CCSL. In general, a CCSL specification cannot be formulated as an UML class diagram.

Future Work

There are many ideas about how to develop CCSL further. Here are the more important ones:

- Support algebraic specifications (that is abstract data types with axioms).
- Generate more proofs for standard results.
- Include support for the powerset type constructor.

For the semantics of CCSL the open questions of Chapter 3 of [Tew02b] are important, especially how iterated specifications behave if they contain binary methods. Also type parameters with negative and mixed variance need clarification. Type parameters are important when they get instantiated with the special type Self. If a type parameter gets only instantiated with constant types, then it behaves more or less like an (unknown) constant. So it seems that in Theorem 8.4 one could allow type parameters with negative or mixed variance under the proviso that they get only instantiated with constant types. However, it is not clear how to get the technicalities right.

CCSL as presented here has the following disadvantages. The logic of CCSL is different from the logic of the target theorem prover. This is a consequence of developing the CCSL compiler as a front end to existing theorem provers. A possible solution would be to build a theorem prover for CCSL or to integrate CCSL into an existing theorem prover. The latter is an interesting idea in conjunction with the generic theorem prover ISABELLE. In principle CCSL could be integrated into ISABELLE/HOL in the same way as the data type package of [BW99].

The second disadvantage of developing CCSL as a front end is that type checking CCSL specifications is a two stage process. First, the CCSL compiler reads and checks a specification, then one has to load the generated theories into the target theorem prover. Some typing errors in the specification might only become apparent after the output of the CCSL compiler has been type checked in the target theorem prover. For instance a wrong use of an accessor function is not reported by the CCSL compiler. It only yields an unprovable type correctness condition in PVS.

Appendix

A. CCSL Case Studies

This appendix reports on some case studies that have been performed with CCSL

The MSMIE Protocol

The Multiprocessor Shared-Memory Information Exchange protocol [BA94] is a protocol for communication between several processors in a real-time control system. The protocol has been used for instance in the embedded software of Westinghouse nuclear system design. In [Mey99] an early version of CCSL is used to analyse the protocol. Meyer develops 4 specifications of the protocol in CCSL and proves several refinements. Finally he implemented the protocol in Java and uses an early version of the LOOP tool [JvdBH+98] to translate the Java sources into (their semantics in) PVS. Then he proved that the Java program forms a valid model of the CCSL specification of the MSMIE protocol.

The YAPI Case Study

The Y–chart Application Programmers Interface [dKE99] is used at Phillips for the development of signal processing systems. For one aspect of the interface, the buffered data transfer, [Lam00] develops a CCSL specification and shows its correctness via refinement.

Case Study on Transaction Mechanisms

Transactions are used to make certain designated sequences of actions atomic. Transaction mechanisms are important in the context of databases or operating systems, but also in the world of smart cards — where there is always the possibility that a smart card is removed before an appropriate sequence of actions is completed, see Chapter 5 of [Che00]. In the joint work [JT01] we provide an abstract specification of a (simplified) transaction mechanism. There are two standard implementation techniques for a transaction mechanism: new value logging and old value logging. For both approaches we derive an assertional refinement from the original specification and prove its correctness in PVS. This case study uses the current CCSL compiler to translate the three specifications into PVS. The complete CCSL and PVS sources are available at the following URL: http://wwwtcs.inf.tu–dresden.de/~tews/Transaction.

The Fiasco Case Study

Fiasco [Hoh00, Hoh98] is a micro kernel operating system developed within the DROPS [HBB+98] project. The DROPS project is hosted at the computer science department of the Technische Universität Dresden (Dresden University of Technology) and aims on the construction of an operating system that supports quality of service requirements. As a micro kernel Fiasco implements only the absolutely necessary operating system functionality: address spaces, processes, and interprocess communication. Fiasco is an implementation of the L4 micro kernel interface in C++. It contains about 20.000 lines of C++ code.

In the FIASCO case study I formalised a part of the internal interface of the memory management in FIASCO. Then I tried to prove that the source code of FIASCO gives rise to a model of my specification. The case study revealed some hidden assumptions in the scrutinised interface, therefore the proof could not be completed. The case study is described in full detail in [Tew00], the source code (comprising all CCSL and PVS source files and also some C++ files) is available in the world wide web at http://wwwtcs.inf.tu-dresden.de/~tews/vfiasco/.

For the FIASCO case study address spaces and virtual memory are important. Virtual memory is the memory that is visible to applications. Physical memory is the main memory that sits on the mother board of the computer. The operating system takes care that each application can use a fair amount of physical memory and that one application cannot access or modify the memory of another application without proper authorisation. This task is accomplished with *address spaces*. An address space defines a partial mapping of (addresses in) virtual memory to (addresses in) physical memory. Each application has its own address space so that the same virtual addresses usually refer to different physical addresses in different applications. Address spaces are *partial* mappings, because not all virtual addresses are mapped to physical addresses. If an application accesses such a non-mapped virtual address then the hardware signals a page fault. Page faults occur as the result of programming errors or when the operating system has swapped parts of the applications memory to hard disc. In the former case the application is usually terminated. In the latter case the operating system loads the data from the swap area and adjusts the address space of the application. If an application needs more memory it has to request it from the operating system. In case the request can be satisfied, the operating system changes the address space of the application. This shows that the manipulation of address spaces is a primary task for an operating system.

In an Intel based personal computer (more precisely in the IA32 architecture) the data structure that represents an address space is called a *page directory*. A page directory is a hierarchical structure of pointers that describe the address mapping. Any manipulation of address spaces boils down to the insertion or deletion of some pointers in a page directory. (For a more detailed account of virtual memory and address mapping see [Int99] or Chapter 4 of [Tew00].)

FIASCO is a particular nice challenge for CCSL because FIASCO was developed following the object–oriented paradigm: The whole micro kernel consists of a set of classes, each capturing some particular functionality. The class space_t provides the internal abstraction of address spaces: Objects of space_t are page directories and the methods of space_t provide suitable services. For instance the methods v_insert and v_delete insert or delete mappings of virtual memory (in the address space that is represented by the object on which these methods are invoked). Another method is switchin_context. It implements the change of the address space (by advising the hardware to use from now on the page directory in the current object for translating virtual addresses into physical ones).

In the case study I decided to investigate two correctness properties of the methods v_insert and switchin_context. The first correctness property is that these two methods should always terminate without itself producing a page fault. The second property is that after the insertion of a super page mapping[34] with a subsequent call of switchin_context the hardware should map the virtual addresses as desired.

To be able to express the two correctness properties I first developed the CCSL specification PhyMem of physical memory. The physical memory provides operations for reading and writing of memory cells. Further it is bounded, that is, accesses to addresses above a memory depended limit go into nowhere.

By exploiting inheritance of CCSL specifications, the physical memory is extended into a specification VirtMem of virtual memory. The virtual memory has a method virt_to_phy to translate virtual addresses into physical ones. The read and write methods are redefined as partial methods that work on virtual addresses now. They are partial methods because they fail if their virtual address argument is not mapped to a physical address by the address space in charge.

The two specifications of virtual and physical memory form the basis of the case study. Therefore I checked their consistency and constructed the final model for both. The inspection of the state space of both models shows that there is no unwanted behaviour in the final models. This provides an informal argument for the correctness of the two memory specifications.

The specification Space_t captures a part of the interface of the class space_t. As assertions it contains the two correctness properties from above, which can now be expressed in terms of read and write operations on the virtual memory of VirtMem. The aim of the case study was then to show that the C++ source code of the methods v_insert and switchin_context of class space_t yields a model of the Space_t specification. For that I translated the C++ source by hand into PVS and tackled the proof. This sounds rather easy but the proof development in PVS was a three–month enterprise. The *hand translation* of the C++ sources into PVS is certainly a weakness of the case study. However, a translation tool that computes the semantics of a C++ program in the logic of PVS was certainly beyond the scope of the case study.[35]

As I indicated above the attempted proof failed because the space_t interface contains some hidden assumptions that have not been formalised in the specification Space_t. During the proof it became apparent that, if one combines certain states of the virtual memory with certain arguments of the method v_insert, then an assert statement[36] in the method v_insert fails (for a more precise formulation see Proposition 6.1 in [Tew00]). The hidden assumption in the interface of v_insert is that one may only insert address mappings for virtual addresses that are not already mapped. The source of FIASCO tests always for this condition.

The case study required about four months of work. It was carried out in the autumn

[34] A super page is a continuous memory area of 4 megabyte aligned at an address that is a multiple of 2^{22}.

[35] Such a translation tool would of course be nonsensically unless it restricts C++ to a well understood subset.

[36] In C++ the assert statement (which is actually a preprocessor macro) tests for a logical condition and aborts the program if the condition is not true.

of 1998 with PVS version 2.2, patch level 1.46. It consists of about 5,000 line of source code. There are 230 lemmas and 150 type correctness conditions that are proved with about 4,000 PVS commands. To type check the whole specification and to run all the proofs takes more than half a hour on a 333MHz Pentium II box.

Although the proof of the main theorem failed it was possible to verify a few properties of the FIASCO source code. Therefore it is fair to say that the case study was successful in the sense it showed that coalgebraic specification can well be applied to operating-system verification.

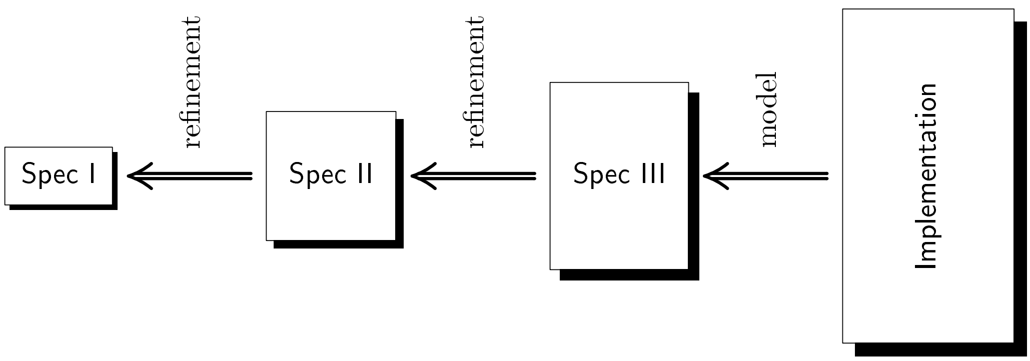
One has to say that PVS did not perform very well under this case study. During the case study I submitted numerous bug reports. Some time after I completed the case study PVS version 2.3 was released. This new version crashed when parsing the source code of the case study. At the time of writing PVS version 2.4 patch level 1 is available. It still contains some bugs that make it impossible to port the case study.

B. Proving Coalgebraic Refinement

This appendix generalises the definition of coalgebraic refinement from [JT01] to parametric class specifications and shows an example refinement of the queue specification from Example 6.15 from Figure 11.

Refinement is a relation between specifications. It links a specification that is considered to be more 'abstract' with a specification that is considered to be more 'concrete'. The intuition is as follows: A concrete specification $\mathcal{S}_C$ *refines* an abstract specification $\mathcal{S}_A$, if all models of $\mathcal{S}_C$ can be transferred into models of $\mathcal{S}_A$. Refinement could also be paraphrased as relative model construction: If $\mathcal{S}_C$ refines $\mathcal{S}_A$ then one can build a model of $\mathcal{S}_A$ by assuming an arbitrary model of $\mathcal{S}_C$. Typically refinement involves a translation of signatures: The operations of the signature of $\mathcal{S}_A$ must be expressed with the operations available in $\mathcal{S}_C$.

Refinement is an important notion in software verification. Instead of relating the implementation directly with the specification one often uses several refinement steps, as depicted below.



As the size of the boxes indicate there is a chain of increasingly complex and increasingly more concrete specifications. The last specification in the chain, Spec III, sufficiently resembles the implementation. So it is feasible to prove that the implementation is a model of

Spec III. The specifications are related by refinements. If the chosen notion of refinement is compositional (i.e., if the refinement relation is transitive) it follows that the implementation is also a model of the specification Spec I.

[Wir90] describes refinement in the context of algebraic specification, but see also [BvW98] for program refinement. For coalgebraic specification refinement was first studied in [Jac97a] and in [Jac97b]. The experience with constructing refinements of CCSL specifications in the theorem prover PVS showed that Jacobs' original notion of coalgebraic refinement is not general enough. A number of generalisation finally lead to two notions of refinement: *assertional refinement* and *behavioural refinement*. Both notions and the need for the generalisations are discussed in detail in the joint work [JT01].

Assertional refinement requires that the assertions of the abstract specification should hold for each translated model of the concrete specification. This implies that for assertional refinement the translation of signatures must cover the complete signature of the abstract specification (because every method of the abstract signature can occur in the assertions). Sometimes this complete coverage is inappropriate. For instance, if the abstract signature contains private methods then one might want to construct a refinement for the public methods only.

In behavioural refinement one requires that each translated model of the concrete specification should be *behaviourally equal* to some abstract model. The behavioural equality can be taken with respect to a subsignature of the abstract specification, for instance to hide the private methods.

In the following I present the definitions from [JT01] in the formal context of this report and explain how one can prove refinements of CCSL class specifications with the theorem prover PVS. Recall from the Definitions 5.1 and 6.14 (on page 32 and 59, respectively) that a class specification $\mathcal{S}$ is a triple $\langle \Sigma, \mathcal{A}_M, \mathcal{A}_C \rangle$, where $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$ is the signature, consisting of the method declarations Σ_M and the constructor declarations Σ_C, and $\mathcal{A}_M$ and $\mathcal{A}_C$ are sets of method and constructor assertions, respectively.

Definition B.1 (Assertional Refinement) Let S_C be a concrete coalgebraic class specification over the signature Σ_C with n type parameters and let S_A be an abstract coalgebraic class specification over the signature Σ_A with m type parameters $\alpha_1, \ldots, \alpha_m$.

1. A *parameter translation* from Σ_A to Σ_C is a n-tuple of types $(\tau_1, \ldots, \tau_n)$ such that every τ_i contains only the type variables $\alpha_1, \ldots, \alpha_n$, that is

$$\alpha_1 : \mathsf{Type}, \ldots, \alpha_m : \mathsf{Type} \quad \vdash \quad \tau_i : \mathsf{Type}$$

   can be derived.

2. Let $(\tau_1, \ldots, \tau_n)$ be a parameter translation from Σ_A to Σ_C. A fixed interpretation $U_1, \ldots, U_m$ of the type parameters $\alpha_1, \ldots, \alpha_m$ induces an interpretation $[\![\tau_i]\!]_{U_1,\ldots,U_m}$ of the types τ_i. A *translation map* (from Σ_C to Σ_A with respect to $(\tau_1, \ldots, \tau_n)$) is a family of mappings $(\phi_{U_1,\ldots,U_m})$ such that for an interpretation $U_1, \ldots, U_m$ of the type parameters $\alpha_1, \ldots, \alpha_m$ the map $\phi_{U_1,\ldots,U_m}$ assigns to every model $M = \langle X, c, a \rangle_{[\![\tau_1]\!],\ldots,[\![\tau_m]\!]}$ of

the specification $\mathcal{S}_C$ a signature model $\phi(M) = \langle X', c', a' \rangle_{U_1,\ldots,U_m}$ of Σ_A such that $X' \subseteq X$.

3. A translation map ϕ is an *assertional refinement*, if $\phi(M)$ is a model of S_A for all models M of S_C.

The notion of parameter translation is not present in [JT01], there we discuss only refinements between class specification without type parameters. The parameter translation deals with the (rare) situation where the number of type parameters of the abstract and the concrete specification differ. In most cases the parameter translation is the identity, that is $\tau_i = \alpha_i$.

The main restriction in the preceding definition of assertional refinement is that the state space of the translated model $\phi(M)$ is a subset of the state space X of the original model. The requirement that X' must only be a subset of X accounts for the fact that in a refinement one might want to exclude certain (unreachable) states from M.

Definition B.2 (Behavioural Refinement)

1. Let $\mathcal{S}$ be a coalgebraic class specification over signature $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$ and let $\Sigma' = \langle \Sigma'_M, \Sigma'_C \rangle$ be a subsignature of Σ with the same set of constructors: $\Sigma_C = \Sigma'_C$. A *behavioural model* (of $\mathcal{S}$ with respect to Σ') is a model $\langle Y, d, b \rangle$ of Σ' such that there exists a model $\langle X, c, a \rangle$ of $\mathcal{S}$ with $\mathrm{Rel}(\llbracket \sigma_\Sigma \rrbracket)({}_c\!\underline{\leftrightarrow}_d)(a, b)$, where σ_Σ is the combined constructor type of Σ (see page 34).

2. Assume a concrete specification, consisting of a coalgebraic class specification $\mathcal{S}_C$ over the signature $\Sigma_C = \langle \Sigma_{CM}, \Sigma_{CC} \rangle$ with n type parameters and a subsignature $\Sigma'_C = \langle \Sigma'_{CM}, \Sigma'_{CC} \rangle$ such that $\Sigma_{CC} = \Sigma'_{CC}$. Let $\mathcal{S}_A$ be an abstract coalgebraic specification over a signature $\Sigma_A = \langle \Sigma_{AM}, \Sigma_{AC} \rangle$ with a subsignature $\Sigma'_A = \langle \Sigma'_{AM}, \Sigma'_{AC} \rangle$. A translation map ϕ from Σ'_C to Σ'_A is called a *behavioural refinement* from $\langle \mathcal{S}_C, \Sigma'_C \rangle$ to $\langle \mathcal{S}_A, \Sigma'_A \rangle$ if ϕ maps behavioural models of $\mathcal{S}_C$ to behavioural models of $\mathcal{S}_A$.

Both notions of refinements are compositional. If one considers behavioural refinements from $\langle \mathcal{S}_C, \Sigma_C \rangle$ to $\langle \mathcal{S}_A, \Sigma_A \rangle$ (i.e., the case where no methods are hidden) then, under certain (reasonable) assumptions on the assertions of the abstract specification, one can prove that assertional refinement coincides with behavioural refinement. See [JT01] for details.

In the remainder of this subsection I construct an assertional refinement for the queue specification of Figure 11 (on page 65). The refinement is based on the idea that lists form queues, if one appends new elements at the end. The refining specification ListQueue is in Figure 23. The complete source code of the refinement and the proofs are available in the material distributed along with my PhD, see http://wwwtcs.inf.tu-dresden.de/ tews/PhD/.

In Figure 23 the ground signature ListOp introduces the function append for the concatenation of lists, which is predefined in PVS. The class specification ListQueue has only two methods contents and set_contents, where the latter is automatically generated by the

```
Begin ListOp[ A : Type ] : GroundSignature
  Constant
    append : [list[A], list[A] -> list[A]];
End ListOp

Begin ListQueue[ A : Type ] : ClassSpec
  Attribute
    contents : Self -> list[A];

  Defining
    put : [Self, A] -> Self
    put(x,a) = set_contents(x, append(contents(x), cons(a,null)));

    top : Self -> Lift[[A, Self]]
    top x = Cases contents x of
                null : bot,
                cons(a,rest) : up(a, set_contents(x,rest))
              EndCases;

  Constructor
    l_new : Self

  Creation
    new_empty : contents(l_new) = null;
End ListQueue
```

Figure 23: A refinement of queues in CCSL

CCSL compiler as an update method for the attribute contents (see Subsection 5.2). The method set_contents has the following type

set_contents : [**Self**, list[A]] –> **Self**

Further the CCSL compiler generates the following assertion.

contents_set_contents : **Forall**(l : list[A]) : contents(set_contents(x, l)) = l

So models of ListQueue are records with one field of type list[A].

An assertional refinement of the queue specification consists of three items: first, a parameter translation, second, a translation function that maps models of ListQueue to models of the queue signature, and, third, a proof that the result of the translation function is a model of Queue. A parameter translation is not necessary in this example, so I choose the identity translation. This means that models of ListQueue[A] get translated into models of Queue[A].

```
QueueRefine[ X, A : Type] : Theory
Begin
  Importing ListQueueBasic[X,A]
  c : Var (ListQueueAssert?)

  Importing QueueBasic[X,A]

  abs_c(c) : QueueSignature[X,A] =
    (# top := top(c),
       put := put(c)    #)

  abs_new(c)(z : (ListQueueCreate?(c))) : QueueConstructors[X,A] =
      (# new := l_new(z) #)

  model : Proposition Forall(z : (ListQueueCreate?(c))) :
      QueueModel?(abs_c(c), abs_new(c)(z))
End QueueRefine
```

Figure 24: The theory ListQueue containing the refinement proof

For the second ingredient of an assertional refinement I need an interpretation of the methods top and put for an arbitrary model of ListQueue. Because it is so obvious how to do that I defined these two methods in the ListQueue specification as definitional extensions.

It remains to prove that the queue assertions hold. This is done in PVS in the theory QueueRefine, see Figure 24. The variable declaration for c on line 4 uses the dependent types of PVS. Recall that ListQueueAssert? is a predicate on ListQueue coalgebras. By putting parenthesis around such a predicate one obtains the (sub–) type of those inhabitants that fulfil the predicate. So c is a ListQueue coalgebra (on state space X) that fulfils the method assertions of ListQueue. Technically, the declaration of c (together with the type parameters X and A) amounts to the assumption of an arbitrary ListQueue model.

The importing statement for QueueBasic makes all necessary notions from the specification Queue available. The functions abs_c and abs_new form the translation map ϕ (of Item 2 of Definition B.1). The definition of abs_new looks a bit complicated because the interpretation z of the constructor of ListQueue cannot be declared as a variable.

It remains to prove the proposition model. The proof is not completely trivial because it involves some reasoning about bisimilarities. In the proof I used the following three utility lemmas:

```
bisim_char : Lemma Forall( x, y : X ) :
    bisim?(c)(x,y)    IFF   contents(c)(x) = contents(c)(y)

abs_bisim : Lemma Forall( x, y : X ) :
    bisim?(c)(x,y)    Implies   bisim?(abs_c(c))(x,y)
```

```
bisim_abs : Lemma Forall( x, y : X ) :
    bisim?(abs_c(c))(x,y)   Implies   contents(c)(x) = contents(c)(y)
```

The first one, bisim_char, gives a characterisation of bisimilarity on models of ListQueue: Two states are bisimilar precisely if their contents field is equal. The second lemma abs_bisim states that two bisimilar states x and y are also bisimilar when considered as states of a queue with respect to the translated coalgebra abs_c(c). The third lemma describes the converse situation.

Behavioural refinements are more difficult to construct in general because usually the existential quantifier, which is hidden in the notion of behavioural models, requires the construction of a suitable abstract model. However, the technical aspects of the translation of a behavioural refinement into PVS are as simple as for assertional refinements. For a behavioural refinement one would prove a proposition similar to the following.

```
same_behaviour : Proposition Forall( z : (ListQueueCreate?(c)) ) :
    bisim?(d(c), abs_c(c))(new(b(c)(z)), new(abs_new(c)(z)))
```

Here abs_c and abs_new form the translation map as before. The functions d and b give the abstract model, which is required in the notion of behavioural models. Usually the abstract model must be chosen in dependence of the concrete model, therefore d and b take the concrete model as arguments.

C. The CCSL Grammar

This appendix contains the complete CCSL grammar. It is given in a BNF–like notation. Brackets [...] denote optional components, braces ⦃...⦄ denote arbitrary repetition (including zero times), and parenthesis (...) denote grouping. Terminals are set in `UPPERCASE TYPEWRITER`, non–terminals in *lowercase slanted*. The terminal symbols for parenthesis and brackets are written as `(`, `)`, `[`, and `]`.

file	::=	⦃ *declaration* ⦄ `EOF`
declaration	::=	*classspec*
	|	*adtspec*
	|	*groundsignature*
	|	*typedef*
	|	*groundtermdef*
classspec	::=	`BEGIN` *identifier* [*parameterlist*] `:` [`FINAL`] `CLASSSPEC` ⦃ *importing* ⦄ ⦃ *classsection* ⦄ `END` *identifier*
parameterlist	::=	`[` *parameters* ⦃ `,` *parameters* ⦄ `]`
parameters	::=	*identifier* ⦃ `,` *identifier* ⦄ `:` [*variance*] `TYPE`
variance	::=	`POS`
	|	`NEG`
	|	`MIXED`
	|	`(` *numberorquestion* `,` *numberorquestion* `)`
numberorquestion	::=	`?`
	|	*number*
classsection	::=	*inheritsection*
	|	[*visibility*] *attributesection* [`;`]
	|	[*visibility*] *methodsection* [`;`]
	|	*definitionsection*
	|	*classconstructorsection* [`;`]
	|	*assertionsection*
	|	*creationsection*
	|	*theoremsection*
	|	*requestsection* [`;`]
visibility	::=	`PUBLIC`
	|	`PRIVATE`
inheritsection	::=	`INHERIT FROM` *ancestor* ⦃ `,` *ancestor* ⦄

ancestor	::=	*identifier* [*argumentlist*] [`RENAMING` *renaming* ⦃ `AND` *renaming* ⦄]
renaming	::=	*identifier* `AS` *identifier*
attributesection	::=	`ATTRIBUTE` *member* ⦃ ; *member* ⦄
methodsection	::=	`METHOD` *member* ⦃ ; *member* ⦄
member	::=	*identifier* : *type* `->` *type*
definitionsection	::=	`DEFINING` *member formula* ; ⦃ *member formula* ; ⦄
classconstructorsection	::=	`CONSTRUCTOR` *classconstructor* ⦃ ; *classconstructor* ⦄
classconstructor	::=	*identifier* : *type*
	|	*identifier* : *type* `->` *type*
assertionsection	::=	`ASSERTION` ⦃ *importing* ⦄ [*assertionselfvar*] ⦃ *freevarlist* ⦄ *namedformula* ⦃ *namedformula* ⦄
assertionselfvar	::=	`SELFVAR` *identifier* : `SELF`
freevarlist	::=	`VAR` *vardecl* ⦃ ; *vardecl* ⦄
creationsection	::=	`CREATION` ⦃ *importing* ⦄ ⦃ *freevarlist* ⦄ *namedformula* ⦃ *namedformula* ⦄
namedformula	::=	*identifier* : *formula* ;
requestsection	::=	`REQUEST` *request* ⦃ ; *request* ⦄
request	::=	*identifier* : *type*
theoremsection	::=	`THEOREM` ⦃ *importing* ⦄ ⦃ *freevarlist* ⦄ *namedformula* ⦃ *namedformula* ⦄
formula	::=	`FORALL` (*vardecl* ⦃ , *vardecl* ⦄) (: | .) *formula*
	|	`EXISTS` (*vardecl* ⦃ , *vardecl* ⦄) (: | .) *formula*
	|	`LAMBDA` (*vardecl* ⦃ , *vardecl* ⦄) (: | .) *formula*
	|	`LET` *binding* ⦃ (; | ,) *binding* ⦄ [; | ,] `IN` *formula*
	|	*formula* `IFF` *formula*
	|	*formula* `IMPLIES` *formula*
	|	*formula* `OR` *formula*
	|	*formula* `AND` *formula*
	|	`IF` *formula* `THEN` *formula* `ELSE` *formula*
	|	`NOT` *formula*
	|	*formula* *infix_operator* *formula*
	|	`ALWAYS` *formula* `FOR` [*identifier* [*argumentlist*] : :] *methodlist*

	|	`EVENTUALLY` *formula* `FOR`
		[*identifier* [*argumentlist*] `::`] *methodlist*
	|	`CASES` *formula* `OF` *caselist* [`;` | `,`] `ENDCASES`
	|	*formula* `WITH` `[` *update* ⦃ `,` *update* ⦄ `]`
	|	*formula* `.` *qualifiedid*
	|	*formula* *formula*
	|	`TRUE`
	|	`FALSE`
	|	`PROJ_N`
	|	*number*
	|	*qualifiedid*
	|	`(` *formula* `:` *type* `)`
	|	`(` *formula* ⦃ `,` *formula* ⦄ `)`
vardecl	::=	*identifier* ⦃ `,` *identifier* ⦄ `:` *type*
methodlist	::=	{ *identifier* ⦃ `,` *identifier* ⦄ }
qualifiedid	::=	*idorinfix*
	|	*identifier* [*argumentlist*] `::` *idorinfix*
idorinfix	::=	`(` *infix_operator* `)`
	|	*identifier*
binding	::=	*identifier* [`:` *type*] `=` *formula*
caselist	::=	*pattern* `:` *formula* ⦃ (`;` | `,`) *pattern* `:` *formula* ⦄
pattern	::=	*identifier* [`(` *identifier* ⦃ `,` *identifier* ⦄ `)`]
update	::=	*formula* `:=` *formula*
adtspec	::=	`BEGIN` *identifier* [*parameterlist*] `:` `ADT`
		⦃ *adtsection* ⦄
		`END` *identifier*
adtsection	::=	*adtconstructorlist* [`;`]
adtconstructorlist	::=	`CONSTRUCTOR` *adtconstructor* ⦃ `;` *adtconstructor* ⦄
adtconstructor	::=	*identifier* [*adtaccessors*] `:` *type*
	|	*identifier* [*adtaccessors*] `:` *type* `->` *type*
adtaccessors	::=	`(` *identifier* ⦃ `,` *identifier* ⦄ `)`
groundsignature	::=	`BEGIN` *identifier* [*parameterlist*] `:` `GROUNDSIGNATURE`
		⦃ *importing* ⦄ ⦃ *signaturesection* ⦄
		`END` *identifier*
signaturesection	::=	*typedef*
	|	*signaturesymbolsection* [`;`]

signaturesymbolsection	::=	`CONSTANT` *termdef* {| `;` *termdef* |}
typedef	::=	`TYPE` *identifier* [*parameterlist*] [`=` *type*]
groundtermdef	::=	`CONSTANT` *termdef* [`;`]
termdef	::=	*idorinfix* [*parameterlist*] `:` *type* [*formula*]
type	::=	`SELF`
	|	`CARRIER`
	|	`BOOL`
	|	`[` *type* {| `,` *type* |} `->` *type* `]`
	|	`[` *type* {| `,` *type* |} `]`
	|	*qualifiedid*
	|	*identifier argumentlist*
argumentlist	::=	`[` *type* {| `,` *type* |} `]`
importing	::=	`IMPORTING` *identifier* [*argumentlist*]

CCSL Keywords

The following words are reserved.

adt	always	and	as	assertion
attribute	begin	bool	carrier	cases
classspec	constant	constructor	creation	defining
else	end	endcases	eventually	exists
false	final	for	forall	from
groundsignature	if	iff	implies	importing
in	inherit	lambda	let	method
mixed	neg	not	of	or
pos	private	public	renaming	request
self	selfvar	then	theorem	true
type	var	with		

Include Directive

The include directive has the following form:

include ::= `#include` "*string*"

The string is interpreted as a file name. The compiler substitutes the contents of the file for the include directive. The directive can stand at any place in the input.

D. References

[ABB+99] Augustsson, L., D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler: *Report on the Programming Language Haskell 98*, February 1999. Available via http://www.haskell.org/.

[AC96] Abadi, M. and L. Cardelli: *A Theory of Objects*. Springer-Verlag, New York, 1996.

[AS85] Alpern, Bowen and Fred B. Schneider: *Defining liveness*. Information Processing Letters, 21(4):181–185, 7 October 1985.

[ASU86] Aho, A. V., R. Sethi, and J. D. Ullman: *Compilers: principles, techniques, tools*. Addison-Wesley, 1986.

[BA94] Bruns, G. and S. Anderson: *The formalization and analysis of a communications protocol*. Formal Aspects of Computing, 6(1):92–112, 1994.

[Bar92] Barendregt, H. P.: *Lambda calculi with types*. In Abramsky, S., D. M. Gabbay, and T.S.E. Maibaum (editors): *Handbook of Logic in Computer Science*, volume 2. Oxford Science Publications, 1992.

[BvW98] Back, R.-J. and J. von Wright: *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[BW99] Berghofer, S. and M. Wenzel: *Inductive datatypes in HOL — lessons learned in formal-logic engineering*. In Bertot, Y., G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry (editors): *Theorem Proving in Higher Order Logics*, volume 1690 of *Lecture Notes in Computer Science*, pages 19–36. Springer, September 1999.

[Cas97] Castagna, Giuseppe: *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkhauser, Boston, 1997.

[CF92] Cockett, R. and T. Fukushima: *About Charity*. Yellow Series Report 92/480/18, Department of Computer Science, University of Calgary, June 1992.

[CHC90] Cook, W. R., W. Hill, and P. S. Canning: *Inheritance is not subtyping*. In ACM (editor): *POPL '90. Proceedings of the seventeenth annual ACM symposium on Principles of programming languages*, pages 125–135, New York, NY, USA, 1990. ACM Press.

[Che00] Chen, Z.: *Java Card Technology for Smart Cards*. The Java Series. Addison-Wesley, 2000.

[Cîr99] CÎRSTEA, C.: *A coalgebraic equational approach to specifying observational structures.* In JACOBS, B. and J. RUTTEN (editors): *Coalgebraic Methods in Computer Science '99*, volume 19 of *ENTCS*. Elsevier, Amsterdam, 1999.

[Cor98] CORRADINI, A.: *A completeness result for equational deduction in coalgebraic specification.* In PRESICCE, F. PARISI (editor): *Recent Trends in Data Type Specification*, volume 1376 of *Lecture Notes in Computer Science*, pages 190–205. Springer, Berlin, 1998.

[CS92] COCKETT, J. R. B. and D. SPENCER: *Strong categorical datatypes I.* In SEELY, R. A. G. (editor): *Proc. of Int. Summer Category Theory Meeting, Montréal, Québec, 23–30 June 1991*, volume 13 of *Canadian Mathematical Society Conf. Proceedings*, pages 141–169. American Mathematical Society, Providence, RI, 1992.

[CW85] CARDELLI, LUCA and PETER WEGNER: *On understanding types, data abstraction, and polymorphism.* ACM Computing Surveys, 17(4):471–522, December 1985.

[DF98] DIACONESCU, R. and K. FUTATSUGI: *CafeOBJ Report: the Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification.* World Scientific, Singapore, 1998.

[dKE99] KOCK, E. A. DE and G. ESSINK: *Y–chart application programmer's interface.* Technical note 0008/99, Philips Naturkundig Laboratorium, March 1999.

[GJS96] GOSLING, J., B. JOY, and G. STEELE: *The Java Language Specification.* Addison-Wesley, 1996.

[GM93] GORDON, M. J. C. and T. F. MELHAM: *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[GM96] GOGUEN, J. and G. MALCOLM: *Algebraic Semantics of Imperative Programs.* MIT Press, Cambridge, Mass., 1 edition, 1996.

[Gol92] GOLDBLATT, R.: *Logics of Time and Computation, Second Edition, Revised and Expanded*, volume 7 of *CSLI Lecture Notes.* CSLI, Stanford, 1992.

[Gol01] GOLDBLATT, R.: *A calculus of terms for coalgebras of polynomial functors.* In CORRADINI, A., M. LENISA, and U. MONTANARI (editors): *Coalgebraic Methods in Computer Science '01*, volume 44 of *ENTCS*. Elsevier, Amsterdam, 2001.

[Gun92] GUNTER, ELSA L.: *Why we can't have SML style* `datatype` *declarations in HOL.* In CLAESE, L. J. M. and M. J. C. GORDON (editors): *Higher Order Logic Theorem Proving and Its Applications*, volume A–20 of *IFIP Transactions*, pages 561–568. North-Holland Press, September 1992.

[HBB+98] Härtig, H., R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter: *DROPS: OS support for distributed multimedia applications.* In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[Hen99] Hensel, U.: *Definition and Proof Principles for Data and Processes.* PhD thesis, Univ. of Dresden, Germany, 1999.

[HHJT98] Hensel, U., M. Huisman, B. Jacobs, and H. Tews: *Reasoning about classes in object–oriented languages: Logical models and tools.* In Hankin, Ch. (editor): *European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*, pages 105–121. Springer, Berlin, 1998.

[HJ97] Hensel, U. and B. Jacobs: *Proof principles for datatypes with iterated recursion.* In Moggi, E. and G. Rosolini (editors): *Category Theory and Computer Science*, volume 1290 of *Lecture Notes in Computer Science*, pages 220–241. Springer, Berlin, 1997.

[HJ98] Hermida, C. and B. Jacobs: *Structural induction and coinduction in a fibrational setting.* Information and Computation, 145(2):107–152, 1998.

[HJ00] Huisman, M. and B. Jacobs: *Inheritance in higher order logic: Modeling and reasoning.* In Aagaard, M. and J. Harrison (editors): *Theorem Proving in Higher Order Logics*, volume 1869 of *Lecture Notes in Computer Science*, pages 301–319. Springer, Berlin, 2000.

[Hoh98] Hohmuth, M.: *The Fiasco kernel: Requirements definition.* Technical Report TUD–FI–12, TU Dresden, December 1998. Available at URL: http://os.inf.tu-dresden.de/fiasco/doc.html.

[Hoh00] Hohmuth, M.: *The Fiasco kernel: System architecture.* Available at URL: http://os.inf.tu-dresden.de/fiasco/doc.html, 2000.

[HPF92] Hudak, P., J. Peterson, and J. H. Fasel: *A Gentle Introduction to Haskell 98.* Available via http://haskell.cs.yale.edu/tutorial/, October 1992.

[Hug01] Hughes, Jesse: *Modal operators for coequations.* In Corradini, A., M. Lenisa, and U. Montanari (editors): *Coalgebraic Methods in Computer Science '01*, volume 44 of *ENTCS*. Elsevier, Amsterdam, 2001.

[Hui01] Huisman, M.: *Reasoning about Java programs in Higher-order logic using PVS and Isabelle.* PhD thesis, University of Nijmegen, The Netherlands, 2001.

[Int99] Intel Corp.: *Intel Architecture Software Developer's Manual, Volume 3: System Programming*, 1999.

[Jac95] JACOBS, B.: *Mongruences and cofree coalgebras.* In ALAGAR, V.S. and M. NIVAT (editors): *Algebraic Methodology and Software Technology*, volume 936 of *Lecture Notes in Computer Science*, pages 245–260. Springer, Berlin, 1995.

[Jac96] JACOBS, B.: *Objects and classes, co–algebraically.* In FREITAG, B., C.B. JONES, C. LENGAUER, and H.-J. SCHEK (editors): *Object–Orientation with Parallelism and Peristence*, pages 83–103. Kluwer Acad. Publ., 1996.

[Jac97a] JACOBS, B.: *Behaviour-refinement of coalgebraic specifications with coinductive correctness proofs.* In BIDOIT, M. and M. DAUCHET (editors): *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 787–802. Springer, Berlin, 1997.

[Jac97b] JACOBS, B.: *Invariants, bisimulations and the correctness of coalgebraic refinements.* In JOHNSON, M. (editor): *Algebraic Methodology and Software Technology*, volume 1349 of *Lecture Notes in Computer Science*, pages 276–291. Springer, Berlin, 1997.

[Jac99] JACOBS, B.: *Categorical Logic and Type Theory*, volume 141 of *Studies in Logic and the Foundations of Mathematics*. North Holland, Elsevier, 1999.

[Jac00] JACOBS, B.: *Many-sorted coalgebraic modal logic: a model-theoretic study.* Technical Report CSI-R0020, University of Nijmegen, 2000.

[Jac02] JACOBS, B.: *Exercises in coalgebraic specification.* In R. BACKHOUSE, R. CROLE and J. GIBBONS (editors): *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*, pages 237–280. Springer, Berlin, 2002.

[Jay96] JAY, C. B.: *Data categories.* In HOULE, M. E. and P. EADES (editors): *Proceedings of Conference on Computing: The Australian Theory Symposium*, pages 21–28. Australian Computer Science Communications, January 1996.

[JT01] JACOBS, B. and H. TEWS: *Assertional and behavioural refinement in coalgebraic specification.* Submitted, 2001.

[JvdBH+98] JACOBS, B., J. VAN DEN BERG, M. HUISMAN, M. VAN BERKUM, U. HENSEL, and H. TEWS: *Reasoning about classes in Java (preliminary report).* In *Object–Oriented Programming, Systems, Languages and Applications*, pages 329–340. ACM Press, 1998.

[Kel97] KELLOMÄKI, P.: *Mechanical Verification of Invariant Properties of DisCo Specifications.* PhD thesis, Tampere University of Technology, Finland, 1997.

[KM00] KAWAHARA, Y. and M. MORI: *A small final coalgebra theorem.* Theoretical Computer Science, 233(1–2):129–145, February 2000.

[Kur98] Kurz, A.: *Specifying coalgebras with modal logic.* In Jacobs, B., L. Moss, H. Reichel, and J. Rutten (editors): *Coalgebraic Methods in Computer Science '98*, volume 11 of *ENTCS*, pages 57–72. Elsevier, 1998.

[Kur00] Kurz, A.: *A co-variety-theorem for modal logic.* In Zakharyaschev, A., K. Segerberg, M. de Rijke, and H. Wansing (editors): *Advances in Modal Logic*, volume 2, pages 385 – 398, 2000.

[Lam94] Lamport, L.: *The temporal logic of actions.* ACM Transactions on Programming Languages and Systems, 16(3):872–923, May 1994.

[Lam00] Lambooij, P.: *The YAPI protocol for buffered data transfer.* Techn. Rep. CSI-R9923, Comput. Sci. Inst., Univ. of Nijmegen. Available at URL http://www.cs.kun.nl/csi/reports/info/CSI-R9923.html, 2000.

[LDG+01] Leroy, X., D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon: *The Objective Caml system, release 3.04*, December 2001. Available at URL http://caml.inria.fr/ocaml/htmlman/.

[LLP+00] Leavens, G. T., K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs: *JML: notations and tools supporting detailed design in Java.* In *OOPSLA 2000 Companion*, pages 105–106, Minneapolis, Minnesota, October 2000. ACM.

[Mey92] Meyer, B.: *Eiffel: The Language.* Prentice Hall, 1992.

[Mey97] Meyer, B.: *Object-Oriented Software Construction, Second Edition.* The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA, 1997.

[Mey99] Meyer, D.: *A case study in object oriented specification and verification: The MSMIE protocol.* Graduation thesis, Catholic University of Nijmegen, January 1999.

[Mos97] Mosses, P. D.: *CoFI: The common framework initiative for algebraic specification and development.* In Bidoit, M. and M. Dauchet (editors): *TAPSOFT '97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 115–140. Springer-Verlag, 1997.

[Mos99] Moss, Lawrence S.: *Coalgebraic logic.* Annals of Pure and Applied Logic, 96(1–3):277–317, 1999.

[Mos00] Mossakowski, T.: *CASL: From semantics to tools.* In Graf, S. and M. Schwartzbach (editors): *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 93–108. Springer, 2000.

[MP88] Mitchell, J. C. and G. D. Plotkin: *Abstract types have existential types.* ACM Trans. on Programming Languages and Systems, 10(3):470–502, 1988.

[MTH91] Milner, R., M. Tofte, and R. Harper: *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1991.

[NPW02a] Nipkow, T., L. C. Paulson, and M. Wenzel: *Isabelle's Logics: HOL*, March 2002. Available via http://isabelle.in.tum.de/.

[NPW02b] Nipkow, T., L.C. Paulson, and M. Wenzel: *Isabelle/HOL, A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[ORR+96] Owre, S., S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas: *PVS: Combining specification, proof checking, and model checking.* In Alur, R. and T.A. Henzinger (editors): *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer, Berlin, 1996.

[ORSvH95] Owre, S., J.M. Rushby, N. Shankar, and F. von Henke: *Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS.* IEEE Trans. on Softw. Eng., 21(2):107–125, 1995.

[OS93] Owre, S. and N. Shankar: *Abstract datatypes in PVS.* Technical Report SRI-CSL-93-9R, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Extensively revised June 1997; Also available as NASA Contractor Report CR-97-206264.

[Pau02] Paulson, L. C.: *Introduction to Isabelle*, March 2002. Available via http://isabelle.in.tum.de/.

[PT94] Pierce, B. C. and D. N. Turner: *Simple type-theoretic foundations for object-oriented programming.* Journal of Functional Programming, 4(2):207–247, April 1994.

[Rei95] Reichel, H.: *An approach to object semantics based on terminal co–algebras.* Mathematical Structure in Computer Science, 5:129–152, 1995.

[Röß99] Rössiger, M.: *Languages for coalgebras on datafunctors.* In Jacobs, B. and J. Rutten (editors): *Coalgebraic Methods in Computer Science*, volume 19 of *ENTCS*. Elsevier, Amsterdam, 1999.

[Röß00a] Rössiger, M.: *Coalgebras and modal logic.* In Reichel, H. (editor): *Coalgebraic Methods in Computer Science '00*, volume 33 of *ENTCS*. Elsevier Science Publishers, 2000.

[Röß00b] Rössiger, M.: *Coalgebras, Clone Theory and Modal Logic.* PhD thesis, Univ. of Dresden, Germany, 2000.

[Rot00] Rothe, J.: *Modal logics for coalgebraic class specification.* Master's thesis, TU Dresden, Germany, 2000. Available at URL: http://wwwtcs.inf.tu-dresden.de/~janr/diplom.ps.gz.

[RTJ01] Rothe, J., H. Tews, and B. Jacobs: *The coalgebraic class specification language CCSL*. Journal of Universal Computer Science, 7(2):175–193, March 2001.

[Rut00] Rutten, J. J. M. M.: *Universal coalgebra: A theory of systems*. Theoretical Computer Science, 249(1):3–80, October 2000.

[Sch90] Schmidt, D. C.: *gperf: a perfect hash function generator*. In *USENIX C++ Conference*, pages 87–101, Berkeley, CA, USA, 1990. USENIX Association.

[Sch97] Schroeder, M. A.: *Higher order charity*. Master's thesis, The University of Calgary, Department of Computer Science, 1997.

[Sec98] Secure Networks: *Custom Attack Simulation Language (CASL), User manual*, January 1998.

[Sti92] Stirling, Colin: *Modal and temporal logics*. In Abramsky, S., D. M. Gabbay, and T.S.E. Maibaum (editors): *Handbook of Logic in Computer Science*, volume 2, chapter Modal and Temporal Logics. Oxford Science Publications, 1992.

[Str97] Stroustrup, B.: *The C++ Programming Language: Third Edition*. Addison-Wesley Publishing Co., Reading, Mass., 1997.

[Tar55] Tarski, Alfred: *A lattice-theoretic fixpoint theorem and its applications*. Pacific Journal of Mathematics, 5(2):285–309, 1955.

[Tew00] Tews, H.: *A case study in coalgebraic specification: Memory management in the* FIASCO *microkernel*. Technical Report TPG2/1/2000, SFB 358, April 2000. Available at URL http://wwwtcs.inf.tu-dresden.de/∼tews/vfiasco/.

[Tew02a] Tews, H.: *The Coalgebraic Class Specification Language CCSL (Reference manual)*, 2002. Available via URL wwwtcs.inf.tu-dresden.de/∼tews/ccsl/.

[Tew02b] Tews, H.: *Coalgebraic Methods for Object-Oriented Specification*. PhD thesis, University of Dresden, 2002. available via wwwtcs.inf.tu–dresden/∼tews/PhD.

[U. 97] U. Schöning: *Theoretische Informatik kurz gefasst*. Akademischer Verlag, 3. Auflage, 1997.

[vdBJ01] Berg, J. van den and B. Jacobs: *The LOOP compiler for Java and JML*. In Margaria, T. and W. Yi (editors): *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312, 2001.

[VEK00] Vigna, G., S.T. Eckmann, and R.A. Kemmerer: *Attack languages*. In *Proceedings of the IEEE Information Survivability Workshop*, Boston, MA, October 2000. available via http://www.cs.ucsb.edu/∼vigna/.

[Wen02] Wenzel, M.: *The Isabelle/Isar Reference Manual*, March 2002. Available via http://isabelle.in.tum.de/.

[Wir90] Wirsing, M.: *Algebraic specification.* In Leeuwen, J. van (editor): *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 13, pages 673–788. Elsevier/MIT Press, 1990.

Notation Index

The notation index is split into six parts: (1) the index of logical entailments, (2) the index of symbols based on the Greek alphabet, (3) the index of symbols based on the Latin alphabet (on page 134), and (4) the index of other symbols (on page 134).

Entailment Index

Greek Symbols

Latin Symbols

Other Symbols

Subject Index

Entries in `UPPERCASE TYPEWRITER` are CCSL keywords. Entries in *slanted* are meta symbols. For both kinds of entries page numbers smaller than 121 refer into the body of this report, larger page numbers refer into Appendix C with the full CCSL grammar.