Relations between Abstract Datatypes modeled as Abstract Datatypes

Hubert Baumeister

Dissertation
zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Technischen Fakultät
der Universität des Saarlandes

Saarbrücken
1998

Dekan:	Prof. Dr. Wolfgang Paul
Gutachter:	Prof. Dr. Harald Ganzinger
	Prof. Dr.-Ing. Jacques Loeckx
Termin des Kolloquiums:	21. Mai 1999

Meinen Eltern,
für alles

Abstract

In this thesis we define a framework for the specification of dynamic behavior of software systems. This framework is motivated by the state as algebra approach and the model-oriented language Z. From the state as algebra approach we use the idea of modeling the environment and the state components as structures of an institution. However, in contrast to the state as algebra approach, states in our framework are modeled by structures from any suitable institution not only those having algebras as their structures. From Z we use the idea that environment, state spaces, and relations between state spaces are specified using the same logic, and how more complex relations can be constructed from simpler ones by means of the schema calculus. However, we differ from Z in that our framework can be instantiated by different institutions while the approach of Z can only work because of the particular logical system used by Z.

Zusammenfassung

In dieser Arbeit schlagen wir einen Ansatz zur Spezifikation des Verhaltens von Softwaresystemen vor. Dieser Ansatz ist motiviert durch Spezifikationssprachen, die den Zustand von Softwaresystemen als Algebren modellieren, und durch die modellorientierte Spezifikationssprache Z. Von der ersten Gruppe von Spezifikationssprachen übernehmen wird das Konzept den Zustand als Struktur einer Logik zu modellieren; allerdings ist unser Ansatz nicht auf Logiken beschränkt, deren Strukturen Algebren sind. Von Z übernehmen wir die Idee Zustandsräume und Zustandsübergänge mit derselben Logik zu spezifizieren, und die Art und Weise wie aus einfachen Zustandsräumen und Zustandsübergängen komplexere Zustandsräume und Zustandsübergänge konstruiert werden können. Im Gegensatz zu Z können wir eine beliebige Logik verwenden, während der Ansatz von Z nur funktioniert weil eine ganz spezielle Logik verwendet wird.

Extended Abstract

In this thesis we study relations between abstract datatypes. Relations between abstract datatypes can be used to specify the dynamic behavior of software systems. An abstract datatype, a pair consisting of a signature and a class of structures over that signature, defines the state-space of a software system, and relations between abstract datatypes define how the state of a software system changes.

Our approach is connected to the state as algebra approach, where the state of a software system is modeled by an algebra, for example, [33, 4, 22, 29, 59, 20, 16, 23]. However, in a similar way as Goguen and Burstall [13] used the notion of institution to define the semantics of the specification language Clear independent from a particular logical system, we use institutions to study relations between abstract datatypes from an arbitrary institution, not only those consisting of a many-sorted signature Σ and a class of Σ-algebras as in the state-as-algebra approach. For example, we define the institution $\mathcal{SET}$ that models the logical system of the model-oriented specification language Z [56], which is based on set-theory. But not only institutions for existing logical systems may be used, one is free to define ones own institution whose structures model in a more appropriate way the states of the class of software systems one is dealing with.

Our approach is also similar to the way the dynamic behavior of software systems is specified in Z. However, Z's method depends on the particular logical systems used by Z. In this thesis we show a way to uncouple Z's method from the underlying logical system. This yields a specification method which is in many aspects similar to Z's method, but can be used with any suitable logical system, including the logical systems used for the state as algebra approach, as well as the logical system used by Z.

The method we use in this thesis is to model relations between abstract datatypes again as abstract datatypes. To this end we define the institution $\mathcal{R}_{\mathcal{I}}$ which is based on a suitable institution $\mathcal{I}$. Then relations between abstract datatypes from $\mathcal{I}$ are modeled by abstract datatypes from $\mathcal{R}_{\mathcal{I}}$. This has the advantage that a lot of results from the institution independent theory of abstract datatypes can be reused for the specification of relations. For example, we use the operations defined by Sannella and Tarlecki [50] as the basis for the specification language $\mathtt{RSL}_{\mathcal{I}}$ used to specify relations. We show also how to reuse the proof calculus for proving properties of abstract datatypes and entailment of abstract datatypes, for example, from Wirising [61], to prove properties of relations and entailment of relations.

The results of this thesis can be used in two ways to specify the dynamic behavior of software systems. First, $\mathtt{RSL}_{\mathcal{I}}$ can be used as the basis for a more practical specification language. However, new tools have to be designed and implemented for that language. Another way is to use the translation $\mathsf{G}^{\mathcal{A}}$ of abstract datatypes in $\mathcal{R}_{\mathcal{I}}$, that is relations, to abstract datatypes

in $\mathcal{I}$, which has the property that it preserves entailment. Then one can reuse any specification language having as semantics abstract datatypes in $\mathcal{I}$ together with its tools to specify relations. This yields a specification style similar to that used by Z.

Ausführliche Zusammenfassung

Das Ziel dieser Arbeit ist das Studium von Relationen zwischen abstrakten Datentypen. Relationen zwischen abstrakten Datentypen werden verwendet um das dynamische Verhalten von Softwaresystemen zu spezifizieren. Dabei definieren abstrakte Datentypen, die aus einer Signatur und einer Klassen von Strukturen über diese Signatur bestehen, Zustandsräume von Softwaresystemen und Relationen zwischen abstrakten Datentypen deren Zustandsübergänge.

Unser Ansatz ist vergleichbar mit den Spezifikationssprachen, die den Zustand eines Softwaresystems als Algebra modellieren, zum Beispiel [33, 4, 22, 29, 59, 20, 16, 23]. Wir verallgemeinern den Ansatz dieser Spezifikationssprachen und erlauben es den Zustand eines Softwaresystems als eine Struktur bzgl. einer beliebigen Institution $\mathcal{I}$ zu modellieren. Zum Beispiel kann man die, in dieser Arbeit definierte, Institution $\mathcal{SET}$ verwenden, die die Logik der modellorientierten Spezifikationssprache Z [56] repräsentiert und auf Mengentheorie basiert. Es ist aber auch möglich eigene Institutionen zu definieren, deren Strukturen die Zustände von Klassen von Softwaresystemen besser modellieren als die Strukturen von Institutionen, die eine existierende Logik repräsentieren.

Aus einer ähnlichen Motivation heraus haben Goguen und Burstall [13] den Begriff der Institution entwickelt um die Semantik der Spezifikationssprache Clear unabhängig von einer speziellen Logik zu formulieren.

Unser Ansatz ist ähnlich der Methode, die in Z verwendet wird um das Verhalten dynamischer Systeme zu spezifizieren. Allerdings hängt die Methode von Z von der speziellen Logik ab die Z verwendet. Ein Resultat dieser Arbeit ist die Trennung dieser Methode von der Logik, die Z verwendet. Das Ergebnis ist eine Spezifikationsmethode, die in vielen Aspekten der von Z entspricht, aber nicht von der Logik von Z abhängt. So ist es zum Beispiel möglich Varianten von Gleichungslogik zu verwenden, aber auch wieder die Logik von Z.

In unserem Ansatz modellieren wir Relation zwischen abstrakten Datentypen wieder als abstrakte Datentypen. Zu diesem Zweck definieren wir eine Institution $\mathcal{R}_{\mathcal{I}}$ basierend auf einer geeigneten Institution $\mathcal{I}$, deren abstrakte Datentypen Relationen zwischen abstrakten Datentypen von $\mathcal{I}$ repräsentieren. Dieses Vorgehen hat den Vorteil, daß die Theorie der abstrakten Datentypen, die unabhängig von einer speziellen Institution ist, auf Relationen zwischen abstrakten Datentypen angewandt werden kann. Zum Beispiel liefern die Operation auf abstrakten Datentypen, wie sie bei Sannella und Tarlecki [50] definiert sind, die Basis für eine rudimentäre Spezifikationssprache $\mathtt{RSL}_{\mathcal{I}}$ zur Spezifikation von Relationen und wir zeigen wie man Inferenzsysteme, die es erlauben Eigenschaften von abstrakten Datentypen abzuleiten (zum Beispiel Wirsing [61]), anwenden kann um Eigenschaften von Relationen abzuleiten.

Die Resultate dieser Arbeit lassen sich in zweifacher Hinsicht zur Spezifikation des dynamischen Verhaltens von Softwaresystemen verwenden. Erstens kann man $\mathtt{RSL}_{\mathcal{I}}$ als Basis einer praktischeren Spezifikationssprache verwenden. Der Nachteil dieser Methode ist, daß neue Werkzeuge für diese Spezifikationssprache entwickelt und implementiert werden müssen. Einen zweiten Weg liefert die Übersetzung $\mathsf{G}^{\mathcal{A}}$, die abstrakte Datentypen in $\mathcal{R}_{\mathcal{I}}$ — also Relationen — in abstrakte Datentypen der Institution $\mathcal{I}$ übersetzt. Damit kann man jede Spezifikationssprache mit ihren Werkzeugen wiederverwenden, deren Semantik abstrakte Datentypen in $\mathcal{I}$ sind. Als Ergebnis erhält man einen Stil zur Spezifikation von dynamischen Verhalten, der dem von Z entspricht.

Acknowledgments

I am indebted to my supervisor Harald Ganzinger for his support, his patients, and for providing the pressure that I needed to finish the thesis. I also thank Jacques Loeckx for agreeing to referee this thesis.

I wish to thank Viorica Sofronie for careful reading several drafts of this thesis and suggesting improvements. Without her comments the thesis would be very much different.

I also like to thank Andrzej Tarlecki for clarifying some subtle points with indexed categories.

Finally, I would like to thank my colleagues at the Max-Planck-Institut für Informatik for fruitful discussions and a lot of fun, especially, Jürgen Stuber, Uwe Waldmann, Andreas Tönne, Ullrich Hustadt, Renate Schmidt, Uwe Brahm, Margus Veanes, and Giorgio Delzanno.

Contents

1 Introduction

It is well recognized that at least some, if not all, aspects of a software system should be formally specified. Among these aspects are the functional requirements and the dynamic behavior of a software system. A specification of the functional requirements is concerned with the data of a software system and how this data is transformed by functions. Usually property-oriented, also known as axiomatic or algebraic, specification languages are used for functional requirement specifications. When writing property-oriented specifications, one first defines the sorts and functions involved; then, their properties are defined by using equations or, more generally, first-order formulas. Examples of property-oriented specification languages are Clear [13], the Larch Shared Language [34], ASL [3], OBJ [28], Obscure [41], ACT [15], and CASL [45].

A specification of the dynamic behavior of a software system is concerned with the changes of the behavior of that system after performing some operations. This means that the system may react differently on the same operation depending on the operations that have been performed in the meantime. An example is the top operation of stacks which returns the top-most element of a stack. The result of two successive top operations yields different elements if a pop operation which removes the top-most element of a stack is performed in between the top operations. This leads to the notion of state of a software system which is an abstraction of the history of a software system. Model-oriented specification languages are often used for dynamic behavior specifications. When writing model-oriented specifications one first constructs an abstract model of the state of the software system and then defines for each operation the effect of this operation on the state. Examples of model-oriented specification languages are Z [56], VDM [40], and B [1].

In general, functional requirement specifications precede dynamic behavior specifications in the software engineering process, in particular, if the target programming language is imperative or object-oriented. The dynamic behavior specification can be seen as a refinement of the functional requirement specifications. Take for example a functional requirement specification of the union operation of sets. Given two sets, the result of the union operation is a set that contains as its only elements all the elements of the argument sets. A dynamic behavior specification specifies in addition whether a new set is created for the result while the arguments remain unchanged, or if the elements of the second set are added to the elements of the first set and thus change the first set.

However, a functional requirement specification need not be followed by a dynamic behavior specification if, for example, the target programming language is a functional language. Similarly, one may omit the functional requirement specification and directly start with a dynamic behavior specification if its style is more appropriate to the software system that is to be specified, for example a database system.

In the two-tiered approach of Larch [34] functional requirement specifications and dynamic behavior specifications are combined. The datatypes which are used to model the states are given as functional requirement specifications. For this the Larch Shared Language is used which is a specification language based on many-sorted total algebras and equational logic with constraints. For each target programming language there exist different Larch Interface Languages for the specifications of the dynamic behavior of the functions, procedures, or methods of that programming language.

A problem with the two-tiered approach of Larch is that for each programming language the state is modeled in a different way. This is in contrast to the state as algebra approaches where not only the used datatypes are defined by algebras, but also the states. This has the advantage that the datatypes used by the components of the state as well as the set of admissible states of a software system can be specified by the same specification language.

There are quite a few specification languages and methods using the state as algebra approach, for example, Gurevich's Abstract State Machines [33], also known as evolving algebras, d-oids by Astesiano and Zucca [4], COLD-K by Feijs and Jonkers [22], transition categories by Große-Rhode [29], equational dynamic logic by Wieringa [59], an informal proposal to dynamic algebras by Ehrig and Orjeas [20], the implicit state approach by Dauchy and Gaudel [16], and the proposal by Ganzinger to model the semantics of programs by transformation of algebraic theories [23].

In this thesis we abstract away from the use of algebras to model states; instead, we use structures from any suitable institution. Goguen and Burstall [13] introduced the notion of institution to define the semantics of the specification language Clear independent from a logical system. In a similar way we use institutions to define a framework for dynamic behavior specifications which can be instantiated, for example, with states modeled as many-sorted total algebras, many-sorted partial algebras, order-sorted algebras, and first-order structures. In addition, we define two new institutions which can be used to model states. The institution $\mathcal{LSL}$ models the logical system of the Larch Shared Language, and the institution $\mathcal{SET}$ models the logical system of the model-oriented specification language Z.

Given a specification language whose semantics are abstract datatypes, that is pairs consisting of a signature and a class of algebras over that signature, then a specification in that specification language defines the state space of a software system. Operations that modify the state of a software system are relations between abstract datatypes or, more precisely, relations between the classes of algebras given by abstract datatypes.

In this thesis we concentrate on the specification of state transformations, that is we study relations between abstract datatypes. We provide a set of basic operations to construct complex relations from simpler ones. These operations are parallel composition, extending the domain of a relation by adding new state components, and restricting a relation by hiding state components. Other operations, like sequential composition, can be defined in terms of these basic operations.

Another way to define state transitions relates the state components before and after a state transformation using formulas. Our approach is based on the observation that we can use the same kind of formulas that were used to define the components of the environment and the set of admissible states to relate the state components before and after a state transformation. These formulas depend on the logical system we use for the states; however, they do not depend on a particular logical system, like equational logic or first-order logic.

We also provide an inference system for proving properties from relations. This inference system is based on an inference system for the institution used to model the states. This allows us to reuse theorem provers for the base institution, if existent, to prove properties of relations.

This work is strongly influenced by the model-oriented specification language Z [56, 57]. In Z states are modeled as elements of schema-type, state spaces as schemata which are sets of elements of schema-type, and operations (relations between state-spaces) again as schemata. The schema-calculus allows to construct complex state spaces and operations from simpler state spaces and operations. Since state spaces and operations are schemata, and schemata are objects in the logical system underlying Z, proving properties of states and proving operations are done using the same inference system.

However, the fact that schemata are objects of the logical-system of Z implies that the method used by Z for the specification of state-based software systems depends on the particular logical system of Z and thus cannot be applied directly to the state as algebra approach.

In this thesis we present a way to uncouple Z's method for the specification of state based systems from the logical system used by Z. This is based on the close relationship between schemata and abstract datatypes already observed by Spivey [56]. This yields a specification method which is in many aspects similar to Z's method, but can be used with any suitable logical system, including the logical systems used for the state as algebra approach.

1.1 Relations between Abstract Datatypes

As an example of a relation between abstract datatypes consider a simple counter with an increment operation. The state space of the counter is modeled as a class of Σ_C-algebras M_C where Σ_C is a many-sorted signature with one sort Nat, an operation succ from Nat to Nat, and constants zero and c of sort Nat; and M_C has the following properties: For each algebra A in M_C $A(\mathtt{Nat})$ is the set of natural numbers $\mathbb{N}$, $A(c)$ is an arbitrary natural number, $A(\mathtt{zero}) = 0$,and $A(\mathtt{succ})(n) = n + 1$ for each natural number n.

The relation $\mathtt{Inc} \subseteq M_C \times M_C$ is defined as all pairs (A, B) such that $B(c) = A(c) + 1$. Note that, because of the way M_C is defined, $A(\mathtt{Nat})$ is the same as $B(\mathtt{Nat})$, $A(\mathtt{succ})$ the same as $B(\mathtt{succ})$, and $B(\mathtt{zero})$ is the same as $A(\mathtt{zero})$. Thus Inc defines an operation incrementing the constant c while leaving the interpretation of Nat, succ, and zero unchanged.

The advantage of viewing a class of Σ-algebras M as an abstract datatype (Σ, M) is the possibility to use a specification language that has abstract datatypes as the semantics of their specifications to define the domain of relations. For example, the abstract datatype (Σ_C, M_C) is given by the following specification using a syntax adopted from the Larch Shared Language:[1]

```
Counter : trait
introduces
   succ: Nat
   zero: Nat
   c: Nat
asserts
   Nat generated freely by succ, zero
```

Now consider the following PASCAL program:[2]

```
program example;
   var c: natural;
   procedure increment;
   begin
      c := c + 1;
   end;
begin
   c := 10;
   increment;
end.
```

Its state can be viewed as a Σ_C-algebra where Σ_C contains as sorts the PASCAL types — in this case `natural` — and for each variable x of type T a constant x of sort T, which in this example is c. Then we can view the semantics of a PASCAL procedure as a relation between abstract datatypes. This makes it possible, for example, to check whether the PASCAL procedure `increment` implements the relation `Inc` defined above.

However, for specification purposes we do not want to limit ourselves to those algebras that model states of PASCAL programs. Instead, we want to use more problem oriented descriptions of the state space, and only after a process of data-refinement reach algebras that model states of PASCAL programs to show that a certain PASCAL program implements its specification.

Thus, we are not limited to the use of constants as state components in the state as algebra approach. For example, we can define the state of a dictionary by an algebra over a signature that has sorts `bool`, E for the elements of the dictionary, K for the keys of the dictionary, and functions `map` from K to E and `dom` from K to `bool`. The intention is that the function `map` records the associations between keys and values, but since `map` is a total function, we need the function `dom` which tells us for each key if the key/value pair is valid. Now an operation

[1] Note that it is not completely true that (Σ_C, M_C) is the semantics of the given specification since for an algebra A in M_C, $A(\texttt{Nat})$ is the *same* as $\mathbb{N}$ while for an algebra B satisfying the LSL-specification we can only ensure that $B(\texttt{Nat})$ is *isomorphic* to $\mathbb{N}$. However, as we will see in later chapters, this poses no problems.

[2] For the purpose of this introduction we assume that PASCAL has a datatype `natural` for the natural numbers.

$\texttt{Update}(k, e)$, adding an association (k, e) to the dictionary, changes the value of map such that $\texttt{map}(k')$ is e if $k' = k$ and is the old value of $\texttt{map}(k')$ if $k \neq k'$. Similarly, dom is changed such that $\texttt{dom}(k')$ yields true if $k' = k$ and returns the old value of $\texttt{dom}(k')$ otherwise.

The theory presented in this thesis is largely independent from the underlying logical systems used to model the states. To make this precise, we use the notion of an institution. The idea of an institution $\mathcal{I}$ is to abstract away from the notion of many-sorted signatures, algebras, equations, and satisfaction to an arbitrary class of signatures such that each signature Σ has associated a class of Σ-structures, a set of Σ-formulas, and a satisfaction relation $\models^{\mathcal{I}}$ which relates Σ-structures to Σ-formulas valid in these structures. The definition of abstract datatypes extends naturally to arbitrary institutions. An abstract datatype with respect to $\mathcal{I}$ is a pair (Σ, M) where Σ is a signature from $\mathcal{I}$ and M a class of Σ-structures.

A variety of well-known logical systems have been shown to be institutions, and a lot of these can be used in the framework of this thesis; for example, the original paper by Goguen and Burstall [25] showed that equational logic, that is many-sorted signatures together with algebras as Σ-structures and (conditional) equations as Σ-formulas, forms an institution. Also first-order logic is an institution. As part of this thesis we show that the logical systems underlying the specification languages LSL, which is a variant of equational logic with constraints, and Z, which is based on set-theory, are institutions. It is also possible to define a new institution appropriate to model the states of a particular class of software systems. For example, one could define an institution which naturally allows to talk about states containing arrays and pointers.

The instantiation of our framework by different institutions results in different specifications for the same problem. For example, if in the dictionary example the states were modeled by partial algebras, the state would probably consist only of a partial function map from K to E. For any key not in the range of the dictionary map would be undefined. The operation $\texttt{Update}(k, e)$ would change $\texttt{map}(k)$ to yield e if map is defined for k and would extend the domain of map to include k if map is not defined for k. If the institution $\mathcal{SET}$ were used, then map most likely would be a subset of $K \times E$ such that if (k, e) and (k, e') are in map, then $e = e'$. The operation $\texttt{Update}(k, e)$ would add the pair (k, e) to map and remove any pair (k, e') with $e \neq e'$ from map.

In this thesis we construct a new institution $\mathcal{R}_{\mathcal{I}}$ based on a suitable institution $\mathcal{I}$ used to describe the states. The signatures of $\mathcal{R}_{\mathcal{I}}$ are, roughly, tuples $\Theta = ((\Sigma_1, M_1), \ldots, (\Sigma_n, M_n))$ of abstract datatypes from $\mathcal{I}$, and Θ-structures are tuples $(A_1, \ldots, A_n)$ where A_i is in M_i for $1 \leq i \leq n$. Now we can view a relation $R \subseteq M_1 \times \ldots \times M_n$ as an abstract datatype (Θ, R) with respect to the institution $\mathcal{R}_{\mathcal{I}}$.

The advantage of this approach is that we can reuse the institution independent theory of abstract datatypes found, for example, in the paper of Sannella and Tarlecki [50] for the definition of relations. The operations on abstract datatypes can be interpreted in $\mathcal{R}_{\mathcal{I}}$ as operations on relations. For example, we show how the relation Inc can be defined as the class of all pairs $(A, B) \in M_C \times M_C$ such that (A, B) satisfies the equation $c' = \texttt{succ}(c)$ where c' refers to the value of $B(c)$ and c to the value of $A(c)$. Note that, by the choice of

M_C, $A(\texttt{succ})$ and $B(\texttt{succ})$ are the same. Other operations to construct relations are union, extension, and restriction. More complex operations on relations, like sequential composition, instantiation etc. can be defined in terms of these primitive operations.

Another consequence of the construction of $\mathcal{R}_\mathcal{I}$ is that we can reuse the institution independent proof calculus for proving properties of abstract datatypes and refinement of abstract datatypes studied, for example, by Sannella and Tarlecki [50], Wirsing [61], and Hennicker, Wirsing and Bidoit [38] to prove properties of relations and entailment of relations.

1.2 Structure of the Thesis

The rest of this chapter gives a brief overview of specification languages based on the state as algebra approach.

The notion of institutions is based on category theory; thus Chapter 2 introduces the terminology from category theory needed for the main construction of this thesis in Chapter 5. The reader familiar with (co)limits, adjunctions, and indexed categories may want to skip this chapter.

Chapter 3 introduces institutions and gives three examples: the institution $\mathcal{EQ}$ of equational logic, $\mathcal{EQC}$ of equational logic with constraints, and the institution $\mathcal{LSL}$, which formalizes the logical system underlying the Larch Shared Language [34].

Chapter 4 summarizes the institution independent theory of abstract datatypes. It introduces the specification language $\texttt{SL}_\mathcal{I}$ for writing specifications of abstract datatypes in an institution $\mathcal{I}$, and we present an inference system for proving specification entailment.

Chapter 5 contains the main construction of this thesis — the institution $\mathcal{R}_\mathcal{I}$ parameterized by a suitable institution $\mathcal{I}$ — and proves properties of this institution, like that the category of signatures of $\mathcal{R}_\mathcal{I}$ is (finitely) cocomplete and that the structure functor preserves colimits. Another important property is the relationship between abstract datatypes in $\mathcal{R}_\mathcal{I}$ and abstract datatypes in $\mathcal{I}$. The chapter ends with the introduction of the language $\texttt{RSL}_\mathcal{I}$ for the definition of relations.

Chapter 6 applies the theoretical results of the previous chapter to the specification of abstract datatypes with states, also called abstract machines. Several styles for the specification of abstract machines are discussed.

Chapter 7 extends the methods for proving specification entailment introduced in Chapter 4 to prove entailment of relations. The main technique is the translation of $\texttt{RSL}_\mathcal{I}$-expressions to $\texttt{SL}_\mathcal{I}$-expressions, which preserves entailment of relations. As an example we apply these results to the refinement of abstract machines and prove the correctness of the implementation of a stack by using a pointer in an array. This stack implementation is also an example of the construction of more complex relations from simpler ones.

In Chapter 8 we add disjunction to our set of operations on specifications and thus to $\texttt{RSL}_\mathcal{I}$. This allows us to define relations on parts of the input domain first and then combine these relations into one relation, which is defined on the whole input domain. Disjunction was not studied before as a specification building operation mainly because, in contrast to the union of specifications, the disjunction of specifications is not in all institutions expressible on the presentation level; for this a sufficently expressive institution is needed. We shall show that $\mathcal{LSL}$ is such an institution.

Chapter 9 compares the approach of this thesis with the approach taken by Z for the specification of sequential systems. To this end we first define the institution $\mathcal{SET}$ of the logical system underlying Z, which is a variant of set theory, and then investigate the relationship between elements of schema-types and abstract datatypes over $\mathcal{SET}$. The outcome is a specification style similar to that used by Z for the specification of sequential systems, but using an algebraic specification language — LSL — instead of Z.

1.3 Related Work

There are mainly two approaches to the specification of state based software systems using algebras. In the first approach, also known as the *explicit state* approach (cf. [16]), a state is modeled as an element of the carrier set of a special state sort. Operations changing the state of a software system are functions having the state sort in its domain and co-domain. To enforce data abstraction, one usually *hides* the state sort which means that one cannot directly compare two elements of the state sort. This leads to the hidden sorted approach (cf. Goguen et al [24, 26]).

The other approach is the state as algebra approach where the state of a software system is modeled as an algebra and a state transformation as a relation or function between classes of algebras. This approach goes back to at least Ganzinger [23], who used this approach to define an algebraic semantics of imperative programming languages.

Semantic of While-Programs Ganzinger defines the semantics of while-programs to be the composition of a free functor and a forgetful functor on algebras representing the state of the program. In principle there are three specifications: $\mathsf{Spec}_{\mathsf{base}}$, Spec_X, and $\mathsf{Spec}^p_{X\cup X'}$. $\mathsf{Spec}_{\mathsf{base}}$ is the specification of some base types defining the domain over which the variables range. Spec_X is a conservative extension of $\mathsf{Spec}_{\mathsf{base}}$ containing constants of the form $x : s$ denoting the variables of the program. Each program p determines a specification $\mathsf{Spec}^p_{X\cup X'}$ which is an enrichment of Spec_X that contains for each $x : s$ in Spec_X a constant $x' : s$. Further, it contains equations defining the semantics of the program p. For example in the case where p is the assignment statement $x := t$, $\mathsf{Spec}^p_{X\cup X'}$ contains the formula $x' = t$. The semantics of a program p is given by $T_p; U_\sigma$ where $T_p : \mathsf{Alg}(\mathsf{Spec}_X) \to \mathsf{Alg}(\mathsf{Spec}^p_{X\cup X'})$ is the left adjoint to the forgetful functor U with respect to the inclusion of Spec_X in $\mathsf{Spec}^p_{X\cup X'}$, and $U_\sigma : \mathsf{Alg}(\mathsf{Spec}^p_{X\cup X'}) \to \mathsf{Alg}(\mathsf{Spec}_X)$ is the forgetful functor for the signature morphism σ that takes each $x : s$ in Spec_X to $x' : s$ in $\mathsf{Spec}^p_{X\cup X'}$. In general T_p will be a persistent functor; however, if p does not terminate then $(T_p; U)(A) \neq A$.

Abstract State Machines (a.k.a. Evolving Algebras) Abstract State Machines, formerly known as evolving algebras, are intended to provide a computing device for formalizing the notion of algorithm similar to the Turing machine model, which formalizes the notion of computable function. The Abstract State Machine Thesis claims that for any algorithm there always exists an appropriate abstract state machine that simulates that algorithm in lock-step on the same abstraction level of that algorithm. Thus, Abstract State Machines are able to directly execute algorithms without translating them to a lower abstraction level.

> Computation models and specification methods seem to be worlds apart. The Abstract State Machine (ASM) project started as an attempt to bridge the gap by improving on Turing's thesis. We sought more versatile machines which would be able to simulate arbitrary algorithms in a direct and essentially coding-free way. Here the term algorithm is taken in a broad sense including programming languages, architectures, distributed and real-time protocols, etc.. The simulator is not supposed to implement the algorithm on a lower abstraction level; the simulation should be performed on the natural abstraction level of the algorithm.
>
> The ASM thesis asserts that ASM's are such versatile machines.

This quote from Gurevich can be found on the web-site on Abstract State Machines[3] and is adapted from Gurevich [33].

The state of an abstract state machine is a one-sorted algebra. Relations and subsorts are modeled using characteristic functions.

A signature Σ is a finite collection of function names, each of a fixed arity and possibly marked as relation name or static name, or both. The set of function names includes the null-ary static function names **true**, **false**, and **undef**; the equal sign as a binary static relation name; and the usual Boolean operations.

A Σ-algebra A (or a state) is a non-empty set X, called super-universe, together with a set of functions for each function name f in the signature. If the arity of f is n, then the algebra contains a function $f^A : X^n \to X$ if it is not marked as a relation name and a function $f^A : X^n \to \{\mathbf{true}, \mathbf{false}\}$ if it is marked as a relation name. The null-ary function names **true**, **false**, and **undef** are required to be interpreted as different elements from the super-universe.

Unary relations U determine subsets of the super-universe and are called universes. Universes do not include **undef**.

All functions and relations are total. Partiality is modeled by returning **undef**. However, note that the characteristic function of a relation can not yield **undef**. Further, the boolean operations $\vee$, $\wedge$, $\neg$ etc. are not relations and therefore may have **undef** as a result. The boolean operations are strict while, in general, functions need not be strict.

The equality sign is interpreted as the characteristic function of the identity on the super-universe. Note that this implies $\mathbf{undef} = \mathbf{undef}$.

[3] `http://www.eecs.umich.edu/gasm/`

Transformations are build from sets of guarded multi-update instructions of the form

$$\textbf{if } g \textbf{ then } R.$$

A multi-update instruction R is a set of update instructions $f(\vec{t}) := t_0$ where f is a function name not marked as static, $\vec{t}$ a vector of ground terms matching the arity of f, and t_0 a ground term. Let $\vec{t}$ evaluate to $\vec{a}$ and t_0 to a_0 in A. If the guard g evaluates to **true** in A, then the update $f(\vec{t}) := t_0$ changes the interpretation of f at location $\vec{a}$ in A to a_0. All update instructions are performed simultaneously. The result of function applications to values not mentioned in an update instruction remain unchanged. In case of conflicting updates, i.e. where there are update instructions for updating $f^A(\vec{a})$ to a_0 and to b_0 ($a_0 \neq b_0$), A is not changed.

A program is a set of guarded multi-updates, and a run of a program is a possible infinite sequence of algebras starting from some initial state such that the next algebra in the sequence is obtained by firing the guarded multi-updates in the previous algebra.

The interpretation of function names marked static cannot change. The set of static names determines a sub-signature $\Sigma_{\mathtt{Env}}$ of Σ.

The execution of a program for an evolving algebra is deterministic. If two conflicting update instructions are to be executed, none of them is executed. This is in contrast to the Evolving Algebra Tutorial [32] where one update was chosen non-deterministically. In the current setting non-determinism is achieved by introducing a new transformation rule

```
choose v in U
   R
endchoose
```

This rule randomly picks an element a of the universe U and executes R in an algebra where v is bound to a. If the universe U is empty, the execution of the evolving algebra halts.

Parallelism is modeled by rules containing variables which are declared with

$$\mathtt{Var}\ v\ \mathtt{ranges\ over}\ U$$

where U is preferably finite. The semantics is that for each instantiation of v with a value from U the updates are performed in parallel.

D-oids D-oids are a model theoretic approach by Astesiano and Zucca [4] to define dynamic datatypes. D-oids consist of a set of algebras over a many-sorted signature together with a set of dynamic operations. A dynamic operation associates to an algebra A and a set of values from A an algebra B a return value from B and a family of functions $f_s : A(s) \to B(s)$ for each sort s called the tracking map.

In the D-oids approach not only the interpretation of the operation symbols may change, but also the interpretation of sorts. The tracking map f_s is used to relate the elements in the interpretation of the sort s before and after an application of a dynamic operation. For example, if a dynamic operation adds a new element of sort s, that is $B(s) = A(s) \cup \{e\}$, then f_s is the injection of $A(s)$ into $B(s)$.

Proposal for Dynamic Abstract Datatypes An approach similar to d-oids is the informal proposal for dynamic abstract datatypes by Ehrig and Orejas [20]. The instant structure specification, a conservative extension of some value type specification, is intended to model class sorts and attribute functions of objects. A dynamic operation denotes a function mapping a model A of the instant structure specification and a set of values from A to some instant algebra B.

Implicit State Approach In the implicit state approach of Dauchy and Gaudel [16] the states are enrichments of an abstract datatype defining the environment by a set of access functions. The set of access functions is divided into elementary access functions and non-elementary access functions. The non-elementary access functions are completely defined in terms of the elementary access functions and the functions in the environment. Modifiers are functors on states built from elementary modifiers by sequential and indifferent composition and using conditionals. Elementary modifiers modify the value of the elementary access functions; however, since the value of the non-elementary access functions depend on the value of the elementary access functions, the non-elementary access functions also change. The signature containing the symbols from the environment together with the elementary and non-elementary access functions is denoted by Σ_{ac}.

The implicit state approach is independent of the specification language used to specify the environment and how the non-elementary access functions relate to the elementary access functions as long as the semantics of a specification in that language is a class of many-sorted algebras.

Given an elementary access function $f : s_1 \to s_2$ with sorts s_1 and s_2 from the environment. An elementary modifier has the form $\mu\text{-}f(t_1, t_2)$ where t_1 is a term over the signature of the environment and t_2 is a term over Σ_{ac} such that all variables in t_2 also occur in t_1. Given a Σ_{ac}-algebra A and an elementary modifier $\mu\text{-}f(t_1, t_2)$, a new Σ_{ac}-algebra B is constructed by letting $B(f)(a) = \tilde{\rho}(t_2)$ if there exists a variable assignment ρ from the variables of t_1 to A such that $\tilde{\rho}(t_1) = a$. In case there is no such assignment, $B(f)(a)$ is the same as $A(f)(a)$. All the other elementary access functions do not change; however, the derived non-elementary access functions may change if they depend on f.

Transition Categories In the approach taken by Große-Rhode in his PhD-thesis [29, 30] the states of a software system are partial algebras over a many-sorted signature $\Sigma = (S, \Omega)$. Σ contains the sorts and operations of the data-types needed to describe the state components, and, in addition, some sorts in S are marked as reference sorts for other sorts in S. A reference sort for a sort s is denoted by $\texttt{ref}(s)$. Not every sort in S needs to have a reference sort, and it is possible to have a reference sort for a reference sort, for example, $\texttt{ref}(\texttt{ref}(s))$. For each reference sort $\texttt{ref}(s)$ there exists an operation $!_s : \texttt{ref}(s) \to s$ in Ω, called the *contents function*. A base specification $\mathsf{BaseSPEC}$ is a pair (Σ, Φ) where Φ is a set of conditional existential equations defining the data-types and the state invariant.

Given a model A of $\mathsf{BaseSPEC}$, a *state* $A^{[d_1:=a_1,\dots,d_n:=a_n]}$ is a free extension of A by the set of existential equations $!_{s_i}(d_i) =e= a_i$ where a_i is an element of $A(s_i)$ and d_i is an element of $A(\texttt{ref}(s_i))$ for every $i = 1, \ldots, n$. In the free extension the values $a \in A(s)$ are added as constants to the signature. Note that a state $A^{[d_1:=a_1,\ldots,d_n:=a_n]}$ need not be a persistent extension of A, that is, the unit morphisms $\eta_A : A \to A^{[d_1:=a_1,\ldots,d_n:=a_n]}$ of the adjunction may not be a family of isomorphisms. This happens, for example, if a formula φ in Φ requires that $!_s(d) =e= !_s(d')$ while we have a state $A^{[d:=a,d':=a']}$ where a and a' are two distinct elements in $A(s)$. In this case the state $A^{[d:=a,d':=a']}$ satisfies the equations $!_s(d) = a$ and $!_s(d') = a'$, which implies together with φ that $a = a'$ and thus $\eta_A(a) = \eta_A(a')$.

A state $A^{[d_1:=a_1,\ldots,d_n:=a_n]}$ is called *persistent* if the state is a persistent extension of A. The intuition is that a persistent state $A^{[d_1:=a_1,\ldots,d_n:=a_n]}$ defines the value of the contents function $A(!_{s_i})$ for $d_i \in A(\texttt{ref}(s_i))$ as a_i for every $i = 1, \ldots, n$ while the interpretation of all other symbols, including the interpretation of the sorts, in $A^{[d_1:=a_1,\ldots,d_n:=a_n]}$ is, up to isomorphism, the same as in A.

A *method* is either a create method, a delete method, or a transformation. A create methods increases the domain of the contents functions, a delete method decreases the domain, and a transformation changes the value of the contents functions.

A transformation method m of type $w, p \in S^*$ is written as $m : \texttt{ref}(w); p$ where $\texttt{ref}(w)$ is an abbreviation for $\texttt{ref}(s_1), \ldots, \texttt{ref}(s_n)$ if $w = s_1, \ldots, s_n$. A method definition for method m is a set of rules

$$u =e= v \to m(\Delta_1, \ldots, \Delta_n, t_1, \ldots, t_m) := r_1, \ldots, r_n$$

where u and v are terms of the same sort, Δ_i is either a constant or a variable of sort $\texttt{ref}(s_i)$, t_i a term of sort p_j, and r_i a term of sort s_i for every $i = 1, \ldots, n$, $j = 1, \ldots, m$, and $p = p_1 \ldots p_m$.

For a given model A of $\mathsf{BaseSPEC}$ and values a_i of $A(p_i)$ for all $i = 1, \ldots, m$ a method expression of the form $m(a_1, \ldots, a_m)$ determines a relation between states A^{St} and $A^{\mathsf{St}'}$ as follows: If A^{St} has the form

$$A^{[d_1:=a_1,\ldots,d_n:=a_n,d_{n+1}:=a_{n+1},\ldots,d_{n+m'}:=a_{n+m'}]}$$

and there exists a variable assignment ρ such that the evaluation of t_i with respect to ρ is the same as a_i for $i = 1, \ldots, n$ and the guard $u =e= v$ is true with respect to ρ, then $A^{\mathsf{St}'}$ has the form $A^{[d_1:=b_1,\ldots,d_n:=b_n,d_{n+1}:=a_{n+1},\ldots,d_{n+m'}:=a_{n+m'}]}$ where b_i is the evaluation of r_i with respect to ρ for $i = 1, \ldots, n$.

A create method is defined similar to a transformation method with the restriction that the evaluation of Δ_i with respect to ρ for every $i = 1, \ldots, n$ yields a value of reference sort not in the domain of the corresponding contents function $!_{s_i}$.

A delete method is given by the expression $\texttt{del}(d_1, \ldots, d_n)$ where d_i is an element of the set $A(\texttt{ref}(s_i))$ for every $i = 1, \ldots, n$. It transforms a state A^{St} where d_i is in the domain of the contents function $!_{s_i}$ into a state $A^{\mathsf{St}'}$ where d_i is not in the domain of the contents function $!_{s_i}$ for every $i = 1, \ldots, n$.

COLD-K In COLD-K [22] the states are many sorted algebras with partial functions and predicates. The state transitions are specified using Harel's dynamic logic [35] or given directly as an expression involving other state transitions.

A *class* consists of a the definition of a many sorted signature $\Sigma = (S, \Omega)$ with annotations which sort-, function-, and predicate symbols are *variable*, together with a set of procedures of the form

```
PROC p : s1,..., sn -> MOD f1,...,fm
```

where p is the procedure name, s_i are sorts from S, and f_j are either sort-, function-, or predicate symbols which have been marked variable. The intention is that p describes a state transition with parameters of sort s_i which is only allowed to change the state components given after the `MOD` keyword.

The semantics of a class consists of

- a set of states `St` together with a function assigning to each state σ in `St` a Σ-algebra $A(\sigma)$,
- an initial state σ_0 in `St` and
- a transition relation $\mathcal{T}_p$ for each procedure p.

A transition relation $\mathcal{T}_p$ is a relation consisting of tuples $(\sigma, x_1, \ldots, x_n, \tau)$; σ and τ are states from `St`, and each x_i is an element of the carrier set for sort s_i of algebra $A(\sigma)$. The interpretation of a symbol from Σ may only differ in $A(\sigma)$ and $A(\tau)$ if it occurs after the `MOD` keyword in the definition of the procedure.

The possible interpretations of classes are constrained by a set of dynamic logic formulas. In addition to the usual first-order language, dynamic logic formulas contain assertions of the form

$$\texttt{INIT},\ [p]\varphi,\ <p>\varphi \text{ and } \texttt{PREV}\ \varphi$$

and expression of the form `PREV` t.

`INIT` is true only in the initial state σ_0 with an empty history. `INIT` can be used to constrain the initial state. For example, the formula `INIT` $\Rightarrow \forall i : \mathbb{N}.\ f(i) = 0$ ensures that in the initial state $A(\sigma_0)$ the interpretation of the operation f is the constant function returning 0.

An assertion $[p]\varphi$ is true in a state σ with history $(\sigma_0, \ldots, \sigma_n)$ if φ is true in all possible successor states τ of σ given by the transition relation $\mathcal{T}_p$ with history $(\sigma_1, \ldots, \sigma_n, \sigma)$. An assertion $<p>$ is true in a state σ with history $(\sigma_0, \ldots, \sigma_n)$ if there exists a successor state τ given by the transition relation $\mathcal{T}$ such that φ is true with history $(\sigma_1, \ldots, \sigma_n, \sigma)$. The assertion `PREV` φ is true in a state σ with history $(\sigma_0, \ldots, \sigma_{n-1}, \sigma_n)$ if φ is true in σ_n with history $(\sigma_0, \ldots, \sigma_{n-1})$. The evaluation of an expression `PREV` t in a state σ with history $(\sigma_1, \ldots, \sigma_n)$ is the evaluation of t in $A(\sigma_n)$. Any other formulas φ not containing any of the above assertions is true in a state σ with history $(\sigma_1, \ldots, \sigma_n)$ if φ is true in $A(\sigma)$.

Usually the specification of a procedure follows a certain pattern. First, one defines the conditions *pre* under which a procedure *proc* has a successor state. This is done by a formula of the form $\mathit{pre} \Rightarrow <\!\mathit{proc}\!>\mathit{true}$. Then the effect of *proc* on the state is given by a formula of the form $[\mathit{proc}]\mathit{post}$, which states that after any state transition associated with *proc* is performed the formula *post* holds. The first formula is a liveness property and the second a safety property.

State components not mentioned in the `MOD` clause of a procedure do not change by the state transition associated to that procedure. Note that for variable state components that are functions it does not suffice to give the new result value for an argument value as in the ASM approach by Gurevich; instead, one also has to state that the function applied to the other values yield the same results as in the previous state. That is, to specify a procedure $\mathit{proc}(p, x)$ intended to change the function f at value p to yield $f(p) = x$; but otherwise remaining the same, one has to write $[\mathit{proc}(p, x)]f(p) = x \wedge \forall i.\ (i \neq p \Rightarrow f(i) = \texttt{PREV}\ f(i))$.

Equational Dynamic Logic In Wieringa [59, 60] Equational Dynamic Logic is defined as a specification language for dynamic objects in the context of object oriented databases.

The environment of base types and sets for object identifiers is provided by the initial algebra A of a conditional equational specification $\textsc{Sp}_{\mathsf{Adt}}$. To define the set of possible states of an object system, an enrichment $\textsc{Sp}_{\mathtt{Stat}}$ of $\textsc{Sp}_{\mathsf{Adt}}$ by attribute functions and class invariants is given. No new sorts are allowed. Attribute functions are operations having a sort of object identifiers as their first argument. The set of possible states, or possible worlds, are all extensions of A satisfying $\textsc{Sp}_{\mathtt{Stat}}$.

Transformations are modeled as terms of a special sort `Event` and are called events. The sort `Event` together with operations of sort `Event` are given by a conservative extension $\textsc{Sp}_{\mathsf{Event}}$ of $\textsc{Sp}_{\mathsf{Adt}}$. An algebra E satisfying $\textsc{Sp}_{\mathsf{Event}}$ and having A as $\textsc{Sp}_{\mathsf{Adt}}$-reduct is chosen as a model for all events. Terms over $\textsc{Sp}_{\mathsf{Event}}$ of sort `Event` denote (non-deterministic) functions on the set of possible worlds defined by $\textsc{Sp}_{\mathtt{Stat}}$ and A such that terms evaluating to the same value in E are the same function on the possible worlds.

The interpretation of elements of sort `Event` as functions on possible worlds is constraint by the use of dynamic equational logic. These are formulas requiring certain properties to hold after performing a certain event. For example, the formula

$$\mathtt{age}(p) = a \Rightarrow [\mathtt{inc\_Age}(p)]\mathtt{age}(p) = a + 1$$

restricts the interpretation of the event $\mathtt{inc\_Age}(p)$ in such a way that if $\mathtt{age}(p) = a$ holds in a state C, then $\mathtt{age}(p) = a + 1$ must hold in any of the possible successor states C given by $\mathtt{inc\_Age}(p)$.

A problem with these kind of specifications is that the semantics of the event `inc_Age` only ensures that in the successor state $\mathtt{age}(p) = a + 1$ holds, but it does not say anything about $\mathtt{age}(q)$ for object identifiers q different from p, and similar it is not ensured that $\mathtt{attr}(p) = a \Rightarrow [\mathtt{inc\_Age}(p)]\mathtt{attr}(p) = a$ for an attribute `attr` different from `age`.

2 Some Category Theory

The framework of institutions and the main results of this paper are formulated using category theory. In the following chapter we shall summarize the results from category theory needed in the rest of thesis. First we introduce categories, functors, and natural transformations and give some examples. Then limits and colimits are defined and their construction is shown in the categories Set and Cat. After that, adjunctions and indexed categories are presented. For a more thorough treatment of category theory see the books of Mac Lane [43], Barr and Wells [5], or Adamek, Herrlich and Strecker [2]. The material about indexed categories is mainly from Tarlecki, Burstall and Goguen [58].

2.1 Categories, Functors and Natural Transformations

A *category* C consists of a collection of *objects* and a collection of *arrows* also called *morphisms.* Each arrow f of a category C has associated two objects in C: the *domain*, $\mathtt{dom}(f)$, and the *codomain* of f, $\mathtt{cod}(f)$. The *composition* $g \circ f$ of two arrows f and g of C is defined if the codomain of f is the same object as the domain of g and yields again an arrow in C with domain the domain of f and codomain the codomain of g. Composition is associative:

$$h \circ (g \circ f) = (h \circ g) \circ f,$$

and for each object c there exists an arrow $\mathtt{id}_c$, the *identity* on c, with domain and codomain being c such that for all arrows f with domain c, and g with codomain c:

$$\mathtt{id}_c \circ f = f \text{ and } g \circ \mathtt{id}_c = g.$$

For an arrow f with domain c and codomain d we shall use the function notation and write $f : c \to d$. We shall also use the diagrammatic notation $f; g$ for the composition of two arrows $f : c \to c'$ and $g : c' \to c$ instead of $g \circ f$.

An arrow $f : c \to c'$ is an *isomorphism* if there exists an arrow $f^{-1} : c' \to c$, the *inverse* of f, such that $f; f^{-1} = \mathtt{id}_c$ and $f^{-1}; f = \mathtt{id}_{c'}$.

A discrete category is a category where all arrows are identities. Any set can be regarded as a discrete category; the objects of the category are the elements of the set and the only arrows are the identities $\mathtt{id}_e : e \to e$ for each element e in the set.

A subcategory D of C is a category such that the objects and arrows of D are included in the objects and arrows of C. We write $D \subseteq C$ to denote that D is a subcategory of C. A full subcategory D of C is a subcategory of C such that for any two objects c and c' from D: if $f : c \to c'$ is an arrow in C then f is an arrow in D.

Examples of categories are:

0 the empty category with no objects and arrows,

1 the category with one object and its identity as arrow,

2 the category with two objects 1 and 2 and just one arrow $f : 1 \to 2$ not the identity,

$\downarrow\downarrow$ the category with two objects 1 and 2 and two arrows $f : 1 \to 2$ and $g : 1 \to 2$ not the identity,

V the category with three objects 0, 1 and 2 and two arrows $f : 0 \to 1$ and $g : 0 \to 2$ not the identity,

K_n the discrete category with objects $\{1, \dots, n\}$,

Set the category having sets as objects and functions between sets as morphisms,

Grp the category having as objects groups and as morphisms group homomorphisms.

The opposite category C^{op} of C has as objects the objects of C and as morphisms $f^{op} : c' \to c$ for every morphism $f : c \to c'$ in C. The composition of arrows $f^{op}; g^{op}$ in C^{op} is defined as $(g; f)^{op}$. Note that $(C^{op})^{op} = C$.

Sometimes it is convenient to depict a category C as a graph with nodes the objects of C and edges the arrows of C. In general, the identity arrows and all arrows composed from other arrows are not shown. For example, the category V is depicted as

$$\begin{tikzcd} 1 & & 2 \\ & 0 \arrow[ul, "f"] \arrow[ur, "g"'] & \end{tikzcd}$$

A category C is *freely generated* by a graph if the objects of C are the nodes of the graph and the arrows are the reflexive and transitive closure of the edges of the graph.

A *(covariant) functor* $F : C \to D$ is a pair of functions from the collection of objects of C to the collection of objects of D and from the collection of arrows of C to the collection of arrows of D such that

1. if $f : c \to d$ then $F(f) : F(c) \to F(d)$,
2. $F(f \circ g) = F(f) \circ F(g)$,
3. $F(\mathtt{id}_c) = \mathtt{id}_{F(c)}$.

Functors preserve isomorphisms. Consider a functor $F : C \to D$ and an isomorphism $f : c \to c'$ in c' then $F(f)$ is an isomorphism with inverse $F(f)^{-1} = F(f^{-1})$ because

$$F(f); F(f)^{-1} = F(f); F(f^{-1}) = F(f; f^{-1}) = F(\mathtt{id}_c) = \mathtt{id}_{F(c)}$$

and similar for $F(f)^{-1}; F(f) = \mathtt{id}_{F(c')}$.

The composition of a functor $G : D \to D'$ with a functor $F : C \to D$ is the composition of their functions on objects and their functions on arrows:

$$\begin{array}{rcll}(G \circ F)(c) & = & G(F(c)) & \text{for all } c \in C \text{ and} \\ (G \circ F)(f) & = & G(F(f)) & \text{for all } f : c \to c' \in C.\end{array}$$

A contravariant functor F from C to D maps objects in C to objects in D and arrows $f : c \to c'$ in C to arrows $F(f) : F(c') \to F(c)$ respecting the identities and the composition of arrows, that is, $F(\mathtt{id}_c) = \mathtt{id}_{F(c)}$ and $F(f; g) = F(g); F(f)$. A contravariant functor F from C to D can be either regarded as a covariant functor from C^{op} to D or as a covariant functor from C to D^{op}, whatever is convenient. The composition of a contravariant functor G from D to D' with a contravariant functor F from C to D yields a covariant functor $F; G$ from C to D', and the composition of a contravariant functor $G : D \to D'$ with a covariant functor $F : C \to D$ (and vice versa) yields a contravariant functor $F; G$ from C to D'.

The category Cat has as objects categories and as arrows functors. The identity morphisms are functors Id_C which are the identity on objects and on arrows from C; the composition of two functors is the composition of the functions on objects and on arrows.

The category FCat is the full subcategory of Cat having as objects only finite categories, that is, categories with a finite set of objects and arrows.

In this thesis we use a naive approach to category theory in that we assume that Cat is large enough to hold all categories of interest, ignoring the foundational problem that the category Cat cannot be an object of itself and thus cannot be the category of *all* categories. An overview of these problems and their solutions can be found at pages 21ff of Mac Lane's book [43].

The *opposite* to a functor $F : C \to D$ is the functor $F^{op} : C^{op} \to D^{op}$ with $F^{op}(c) = F(c)$ and $F^{op}(f^{op} : c' \to c) = F(f : c \to c')^{op}$. Note that $(F^{op})^{op} = F$. Thus $\_^{op}$ is a functor from Cat to Cat mapping categories to their opposite categories and functors to their opposite functor.

Given two functors F and G from C to D. A natural transformation $\mu : F \Rightarrow G$ is a family of arrows $\mu_c : F(c) \to G(c)$ in D such that for any arrow $f : c \to c'$ in C the following diagram commutes:

$$\begin{tikzcd} F(c) \arrow[r, "F(f)"] \arrow[d, "\mu_c"'] & F(c') \arrow[d, "\mu_{c'}"] \\ G(c) \arrow[r, "G(f)"'] & G(c') \end{tikzcd}$$

Let F be a functor from D to D' and μ a natural transformation from G to H, where G and H are functors from C to D. The composition of μ with F is the natural transformation from $G; F$ to $H; F$ defined by:

$$(\mu; F)_c = F(\mu_c).$$

Similarly, let F' be a functor from C' to C then the natural transformation $F';\mu$ from $F';G$ to $F';H$ is defined by:

$$(F';\mu)_{c'} = \mu_{F'(c')}$$

Two functors F and G from C to D are *natural isomorphic* if there exists a natural transformation $\mu : F \Rightarrow G$ such that $\mu_c : F(c) \to G(c)$ is an isomorphism for each $c \in C$. Note that the family of morphisms $(\mu_c)^{-1}$ for each $c \in C$ yields again a natural transformation $\mu^{-1} : G \Rightarrow F$.

Given two categories C and D, the objects of the *functor category* D^C are functors $F : C \to D$ and the morphisms are natural transformations.

The identity in D^C for a functor $F : C \to D$ is the natural transformation $\mathtt{id}_F : F \Rightarrow F$ given by $(\mathtt{id}_F)_c = \mathtt{id}_{F(c)}$.

The composition of two natural transformations $\mu : F \Rightarrow G$ and $\nu : G \Rightarrow H$ is defined by $(\mu;\nu)_c = \mu_c;\nu_c$.

2.2 Colimits

An object c of a category C is an *initial object of* C if there exists for all objects d in C a unique arrow $h : c \to d$ in C. A category may have no, one or several initial objects, which, in the latter case, are all isomorphic.

Note that if c is an initial object then the identity is the unique arrow from c to c.

The initial object of Set is the empty set and the initial object of Cat is the category $\mathbf{0}$ with no objects and no arrows.

For categories C and J the *diagonal functor* $\Delta : C \to C^J$ maps an object c of C to the constant functor $\Delta c : J \to C$ given by

$$\Delta c(i) = c \text{ and } \Delta c(f : i \to j) = \mathtt{id}_c$$

and an arrow $g : c \to c'$ to the constant natural transformation $\Delta g : \Delta c \Rightarrow \Delta c'$ given by $(\Delta g)_i = g$.

Definition 2.1 (Colimit) *Given a functor $F : J \to C$. The* colimit *of F consists of a pair $(\iota^F, \coprod_J F)$ where $\coprod_J F$ is an object of C and ι^F is a natural transformation $\iota^F : F \Rightarrow \Delta \coprod_J F$ such that the following universal property is satisfied: For any object c and natural transformation $\mu : F \Rightarrow \Delta c$ from C there exists a unique arrow $h_\mu : \coprod_J F \to c$ such that the following diagram commutes for each i in J:*

$$\begin{tikzcd}
\coprod_J F \arrow[r, "h_\mu"] & c \\
F(i) \arrow[u, "\iota^F_i"] \arrow[ur, "\mu_i"'] &
\end{tikzcd}$$

We call the ι_i^F *co-cone morphisms* from $F(i)$ into $\coprod_J F$ and J the *shape* of F. Usually, we will omit J in $\coprod_J F$ and write $\coprod F$ instead.

Note that the colimit object is only unique up to isomorphism.

The initial object of a category C is the colimit object $\coprod F$ of the unique functor $F : \mathbf{0} \to C$. The co-cone morphisms are given by the unique natural transformation from F to $\Delta(\coprod F)$. Since there exists for all objects c from C a unique natural transformation μ from F to Δc, the universal property of $\coprod F$ guarantees the existence of a unique morphism $h_\mu : \coprod F \to c$.

Let F and G be functors from J to C and μ a natural transformation from F to G. We write $\coprod \mu$ for the unique morphism from $\coprod F$ to $\coprod G$ given by the universal property of $\coprod F$ and the natural transformation ν from F to $\Delta(\coprod G)$ defined by $\nu_i = \mu_i; \iota_i^G$.

$$\begin{tikzcd}
\coprod F \arrow[r, "\coprod\mu"] & \coprod G \\
F(i) \arrow[u, "\iota_i^F"] \arrow[r, "\mu_i"'] \arrow[ur, dashed, "\nu_i"'] & G(i) \arrow[u, "\iota_i^G"']
\end{tikzcd}$$

Note that if G is Δc then $\coprod \nu$ is the unique morphism given by the universal property of $\coprod F$ and ν because $\coprod(\Delta c) = c$:

$$\begin{tikzcd}
\coprod F \arrow[r, "\coprod\nu"] & c \\
F(i) \arrow[u, "\iota_i^F"] \arrow[r, "\nu_i"'] \arrow[ur, dashed, "\nu_i"'] & c \arrow[u, equal]
\end{tikzcd}$$

Definition 2.2 (Cocomplete) *A category C is (finitely) cocomplete if any (finite) functor $F : J \to C$ has a colimit for (finite) categories J. A category is finite its collection of objects and arrows are finite.*

Fact 2.3 *The categories* Set *and* Cat *are cocomplete.*

It is well known that the categories Set and Cat are cocomplete. In Set a colimit of a functor F from J to Set is the set $\{(i, a) \mid a \in F(i)\}/\equiv$. The relation $\equiv$ is the smallest equivalence relation such that $(i, a) \equiv (j, b)$ if there exists $f : i \to j$ in J with $F(f)(a) = b$.

The co-cone morphisms ι_i^F from $F(i)$ to the colimit of F are defined by $\iota_i^F(a) = [(i, a)]_\equiv$.

The construction of the colimit category of $F : J \to$ Cat is similar to the construction of the colimit in Set. The objects of $\coprod F$ are the equivalence classes of the relation $\equiv$. The relation $\equiv$ is the smallest relation such that $(i, c_i) \equiv (j, c_j)$ if there exists $f : i \to j$ in J with $F(f)(c_i) = c_j$ for objects $c_i \in F(i)$, $c_j \in F(j)$, and $i, j \in J$.

Similarly, the collection of arrows of $\coprod F$ are the equivalence classes of the smallest equivalence relation satisfying $(i, g_i : c_i \to c'_i)\ S\ (j, g_j : c_j \to c'_j)$ if there exists $f : i \to j$ in J with $F(f)(g_i) = g_j$.

Note that the category FCat of finite categories is only finitely cocomplete since, for example, the colimit of the functor F from a discrete category J to Cat mapping each $i \in J$ to the category $\mathbf{1}$ is the discrete category with objects pairs $(i, 1)$ for each $i \in J$. This category is infinite if J is infinite.

Fact 2.4 *The functor category D^C is (finitely) cocomplete if D is.*

Proof. Let F be a functor from J to D^C. Define the family of functors $F_c : J \to D$ by $F_c(i) = F(i)(c)$ and $F_c(f) = F(f)(c)$ for objects i and morphisms $f : i \to j$ in J. The colimit $\coprod F$ is a functor from C to D given by $(\coprod F)(c) = \coprod F_c$ and $(\coprod F)(g : c \to c') = \coprod \mu$ where μ is the natural transformation from F_c to $\Delta(\coprod F_{c'})$ defined by $\mu_i = F(i)(g)$ for each $i \in J$. □

In particular, the initial object in D^C is the functor mapping every object in C to the initial object in D and every morphism to the identity on the initial object.

In case where F is a functor from a discrete category $\{1, 2\}$ to C, the colimit $\coprod F$ is called the coproduct $F(1) \uplus F(2)$ of $F(1)$ and $F(2)$. The disjoint union of two sets S_1 and S_2,

$$S_1 \uplus S_2 = \{(1, s_1) \mid s_1 \in S_1\} \cup \{(2, s_2) \mid s_2 \in S_2\},$$

is a coproduct of S_1 and S_2 with the co-cone morphisms $\iota_i : S_i \to S_1 \uplus S_2$ being defined by $\iota_i(s) = (i, s)$ for all s in S_i and $i \in \{1, 2\}$.

The coproduct of two categories C_1 and C_2 in Cat is the disjoint union of their collection of objects and the disjoint union of their collection of arrows.

The notion of coproducts can be extended to functors of the form $F : \mathrm{K}_n \to C$ for arbitrary n. We may write $F(1) \uplus \cdots \uplus F(n)$ instead of $\coprod_{\mathrm{K}_n} F$.

Definition 2.5 (Coequalizer) *A coequalizer is the colimit of a functor F from $\downdownarrows$ to C.*

This means that for any object c and arrow $h : F(2) \to c$ in C with $F(f); h = F(g); h$ there exists a unique arrow h' from $\coprod F$ to c such that $\iota_2; h' = h$:

$$\begin{tikzcd}
F(1) \arrow[r, shift left, "F(f)"] \arrow[r, shift right, "F(g)"'] & F(2) \arrow[r, "\iota_2"] \arrow[dr, "h"'] & \coprod F \arrow[d, dashed, "\exists_1 h'"] \\
& & c
\end{tikzcd}$$

The coequalizer of two functions $f_1, f_2 : S_1 \to S_2$ in Set is given by the quotient $S_2/_{\equiv}$ where $\equiv$ is the smallest equivalence relation generated by the relation

$$\{(s, s') \mid \exists s_1 \in S_1 \ \ f_1(s_1) = s \text{ and } f_2(s_1) = s'\}.$$

The co-cone morphism ι_2 maps $s_2 \in S_2$ to $[s_2]_\equiv$ and ι_1 is $f_1; \iota_2$, which is the same as $f_2; \iota_2$.

The collection of objects of the coequalizer in Cat of two functors $F_1, F_2 : C \to D$ is the coequalizer of the collection of objects of C and D, and the collection of arrows of the coequalizer is the coequalizer of the collection of arrows of C and D.

Definition 2.6 (Pushout) *A pushout is the colimit of a functor F from* V *to C.*

For any object c in C and pair of arrows $\mu_1 : F(1) \to c$ and $\mu_2 : F(2) \to c$ such that $F(g); \mu_1 = F(f); \mu_2$ there exists a unique arrow $h_\mu : \coprod F \to c$ such that $\iota_1; h_\mu = \mu_1$ and $\iota_2; h_\mu = \mu_2$:

$$\begin{tikzcd}
& c & \\
& \coprod F \arrow[u, dotted, "h_\mu"] & \\
F(1) \arrow[uur, "\mu_1"] \arrow[ur, dashed, "\iota_1"'] & & F(2) \arrow[uul, "\mu_2"'] \arrow[ul, dashed, "\iota_2"] \\
& F(0) \arrow[ul, "F(f)"] \arrow[ur, "F(g)"'] \arrow[uu, dashed, "\iota_0"] &
\end{tikzcd}$$

For the pushout of a functor $F : \mathrm{V} \to C$ we may write $F(1) +_{(F(f),F(g))} F(2)$. We may write $F(1) +_{F(0)} F(2)$ if $F(f)$ and $F(g)$ are "canonic", for example, in Set this means that $F(f)$ and $F(g)$ are the inclusions of $F(0)$ into $F(1)$ and $F(g)$, respectively.

Pushouts can be constructed by first taking the coproduct $F(1) \uplus F(2)$ and then the coequalizer of $F(f); \iota_1$ and $F(g); \iota_2$

$$\begin{tikzcd}
& \coprod F & \\
& F(1) \uplus F(2) \arrow[u, "\iota"] & \\
F(1) \arrow[ur, "\iota_1"'] \arrow[uur, dashed, "\iota_1^F"] & & F(2) \arrow[ul, "\iota_2"] \arrow[uul, dashed, "\iota_2^F"'] \\
& F(0) \arrow[ul, "F(f)"] \arrow[ur, "F(g)"'] &
\end{tikzcd}$$

In Set the pushout is $(F(1) \uplus F(2))/_\equiv$ with $\equiv$ being the smallest equivalence relation generated by

$$\{((1, s_1), (2, s_2)) \mid \exists s_0 \in F(0)\ F(f)(s_0) = s_1 \text{ and } F(g)(s_0) = s_2\}$$

The co-cone morphisms are $\iota_i(s_i) = [(i, s_i)]_\equiv$ for $i \in \{1, 2\}$ and

$$\iota_0(s_0) = [(1, F(f)(s_0))]_\equiv = [(2, F(g(s_1)))]_\equiv.$$

Fact 2.7 *A category is cocomplete if it has initial objects and pushouts or initial objects, coequalizer and all coproducts. A category C has all coproducts if every functor from a discrete category to C has a colimit.*

This fact is useful since it allows to show that a category is cocomplete by exhibiting initial objects and the construction of pushouts.

A proof of Fact 2.7 can be found, for example, in the book by Mac Lane [43]. Above we have seen how to construct pushouts in terms of coproducts and coequalizers. Conversely, coproducts $F(1) \uplus F(2)$ can be expressed by pushouts $F(1) +_{\perp} F(2)$ where $\perp$ is the initial object of the category.

Definition 2.8 (Preservation of Colimits) *The functor $G : C \to D$ preserves colimits if for any functor $F : J \to C$ for which the colimit $(\iota^F, \coprod F)$ exists, $(G(\iota^F), G(\coprod F))$ is a colimit of the functor $F; G$.*

The next fact is a consequence of Fact 2.7.

Fact 2.9 *Let C be cocomplete then a functor F from C to D preserves colimits if it preserves initial objects and pushouts, or if it preserves initial objects, all coproducts and coequalizers.*

It is important that C is cocomplete to ensure that the colimit of a functor $F : J \to C$ can be decomposed in initial objects and pushouts, or initial objects, coproducts and coequalizer. If C is not cocomplete then the corresponding diagrams may not have colimits and thus one cannot use the preservation of initial objects etc. to construct the colimit in D.

Definition 2.10 (Creation of Colimits) *A functor $G : C \to D$ creates colimits if for any functor $F : J \to C$ for which the colimit $(\iota^{F;G}, \coprod(F; G))$ of $F; G$ exists there exists a colimit $(\iota^F, \coprod F)$ of F such that*

$$G(\coprod F) = \coprod(F; G) \text{ and } \iota^F; G = \iota^{F;G}.$$

It is easy to see that if $F : C \to C'$ and $G : C' \to D$ both preserve colimits, so does the composition $F; G$, and similar for creation of colimits.

Fact 2.11 *If a functor $G : C \to D$ creates colimits and D is cocomplete then C is cocomplete.*

Proof. Consider a functor $F : J \to C$. Since D is cocomplete, the colimit of $F; G$ always exists and therefore the colimit of F exists since G creates colimits. □

2.3 Limits

A concept dual to that of colimits is the concept of limits. If we reverse the direction of all arrows in the definition of colimits, we get the definition of limits.

Definition 2.12 (Limit) *Given a functor $F : J \to C$. The limit of F consists of a pair $(\pi^F, \prod_J F)$ where $\prod_J F$ is an object of C and π^F a natural transformation $\pi^F : \Delta(\prod_J F) \Rightarrow F$ such that for any object c and natural transformation $\mu : \Delta c \Rightarrow F$ from C there exists a unique arrow $h_\mu : c \to \prod_J F$ such that $h_\mu; \pi_i^F = \mu_i$ for all $i \in J$, that is, the following diagram commutes for all $i \in J$:*

$$\begin{tikzcd}
\prod_J F \arrow[d, "\pi_i^F"'] & c \arrow[l, "h_\mu"'] \arrow[ld, "\mu_i"] \\
F(i) &
\end{tikzcd}$$

We call the morphisms π_i^F the *cone morphisms* of the limit of F.

As with colimits, limits are only unique up to isomorphism.

A *terminal object* c of a category C is an object of C such that for all objects d in C there exists a unique arrow $h : d \to c$ in C. The concept of terminal objects is dual to that of initial objects. As with initial objects, terminal objects are only unique up to isomorphism. The terminal object is the limit of the unique functor $F : \mathbf{0} \to C$.

Any set with one element is a terminal object in Set and any category with one object and one arrow, the identity, is a terminal object in Cat. In particular the category $\mathbf{1}$ is a terminal object in Cat.

We can now formally establish the duality between limits and colimits. Consider a functor $F : J \to C$ with colimit $(\iota^F, \coprod F)$ then $((\iota^F)^{op}, \coprod F)$ is a limit of the functor $F^{op} : J^{op} \to C^{op}$. Similarly, if $(\pi^F, \prod F)$ is a limit of F then $((\pi^F)^{op}, \prod F)$ is a colimit of F^{op}.

Given two functors F and G from C to D such that both F and G have a limit. Let μ be a natural transformation from F to G then $\prod \mu$ denotes the unique morphism from $\prod F$ to $\prod G$ given by the limit property of $\prod G$ with respect to the natural transformation $\nu = \pi^F; \mu$.

$$\begin{tikzcd}
\prod F \arrow[r, "\Pi \mu"] \arrow[d, "\pi_i^F"'] \arrow[rd, dashed, "\nu_i"'] & \prod G \arrow[d, "\pi_i^G"] \\
F(i) \arrow[r, "\mu_i"'] & G(i)
\end{tikzcd}$$

The concept dual to that of coproducts is that of products. In case where F is a functor from the discrete category $\{1, 2\}$ to C, the limit of F is called the product $F(1) \times F(2)$ of $F(1)$ and $F(2)$. The cartesian product of two sets S_1 and S_2 is a product of S_1 and S_2 in Set:

$$S_1 \times S_2 = \{(s_1, s_2) \mid s_1 \in S_1 \text{ and } s_2 \in S_2\}$$

The cone morphisms $\pi_i : S_1 \times S_2 \to S_i$ are defined by $\pi_i((s_1, s_2)) = s_i$ for all s_i in S_i and $i \in \{1, 2\}$.

The product $C_1 \times C_2$ of two categories C_1 and C_2 is the cartesian product of their collection of objects and arrows. Given a pair of arrows $f : c_1 \to c_1'$ in C_1 and $g : c_2 \to c_2'$ in C_2 then $(f, g) : (c_1, c_2) \to (c_1', c_2')$ is an arrow in $C_1 \times C_2$.

As with coproducts, we write $F(1) \times \ldots \times F(n)$ for the limit of a functor $F : \mathrm{K}_n \to C$.

Definition 2.13 (Equalizer) *An equalizer is the limit of a functor F from $\downarrow\downarrow^{op}$ to C.*

This means that for any object c and arrow $h : c \to F(2)$ in C, with $h; F(f) = h; F(g)$, there exists a unique arrow h' from c to $\prod F$ such that $h'; \pi_2 = h$.

$$\begin{tikzcd}
F(1) \arrow[r, <-, shift left, "F(g)"] \arrow[r, <-, shift right, "F(f)"'] & F(2) & \prod F \arrow[l, "\pi_2"'] \\
 & & c \arrow[ul, "h"] \arrow[u, dashed, "\exists_1 h'"']
\end{tikzcd}$$

The equalizer of two functions $f_1, f_2 : S_1 \to S_2$ in Set is the set

$$\{s \in S_2 \mid f_1(s) = f_2(s)\}$$

with $\pi_2(s) = s$ as the cone morphism.

The equalizer of two functors $F_1, F_2 : C_1 \to C_2$ in Cat is the subcategory of C_2 with objects c from C_1 such that $F_1(c) = F_2(c)$ and morphisms $f : c \to c'$ such that $F_1(f) = F_2(f)$.

Definition 2.14 (Pullback) *A pullback is the limit of a functor $F : \mathrm{V}^{op} \to C$.*

If $\mu_1 : c \to F(1)$ and $\mu_2 : c \to F(2)$ are two arrows in C with $\mu_1; F(f) = \mu_2; F(g)$ then there exists a unique $h : c \to \prod F$ with $\mu_1 = h; \pi_1$ and $\mu_2 = h; \pi_2$.

$$\begin{tikzcd}
 & c \arrow[d, dotted, "h"] \arrow[ddl, "\mu_1"'] \arrow[ddr, "\mu_2"] & \\
 & \prod F \arrow[dl, dashed, "\pi_1"] \arrow[dr, dashed, "\pi_2"'] & \\
F(1) \arrow[dr, "F(f)"'] & & F(2) \arrow[dl, "F(g)"] \\
 & F(0) &
\end{tikzcd}$$

For the pullback of a functor $F : \mathrm{V}^{op} \to C$ we may write $F(1) \times_{(F(f),F(g))} F(2)$ or, if $F(f)$ and $F(g)$ are "canonic", for example, inclusions in Set, we may write $F(1) \times_{F(0)} F(2)$.

In Set the pullback is the set

$$\{(s_1, s_2) \in F(1) \times F(2) \mid F(f)(s_1) = F(g)(s_2)\}$$

with cone morphisms $\pi_i((s_1, s_2)) = s_i$ for $i \in \{1, 2\}$ and

$$\pi_0((s_1, s_2)) = F(f(s_1)) = F(g(s_2))$$

The pullback in Cat is the subcategory of $F(1) \times F(2)$ with objects

$$\{(s_1, s_2) \in F(1) \times F(2) \mid F(f)(s_1) = F(g)(s_2)\}$$

and morphisms $(f, g) : (c_1, c_2) \to (c_1', c_2')$ such that $F(f)(f) = F(g)(g)$.

Definition 2.15 (Complete) *A category C is (finitely) complete if every (finite) functor $F : J \to C$ has a limit.*

Fact 2.16 *A category is complete if it has terminal objects and pullbacks or terminal objects, equalizers and all products.*

A category C has all products if the limit of every functor from a discrete category to C exists.

Fact 2.17 *The categories* Set *and* Cat *are complete and the category* FCat *is finitely complete.*

We have already shown that Set and Cat have terminal objects and pullbacks. However, to give a feeling of limits in Set, we give one possible construction for them.

The limit of a functor $F : J \to \text{Set}$ is the set

$$\{(m_1, \dots, m_i, \dots) \mid m_i \in F(i),\ i \in J,\ F(f)(m_i) = m_j,\ f : i \to j \in J\}$$

with cone morphisms π_i^F, mapping $(m_1, \dots, m_i, \dots)$ to m_i.

Fact 2.18 *The functor category C^D is complete if C is complete.*

Proof. For a functor $F : J \to C^D$, the limit is the following functor $\prod F : D \to C$ defined by

$$(\prod F)(d) = \prod F_d,$$

where F_d is a functor from J to C defined as $F_d(i) = F(i)(d)$ and $F_d(f) = F(f)(d)$ for objects i and morphism $f : i \to j$ in J.

Let μ be a natural transformation from ΔG to F. Define the natural transformation $h_\mu : G \Rightarrow \prod F$ with the help of the unique morphisms $h_{\mu_d} : G(d) \to \prod F_d$ given by the universal property of $\prod F_d$ and $\mu_d : \Delta(G(d)) \Rightarrow \prod F_d$, defined by $(\mu_d)_i = (\mu_i)_d$. □

Theorem 2.19 *Given two natural isomorphic functors G and H from J to C then G has a (co)limit if and only if H has a (co)limit.*

Proof. Let $\mu : G \Rightarrow H$ be the natural isomorphism from G to H with inverse $\mu^{-1} : H \Rightarrow G$. Let c be a colimit of G and $\iota^G : G \Rightarrow \Delta c$ the co-cone morphisms. Then c is a colimit of H with co-cone morphisms $\iota^H = \mu^{-1}; \iota^G$.

Given a natural transformation $\nu : H \Rightarrow \Delta c'$ then $\bar{\nu} = \mu; \nu$ is a natural transformation from G to $\Delta c'$ and since (ι^G, c) is a colimit of G, there exists a unique homomorphism $h : c \to c'$ with $\iota^G_i; h = \bar{\nu}_i$ for all $i \in J$. We have to show that also $\iota^H_i; h = \nu_i$ for all $i \in J$ holds. Let i be an object of J then

$$\begin{aligned}
\iota^H_i; h &= \mu^{-1}_i; \iota^G; h && \mid \iota^H_i = \mu^{-1}_i; \iota^G_i \\
&= \mu^{-1}_i; \bar{\nu}_i && \mid \iota^G_i; h = \bar{\nu}_i \\
&= \mu^{-1}_i; \mu_i; \nu_i && \mid \bar{\nu}_i = \mu_i; \nu_i \\
&= \nu_i.
\end{aligned}$$

The proof for limits is analog. □

Theorem 2.20 *Let G and H be two natural isomorphic functors from C to D then G preserves (co)limits if and only if H preserves (co)limits.*

Proof. Assume that G preserves colimits. Given a functor $F : J \to C$ with colimit (ι^F, c) then $(G; \iota^F, G(c))$ is a colimit of $F; G$. Since G and H are natural isomorphic with isomorphism $\mu : G \Rightarrow H$, so are $F; G$ and $F; H$ with isomorphism $F; \mu$ and thus $G(c)$ is a colimit of $F; H$. Because $G(c)$ is isomorphic to $H(c)$, $H(c)$ is also a colimit of $F; H$ with co-cone morphisms $\iota^{F;G}; \mu_c$.

The proof for limits is analog. □

2.4 Adjoints

Given two categories C and D. Two functors $F : C \to D$ and $U : D \to C$ form an *adjunction* if there is a bijection φ between the morphisms from c to $U(d)$ and the morphisms from $F(c)$ to d for each $c \in C$ and $d \in D$:

$$\frac{c \xrightarrow{f} U(d)}{F(c) \xrightarrow{\varphi(f)} d}.$$

An example of an adjunction in the domain of algebraic specifications is the free term functor, which, given a signature $\Sigma = (S, \Omega)$, constructs the free term algebra $\mathsf{T}_\Sigma(X)$ of an S-indexed family of sets X, together with the forgetful functor that maps a Σ-algebra to its family of carrier sets (cf. Section 3.1).

Let C be the category of S-indexed families of sets $\textsc{Set}^S$ and D the category $\mathsf{Alg}(\Sigma)$ of Σ-algebras. The functor U takes a Σ-algebra A and yields the S-indexed family of carrier sets of A. The functor F takes an S-indexed family of carrier sets $X : S \to \textsc{Set}$ and returns the free term algebra $\mathsf{T}_\Sigma(X)$ over X. The free term algebra has as carrier sets the smallest family of sets such that

- $X(s) \subseteq \mathsf{T}_\Sigma(X)(s)$ for $s \in S$ and
- $\omega(t_1, \ldots, t_n) \in \mathsf{T}_\Sigma(X)(s)$ for all $\omega \in \Omega(s_1, \ldots, s_n, s)$, $t_i \in \mathsf{T}_\Sigma(X)(s_i)$, s in S and s_i in S for all $1 \leq i \leq n$.

For operations ω in $\Omega(s_1, .., s_n, s)$ we define

$$\mathsf{T}_\Sigma(X)(\omega)(t_1, \ldots, t_n) = \omega(t_1, \ldots, t_n)$$

for each $t_i \in \mathsf{T}_\Sigma(s_i)$ and $1 \leq i \leq n$.

Any S-indexed family of functions h between X and the carrier sets of a Σ-algebra B extends uniquely to a homomorphism $\varphi(h)$ from $\mathsf{T}_\Sigma(X)$ to B by

- $\varphi(h)_s(x) = h(x)$ for $x \in X(s)$ and
- $\varphi(h)_s(\omega(t_1, \ldots, t_n)) = B(\omega)(\varphi(h)_{s_1}(t_1), \ldots, \varphi(h)_{s_n}(t_n))$
  for $\omega \in \Omega(s_1, \ldots, s_n, s)$, $t_i \in \mathsf{T}_\Sigma(X)_{s_i}$ and $1 \leq i \leq n$.

On the other hand, if g is a homomorphism between $\mathsf{T}_\Sigma(X)$ and a Σ-algebra B then a family of functions $\varphi^{-1}(g)$ is given by $\varphi^{-1}(g)_s(x) = g_s(x)$ for all $s \in S$ and $x \in X(s)$.

Thus we have a bijection between natural transformations $h : X \Rightarrow U(B)$, which are the morphisms in $\textsc{Set}^S$ and homomorphisms $\varphi(h) : \mathsf{T}_\Sigma(X) \to B$:

$$\frac{X \xRightarrow{h} U(B)}{\mathsf{T}_\Sigma(X) \xrightarrow{\varphi(h)} B}.$$

Definition 2.21 (Adjunction) *An adjunction from a category C to a category D is a pair of functors $F : C \to D$ and $U : D \to C$ such that there exists a bijection φ between the collection of arrows from c to $U(d)$ in C and the collection of arrows from $F(c)$ to d in D for each object $c \in C$ and $d \in D$, which is natural in c and d. The naturality conditions require that*

$$\begin{tikzcd} c \arrow[r, "g_1"] \arrow[d, "f_1"'] & U(d) \arrow[d, "U(f_2)"] \\ c' \arrow[r, "g_2"'] & U(d') \end{tikzcd} \quad \textit{commutes if and only if} \quad \begin{tikzcd} F(c) \arrow[r, "\varphi(g_1)"] \arrow[d, "F(f_1)"'] & d \arrow[d, "f_2"] \\ F(c') \arrow[r, "\varphi(g_2)"'] & d' \end{tikzcd}$$

commutes for morphisms g_1, g_2 and f_1 in C and f_2 in D.

An alternative definition of an adjunction is:

Definition 2.22 *Let U be a functor from D to C then the functor $F : C \to D$ is left adjoint for U, if there exists a natural transformation $\eta : \mathsf{Id}_C \Rightarrow F;U$ such that for every object $d \in D$ and morphism $f : c \to U(d)$ in C there exists a unique morphism $\tilde{f} : F(c) \to d$ such that the following diagram commutes:*

$$\begin{tikzcd} c \arrow[r, "\eta_c"] \arrow[dr, "f"'] & U(F(c)) \arrow[d, "U(\tilde{f})"] \\ & U(d) \end{tikzcd}$$

Similarly, let F be a functor from C to D then the functor $U : D \to C$ is right adjoint for F if there exists a natural transformation $\epsilon : U;F \Rightarrow \mathsf{Id}_D$ such that for every object $c \in C$ and arrow $g : F(c) \to d$ in D there exists a unique morphism $\tilde{g} : c \to U(d)$ such that the following diagram commutes:

$$\begin{tikzcd} d & F(U(d)) \arrow[l, "\epsilon_d"'] \\ & F(c) \arrow[ul, "g"] \arrow[u, "F(\tilde{g})"'] \end{tikzcd}$$

Given an adjunction (F, U, φ) from C to D then F is a left adjoint for U and U is a right adjoint for F. We call the natural transformations $\eta : \mathsf{Id}_C \Rightarrow F;U$ the unit and $\epsilon : U;F \Rightarrow \mathsf{Id}_D$ the co-unit of the adjunction.

Theorem 2.23 *Given functors $F : C \to D$ and $U : D \to C$. The following definitions are equivalent:*

1. *(F, U, φ) is an adjunction from C to D,*
2. *F is left adjoint for U and*
3. *U is right adjoint for F.*

Proof. Let (F, U, φ) be an adjunction from C to D. Define $\eta : \mathsf{Id}_C \Rightarrow F;U$ by $\eta_c = \varphi^{-1}(\mathtt{id}_{F(c)})$ for every object $c \in C$, and for a given morphism $f : c \to U(d)$ let $\tilde{f} = \varphi(f)$. The naturality of φ ensures that η is a natural transformation, that $\eta_c;U(\tilde{f}) = f$ and that $\tilde{f}$ is unique with that property. This makes F a left adjoint for U.

Similarly, define $\epsilon : U;F \Rightarrow \mathsf{Id}_D$ by $\epsilon_d = \varphi(\mathtt{id}_{U(d)})$ for every object d in D, and given a morphism $g : F(c) \to d$ in D let $\tilde{g} = \varphi^{-1}(g)$. This makes U a right adjoint for F.

On the other hand, if F is a left adjoint for U, define $\varphi(f) = \tilde{f}$ for every morphism $f : c \to U(d)$. Then naturality of φ is given by the uniqueness property of $\tilde{f}$ and the fact that $\tilde{\eta}_c : F(c) \to F(c)$ is the identity on $F(c)$. Similarly, if U is right adjoint for F then $\varphi^{-1}(g : F(c) \to d) = \tilde{g}$. □

Fact 2.24 *Let (F, U, φ) be an adjunction from C to D and (F', U', φ') an adjunction from D to D'. Then $(F; F', U'; U, \varphi; \varphi')$ is an adjunction from C to D'.*

This fact can be rephrased as follows: the composition $F; F'$ of a left adjoint F for U with a left adjoint F' for U' yields a left adjoint for $U'; U$ and similar for right adjoints.

Theorem 2.25 *A functor F preserves colimits if F has a right adjoint U.*

Proof. Let (F, U, φ) be an adjunction from C to D and $G : J \to C$ a functor that has a colimit $(\iota^G, \coprod G)$. We want to show that $F(\coprod G)$ is a colimit of $G; F$ with co-cone morphisms $\iota^G; F$. Assume an object d in D and a natural transformation $\nu : G; F \Rightarrow \Delta d$ then we define a natural transformation $\nu' : G \Rightarrow \Delta U(d)$ by $\nu'_i = \varphi^{-1}(\nu_i)$ for every $i \in J$. Because G has a colimit, there exists a unique morphism h from $\coprod G$ to $U(d)$ making for each $i \in J$ the following diagram commute:

$$\begin{tikzcd}
\coprod G \arrow[r, "h"] & U(d) \\
G(i) \arrow[u, "\iota_i^G"] \arrow[ur, "\nu'_i=\varphi^{-1}(\nu_i)"'] &
\end{tikzcd}$$

Then $\varphi(h)$ is a morphism from $F(\coprod G)$ to d and for each $i \in J$ the following diagram commutes because of the naturality of φ:

$$\begin{tikzcd}
F(\coprod G) \arrow[r, "\varphi(h)"] & U(d) \\
F(G(i)) \arrow[u, "F(\iota_i^G)"] \arrow[ur, "\nu_i"'] &
\end{tikzcd}$$

□

Note that if F has a right adjoint U then F is left adjoint to U. Thus the last theorem is equivalent to saying that left adjoint functors preserve colimits.

The dual of Theorem 2.25 is the following theorem:

Theorem 2.26 *A functor U preserves limits if it has a left adjoint F.*

2.5 Indexed Categories

In [58] Tarlecki, Burstall and Goguen develop indexed categories as a tool for theoretical computer science. An indexed category models in a uniform way a family of categories indexed by the objects of an index category. The morphisms in the index category generate additional relations between the component categories.

An example of an indexed category is the family of the categories of Σ-algebras $\mathsf{Alg}(\Sigma)$ for each many-sorted signature Σ. The index category is the category of many-sorted signatures $\textsc{Sig}$ and the component categories are the categories of Σ-algebras $\mathsf{Alg}(\Sigma)$. For each signature morphism $\sigma : \Sigma \to \Sigma'$ in $\textsc{Sig}$ there exists a functor $\mathsf{Alg}(\sigma) : \mathsf{Alg}(\Sigma') \to \mathsf{Alg}(\Sigma)$, the σ-reduct (cf. Section 3.1).

Each indexed category has associated a single category given by the disjoint union of the objects of the component categories plus some additional morphisms. This category is the flat category associated with an indexed category. Together with a projection functor from the flat category to the index category, which assigns to each object in the flat category the index of the category where it originates from, this yields the "fibered category" presented by the indexed category (cf. Grothendieck [31]).

An indexed category C indexed by a category E is a family of categories C_e for each $e \in E$ and a family of functors C_f from $C_{e'}$ to C_e for each morphism $f : e \to e'$ in E such that $C_{\mathtt{id}_e}$ is the identity functor Id_{C_e} and if $f : e \to e'$ and $g : e' \to e''$ are morphisms in E then $C_{f;g} = C_g; C_f$.

Thus an indexed category C determines a contravariant functor C from E to $\textsc{Cat}$ defined by $C(e) = C_e$ and $C(f) = C_f$ for each object e and morphism f in E.

Flattening an indexed category C yields the category $\mathsf{Flat}(C)$, whose collection of objects is the disjoint union of the collection of objects of C_e for every $e \in E$. Thus the objects of $\mathsf{Flat}(C)$ are pairs (e, c) where e is an object of E and c is an object of C_e. An arrow from (e, c) to (e', c') in $\mathsf{Flat}(C)$ is a pair (f, g) where $f : e \to e'$ is a morphism in E and $g : c \to C_g(c')$ is a morphism in C_e. The identity of an object (e, c) in $\mathsf{Flat}(C)$ is the pair $(\mathtt{id}_e, \mathtt{id}_c)$. The composition of two arrows (f_1, g_1) from (e, c) to (e', c') and (f_2, g_2) from (e', c') to (e'', c'') is given by $(f_1; f_2, g_1; C_{f_1}(g_2))$.

$$\begin{tikzcd}
c \arrow[r, "g_1"] & C_{f_1}(c') \arrow[r, "C_{f_1}(g_2)"] & C_{f_1}(C_{f_2}(c'')) \\
 & c' \arrow[u, mapsto] \arrow[r, "g_2"'] & C_{f_2}(c'') \arrow[u, mapsto]
\end{tikzcd}$$

Flattening the indexed category $\mathsf{Alg} : \textsc{Sign}^{op} \to \textsc{Cat}$ yields the category $\mathsf{Flat}(\mathsf{Alg})$. Objects in $\mathsf{Flat}(\mathsf{Alg})$ are pairs (Σ, A) where Σ is a signature and A is a Σ-algebra. Morphisms from (Σ, A) to (Σ', A') in $\mathsf{Flat}(\mathsf{Alg})$ are pairs (σ, h) where σ is a signature morphism from Σ to Σ' and h is a homomorphism from A to $A|_\sigma$.

Another example of flattening an indexed category is the category of many-sorted signatures $\textsc{Sig}$ (cf. Section 3.1). The indexed category MS is a contravariant functor $\mathsf{MS} : \textsc{Set}^{op} \to \textsc{Cat}$. An object Ω in MS_S is an S^+ indexed family of operation symbols, that is, an element of $\textsc{Set}^{S^+}$. A function $\sigma_s : S \to S'$ induces a function $\overline{\sigma}_s$ from S^+ to S'^+ by $\overline{\sigma}_s(s_1, \ldots, s_n) = \sigma_s(s_1), \ldots, \sigma_s(s_n)$. Then MS_{σ_s} is a functor from $\textsc{Set}^{S'^+}$ to $\textsc{Set}^{S^+}$ given by $\mathsf{MS}_{\sigma_s}(\Omega') = \overline{\sigma}_s; \Omega'$, that is, $\mathsf{MS}_{\sigma_s}(\Omega')(w) = \Omega'(\overline{\sigma}_s(w))$ for every $w \in S^+$, and $\mathsf{MS}_{\sigma_s}(\sigma_\Omega) = \overline{\sigma}_s$ for a natural transformation $\sigma_\Omega : \Omega_1 \Rightarrow \Omega_2$, that is, $\mathsf{MS}_{\sigma_s}(\sigma_\Omega)_w = (\sigma_\Omega)_{\overline{\sigma}_s(w)}$ for every $w \in S^+$.

Then $\mathsf{Flat}(\mathsf{MS})$ has as objects pairs (S,Ω) where S is a set and Ω is an S^+-indexed family of operations names. A morphism from (S,Ω) to (S',Ω') in $\mathsf{Flat}(\mathsf{MS})$ is a pair (σ_s,σ_Ω) where σ_s is a function from S to S' and σ_Ω a natural transformation from Ω to $\mathsf{MS}_{\sigma_s}(\Omega')$, which is an S^+ indexed family of functions $(\sigma_\Omega)_w : \Omega(w) \to \Omega'(\overline{\sigma}_s(w))$, mapping operation symbols of type w in Ω to operations symbols of type $\overline{\sigma}_s(w)$ in Ω'. Thus $\mathsf{Flat}(\mathsf{MS})$ is the category of many-sorted signatures Sig.

Similar to indexed categories one can define indexed functors between indexed categories. An indexed functor $F : C \to D$ with index category E is a family of functors $F_e : C_e \to D_e$ such the following diagram commutes

$$\begin{tikzcd}
C_e \arrow[d, "F_e"'] & C_{e'} \arrow[l, "C_f"'] \arrow[d, "F_{e'}"] \\
D_e & D_{e'} \arrow[l, "D_f"]
\end{tikzcd}$$

for all morphisms $f : e \to e'$ in E. In other words, F s a natural transformation between the indexed categories $C : E^{op} \to \text{Cat}$ and $D : E^{op} \to \text{Cat}$.

As with indexed categories, we can flatten an indexed functor, which yields the functor $\mathsf{Flat}(F) : \mathsf{Flat}(C) \to \mathsf{Flat}(D)$ given by

- $\mathsf{Flat}(F)((e,c)) = F_e(c)$ and
- $\mathsf{Flat}(F)((f,g)) = (f, F_e(g))$

for each object (e,c) and morphism $(f,g) : (e,c) \to (e',c')$ in $\mathsf{Flat}(C)$. $F_e(g)$ is indeed a morphism from $F_e(c)$ to $C_f(F_{e'}(c'))$ because, by the definition of indexed functors, $F_{e'}; D_f$ is the same as $C_f; F_e$.

The limit of a functor $F : J \to \mathsf{Flat}(C)$ in $\mathsf{Flat}(C)$ is a pair (e,c) where e is the limit of the projection of F to the objects and morphisms in E; this requires E to have limits of shape J. The object c in C_e is given by the limit of the functor $\overline{F} : J \to C_e$, which is defined by constructing for each $i \in J$ and $F(i) = (e_i, c_i)$ an appropriate object $\overline{c}_i$ in C_e. This requires that C_e has colimits of shape J.

Theorem 2.27 *(Tarlecki, Burstall and Goguen [58]) Given an indexed category C with index the category E. The category $\mathsf{Flat}(C)$ has limits of shape J if*

- *E has limits of shape J and*
- *the category C_e has limits of shape J for every $e \in E$.*
- *the functor $C_f : C_{e'} \to C_e$ preserves limits of shape J for every morphism $f : e \to e'$ in E.*

Proof. Let F be a functor from J to $\mathsf{Flat}(C)$. Let $F(i)$ be (e_i, c_i) where $e_i \in E$ and $c_i \in C_{e_i}$ for $i \in J$ and let $F(f)$ be $(f^f : e_i \to e_j, f^g : c_i \to C_{f^f}(c_j))$ for $f : i \to j \in J$.

To construct the limit (e, c) of F, we first construct the limit of the functor $F_E : J \to E$ defined by $F_E(i) = e_i$ and $F_E(f) = f^f$. The limit e of F_E exists because E has limits of shape J.

In the next step we define a functor $\overline{F} : J \to C_e$ by $\overline{F}(i) = C_{\pi_i^{F_E}}(c_i)$ for $i \in J$ and $\overline{F}(f) = C_{\pi_i^{F_E}}(f^g)$ for $f : i \to j \in J$.

Note that f^g is a morphism from c_i to $C_{f^f}(c_j)$ in C_i and $C_{\pi_i^{F_E}}$ maps f^g to a morphism from $C_{\pi_i^{F_E}}(c_i)$ to $C_{\pi_i^{F_E}}(C_{f^f}(c_j))$ in C_e. Because we have $\pi_i^{F_E}; f^f = \pi_j^{F_E}$ we get

$$C_{\pi_i^{F_E}}(C_{f^f}(c_j)) = C_{\pi_i^{F_E}; f^f}(c_j) = C_{\pi_j^{F_E}}(c_j)$$

and thus $\overline{F}$ is well-defined.

Now the limit (e, c) of F is given by the limit e of F_E in E and the limit c of $\overline{F}$ in C_e, which exists because C_i has limits of shape J for all $i \in J$. The cone morphisms $\pi_i^F : (e, c) \to (e_i, c_i)$ are given by the pairs $(\pi_i^{F_E}, \pi_i^{\overline{F}})$ for each $i \in J$.

To show that (e, c) is indeed a limit of F, consider a natural transformation $\mu : \Delta(e', c') \Rightarrow (e, c)$. Then we have to show that there exists a unique morphism $h = (h^f, h^g)$ from (e', c') such that the following diagram commutes for all $i \in J$:

$$\begin{tikzcd}
(e,c) \arrow[d, "{(\pi_i^{F_E},\pi_i^{\overline{F}})}"'] & (e',c') \arrow[l, "{(h^f,h^g)}"'] \arrow[dl, "{(\mu_i^f,\mu_i^g)}"] \\
(e_i,c_i) &
\end{tikzcd}$$

That is, we have to show that the following two diagrams commute for all $i \in J$:

$$\begin{tikzcd}
e \arrow[d, "\pi_i^{F_E}"'] & e' \arrow[l, "h^f"'] \arrow[dl, "\mu_i^f"] \\
e_i &
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
C_{h^f}(c) \arrow[d, "C_{h^f}(\pi_i^{\overline{F}})"'] & c' \arrow[l, "h^g"'] \arrow[dl, "\mu_i^g"] \\
C_{\mu_i^f}(c_i) &
\end{tikzcd}$$

The morphism $h^f : e \to e'$ with the property $h^f; \pi_i^{F_E} = \mu_i^f$ for all $i \in J$ is uniquely given by the limit property of e with respect to F_E and the natural transformation given by the family of morphisms $\mu_i^f : e' \to e_i$ for all $i \in J$.

Now we have to find $h^g : c' \to C_{h^f}(c)$. Since C_g preserves limits of shape J for all $g : e_1 \to e_2$ in E, we know that $C_{h^f}(c)$ is a limit of the functor $\overline{F}; C_{h^f}$. Further, because $h^f; \pi_i^{F_E} = \mu_i^f$, we get

$$C_{\mu_i^f}(c_i) = C_{h^f; \pi_i^{F_E}}(c_i) = C_{h^f}(\overline{F}(i)).$$

Thus, $\mu_i^g : c' \to C_{\mu_i^f}(c_i)$ for $i \in J$ is also a family of morphisms from c' to $C_{h^f}(\overline{F}(i))$. Using the limit property of $\overline{F}; C_{h^f}$ we get a morphism $h^g : c' \to C_{h^f}(c)$, unique with the property

$h^g; C_{h^f}(\pi_i^{\overline{F}}) = \mu_i^g$ for all $i \in J$:

$$\begin{tikzcd}
C_{h^f}(c) \arrow[d, "C_{h^f}(\pi_i^{\overline{F}})"'] & c' \arrow[l, "h^g"'] \arrow[dl, "\mu_i^g"] \\
C_{h^f}(\overline{F}(i)) &
\end{tikzcd}$$

□

We can use the last theorem to check if the category of many-sorted signatures has limits of shape J. To apply the theorem we have to check that $\textsc{Set}$ has limits of shape J and that $\textsc{Set}^{S^+}$ has limits of shape J. We have seen in the previous sections that $\textsc{Set}$ is complete and thus has limits for all categories J and that $\textsc{Set}^{S^+}$ has limits of shape J if $\textsc{Set}$ has limits of shape J.

It remains to show that $\mathsf{MS}_{\sigma_S} : \textsc{Set}^{S'^+} \to \textsc{Set}^{S^+}$ preserves limits for every function $\sigma_S : S \to S'$. Let Ω' be the limit of the functor $F : J \to \textsc{Set}^{S'^+}$ and Ω be the limit of the functor $F; \mathsf{MS}_{\sigma_S}$. It suffices if we can show that $\mathsf{MS}_{\sigma_S}(\Omega') = \sigma_S; \Omega'$ is the same as Ω.

By the construction of limits in functor categories we get $\Omega'(w') = \prod_J F_{w'}$ where $F_{w'}(i) = F(w')(i)$ and $F_{w'}(f) = F(f)_{w'}$ for all $w' \in S'^+$, $f : i \to j \in J$ and $i \in J$. Further, for each $w \in S^+$ we have $\Omega(w) = \prod_J (F; \mathsf{MS}_{\sigma_S})_w$ where $(F; \mathsf{MS}_{\sigma_S})_w(i) = (F; \mathsf{MS}_{\sigma_S})(i)(w) = F(i)(\sigma_S(w))$ and $(F; \mathsf{MS}_{\sigma_S})_w(f) = F(f)_{\sigma_S(w)}$ for each $f : i \to j \in J$ and $i \in J$

Thus for every $w \in S^+$ we have

$$(\sigma_S; \Omega')(w) = \Omega'(\sigma_S(w)) = \prod_J F_{\sigma_S(w)} = \prod_J (F; \mathsf{MS}_{\sigma_S})_w = \Omega(w).$$

As a result we get $\textsc{Sig} = \mathsf{Flat}(\mathsf{MS})$ has limits of shape J for all categories J, in other words, $\textsc{Sig}$ is complete.

Note that the terminal object $\Sigma_\top$ in $\textsc{Sig}$ is $(\{1\}, \Omega_\top)$ where $\Omega_\top : \{1\}^+ \to \textsc{Set}$ maps each non-empty word over $\{1\}$ to the one element set. Thus $\Sigma_\top$ is a signature with one sort and infinite many operation symbols, for each possible function type one.

Colimits in $\mathsf{Flat}(C)$ can be constructed in a similar way as limits, that is, as colimits of functors $F_E : J \to E$ and $\overline{F} : J \to C_e$. However, there is one problem in defining $\overline{F}$. Since the co-cone morphism $\iota_i^{F_E}$ is a morphism from e_i to e and thus $C_{\iota_i^{F_E}}$ is a functor from C_e to C_{e_i}, we cannot use $C_{\iota_i^{F_E}}$ to define $\overline{F}(i)$. A solution to this problem is to require that each functor $C_f : C_{e'} \to C_e$ has a left adjoint $L_f : C_e \to C_{e'}$ for each $f : e \to e'$ in E.

Note that, because C_f is required to have a left adjoint, C_f preserves limits; however, this property is not used in the following proof.

Theorem 2.28 *(Tarlecki, Burstall and Goguen [58]) Given an indexed category C with index the category E. The category* $\mathsf{Flat}(C)$ *is cocomplete if*

- *E is cocomplete,*
- *the category C_e is cocomplete for every $e \in E$ and*
- *the functor $C_f : C_{e'} \to C_e$ has a left adjoint for every morphism $f : e \to e'$ in E.*

Proof. Let C be an indexed category from E to Cat and F a functor from J to $\mathsf{Flat}(C)$. Let further $F(i)$ be (e_i, c_i) where $e_i \in E$ and $c_i \in C_{e_i}$ for $i \in J$ and let $F(f : i \to j)$ be $(f^f : e_i \to e_j, f^g : c_i \to C_{f^f}(c_j))$ for $f \in J$.

To construct the colimit (e, c) of F we first construct the colimit of the functor $F_E : J \to E$ defined by $F_E(i) = e_i$ and $F_E(f) = f^f$. Since we have assumed that E is cocomplete, the colimit e of F_E with co-cone morphisms $\iota_i : e_i \to e$ exists.

Then each c_i is translated by the left adjoint of C_{ι_i} to an object $\bar{c}_i$ of C_e and each morphism $f^g : c_i \to C_{f^f}(c_j)$ gives rise to a unique morphism $\bar{f}^g$ from $\bar{c}_i$ to $\bar{c}_j$ by the universal property of $\bar{c}_i$. Note that $\iota_i = f^f; \iota_j$ because ι_i and ι_j are co-cone morphism, which implies that C_{ι_i} and $C_{\iota_j}; C_{f^f}$ are the same. Let η_{c_j} be the unit morphism from c_j to $C_{\iota_j}(\bar{c}_j)$ given by the left adjoint of C_{ι_j}. Then, for the morphism $f^g; C_{f^f}(\eta_{c_j})$ from c_i to $C_{f^f}(C_{\iota_j}(\bar{c}_j) = C_{\iota_i}(\bar{c}_j)$ there exists a unique morphism $\bar{f}^g$ from $\bar{c}_i$ to $\bar{c}_j$ such that the following diagram commutes:

$$\begin{tikzcd}
c_i \arrow[rr, "\eta_{c_i}"] \arrow[dr, "f^g"'] & & C_{\iota_i}(\bar{c}_i) \arrow[d, "C_{\iota_i}(\bar{f}^g)"] \\
& C_{f^f}(c_j) \arrow[r, "C_{f^f}(\eta_{c_j})"'] & C_{\iota_i}(\bar{c}_j)
\end{tikzcd}$$

Thus the functor $F : J \to \mathsf{Flat}(C)$ gives rise to a functor $\bar{F} : J \to C_e$ by $\bar{F}(i) = \bar{c}_i$ and $\bar{F}(f) = \bar{f}^g$. Then c is the colimit of $\bar{F}$, which exists since we have assumed that C_e is cocomplete. The co-cone morphisms of the colimit of F are the pairs (ι_i, ι_i^F) given by the co-cone morphisms of F_E and $\bar{F}$ where ι_i^F from c_i to $C_{\iota_i}(c)$ is $\eta_{c_i}; C_{\iota_i}(\iota_i^{\bar{F}})$.

To show that (e, c) is a colimit of F, consider a natural transformation $\mu : F \to \Delta(e', c')$. Then we have to show the existence of a unique morphism $h = (h^f, h^g)$ from (e, c) to (e', c') such that the following diagram commutes for all $i \in J$:

$$\begin{tikzcd}
(e, c) \arrow[r, "(h^f, h^g)"] & (e', c') \\
(e_i, c_i) \arrow[u, "(\iota_i, \iota_i^F)"] \arrow[ur, "(\mu_i^f, \mu_i^g)"'] &
\end{tikzcd}$$

that is, we have to show

$$\begin{tikzcd}
e \arrow[r, "h^f"] & e' \\
e_i \arrow[u, "\iota_i"] \arrow[ur, "\mu_i^f"'] &
\end{tikzcd}
\text{ and }
\begin{tikzcd}
C_{\iota_i}(c) \arrow[r, "C_{\iota_i}(h^g)"] & C_{\iota_i}(C_{h^f}(c')) \\
c_i \arrow[u, "\iota_i^F"] \arrow[ur, "\mu_i^g"'] &
\end{tikzcd}$$

The morphism h^f is given unique with the above property by the colimit property of e with respect to the functor F_E and the natural transformation given by the family of functions μ_i^f for each $i \in J$.

The morphism $h^g : c \to C_{h^f}(c')$ is given by colimit property of c with respect to $\bar{F}$ and the natural transformation given by $\tilde{\mu}_i^g : \bar{F}(i) \to C_{h^f}(c')$ for each $i \in J$:

$$\begin{tikzcd}
c \arrow[r, "h^g"] & C_{h^f}(c') \\
\bar{c}_i \arrow[u, "\iota_i^{\bar{F}}"] \arrow[ur, "\tilde{\mu}_i^g"'] &
\end{tikzcd}$$

We get $\tilde{\mu}_i^g$ by the left adjoint of C_{ι_i}. First note that $C_{\mu^f}(c') = C_{\iota_i}(C_{h^f}(c'))$ because $\iota_i; h^f = \mu_i^f$ by the colimit property of e. Since $\bar{c}_i$ is a free extension of c_i with respect to C_{ι_i}, we can extend a morphism $\mu_i^g : c_i \to C_{\iota_i}(C_{h^f}(c'))$ in a unique way to a morphism $\tilde{\mu}_i^g : \bar{c}_i \to C_{h^f}(c')$ such that the following diagram commutes:

$$\begin{tikzcd}
c_i \arrow[r, "\eta_{c_i}"] \arrow[dr, "\mu_i^g"'] & C_{\iota_i}(\bar{c}_i) \arrow[d, "C_{\iota_i}(\tilde{\mu}_i^g)"] \\
& C_{\iota_i}(C_{h_f}(c'))
\end{tikzcd}$$

The following diagram is a combination of the last two diagrams where the first diagram is translated by C_{ι_i}:

$$\begin{tikzcd}
C_{\iota_i}(c) \arrow[r, "C_{\iota_i}(h^g)"] & C_{\iota_i}(C_{h^f}(c')) \\
C_{\iota_i}(\bar{c}_i) \arrow[u, "C_{\iota_i}(\iota_i^{\bar{F}})"] \arrow[ur, "C_{\iota_i}(\tilde{\mu}_i^g)"] & \\
c_i \arrow[u, "\eta_{c_i}"] \arrow[uur, "\mu_i^g"'] &
\end{tikzcd}$$

Note that ι_i^F is defined as $\eta_{c_i}; C_{\iota_i}(\iota_i^{\bar{F}})$ and thus we get $\iota_i^F; C_{\iota_i}(h^g) = \mu_i^g$, which finishes our proof. □

To show that the category of many-sorted signatures has colimits of shape J, we have to show that Set has colimits of shape J, that Set^{S^+} has colimits of shape J for each set S and that each functor $\mathsf{MS}_{\sigma_s} : \mathrm{Set}^{S'^+} \to \mathrm{Set}^{S^+}$ has a left adjoint $L_{\sigma_s} : \mathrm{Set}^{S^+} \to \mathrm{Set}^{S'^+}$. Set is cocomplete for categories J and Set^{S^+} has colimits of shape J, whenever Set has colimits of shape J. Thus it remains to show that each functor MS_{σ_s} has a left adjoint L_{σ_s}.

Given a family of operation symbols $\Omega : S \to \mathrm{Set}$, we define a family of operation symbols $\Omega' : S' \to \mathrm{Set}$ by letting $\Omega'(w')$ be the disjoint union of all sets $\Omega(w)$ such that $\overline{\sigma}_s(w) = w'$ for all $w' \in S'^+$. If there is no such w then $\Omega'(w')$ is the empty set. Given a natural transformation $\sigma_\Omega; \Omega_1 \Rightarrow \Omega_2$ in Set^{S^+} then $L_{\sigma_s}(\sigma_\Omega)$ is a natural transformation from $L_{\sigma_s}(\Omega_1)$

to $L_{\sigma_s}(\Omega_2)$. Let w' be a non-empty word over S' and op an operation symbol in $L_{\sigma_s}(\Omega_1)(w')$ then $L_{\sigma_s}(\sigma_\Omega)_{w'}(op) = (\sigma_\Omega)_w(op)$ if $op \in \Omega_1(w)$ and $\overline{\sigma}_s(w) = w'$.

The unit morphism $\mu : \mathsf{Id}_\Omega \Rightarrow L_{\sigma_s}$ is the family of inclusions of Ω_w into $L_{\sigma_s}(\Omega)_{\overline{\sigma}_s(w)}$ for every $w \in S^+$.

Given a functor $F : J \to \textsc{Sig}$ with $F(i) = (S_i, \Omega_i)$ for $i \in J$ and $F(f) = (\sigma_s^f, \sigma_\Omega^f)$ for $f : i \to j$ in J then the colimit of F is given as (S, Ω) where S is the colimit of F_S defined by $F_S(i) = S_i$ and $F_S(f) = \sigma_s$ and Ω is the colimit of the functor $\overline{F} : J \to \textsc{Set}^{S^+}$ defined by $\overline{F}(f)_w(op) = L_{\sigma_s}(f)_w(op)$ for all $w \in S^+$ and $op \in \overline{F}(i)(w)$.

3 Institutions

The notion of institutions is an attempt to formalize the informal notion of a logical system and was developed by Goguen and Burstall [25] as a means to define the semantics of the specification language Clear [13] independent from a particular logic. Many constructs and results in the theory of abstract datatypes are independent of the underlying logical system, and in Chapter 4 we shall give a brief presentation of these constructs and results.

The main aspect of an institution is the notion of satisfaction between formulas and models. Both, formulas and models are given with respect to a certain vocabulary, called signature. Signatures are related by signature morphisms and induce a translation from formulas over one signature to formulas over the other, going in the direction of the signature morphism, and a translation of models in the reverse direction. We require that the notion of satisfaction is compatible with these translations; this requirement is called the satisfaction condition.

The notion of institutions does not cover proof theoretic aspects of a logical system like, for example, a deduction system. To address this problem, for example, Meseguer [44] combines institutions with entailment systems.

The main construction of this thesis will be the institution $\mathcal{R}_\mathcal{I}$ presented in Section 5.3, which is parameterized by a another suitable institution $\mathcal{I}$.

Here, we will only recall the definition of institutions; for more details see the article by Goguen and Burstall [25].

Definition 3.1 (Institution) *An institution $\mathcal{I} = \langle \textsc{Sign}_\mathcal{I}, \mathsf{Str}_\mathcal{I}, \mathsf{Sen}_\mathcal{I}, \models^\mathcal{I} \rangle$ consists of*

- *a category of* signatures $\textsc{Sign}_\mathcal{I}$,
- *a contravariant functor* $\mathsf{Str}_\mathcal{I} : \textsc{Sign}_\mathcal{I}^{op} \to \textsc{Cat}$ *assigning to each signature* Σ *the category of* Σ*-structures and to each signature morphism* $\sigma : \Sigma \to \Sigma'$ *a forgetful functor* $\_|_\sigma : \mathsf{Str}_\mathcal{I}(\Sigma') \to \mathsf{Str}_\mathcal{I}(\Sigma)$,
- *a functor* $\mathsf{Sen}_\mathcal{I} : \textsc{Sign}_\mathcal{I} \to \textsc{Set}$ *assigning to each signature* Σ *the set of* Σ-formulas *and to each signature morphism* $\sigma : \Sigma \to \Sigma'$ *a translation* $\overline{\sigma}$ *of* Σ*-formulas to* Σ'*-formulas, and*
- *a family of* satisfaction relations ${\models^\mathcal{I}}_\Sigma \subseteq \mathsf{Str}_\mathcal{I}(\Sigma) \times \mathsf{Sen}_\mathcal{I}(\Sigma)$ *for* $\Sigma \in \textsc{Sign}_\mathcal{I}$ *indicating whether a* Σ*-formula* φ *is valid in a* Σ*-structure* m*, written* $m \, {\models^\mathcal{I}}_\Sigma \, \varphi$ *or for short* $m \models^\mathcal{I} \varphi$*,*

such that the satisfaction condition *holds: for all signature morphisms* $\sigma : \Sigma \to \Sigma'$*, formulas*

$\varphi \in \mathsf{Sen}_{\mathcal{I}}(\Sigma)$, *and structures* $m' \in \mathsf{Str}_{\mathcal{I}}(\Sigma')$ *we have*

$$m'|_{\sigma} \models^{\mathcal{I}} \varphi \text{ if and only if } m' \models^{\mathcal{I}} \overline{\sigma}(\varphi).$$

For notational convenience we may write $M \models^{\mathcal{I}} \varphi$ for a class of Σ-structures and a Σ-formula instead of $\forall m \in M . m \models^{\mathcal{I}} \varphi$, and similar for $m \models^{\mathcal{I}} \Phi$ and $M \models^{\mathcal{I}} \Phi$ for a set Φ of Σ-formulas. We may also write $\Phi \models^{\mathcal{I}} \Phi'$ for two sets of Σ-formulas Φ and Φ' to denote that if $m \models^{\mathcal{I}} \Phi$ then $m \models^{\mathcal{I}} \Phi'$ for all $m \in \mathsf{Str}_{\mathcal{I}}(\Sigma)$. We shall omit the superscript $\mathcal{I}$ if the institution is clear from the context.

In the following sections we give three examples of institutions. The first section presents the well-known institution of equational logic $\mathcal{EQ}$. Using the techniques of Goguen and Burstall [25], we add data– and partitioned-by constraints to $\mathcal{EQ}$ and get the institution $\mathcal{EQC}$.

The last section introduces the institution $\mathcal{LSL}$, which formalizes the logical system underlying the Larch Shared Language [34]. The Larch Shared Language will be used in some of our examples. The main difference of $\mathcal{LSL}$ with respect to $\mathcal{EQC}$ is that signatures in $\mathcal{LSL}$ contain the sort bool with its usual operations, and that for each sort in the signature we have the equality as a boolean function. The structures in $\mathcal{LSL}$ are those algebras where these symbols have their usual meaning. This makes it quite convenient to write specifications in $\mathcal{LSL}$; however, since the sort bool with its operations and the equality can be specified using equations and constraints, we can use $\mathcal{EQC}$ to reason about specifications in $\mathcal{LSL}$ (cf. Theorem 3.42).

Another example of an institution is the institution $\mathcal{SET}$ presented in Section 9.1, which formalizes the logical system underlying the specification language Z [56].

3.1 Equational Logic

To pave the road for the definition of the institution $\mathcal{LSL}$ in Section 3.4, we shall first present the institutions $\mathcal{EQ}$ of equational logic and $\mathcal{EQC}$ of equational logic with constraints. Signatures in these institutions are many sorted signatures, structures are algebras, and formulas are equations $\forall X\ s = t$. In addition to equations $\mathcal{EQC}$ has generating and partitioned-by constraints as formulas.

The institutions for equational logic and equational logic with data-constraints are already given in Goguen and Burstall [25]. Adding partitioned-by constraints and the institution $\mathcal{LSL}$ are new.

Signatures

Definition 3.2 (Many-sorted Signatures) *A many sorted* signature $\Sigma = (S, \Omega)$ *consists of a set S of* sorts *and a functor $\Omega : S^* \times S \to \mathrm{Set}$ mapping pairs (w, s) to sets of* operation symbols $\Omega(w, s)$. *We require that for all $w \in S^*$ and $s, s' \in S$ we have $\Omega(w, s) \cap \Omega(w, s') = \{\}$. For an operation symbol $\omega \in \Omega(w, s)$ we shall write $\omega : w \to s$. If w is the empty string, ω is called a constant of sort s and we write $\omega : s$. A signature is* finite *if the set of sorts and the union of all $\Omega(w, s)$ is finite.*

A signature morphism *σ from (S, Ω) to (S', Ω') is a pair of a function $\sigma : S \to S'$ and a natural transformation $\sigma : \Omega \Rightarrow \hat{\sigma}; \Omega'$ where $\hat{\sigma}$ is the extension of $\sigma : S \to S'$ to a function $\hat{\sigma} : S^* \times S \to S'^* \times S'$ given by*

$$\hat{\sigma}((s_1 \dots s_n), s) = (\sigma(s_1) \dots \sigma(s_n), \sigma(s)),$$

that is, σ is an $S^* \times S$ indexed family of functions

$$\sigma_{(w,s)} : \Omega((w, s)) \to \Omega(\hat{\sigma}((w, s))).$$

We do not require that the sets $\Omega(w, s)$ are disjoint. This allows us to *overload* operation names, that is, have the same operation symbol with different types. A possible application is the use of the same symbol + for the addition of natural numbers and for the addition of integers.

However, to ensure that certain constructions on terms, like the evaluation of terms, are well-defined, we need that the sets of terms of sort $s \in S$ are disjoint. A sufficient criteria is that the sets $\Omega(w, s)$ and $\Omega(w, s')$ are disjoint for all $w \in S^*$ and $s, s' \in S$ (cf. Loeckx et al. [42]).

Note that in the definition of Ω as a functor from $S^* \times S$ to Set, $S^* \times S$ is viewed a discrete category.

The category Sig has as objects many-sorted signatures and as arrows signature-morphisms. The category FSig has as objects finite many-sorted signatures and is a full subcategory of Sig.

The following fact is well-known, for example [19]:

Fact 3.3 *The category* Sig *is cocomplete.*

Proof. Let $F : J \to \mathrm{Sig}$ be a functor with $F(i) = (S_i, \Omega_i)$ and $F(f : i \to j) = \sigma_f : (S_i, \Omega_i) \to (S_j, \Omega_j)$ for i and $f : i \to j$ in J. In the following we construct the colimit (S_{cl}, Ω_{cl}) of F with the co-cone morphisms ι_i^F. The set S_{cl} is given by the colimit of the functor $F_S : J \to \mathrm{Set}$, and $\Omega_{cl} : S_{cl}^* \times S_{cl} \to \mathrm{Set}$ is the colimit of the functor $F_\Omega : J \to \mathrm{Set}^{S_{cl}^* \times S_{cl}}$.

$F_S(i)$ is S_i for objects i of J, and $F_S(f)$ is the sort function of σ_f for arrows f of J.

Let (w, s) be an element of $S_{cl}^* \times S_{cl}$, then $F_\Omega(i)(w, s)$ is the disjoint union of all sets $\Omega_i(w_i, s_i)$ with $\iota_i^{F_S}(w_i, s_i) = (w, s)$ for some $(w_i, s_si) \in S_i^+$.

$F_\Omega(f : i \to j)(w,s)$ is the extension of the family of functions $\sigma_f(w_i,s_i) : \Omega((w_i,s_i)) \to \Omega(\sigma_f((w_i,s_i)))$ with $\iota_i^{F_S}((w_i,s_i)) = (w,s)$ to a function from $F_\Omega(i)(w,s)$ to $F_\Omega(j)(w,s)$.

We have to take the disjoint union because we did not require the $\Omega_i(w_i,s_i)$ to be disjoint, and if $\iota_i^{F_S}$ identifies sorts, then two operators with the same name but different sorts could be mapped to the same operator. Note that the colimit of F_Ω exists since the functor category $\textsc{Set}^{S_{cl}^* \times S_{cl}}$ is cocomplete because $\textsc{Set}$ is cocomplete (cf. Fact 2.4).

The sort part of the co-cone morphisms ι_i^F is given by the co-cone morphism $\iota_i^{F_S} : S_i \to S_{cl}$, and the operation part is the composition of the inclusion of $\Omega_i(w_i,s_i)$ into $F_\Omega(i)(w,s)$ with $\iota_i^{F_\Omega^{w,s}}$ for every i in J.

Then the colimit property of (S_{cl},Ω_{cl}) is a consequence of the definition of S_{cl} and Ω_{cl} as colimits of F_S and F_Ω. □

The initial object in $\textsc{Sig}$ is $\Sigma_\perp = (\{\},\Omega_\perp)$ where $\Omega_\perp$ is the unique function from $\{\}$ to $\textsc{Set}$ because $\{\}$ is the initial object in $\textsc{Set}$.

Pushouts (S_{po},Ω_{po}) for functors

$$\begin{tikzcd}
\Sigma(1) & & \Sigma(2) \\
& \Sigma(0) \arrow[ul, "\sigma_1"] \arrow[ur, "\sigma_2"'] &
\end{tikzcd}$$

are given by the following pushouts in $\textsc{Set}$:

$$\begin{tikzcd}
& S_{po} & \\
S_1 \arrow[ur, dashed, "\iota_1"] & & S_2 \arrow[ul, dashed, "\iota_2"'] \\
& S_0 \arrow[ul, "\sigma_1"] \arrow[ur, "\sigma_2"'] \arrow[uu, dashed, "\iota_0"] &
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
& \Omega_{po}(w,s) & \\
\overline{\Omega}_1(w,s) \arrow[ur, dashed, "\iota_1(w,s)"] & & \overline{\Omega}_2(w,s) \arrow[ul, dashed, "\iota_2(w,s)"'] \\
& \overline{\Omega}_0(w,s) \arrow[ul, "\overline{\sigma}_1(w,s)"] \arrow[ur, "\overline{\sigma}_2(w,s)"'] &
\end{tikzcd}$$

where $\overline{\Omega}_i(w,s)$ is the disjoint union of all $\Omega_i(w_i,s_i)$ such that $\iota_i(w_i,s_i) = (w,s)$ for every $(w,s) \in S_{po}^* \times S_{po}$, $(w_i,s_i) \in S_i^* \times S_i$, and $i \in \{0,1,2\}$.

The union of signatures $\Sigma_1 \cup \Sigma_2$ where $\Sigma_i = (S_i,\Omega_i)$ for $i \in \{1,2\}$ is the signature (S,Ω) where S is the union of S_1 and S_2, and $\Omega(w,s) = \bar{\Omega}_1(w,s) \cup \bar{\Omega}_2(w,s)$ for $w \in S^*$ and $s \in S$.

$\bar{\Omega}_i$ is the extension of Ω_i from an $S_i^* \times S_i$-indexed family of sets to an $S^* \times S$-indexed family of sets and is given by $\bar{\Omega}_i(w,s) = \Omega_i(w,s)$ if $(w,s) \in S_i^* \times S_i$ and $\Omega(w,s) = \{\}$ otherwise for $i \in \{1,2\}$.

Note that the union of signatures can be viewed as a pushout of Σ_1 and Σ_2 with respect to $\Sigma_1 \cap \Sigma_2$ and the inclusions of $\Sigma_1 \cap \Sigma_2$ into Σ_1 and Σ_2, respectively, where $\Sigma_1 \cap \Sigma_2$ has as sorts $S = S_1 \cap S_2$ and as operations $\Omega(w,s) = \Omega_1(w,s) \cap \Omega_2(w,s)$ for $(w,s) \in S^* \times S$.

Note that the colimit of an infinite diagram of signatures, in general, is not a finite signature. For instance, consider a functor $\mathsf{F} : J \to \textsc{Sig}$ where J is an infinite, discrete category. For each $i \in J$, $F(i)$ is the signature $(\{s\}, \{\omega : s \to s\})$. By the construction of colimits in $\textsc{Sig}$ the colimit of F is a signature with sorts the set $\{(i,s) \mid i \in J\}$ and operations $\{(i,\omega) : (i,s) \to (i,s) \mid i \in J\}$. Since J is infinite, the resulting signature has infinitely many sorts and operations. Thus, while $\textsc{Sig}$ is cocomplete, $\text{F}\textsc{Sig}$ is only finitely cocomplete.

Algebras

Definition 3.4 (Algebras and Homomorphisms) *Let $\Sigma = (S, \Omega)$ be a signature. A* Σ-algebra *A consists of a functor A from the set S, viewed as a discrete category, to* $\textsc{Set}$ *mapping each sort s in S to its* carrier set *$A(s)$, together with a family of total functions $A(\omega)$ from $A(w)$ to $A(s)$ for each $\omega \in \Omega(w,s)$ where $A(w) = A(s_1) \times \ldots \times A(s_n)$ for $w = s_1 \ldots s_n$.*

A Σ-homomorphism *$h : A \to B$ is a natural transformation $h : A \Rightarrow B$ between the functors A and B from S to* $\textsc{Set}$ *compatible with the operations in Ω, that is, for each $\omega : w \to s$ and $a_1, \ldots, a_n$ in $A(w)$*

$$h_s(A(\omega)(a_1, \ldots, a_n)) = B(\omega)(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)).$$

Σ-algebras and Σ-homomorphisms form the category $\mathsf{Alg}(\Sigma)$. A homomorphism $h : A \to B$ is an *isomorphism* if h_s is an isomorphism between $A(s)$ and $B(s)$ for every sort s. Two algebras A and B are *isomorphic* if there exists an isomorphism $h : A \to B$.

The forgetful functor U_Σ from the category of Σ-algebras to the functor category $\textsc{Set}^S$ maps Σ-algebras A to their family of carrier sets and Σ-homomorphisms $h : A \to B$ to their natural transformations $h : A \Rightarrow B$ between their family of carrier sets.

Definition 3.5 (σ-Reduct) *Given a signature morphism σ from (S, Ω) to (S', Ω') and a Σ'-algebra A, the* σ-reduct *of A, written $A|_\sigma$, is the Σ-algebra defined by*

$$\begin{gathered} A|_\sigma = \sigma; A \\ (A|_\sigma)(\omega) = A(\sigma(\omega)) \textit{ for all } \omega : w \to s \textit{ and } (w,s) \in S^+. \end{gathered}$$

For each Σ'-homomorphism $h : A \to B$ a Σ-homomorphism $h|_\sigma : A|_\sigma \to B|_\sigma$ is defined by

$$h|_\sigma = \sigma; h.$$

The σ-reduct is a functor from the category of Σ'-algebras to the category of Σ-algebras.

Definition 3.6 (Alg) Alg *is a contravariant functor from* $\textsc{Sig}$ *to* $\textsc{Cat}$ *mapping every signature Σ to the category of Σ-algebras* $\mathsf{Alg}(\Sigma)$ *and every signature morphism $\sigma : \Sigma \to \Sigma'$ to the functor $\_|_\sigma : \mathsf{Alg}(\Sigma') \to \mathsf{Alg}(\Sigma)$.*

Equations

Definition 3.7 (Term-algebra) *Let $\Sigma = (S, \Omega)$ be a signature and X an S-indexed family of sets, the term algebra $T_\Sigma(X)$ has as carrier sets the smallest S-indexed family of sets such that*

$$\begin{aligned} X(s) &\subseteq T_\Sigma(X)(s) \textit{ for each } s \in S \textit{ and} \\ \omega(t_1, \ldots, t_n) &\in T_\Sigma(X)(s) \textit{ for each } \omega : w \to s \textit{ in } \Omega \textit{ and } t_1, \ldots, t_n \in T_\Sigma(X)(w) \end{aligned}$$

and for every operation symbol $\omega : w \to s$ in Ω

$$T_\Sigma(X)(\omega)(t_1, \ldots, t_n) = \omega(t_1, \ldots, t_n).$$

The elements in $T_\Sigma(X)(s)$ are called terms *of sort s.*

If X is the family of empty sets, that is, $X(s) = \{\}$ for all sorts s in S, then $T_\Sigma(X)$ is called the *ground term algebra*, and we may write T_Σ instead of $T_\Sigma(X)$.

The functor T_Σ from $\textsc{Set}^S$ to $\mathsf{Alg}(\Sigma)$ maps each family of sets $X : S \to \textsc{Set}$ to the term algebra $T_\Sigma(X)$ and each natural transformation $h : X \Rightarrow Y$ to the unique homomorphism from $\mathsf{T}_\Sigma(X)$ to $\mathsf{T}_\Sigma(Y)$ satisfying $\mathsf{T}_\Sigma(h)_s(x) = h_s(x)$ for every $x \in X(s)$, $s \in S$ and

$$\mathsf{T}_\Sigma(h)_s(\omega(t_1, \ldots, t_n)) = \omega(\mathsf{T}_\Sigma(h)_{s_1}(t_1), \ldots, \mathsf{T}_\Sigma(h)_{s_n}(t_n))$$

for every operation $\omega : w \to s \in \Omega$.

The functor T_Σ is left adjoint to the forgetful functor $\mathsf{U}_\Sigma : \mathsf{Alg}(\Sigma) \to \textsc{Set}^S$, which means that for each algebra B and natural transformation $h : X \Rightarrow \mathsf{U}_\Sigma(B)$ there exists a unique extension of h to a homomorphism $\tilde{h}$ from $\mathsf{T}_\Sigma(X)$ to B such that $h = \eta; \mathsf{U}_\Sigma(\tilde{h})$ where η is the natural transformation from X to $\mathsf{U}_\Sigma(\mathsf{T}_\Sigma(X))$ given by the inclusions of $X(s)$ into $\mathsf{T}_\Sigma(X)(s)$. The homomorphism $\tilde{h}$ is uniquely defined by $\tilde{h}_s(x) = h_s(x)$ for $x \in X(s)$, and $s \in S$ and $\tilde{h}(\omega(t_1, \ldots, t_n)) = B(\omega)(\tilde{h}_{s_1}(t_1), \ldots, \tilde{h}_{s_n}(t_n))$ for $\omega : w \to s \in \Omega$.

The natural transformation $\tilde{h}$ from $\mathsf{T}_\Sigma(X)$ to B *evaluates* a term $t \in \mathsf{T}_\Sigma(X)$ in B with respect to h. If X is the family of empty sets then there exists a unique natural transformation $h : X \Rightarrow \mathsf{U}_\Sigma(B)$ for each Σ-algebra B, and we shall write $B(t)$ for the evaluation of a ground term $t \in T_\Sigma$ in B instead of $\tilde{h}(t)$.

The ground-term-algebra T_Σ is an initial object in $\mathsf{Alg}(\Sigma)$ because the family of empty sets is an initial object in $\textsc{Set}^S$ and left adjoints preserve initial objects (cf. Section 2.4). The unique homomorphism h_B from T_Σ to an arbitrary Σ-algebra B is given by $h_B(t) = B(t)$.

Note that in the definition of algebras and homomorphisms (cf. Definition 3.4) we used the same symbol A for a Σ-algebra A and its family of carrier sets $A : S \to \textsc{Set}$, and we used the same symbol h for a Σ-homomorphism from A to B and the natural transformation $h : A \Rightarrow B$ in $\textsc{Set}^S$. Thus, in the following, we shall leave the application of the forgetful functor $\mathsf{U}_\Sigma : \mathsf{Alg}(\Sigma) \to \textsc{Set}^S$ implicit. For example, if A is a Σ-algebra, we write $\mathsf{T}_\Sigma(A)$ instead of $\mathsf{T}_\Sigma(\mathsf{U}_\Sigma(A))$.

Fact 3.8 *Let A be a Σ-algebra and $\eta : \mathsf{Id}_{\mathrm{SET}^S} \Rightarrow \mathsf{T}_\Sigma ; \mathsf{U}_\Sigma$ be the unit of the adjunction of T_Σ and U_Σ. Then we have*

$$\eta_A ; \widetilde{\mathtt{id}}_A = \mathtt{id}_A$$

in SET^S.

Proof. By definition, $\widetilde{\mathtt{id}}_A$ is the the unique homomorphism from $\mathsf{T}_\Sigma(X)$ to A such that $\eta_A ; \widetilde{\mathtt{id}}_A = \mathtt{id}_A$ in SET^S. □

Fact 3.9 *The following diagram in $\mathsf{Alg}(\Sigma)$ commutes for every homomorphism $h : A \to B$ in $\mathsf{Alg}(\Sigma)$:*

$$\begin{tikzcd}
\mathsf{T}_\Sigma(A) \arrow[r, "\mathsf{T}_\Sigma(h)"] \arrow[d, "\widetilde{\mathtt{id}}_A"'] \arrow[dr, "\tilde{h}"] & \mathsf{T}_\Sigma(B) \arrow[d, "\widetilde{\mathtt{id}}_B"] \\
A \arrow[r, "h"'] & B
\end{tikzcd}$$

Proof. Let $\Sigma = (S, \Omega)$ be a many-sorted signature. First, we show that

$$\begin{tikzcd}
\mathsf{T}_\Sigma(A) \arrow[d, "\widetilde{\mathtt{id}}_A"'] \arrow[dr, "\tilde{h}"] & \\
A \arrow[r, "h"'] & B
\end{tikzcd}$$

commutes in $\mathsf{Alg}(\Sigma)$. Since we have

$$\eta_A ; \widetilde{\mathtt{id}}_A ; h = \mathtt{id}_A ; h = h$$

in SET^S, and the homomorphism $\tilde{h}$ is unique with the property $\eta_A ; \tilde{h} = h$, we have $\tilde{h} = \widetilde{\mathtt{id}}_A ; h$ in $\mathsf{Alg}(\Sigma)$.

Next, we show that the following diagram commutes in $\mathsf{Alg}(\Sigma)$:

$$\begin{tikzcd}
\mathsf{T}_\Sigma(A) \arrow[r, "\mathsf{T}_\Sigma(h)"] \arrow[dr, "\tilde{h}"'] & \mathsf{T}_\Sigma(B) \arrow[d, "\widetilde{\mathtt{id}}_B"] \\
 & B
\end{tikzcd}$$

Since η is a natural transformation we have $\eta_A ; \mathsf{T}_\Sigma(h) = h ; \eta_B$. By composing both sides of the equation with $\widetilde{\mathtt{id}}_B$ we get

$$\eta_A ; \mathsf{T}_\Sigma(h) ; \widetilde{\mathtt{id}}_B = h ; \eta_B ; \widetilde{\mathtt{id}}_B = h.$$

Again, since $\tilde{h}$ is unique with the property $\eta_A ; \tilde{h} = h$, we get that $\tilde{h} = \mathsf{T}_\Sigma(h) ; \widetilde{\mathtt{id}}_B$ in $\mathsf{Alg}(\Sigma)$. □

Given a recursive enumerable set V. A functor $X : S \to \textsc{Set}$ is a family of *variables* for the signature $\Sigma = (S, \Omega)$ if $X(s) \subseteq V$ for each sort s in S and $X(s) \cap X(s') = \{\}$ for two sorts $s \neq s'$ in S.

For a Σ-algebra A and a family of variables $X \in \textsc{Set}^S$, a natural transformation ρ from X to the family of carrier sets of A is called a *variable assignment*.

In the following we may write, for example, $X = \{x, y : s_1, z : s_2\}$ for the family of variables $X : \{s_1, s_2, s_3\} \to \textsc{Set}$ given by $X(s_1) = \{x, y\}$, $X(s_2) = z$ and $X(s_3) = \{\}$. If the sorts of the variables are clear from the context we may omit the declaration of sorts and just write $X = \{x, y, z\}$.

Definition 3.10 (Translation of Terms) *Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism with $\Sigma = (S, \Omega)$ and $\Sigma' = (S', \Omega')$, and let $X : S \to \textsc{Set}$ be a family of variables. Define a new family of variables $X' : S' \to \textsc{Set}$ by taking for $X'(s')$ the union of all $X(s)$ such that $\sigma(s) = s'$ for each s' in S'. We set $X'(s') = \{\}$ if there is no $s \in S$ with $\sigma(s) = s'$.*

The signature morphism σ defines an S-indexed family of functions $\overline{\sigma}_s$ from $\mathsf{T}_\Sigma(X)(s)$ to $\mathsf{T}_\Sigma(X')(\sigma(s))$ by

$$\overline{\sigma}(x) = x \quad \text{for } x \in X(s) \text{ and } s \in S$$
$$\overline{\sigma}(\omega(t_1, \ldots, t_n)) = \sigma(\omega)(\overline{\sigma}(t_1), \ldots, \overline{\sigma}(t_n))$$

for $\omega \in \Omega(w, s)$, $t_1, \ldots, t_n \in \mathsf{T}_\Sigma(w)$ and $w \in S^*$.

A Σ-*equation* has the form $\forall X\ t = r$ where X is a family of variables and t and r are terms in $\mathsf{T}_\Sigma(X)(s)$ for some sort $s \in S$.

Definition 3.11 (Eqn) *The functor Eqn from $\textsc{Sig}$ to $\textsc{Set}$ maps a signature Σ to the set of Σ-equations and a signature morphism $\sigma : \Sigma \to \Sigma'$ to the function $\mathsf{Eqn}(\sigma)$ mapping a Σ-equation $\forall X\ r = t$ to a Σ'-equation $\forall X'\ \overline{\sigma}(r) = \overline{\sigma}(t)$.*

Definition 3.12 (Satisfaction) *A Σ-algebra A satisfies a Σ-equation $\forall X\ t = r$, written $A \models \forall X\ t = r$, if $\tilde{\rho}(t) = \tilde{\rho}(r)$ for all variable assignments $\rho : X \Rightarrow A$.*

When writing an equation $\forall \{x_1 : s_1, \ldots, x_n : s_n\}\ r = t$, we shall omit the braces and write $\forall x_1 : s_1, \ldots, x_n : s_n\ r = t$ instead, or $\forall x_1, \ldots, x_n\ r = t$ if the sorts of the variables are clear from the context.

Lemma 3.13 *If A and B are isomorphic Σ-algebras, then*

$$A \models \forall X\ t = r \text{ if and only if } B \models \forall X\ t = r$$

for all Σ-equations $\forall X\ t = r$.

Proof. ($\Leftarrow$) Let $B \models \forall X\ t = r$ and ρ be an arbitrary variable assignment from X to A. We have to show that $\tilde{\rho}(t) = \tilde{\rho}(r)$.

Define a variable assignment from X to B by $\rho' = \rho; \iota$ where ι is the isomorphism from A to B. Note that $\tilde{\rho}'$ is the same as $\tilde{\rho}; \iota$. Then we have $\tilde{\rho}'(t) = \tilde{\rho}'(r)$ because $B \models \forall X \ t = r$.

By applying ι^{-1} to each side of the equation $\tilde{\rho}'(r) = \tilde{\rho}'(t)$ we get $\tilde{\rho}'; \iota^{-1}(r) = \tilde{\rho}; \iota; \iota^{-1}(r) = \tilde{\rho}(r)$ and $\tilde{\rho}'; \iota^{-1}(t) = \tilde{\rho}; \iota; \iota^{-1}(t) = \tilde{\rho}(t)$ and thus $\tilde{\rho}(r) = \tilde{\rho}(t)$.

($\Rightarrow$) The argument is the same as for the previous case but with the rôles of ι an ι^{-1} reversed. □

Definition 3.14 (Institution $\mathcal{EQ}$) *The institution of equational logic $\mathcal{EQ}$ is given by the category* Sig *of many sorted signatures, the contravariant functor* Alg *from* Sig *to* Cat*, mapping many sorted signatures Σ to the category of Σ-algebras, the functor* Eqn *from* Sig *to* Set*, mapping signatures Σ to the set of Σ-equations, and the family of satisfaction relations $\models_\Sigma \subseteq \mathsf{Alg}(\Sigma) \times \mathsf{Eqn}(\Sigma)$.*

It remains to show that the satisfaction conditions holds for $\mathcal{EQ}$. To show this, the following lemma is needed.

Lemma 3.15 *Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism, A a Σ'-algebra, $\rho : X \Rightarrow A|_\sigma$ a variable assignment ,and let $\rho' : X' \Rightarrow A$ be the variable assignment defined by $\rho'_{s'}(x) = \rho_s(x)$ if $x \in X(s)$ and $s' = \sigma(s)$ where X' is defined as in 3.10. Then*

$$\tilde{\rho}(t) = \tilde{\rho'}(\overline{\sigma}(t))$$

for all terms t from $\mathsf{T}_\Sigma(X)$.

Note that ρ' is well-defined because $X(s) \cap X(s') = \{\}$ for all $s, s' \in S$ with $s \neq s'$.

Proof. The proof is done by induction on the structure of terms from $\mathsf{T}_\Sigma(X)$. If $t = x$ and $x \in X(s)$, then

$$\tilde{\rho}_s(x) = \rho_s(x) = \rho'_{\sigma(s)}(x) = \tilde{\rho}'_{\sigma(s)}(\overline{\sigma}(x))$$

and if $t = \omega(t_1, \ldots, t_n)$, then

$$\begin{aligned} \tilde{\rho}(\omega(t_1, \ldots, t_n)) &= (A|_\sigma)(\omega)(\tilde{\rho}(t_1, \ldots, t_n)) && | \text{ Def. of } \tilde{\rho} \\ &= A(\sigma(\omega))(\tilde{\rho}'(\overline{\sigma}(t_1), \ldots, \overline{\sigma}(t_n))) && | \text{ Def. of } A|_\sigma \text{ and Ind. Hyp.} \\ &= \tilde{\rho}'(\overline{\sigma}(\omega(t_1, \ldots, t_n))) && | \text{ Def. of } \tilde{\rho}' \end{aligned}$$

□

Theorem 3.16 (Satisfaction condition) *Let σ be a signature morphism from Σ to Σ', A a Σ'-algebra, and $\forall X \ r = t$ a Σ-equation. Then*

$$A|_\sigma \models \forall X \ r = t \textit{ if and only if } A \models \mathsf{Eqn}(\sigma)(\forall X \ r = t)$$

Proof. ($\Leftarrow$) Let $A \models \mathsf{Eqn}(\sigma)(\forall X\ t = r)$. Fix an arbitrary $\rho : X \Rightarrow A|_\sigma$ and define $\rho' : X' \Rightarrow A$ as in Lemma 3.15. Since $A \models \overline{\sigma}(\forall X\ r = t)$, we have $\tilde{\rho}'(\overline{\sigma}(r)) = \tilde{\rho}'(\overline{\sigma}(t))$, and because of Lemma 3.15 we get $\tilde{\rho}(r) = \tilde{\rho}'(\overline{\sigma}(r)) = \tilde{\rho}'(\overline{\sigma}(t)) = \tilde{\rho}(t)$.

($\Rightarrow$) Let $A|_\sigma \models \forall X\ r = t$ and ρ' be a variable assignment from X' to A. Define $\rho : X \Rightarrow A|_\sigma$ by $\rho_s(x) = \rho'_{\sigma(s)}(x)$. Since $A|_\sigma \models \forall X\ r = t$ and because of Lemma 3.15, we have $\tilde{\rho}'(\overline{\sigma}(r)) = \tilde{\rho}(r) = \tilde{\rho}(t) = \tilde{\rho}'(\overline{\sigma}(t))$. □

3.2 Amalgamation

An important property of institutions, that is also required for the construction of the institution $\mathcal{R}_\mathcal{I}$ in Section 5.3, is that the contravariant structure functor $\mathsf{Str}_\mathcal{I}$ from $\textsc{Sign}_\mathcal{I}$ to $\textsc{Cat}$ preserves colimits. That is, if F is a functor from J to $\textsc{Sign}_\mathcal{I}$ such that the colimit $(\iota^F, \coprod F)$ of F exists, then $(\iota^F; \mathsf{Str}_\mathcal{I}, \mathsf{Str}_\mathcal{I}(\coprod F))$ is a limit of the functor $F; \mathsf{Str}_\mathcal{I} : J^{op} \to \textsc{Cat}$.

Given a colimit diagram $F : I \to \textsc{Sign}_\mathcal{I}$ with $F(i) = \Sigma_i$ and a Σ_i-structure m_i for each $i \in I$, then the m_i can be glued together to form one structure m over the colimit signature of F provided that the colimit of F exists and that $m_j|_{F(f)} = m_i$ for all morphisms $f : i \to j$ in J.

$\mathsf{Str}_\mathcal{I}$ preserves colimits if $\mathsf{Str}_\mathcal{I}$ maps the initial object in $\textsc{Sign}_\mathcal{I}$ to the terminal object in $\textsc{Cat}$ and if pushouts in $\textsc{Sign}_\mathcal{I}$ are mapped to pullbacks in $\textsc{Cat}$.

Definition 3.17 *We call an institution* exact *if the contravariant structure functor* $\mathsf{Str}_\mathcal{I}$ *from* $\textsc{Sign}_\mathcal{I}$ *to* $\textsc{Cat}$ *preserves colimits.*

The institutions $\mathcal{EQ}$, $\mathcal{EQC}$, $\mathcal{LSL}$, and $\mathcal{SET}$ all have amalgamation, and since their structure functors also preserve initial objects, they are all exact.

Definition 3.18 (Amalgamated Sum) *An institution $\mathcal{I}$ has amalgamation if for any pushout diagram $F : \mathrm{V} \to \textsc{Sign}_\mathcal{I}$ for which a pushout $\coprod F$ exists and for any pair of structures $m_1 \in \mathsf{Str}_\mathcal{I}(F(1))$ and $m_2 \in \mathsf{Str}_\mathcal{I}(F(2))$ such that $m_1|_{F(f)} = m_0 = m_2|_{F(g)}$ there exists a unique $m \in \mathsf{Str}_\mathcal{I}(\coprod F)$ with $m|_{\iota_1} = m_1$ and $m|_{\iota_2} = m_2$. m is called the amalgamated sum of m_1 and m_2 with respect to m_0 and is written as $m_1 +_{m_0} m_2$.*

Similarly, there has to exist for each pair of homomorphisms $h_1 : m_1 \to m'_1$ and $h_2 : m_2 \to m'_2$ with $h_1|_{F(f)} = h_2|_{F(g)}$ a unique homomorphism h from the amalgamated sum of m_1 and m'_1 to the amalgamated sum of m_2 and m'_2 such that $h|_{\iota_1} = h_1$ and $h|_{\iota_2} = h_2$.

Note that if an institution has amalgamation, the objects and homomorphisms in $\mathsf{Str}_\mathcal{I}(\coprod F)$ are exactly the amalgamated sums of objects and homomorphisms from $\mathsf{Str}_\mathcal{I}(F(1))$ and $\mathsf{Str}_\mathcal{I}(F(2))$. To see this, consider an object m in $\mathsf{Str}_\mathcal{I}(\coprod F)$ and let $m_0 = m|_{\iota_0}$, $m_1 = m|_{\iota_1}$, and $m_2 = m|_{\iota_2}$. Because ι_0, ι_1, and ι_2 are co-cone morphisms we have $\sigma_1; \iota_1 = \iota_0 = \sigma_2; \iota_2$, and therefore $m_1|_{\sigma_1} = m_0 = m_2|_{\sigma_2}$. Thus the amalgamated sum of m_1 and m_2 exists. Then $m_1 +_{m_0} m_2$ is the same as m because of the uniqueness of the amalgamated sum.

FACT 3.19 *An institution $\mathcal{I}$ has amalgamation if and only if the contravariant functor* $\mathsf{Str}_{\mathcal{I}}$ *preserves pushouts.*

PROOF. ($\Rightarrow$) Assume that $\mathcal{I}$ has amalgamation. Let $\coprod F$ be a pushout of the functor $F : \mathrm{V} \to \mathrm{SIGN}_{\mathcal{I}}$, then we have to show that $\mathsf{Str}_{\mathcal{I}}(\coprod F)$ is a pullback in CAT. Consider a category C with two functors $F : C \to \mathsf{Str}_{\mathcal{I}}(F(1))$ and $G : C \to \mathsf{Str}_{\mathcal{I}}(F(2))$ such that $F; \mathsf{Str}_{\mathcal{I}}(\sigma_1) = G; \mathsf{Str}_{\mathcal{I}}(\sigma_2)$. We have to construct a unique functor $H : C \to \mathsf{Str}_{\mathcal{I}}(\coprod F)$ with $H; \mathsf{Str}_{\mathcal{I}}(\iota_1) = F$ and $H; \mathsf{Str}_{\mathcal{I}}(\iota_2) = G$. Define $H(c) = F(c) +_{m_0} G(c)$ where $F(c)|_{\sigma_1} = m_0 = G(c)|_{\sigma_2}$, and in a similar manner $H(h : c \to c') = F(h) +_{h_0} G(h)$. Note that H is unique because of the uniqueness of the amalgamated sum. Now we have

$$\mathsf{Str}_{\mathcal{I}}(\iota_1)(H(c)) = (F(c) +_{m_0} G(c))|_{\iota_1} = F(c)$$

for all $c \in C$ and similar

$$\mathsf{Str}_{\mathcal{I}}(\sigma_2)(H(c)) = (F(c) +_{m_0} G(c))|_{\iota_2} = G(c).$$

($\Leftarrow$) Assume that the $\mathsf{Str}_{\mathcal{I}}$ preserves colimits and let F be a functor from V to $\mathrm{SIGN}_{\mathcal{I}}$. We have to show that $\mathcal{I}$ has amalgamation. Consider a $F(1)$-structure m_1 and a $F(2)$-structure m_2. These structures define functors M_1 from $\mathbf{1}$ to $\mathsf{Str}_{\mathcal{I}}(F(1))$ and $\mathsf{Str}_{\mathcal{I}}(F(2))$ by mapping the only element of $\mathbf{1}$ to m_1 and m_2, respectively. If $m_1|_{\sigma_1} = m_0 = m_2|_{\sigma_2}$, then $M_1; \mathsf{Str}_{\mathcal{I}}(\sigma_1) = M_2; \mathsf{Str}_{\mathcal{I}}(\sigma_1)$, and thus there exists a unique functor M from $\mathbf{1}$ to $\mathsf{Str}_{\mathcal{I}}(\coprod F)$ with $M; \mathsf{Str}_{\mathcal{I}}(\iota_1) = M_1$ and $M; \mathsf{Str}_{\mathcal{I}}(\iota_2) = M_2$ because $\mathsf{Str}_{\mathcal{I}}(\coprod F)$ is a pullback in CAT. Now $M(1)$ has all the required properties of an amalgamated sum of m_1 and m_2 with respect to m_0.

Similarly, two homomorphisms $h_1 : m_1 \to m_1'$ and $h_2 : m_2 \to m_2'$ define two functors H_1 and H_2 from $\mathbf{2}$ to $\mathsf{Str}_{\mathcal{I}}(F(1))$ and $\mathsf{Str}_{\mathcal{I}}(F(2))$ by mapping the arrow f from 1 to 2 in $\mathbf{2}$ to h_1 and h_2, respectively. As before, if $h_1|_{\sigma_1} = h_0 = h_2|_{\sigma_2}$, then there exists a unique functor H from $\mathbf{2}$ to $\mathsf{Str}_{\mathcal{I}}(\coprod F)$ with $H; \mathsf{Str}_{\mathcal{I}}(\iota_1) = H_1$ and $H; \mathsf{Str}_{\mathcal{I}}(\iota_2) = H_2$ and $H(f)$ is the amalgamated sum of h_1 and h_2 with respect to h_0. □

FACT 3.20 *The institution $\mathcal{EQ}$ has amalgamation.*

PROOF. Given a pushout diagram $F : \mathrm{V} \to \mathrm{SIG}$ with pushout object $\coprod F$ and co-cone morphisms $\iota_i : F(i) \to \coprod F$.

$$\begin{tikzcd}
& \coprod F & \\
F(1) \arrow[ur, dashed, "\iota_1"] & & F(2) \arrow[ul, dashed, "\iota_2"'] \\
& F(0) \arrow[ul, "\sigma_1"] \arrow[ur, "\sigma_2"'] \arrow[uu, dashed, "\iota_0"] &
\end{tikzcd}$$

Let A_1 be in $\mathsf{Alg}(F(1))$ and A_2 in $\mathsf{Alg}(F(2))$ such that $A_1|_{\sigma_1} = A_0 = A_2|_{\sigma_2}$. The amalgamated sum of A_1 and A_2 with respect to A_0 is given by:

$$(A_1 +_{A_0} A_2)(s) = A_i(s_i) \text{ for } \iota_i(s_i) = s \text{ and } s_i \text{ is a sort of } F(i)$$

for each sort s of $\coprod F$ and

$$(A_1 +_{A_0} A_2)(\omega) = A_i(\omega_i) \text{ for } \iota_i(\omega_i) = \omega \text{ and } \omega_i \text{ is an operation of } F(i)$$

for each operation ω of $\coprod F$.

To show that $A = A_1 +_{A_0} A_2$ is well-defined, consider s_i and s_j with $\iota_i(s_i) = s = \iota_j(s_j)$. It suffices to assume that there exists a morphism f from i to j in V and thus a signature morphism $\Sigma(f) = \sigma$ from $\Sigma(i)$ to $\Sigma(j)$. In this case we have $\sigma(s_i) = s_j$ and $A_j|_\sigma = A_i$, and therefore $A(s) = A_i(s_i) = A_j|_\sigma(s_i) = A_j(\sigma(s_i)) = A_j(s_j)$.

A similar argument shows that $A(\omega)$ is well-defined.

Let $g : A_1 \to A_1'$ be a $F(1)$-homomorphism and $h : A_2 \to A_2'$ be a $F(2)$-homomorphism such that $g|_{\sigma_1} = f = h|_{\sigma_2}$. Then a $\coprod F$-homomorphism $g +_f h$ from $A_1 +_{A_0} B$ to $A_1' +_{A_0'} A_2'$ is defined by:

$$(g +_f h)_s = \begin{cases} g_{s_1} & \text{if } \iota_1(s_1) = s \text{ and } s_1 \text{ is a sort of } F(1) \\ h_{s_2} & \text{if } \iota_2(s_2) = s \text{ and } s_2 \text{ is a sort of } F(2) \end{cases}$$

□

Corollary 3.21 *The contravariant functor* $\mathsf{Alg} : \textsc{Sig}^{op} \to \textsc{Cat}$ *preserves colimits.*

Proof. A functor preserves colimits if it preserves pushouts and initial objects (cf. Lemma 2.9). Since $\mathcal{EQ}$ has amalgamation, the functor Alg preserves pushouts. The initial object of Sig is the signature $(\{\}, \Omega_\perp)$ with no sorts and no operations. The only object of the category of $(\{\}, \Omega_\perp)$-algebras is the algebra with no carrier sets and no functions. Thus $\mathsf{Alg}((\{\}, \Omega_\perp))$ is isomorphic to **1** and therefore a terminal object of Cat. □

3.3 Constraints

The problem with the institution $\mathcal{EQ}$ is that $\mathcal{EQ}$ is not powerful enough to express certain classes of algebras by sets of equations only. For example, it is impossible to specify a class of algebras where the carrier set of a sort is isomorphic to the natural numbers.

One reason for this is that algebras whose carrier sets contain junk, that is, some elements in their carrier sets cannot be expressed by ground-terms, cannot be excluded.

Another reason is that the class of algebras satisfying a set of equations always contains the trivial algebra T. The trivial algebra for a signature $\Sigma = (S, \Omega)$ has either $T(s) = \{1\}$ or $T(s) = \{\}$ for a sort s from S depending on whether the ground term algebra has an element of this sort or not. The operations $\omega : s_1 \dots s_n \to s \in \Omega$ are interpreted as the unique function from the empty set to any other set if $T(s_i) = \{\}$ for some $1 \leq i \leq n$, that is, $T(s_1) \times \dots \times T(s_n)$ is isomorphic to the empty set, or $T(\omega)(1, \dots, 1) = 1$. Note that, if $T(s_i)$ is not empty for $1 \leq i \leq n$, then $T(s)$ cannot be empty because if $T(s_i)$ is not empty, there exists ground terms t_i of sort s_i. Then $\omega(t_1, \ldots, t_n)$ is a ground term of sort s and thus $T(s) = \{1\}$.

The trivial algebra T satisfies all equations $\forall X\ t = r$ from $\mathsf{Eqn}(\Sigma)$. In order to see this, consider a variable assignment $\rho : X \Rightarrow T$. If a variable assignment ρ from X to T exists, then either $X(s)$ is empty or $X(s)$ is non-empty; in this case $T(s)$ has to be non-empty too. Then $\rho(x) = 1$ for all $x \in X(s)$ which implies $\tilde{\rho}(r) = 1 = \tilde{\rho}(t)$ and thus $T \models \forall X\ t = r$.

Adding constraints as formulas to $\mathsf{Eqn}(\Sigma)$ enhances the expressive power of equational specifications. Generating constraints address the junk-problem by requiring that each element of a generated sort can be expressed by an appropriate term of that sort. For proving properties of elements of generated sorts one can use induction over the term structure. Free generating constraints add disequalities between the constructors of a generated sort and thus help exclude the trivial algebra while partitioned-by constraints add a limited form of behavioral equivalence.

For example, the constraint

$$N \ \mathsf{generated\ by}\ \mathtt{zero}, \mathtt{succ}$$

ensures that for each algebra A satisfying that constraint and each element e of $A(N)$ there exists a term t of the form $\mathtt{succ}(\ldots(\mathtt{zero})\ldots)$ such that $A(t) = e$. Since each element of $A(N)$ can be represented by a term, one can use induction on the structure of terms to prove properties φ about elements of sort N, i.e. if $A \models \varphi[x/\mathtt{zero}]$ and if $A \models \varphi$ implies $A \models \varphi[x/\mathtt{succ}(x)]$, then $A \models \varphi$. However, the generating constraint does not ensure that different terms correspond to different elements in $A(N)$. Thus an algebra A with $A(N) = \{0\}$, $A(\mathtt{zero}) = 0$ and $A(\mathtt{succ}) = \lambda x.x$ satisfies the generating constraint. To make sure that $A(\mathtt{succ})(x) \neq x$ one has to use the following free generating constraint:

$$N \ \mathsf{generated\ freely\ by}\ \mathtt{zero}, \mathtt{succ}.$$

Let S be the sort of sets with elements of sort E, $\mathtt{isEmpty}$ a function from S to $\mathtt{bool}$ testing whether a set is empty or not, and a function $\in$ from $E \times S$ to $\mathtt{bool}$ yielding true if the element e is in the set s and false otherwise. For all algebras A that satisfy the constraint

$$S \ \mathsf{partitioned\ by}\ \mathtt{isEmpty}, \in$$

and elements s_1 and s_2 of $A(S)$: if s_1 and s_2 are not equal, then they can be distinguished by using the operations $\mathtt{isEmpty}$ or $\in$, that is, either $\mathtt{isEmpty}(s_1)$ and not $\mathtt{isEmpty}(s_2)$ or vice versa, or there exists an element e of $A(E)$ such that $e \in s_1$ and $e \notin s_2$ or $e \notin s_1$ and $e \in s_2$.

Thus we can assume the following deduction rule:

$$\text{if } \mathtt{isEmpty}(s_1) \Leftrightarrow \mathtt{isEmpty}(s_2) \text{ and } \forall e : E\ e \in s_1 \Leftrightarrow e \in s_2 \text{ then } s_1 = s_2.$$

This rule cannot be expressed as an equation even if we have the boolean operations with their usual meanings in our signature since the universal quantification of e appears on the left side of an implication. Replacing the implication by a disjunction we get an existential quantifier

$$\forall s_1, s_2 : S\ \ \mathtt{isEmpty}(s_1) \not\Leftrightarrow \mathtt{isEmpty}(s_2) \vee \exists e : E.\ e \in s_1 \not\Leftrightarrow e \in s_2 \vee s_1 = s_2.$$

Data Constraints

In the following we add data-constraints to $\mathcal{EQ}$ using the method of Goguen and Burstall [25].

Definition 3.22 (Σ-generating constraints) *Let $\Sigma' = (S', \Omega')$ be a signature, s a sort from S', $\Omega_c \subseteq \Omega'$ a set of operations and σ a signature morphism from Σ' to $\Sigma = (S, \Omega)$. The following are Σ-constraints:*

$$s \ \textsf{generated by} \ \Omega_c \ \textsf{wrt} \ \sigma$$
$$s \ \textsf{generated freely by} \ \Omega_c \ \textsf{wrt} \ \sigma$$

If σ is the identity we write

$$s \ \textsf{generated (freely) by} \ \Omega_c$$

instead of s generated (freely) by Ω_c wrt `id`.

Definition 3.23 (Satisfaction) *Given a Σ-algebra A and a Σ-constraint*

$$c = s \ \textsf{generated (freely) by} \ \Omega_c \ \textsf{wrt} \ \sigma.$$

Then A satisfies c, $A \models c$, if $A|_\sigma$ satisfies the constraint

$$c' = s \ \textsf{generated (freely) by} \ \Omega_c \ \textsf{wrt} \ \texttt{id}.$$

Let Σ_c be (S, Ω_c). The Σ-algebra A satisfies the constraint s generated by Ω_c,

$$A \models s \ \textsf{generated by} \ \Omega_c,$$

if for all a in $A(s)$ there exists a term t in $\mathsf{T}_{\Sigma_c}(\bar{A})$ such that $\tilde{h}(t) = a$ where $\bar{A} \in \text{Set}^S$ is defined by $\bar{A}(s') = A(s')$ for every sort $s' \neq s$ in S and $\bar{A}(s) = \{\}$; and $h : \bar{A} \Rightarrow A$ is given by $h_{s'}(a) = a$ for every sort $s' \neq s$ in S and every $a \in \bar{A}(s')$ and h_s is the unique function from the empty set to $A(s)$.

A satisfies s generated freely by Ω_c if t is unique.

Definition 3.24 (Translation of constraints) *Let σ from Σ_1 to Σ_2 and μ from Σ to Σ_1 be two signature morphisms then*

$$\overline{\sigma}(s \ \textsf{generated (freely) by} \ \Omega_c \ \textsf{wrt} \ \mu) = s \ \textsf{generated (freely) by} \ \Omega_c \ \textsf{wrt} \ \mu; \sigma$$

Note that $\overline{\sigma}(c)$ is not defined as $\sigma(s)$ generated (freely) by $\sigma(\Omega_c)$ as it is done in the Larch Shared Language because the satisfaction condition would not hold. Consider a signature Σ with one sort and two constants c_1 and c_2 and a signature morphism from Σ to a signature Σ' with one sort and one constant c, mapping c_1 and c_2 to c. Let A be a Σ'-algebra and let $A(s)$ contain two elements then $A|_\sigma$ is generated by c_1 and c_2, however A is not generated by $\sigma(c_1) = c = \sigma(c_2)$.

THEOREM 3.25 *Given a Σ'-algebra A, a Σ-constraint s* generated by Ω_c wrt μ *and a signature morphism $\sigma : \Sigma \to \Sigma'$ then*

$$A \models \overline{\sigma}(s \text{ generated by } \Omega_c \text{ wrt } \mu; \sigma) \textit{ iff } A|_\sigma \models s \text{ generated by } \Omega_c \text{ wrt } \mu.$$

and similar for a constraint s generated freely by Ω_c wrt μ.

PROOF.

$$\begin{aligned} A \models s \text{ generated by } \Omega_c \text{ wrt } \mu;\sigma \ &\text{iff } A|_{(\mu;\sigma)} \models s \text{ generated by } \Omega_c \\ &\text{iff } (A|_\sigma)|_\mu \models s \text{ generated by } \Omega_c \\ &\text{iff } A|_\sigma \models s \text{ generated by } \Omega_c \text{ wrt } \mu \end{aligned}$$

□

THEOREM 3.26 *The satisfaction of data-constraints is closed under isomorphism.*

PROOF. Let A and A' be two isomorphic Σ-algebras with isomorphism $\iota : A \to A'$. We have to show that $A \models s$ generated (freely) by Ω_c if and only if $A' \models s$ generated (freely) by Ω_c.

Assume that $A \models s$ generated (freely) by Ω_c then we have to find for each $a' \in A'(s)$ a term $t' \in \mathsf{T}_\Sigma(\bar{A}')$ with $\tilde{h}'(t') = a'$ where $\bar{A}'(s') = A(s')$ for all sorts $s' \neq s$ and $\bar{A}'(s) = \{\}$, and $h'_{s'}(a) = a$ for all $a \in \bar{A}'$ and sorts $s' \neq s$ and h'_s is the unique morphism from $\{\}$ to $A'(s)$.

Since $A \models s$ generated (freely) by Ω_c and $\iota^{-1}(a')$ is in $A(s)$, there exists a term $t \in \mathsf{T}_\Sigma(\bar{A})$ such that $\tilde{h}(t) = \iota^{-1}(a')$ where $\bar{A}$ and $h : \bar{A} \to A$ are defined as in Definition 3.23. Let $\bar{\iota}$ be the natural transformation from $\bar{A}$ to $\bar{A}'$, given by $\bar{\iota}_{s'}(a) = \iota_{s'}(a)$ for every sort $s' \neq s$ and $\bar{\iota}_s$ is the unique function from the empty set to the empty set.

Choose t' as $\mathsf{T}_\Sigma(\bar{\iota})(t)$. It remains to show that $\tilde{h}'(t') = a'$.

The definition of h, h' and $\bar{\iota}$ makes the following diagram commute in SET^S:

$$\begin{tikzcd}
\bar{A} \arrow[r, "h"] \arrow[d, "\bar{\iota}"'] & A \arrow[d, "\iota"] \\
\bar{A}' \arrow[r, "h'"'] & A'
\end{tikzcd}$$

By the functor property of T_Σ this implies that the left rectangle of the following diagram in $\mathsf{Alg}(\Sigma)$ commutes:

$$\begin{tikzcd}
\mathsf{T}_\Sigma(\bar{A}) \arrow[r, "h"] \arrow[d, "\mathsf{T}_\Sigma(\bar{\iota})"'] & \mathsf{T}_\Sigma(A) \arrow[r, "\tilde{\mathtt{id}}_A"] \arrow[d, "\mathsf{T}_\Sigma(\iota)"] & A \arrow[d, "\iota"] \\
\mathsf{T}_\Sigma(\bar{A}') \arrow[r, "\mathsf{T}_\Sigma(h')"'] & \mathsf{T}_\Sigma(A') \arrow[r, "\tilde{\mathtt{id}}_{A'}"'] & A'
\end{tikzcd}$$

The right rectangle commutes because of Theorem 3.9. Also because of theorem 3.9, we have that $\tilde{h} = \mathsf{T}_\Sigma(h); \tilde{\mathtt{id}}_A$ and $\tilde{h}' = \mathsf{T}_\Sigma(h'); \tilde{\mathtt{id}}_{A'}$. Thus we get:

$$\tilde{h}'(t') = \tilde{h}'(\mathsf{T}_\Sigma(\bar{\iota})(t)) = \iota(\tilde{h}(t)) = \iota(\iota^{-1}(a')) = a'.$$

For free generating constraints, the uniqueness of t' is a consequence of the facts that ι and $\bar{\iota}$ are isomorphisms and that functors preserve isomorphisms.

The proof for the other direction, that is, if A' satisfies the data-constraint then A satisfies the data-constraint, is similar. □

Partitioned-By Constraints

A partitioned-by constraint s partitioned by Ω_o defines with Ω_o a set of *observers* that can be used to distinguish between elements of the sort s. An algebra A satisfies a partioned-by constraint if for two values a and b of sort s there exists an operation ω in Ω_o and values $a_1, \ldots, a_n$ such that $A(\omega)(a_1, \ldots, a, \ldots, a_n)$ does not equal $A(\omega)(a_1, \ldots, b, \ldots, a_n)$. On the other hand, if for all operations ω in Ω_o and values $a_1, \ldots, a_n$ we have

$$A(\omega)(a_1, \ldots, a, \ldots, a_n) = A(\omega)(a_1, \ldots, b, \ldots, a_n)$$

then $a = b$.

For example, when given a specification of sets, one would like to have each model A of that specification satisfy the constraint

$$\mathsf{Set\ partitioned\ by\ }\mathtt{isEmpty}, \in ,$$

as we would like to regard to sets s_1 and s_2 as equal, if either both sets are empty or not empty or if e is an element of s_1 then e is also an element of s_2 and vice versa.

Behavioral abstraction [49, 24, 9] is similar to the partioned-by constraint in that it defines observations used to distinguish elements from some sort. The difference is that with behavioral abstraction, in general, infinitely many observations have to be checked, while with the partitioned-by constraint only finitely many observations need to be checked. However, Bidoit and Hennicker [9] have shown that with the help of additional function symbols it is possible to check behavioral equivalence with finite many observations and to express behavioral equivalence using partitioned-by constraints.

Definition 3.27 (Partitioned-by) *Given a signature $\Sigma = (S, \Omega)$, a sort s in S and a set of operation symbols $\Omega_o \subseteq \Omega$. A partitioned-by constraint over Σ is of the form*

$$s \mathsf{\ partitioned\ by\ } \Omega_o$$

Given a signature morphism $\sigma : (S, \Omega) \to (S', \Omega')$ then the translation of s partitioned by Ω_o by σ is

$$\sigma(s) \mathsf{\ partitioned\ by\ } \sigma(\Omega_o).$$

Definition 3.28 (Satisfaction) *Given a signature Σ and Σ-algebra A then A satisfies the constraint s* partitioned by *Ω_o,*

$$A \models s \textsf{ partitioned by } \Omega_o,$$

if for all elements $a \neq b$ of $A(s)$ there exists an operation $\omega : s_1 \dots s_n \to s$ in Ω_o, elements $a_i \in A(s_i)$ for $i = 1 \dots n$ and an argument position $j \in \{1, \dots, n\}$ with $s_j = s$ such that

$$A(\omega)(a_1, \dots, a_{j-1}, a, a_{j+1}, \dots, a_n) \neq A(\omega)(a_1, \dots, a_{j-1}, b, a_{j+1}, \dots, a_n).$$

Theorem 3.29 *For all signature morphisms $\sigma : \Sigma \to \Sigma'$ and Σ'-algebras A we have:*

$$A|_\sigma \models s \textsf{ partitioned by } \Omega_o \textit{ iff } A \models \sigma(s) \textsf{ partitioned by } \sigma(\Omega_o).$$

Proof. ($\Rightarrow$) For two elements $a \neq b$ in $A(\sigma(s))$ we have to find an operation $\omega' : s'_1 \dots s'_n \to \sigma(s)$ in $\sigma(\Omega_o)$ and values $a'_i \in A(s'_i)$ such that

$$A(\omega')(a'_1, \dots, a, \dots, a'_n) \neq A(\omega')(a'_1, \dots, b, \dots, a'_n).$$

Since $A|_\sigma(s)$ is defined as $A(\sigma(s))$, the values a and b are in $A|_\sigma(s)$ and because $A|_\sigma$ satisfies s partitioned by Ω_o there exists $\omega : s_1 \dots s_n \to s$ and values $a_i \in A|_\sigma(s_i)$ with

$$A|_\sigma(\omega)(a_1, \dots, a, \dots, a_n) \neq A|_\sigma(\omega)(a_1, \dots, b, \dots, a_n).$$

Now choose ω' as $\sigma(\omega)$ and a'_i as a_i. Then

$$\begin{array}{c} A(\sigma(\omega))(a_1, \dots, a, \dots, a_n) = A|_\sigma(\omega)(a_1, \dots, a, \dots, a_n) \\ \neq \\ A|_\sigma(\omega)(a_1, \dots, b, \dots, a_n) = A(\sigma(\omega))(a_1, \dots, b, \dots, a_n) \end{array}$$

($\Leftarrow$) This direction is similar to $\Rightarrow$. However, for one operation ω' in $\sigma(\Omega_o)$ there may be several operations ω_i such that $\sigma(\omega_i) = \omega'$. This does not pose a problem because they have all the same interpretation, that is, $A|_\sigma(\omega_i) = A(\sigma(\omega_i)) = A(\omega')$. □

Theorem 3.30 *The satisfaction of partitioned-by constraints is closed under isomorphism.*

Proof. Let A and A' be isomorphic Σ-algebras where the isomorphism is $\iota : A \to A'$ then we have to show that $A \models s$ partitioned by Ω_o if and only if $A' \models s$ partitioned by Ω_o.

Assume that $A \models s$ partitioned by Ω_o. Then we have to find for every pair (a', b') from $A'(s)$ such that $a' \neq b'$ an operation $\omega' : s_1 \dots s_n \to s$ in Ω_o, elements $a'_i \in A'(s_i)$ and an argument position $j' \in \{1, \dots, n\}$ with

$$\begin{aligned} &A'(\omega')(a'_1, \dots, a'_{j'-1}, a', a'_{j'+1}, \dots, a'_n) \\ &\qquad\qquad \neq A'(\omega')(a'_1, \dots, a'_{j'-1}, b', a'_{j'+1}, \dots, a'_n). \end{aligned}$$

Let (a', b') be an arbitrary pair from $A'(s)$ with $a' \neq b'$ then $(\iota^{-1}(a'), \iota^{-1}(b'))$ is a pair of elements from $A(s)$ with $\iota^{-1}(a') \neq \iota^{-1}(b')$ because ι is an isomorphisms. Since $A \models s$ $\mathsf{partitioned\ by}$ Ω_o, we have an operation $\omega : s_1 \dots s_m \to s$ in Ω_o, elements $a_i \in A(s_i)$ and an argument position $j \in \{1, \dots, m\}$ with

$$\begin{aligned}&A(\omega)(a_1, \dots, a_{j-1}, \iota^{-1}(a'), a_{j+1}, \dots, a_m)\\ &\qquad\qquad\qquad \neq A(\omega)(a_1, \dots, a_{j-1}, \iota^{-1}(b'), a_{j+1}, \dots, a_m).\end{aligned}$$

Now set $\omega' = \omega$, $n = m$, $a'_i = \iota(a_i)$ for all $i \in \{1, \dots, m\}$ and $j' = j$. Then we have

$$\begin{aligned}&A'(\omega')(a'_1, \dots, a_{j'-1}, a', a'_{j'+1}, \dots, a'_n)\\ &\qquad = A'(\omega)(\iota(a_1), \dots, \iota(a_{j-1}), \iota(\iota^{-1}(a')), \iota(a_{j+1}), \dots, \iota(a_m))\\ &\qquad\qquad = \iota(A(\omega)(a_1, \dots, a_{j-1}, \iota^{-1}(a'), a_{j+1}, \dots, a_m))\end{aligned}$$

because $\iota; \iota^{-1} = \mathtt{id}_A$ and ι is a homomorphism, and similar

$$\begin{aligned}&A'(\omega')(a'_1, \dots, a'_{j'-1}, b', a'_{j'+1}, \dots, a'_n)\\ &\qquad\qquad = \iota(A(\omega)(a_1, \dots, a_{j-1}, \iota^{-1}(b'), a_{j+1}, \dots, a_m)).\end{aligned}$$

Since

$$\begin{aligned}&A(\omega)(a_1, \dots, a_{j-1}, \iota^{-1}(a'), a_{j+1}, \dots, a_m)\\ &\qquad\qquad\qquad \neq A(\omega)(a_1, \dots, a_{j-1}, \iota^{-1}(b'), a_{j+1}, \dots, a_m)\end{aligned}$$

and ι is an isomorphism, we have

$$A'(\omega')(a'_1, \dots, a''_{j'-1}, a', a'_{j+1}, \dots, a'_n) \neq A'(\omega')(a'_1, \dots, a_{j'-1}, b', a'_{j+1}, \dots, a'_n).$$

The proof that $A' \models s$ $\mathsf{partitioned\ by}$ Ω_o implies $A \models s$ $\mathsf{partitioned\ by}$ Ω_o is similar. □

The Institution $\mathcal{EQC}$

The institution $\mathcal{EQC}$ of equational logic with constraints is similar to $\mathcal{EQ}$; the difference being that data-constraints and partitioned-by constraints are added to the sentences of $\mathcal{EQ}$.

Definition 3.31 (Institution $\mathcal{EQC}$) *The institution $\mathcal{EQC}$ has* Sig *as its category of signatures,* Alg *as its structure functor and* EqnC *as the sentence functor where* EqnC(Σ) *is the union of* Eqn(Σ) *with the set of (free) generating and partitioned-by constraints over Σ. The satisfaction relation is the satisfaction relation of $\mathcal{EQ}$ extended by the satisfaction of constraints.*

The satisfaction condition of $\mathcal{EQC}$ is a consequence of Facts 3.16, 3.25 and 3.29.

Note that, because the category of signatures and the structure functor of $\mathcal{EQC}$ are the same as in the institution $\mathcal{EQ}$, the category of signatures of $\mathcal{EQC}$ is finitely cocomplete and $\mathcal{EQC}$ is exact.

3.4 The Institution $\mathcal{LSL}$

In the following section we define the institution $\mathcal{LSL}$, which is an approximation of the logic used by the Larch Shared Language [34]. Basically $\mathcal{LSL}$ is the institution $\mathcal{EQC}$ of equational logic with constraints. The differences are that each signature in $\mathcal{LSL}$ includes the sort `bool` together with its usual operations $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$; for each sort s of the signature the equality symbol as a boolean function $\equiv : s, s \to \mathtt{bool}$; and the conditional

$$\mathsf{if}\ \_\ \mathsf{then}\ \_\ \mathsf{else}\ \_ : \mathtt{bool}, s, s \to s.$$

Σ-structures in $\mathcal{LSL}$ are Σ-algebras such that the interpretation of the equality symbols, conditionals, and the sort `bool` with its operations have their usual meaning.

This allows to write formulas in $\mathcal{LSL}$ in a more natural form than possible in $\mathcal{EQC}$. If we adopt the convention that we write p instead of $p = \mathbf{true}$ for terms p of sort `bool`, then we can write formulas like $\forall X\ s_1 \equiv t_1 \Leftrightarrow s_0 \equiv t_0$, instead of $\forall X\ (s_1 \equiv t_1 \Leftrightarrow s_0 \equiv t_0) = \mathbf{true}$.

Note that $\equiv$ is different from $=$. While the first is a boolean function that can appear anywhere where a term of sort `bool` is expected, the second is the logical equality only allowed to build formulas $\forall X\ r = t$ from $\mathsf{EqnC}(\Sigma)$. But due to the restrictions we pose on the structures of $\mathcal{LSL}$, $\equiv$ and $=$ are interpreted as the identity, that is,

$$\begin{gathered} A \models \forall X\ (s \equiv t) = \mathbf{true} \\ \text{iff} \\ A \models \forall X\ s = t \\ \text{iff} \\ \tilde{\rho}(s) = \tilde{\rho}(t) \text{ for all } \rho : X \Rightarrow A. \end{gathered}$$

For this reason we shall use the symbol $=$ for both types of equality in later sections. For example, we shall write $\forall X\ s_1 = t_1 \Leftrightarrow s_0 = t_0$ instead of $\forall X\ (s_1 \equiv s_1 \Leftrightarrow s_0 \equiv t_0) = \mathbf{true}$.

Bool

The signature $\Sigma_{\mathtt{Bool}}$ is given by the sort `bool`, the constants **true** and **false** of sort `bool` and operation symbols $\Rightarrow$, $\Leftrightarrow$, $\wedge$ and $\vee$ of type $\mathtt{bool}, \mathtt{bool} \to \mathtt{bool}$ and $\neg$ of type $\mathtt{bool} \to \mathtt{bool}$. The set $\Phi_{\mathtt{Bool}}$ is given by the following list of equations and constraints.

```
bool freely generated by true, false
¬true = false
¬false = true
∀ b:bool  (true ∧ b) = b
∀ b:bool  (false ∧ b) = false
∀ b:bool  (false ∨ b) = b
∀ b:bool  (true ∨ b) = true
∀ b:bool  (true ⇒ b) = b
∀ b:bool  (false ⇒ b) = true
∀ b:bool  (true ⇔ b) = b
∀ b:bool  (false ⇔ b) = ¬b
```

The generating constraint ensures that if a $\Sigma_{\texttt{Bool}}$-algebra A satisfies $\Phi_{\texttt{Bool}}$ then $A(\texttt{bool}) = \{A(\mathbf{true}), A(\mathbf{false})\}$ and $A(\mathbf{true}) \neq A(\mathbf{false})$.

For all $\Sigma_{\texttt{bool}}$-algebras A satisfying $\Phi_{\texttt{Bool}}$ the operation symbols in $\Sigma_{\texttt{bool}}$ are interpreted as expected. For example the graph of $A(\wedge)$ is:

$$\begin{aligned} A(\wedge) = \{ \; & (A(\mathbf{true}), A(\mathbf{true}), A(\mathbf{true})), \\ & (A(\mathbf{true}), A(\mathbf{false}), A(\mathbf{false})), \\ & (A(\mathbf{false}), A(\mathbf{true}), A(\mathbf{false})), \\ & (A(\mathbf{false}), A(\mathbf{false}), A(\mathbf{false})) \; \}. \end{aligned}$$

This can be checked by choosing for each element in the graph of $A(\texttt{bool})$ an appropriate axiom from $\Phi_{\texttt{Bool}}$ and a variable assignment from the variables of the axiom to A. For example, to show that $(A(\mathbf{false}), A(\mathbf{true}), A(\mathbf{false}))$ is in the graph of $A(\wedge)$ we choose the axiom $\mathbf{false} \wedge b = \mathbf{false}$ and define ρ from $\{b : \texttt{bool}\}$ to A by $\rho(b) = A(\mathbf{true})$. Now we get

$$\tilde{\rho}(\mathbf{false} \wedge b) = A(\wedge)(A(\mathbf{false}), \rho(b)) = A(\mathbf{false}) = \tilde{\rho}(\mathbf{false})$$

and therefore $A(\wedge)(A(\mathbf{false}), A(\mathbf{true})) = A(\mathbf{false})$.

Theorem 3.32 *Let A be a Σ-algebra, σ a signature-morphism from $\Sigma_{\texttt{Bool}}$ to Σ such that $A|_\sigma \models \Phi_{\texttt{Bool}}$ and p and q two terms in $\mathsf{T}_\Sigma(X)_{\sigma(\texttt{bool})}$ then*

$$A \models \forall X \; p \wedge q = \mathbf{true} \textit{ iff } A \models \forall X \; p = \mathbf{true} \textit{ and } A \models \forall X \; q = \mathbf{true}.$$

Though it is possible that σ renames $\wedge$ and bool, we assume this is not the case to avoid writing $\sigma(\wedge)$ and $\sigma(\texttt{bool})$ all the time.

Proof. ($\Rightarrow$) Let ρ_1 and ρ_2 from X to A be two arbitrary variable assignments then we have to show that $\tilde{\rho}_1(p) = A(\mathbf{true})$ and $\tilde{\rho}_2(q) = A(\mathbf{true})$. Since $A \models \forall X \; p \wedge q = \mathbf{true}$ we have

$$\begin{aligned} \tilde{\rho}_1(p \wedge q) &= A(\wedge)(\tilde{\rho}_1(p), \tilde{\rho}_1(q)) \\ &= A(\mathbf{true}) \end{aligned}$$

which implies $\tilde{\rho}_1(p) = A(\mathbf{true})$ because the graph of $A(\wedge)$ is the same as the graph of $\wedge$. Similar we get $\tilde{\rho}_2(q) = A(\mathbf{true})$.

($\Leftarrow$) Let ρ be a variable assignment from X to A then we have to show that $A(\wedge)(\tilde{\rho}(p), \tilde{\rho}(q))$. Since we have $A \models \forall X \; p = \mathbf{true}$ and $A \models \forall X \; q = \mathbf{true}$ we have $\tilde{\rho}(p) = \mathbf{true}$ and $\tilde{\rho}(q) = \mathbf{true}$ and thus $A(\wedge)(\tilde{\rho}(p), \tilde{\rho}(q)) = A(\mathbf{true})$. □

Note that, in general, $A \models \forall X \; p \vee q = \mathbf{true}$ is not equivalent to $A \models \forall X \; p = \mathbf{true}$ or $A \models \forall X \; q = \mathbf{true}$. Consider: $\forall x \; x = 3 \vee x = 4$ and $(\forall x \; x = 3) \vee (\forall x \; x = 4)$. However, if we first rename x to y in the second equation and then from the disjunction then $\forall \{x, y\} \; x = 3 \vee y = 4$ and $(\forall x \; x = 3) \vee (\forall x \; x = 4)$ are equivalent.

Lemma 3.33 *Let A be a Σ-algebra, σ a signature-morphism from $\Sigma_{\texttt{Bool}}$ to Σ such that $A|_\sigma \models \Phi_{\texttt{Bool}}$, $p \in \mathsf{T}_\Sigma(X_1)_{\sigma(\texttt{bool})}$ and $q \in \mathsf{T}_\Sigma(X_2)_{\sigma(\texttt{bool})}$ then*

$$A \models \forall X\ p' \vee q' = \mathbf{true} \textit{ iff } A \models \forall X_1\ p = \mathbf{true} \textit{ or } A \models \forall X_2\ q = \mathbf{true},$$

where $X(s)$ is the disjoint union of $X_1(s)$ and $X_2(s)$ for each sort s of Σ and p' and q' are p and q translated by the injections ι_1 and ι_2 from X_1 and X_2 into X, respectively.

Proof. ($\Rightarrow$) Assuming that $\tilde{\rho}(p' \wedge q') = A(\mathbf{true})$ for all variable assignments $\rho : X \Rightarrow A$, we have to show that $\tilde{\rho}_1(p) = A(\mathbf{true})$ for all variable assignments $\rho_1 : X_1 \Rightarrow A$ or $\tilde{\rho}_2(q) = A(\mathbf{true})$ for all variable assignments $\rho_2 : X_2 \Rightarrow A$.

Let the variable assignment $\rho : X \Rightarrow A$ be the unique family of morphisms with the property $(\iota_1)_s; \rho_s = (\rho_1)_s$ and $(\iota_2)_s; \rho_s = (\rho_2)_s$ for every sort s in Σ, given by the universal property of the disjoint union. Then $\tilde{\rho}_1(p) = (\iota_1; \tilde{\rho})(p) = \tilde{\rho}(p')$ and $\tilde{\rho}_2(q) = \tilde{\rho}(q')$. Thus we have to show

$$\tilde{\rho}(p') = A(\mathbf{true}) \text{ or } \tilde{\rho}(p') = A(\mathbf{true}), \text{ for all } \rho : X \Rightarrow A.$$

Assume now that $A \models \forall X\ p' \vee q' = \mathbf{true}$ then $\tilde{\rho}(p' \vee q') = A(\mathbf{true})$, which is equivalent to $\tilde{\rho}(p') = A(\mathbf{true})$ or $\tilde{\rho}(q') = A(\mathbf{true})$.

($\Leftarrow$) Because of the universal property of disjoint union, the class of variable assignments ρ from X to A is the same the as the class of variable assignments ρ_1 from X_1 to A and ρ_2 from X_2 to A. Therefore all steps above are equivalent transformations. □

Equality

The signature $\Sigma_{\texttt{Eq}(s)}$ for some sort s includes the signature $\Sigma_{\texttt{bool}}$ and has as sorts s and `bool` with two operation symbols $\equiv$ and $\not\equiv$ of type $s, s \rightarrow$ `bool`. The set $\Phi_{\texttt{Eq}(s)}$ is given by the union of $\Phi_{\texttt{Bool}}$ with the following list of $\mathcal{EQC}$-formulas:

```
s partitioned by ≡
∀ x:s  (x ≡ x) = true
∀ x,y:s  (x ≡ y) = (y ≡ x)
∀ x,y:s  ((x ≡ y) ∧ (y ≡ z) ⇒ (x ≡ z)) = true
∀ x,y:s  (x ≢ y) = ¬(x ≡ y)
```

Note that the signature $\Sigma_{\texttt{Eq(bool)}}$ has only one sort, which is `bool`.

Note that the partitioned-by constraint cannot be replaced by the formula $\forall\{x, y : s\}.(x \equiv y) = \mathit{true} \Rightarrow x = y$ because this is not a formula of $\mathcal{EQC}$.

If an algebra A satisfies $\Phi_{\texttt{Eq}(s)}$ then the interpretation of $\equiv$ in A is the same as the identity, that is:

Theorem 3.34 *Let A be $\Sigma_{\texttt{Eq}(s)}$-algebra with $A \models \Phi_{\texttt{Eq}(s)}$ then*

$$A(\equiv)(a, b) = A(\mathit{true}) \textit{ if and only if } a = b,$$

for all a, b in $A(s)$.

PROOF. Note that $A(\equiv)$ can be viewed as relation in the following way:

$$(a,b) \in A(\equiv) \text{ iff } A(\equiv)(a,b) = A(\mathbf{true}).$$

Then $A(\equiv)$ is reflexive, transitive and symmetric because of the corresponding axioms of $\mathtt{Eq}(s)$.

($\Leftarrow$) This direction is trivial since $A(\equiv)(a,a) = A(\mathbf{true})$ holds because of the reflexivity axiom of $\mathtt{Eq}(s)$.

($\Rightarrow$) Assume that $a \neq b$ then we have to show that $A(\equiv)(a,b) \neq A(\mathbf{true})$. Note that this is equivalent to $A(\equiv)(a,b) = A(\mathbf{false})$ because $\Phi_{\mathtt{Eq}(s)}$ includes $\Phi_{\mathtt{Bool}}$. Since A models s partitioned by $\equiv$, we can find a value c from $A(s)$ with $A(\equiv)(c,a) = A(\mathbf{true})$ and $A(\equiv)(c,b) \neq A(\mathbf{true})$. Assume now that $A(\equiv)(a,b)$ is $A(\mathbf{true})$ then by transitivity of $A(\equiv)$ we get $A(c,b)$ is $A(\mathbf{true})$, which contradicts $A(c,b) \neq A(\mathbf{true})$. □

THEOREM 3.35 *Given a Σ-algebra A and a signature morphism σ from $\Sigma_{\mathtt{Eq}(s)}$ to Σ such that $A|_\sigma \models \Phi_{\mathtt{Eq}(s)}$ then*

$$A \models \forall X\ (t \equiv r) = \mathbf{true} \textit{ if and only if } A \models \forall X\ t = r.$$

PROOF. Assume that $\rho : X \Rightarrow A$ is a variable assignment then

$$\begin{aligned} \tilde{\rho}(s \equiv t) = A(\mathbf{true}) &\quad \text{iff} \\ A(\equiv)(\tilde{\rho}(s), \tilde{\rho}(t)) = A(\mathbf{true}) &\quad \text{iff} \\ \tilde{\rho}(s) = \tilde{\rho}(t) \end{aligned}$$

Thus we have $A \models \forall X\ (s \equiv t) = \mathbf{true}$ if and only if $A \models \forall X\ s = t$. □

Conditional

The signature $\Sigma_{\mathtt{Cond}(s)}$ contains for some sort s the signature $\Sigma_{\mathtt{bool}}$, the sorts s and `bool` and an operation if _ then _ else _ of type $\mathtt{bool}, s \to s$. $\Phi_{\mathtt{Cond}(s)}$ is the union of $\Phi_{\mathtt{Bool}}$ with:

```
∀ x,y:s  (if true then x else y) = x
∀ x,y:s  (if false then x else y) = y
```

Bse(S)

For a sort s, the signature $\Sigma_{\mathtt{Bse}(s)}$ is the union of the signatures $\Sigma_{\mathtt{Eq}(s)}$ and $\Sigma_{\mathtt{Cond}(s)}$. For a set of sorts S, the signature $\Sigma_{\mathtt{Bse}(S)}$ is the union of all signatures $\Sigma_{\mathtt{Bse}(s)}$ for $s \in S$. Similar $\Phi_{\mathtt{Bse}(s)}$ is the union of $\Phi_{\mathtt{Eq}(s)}$ and $\Phi_{\mathtt{Cond}(s)}$ for a sort s and $\Phi_{\mathtt{Bse}(S)}$ is the union of all $\Phi_{\mathtt{Bse}(s)}$ for $s \in S$.

FACT 3.36 *Let S be a set of sorts and ν be a signature morphism from $\Sigma_{\mathtt{Bool}(S)}$ to $\Sigma_{\mathtt{Bool}(\nu(S))}$, which is the identity on operation symbols and on the sort* `bool`. *Then*

$$\nu(\Phi_{\mathtt{Bse}(s)}) = \Phi_{\mathtt{Bse}(\nu(s))}.$$

Fact 3.37 *For two sets of sorts S_1 and S_2 we have*

$$\Sigma_{\mathtt{Bse}(S_1 \cup S_2)} = \Sigma_{\mathtt{Bse}(S_1)} \cup \Sigma_{\mathtt{Bse}(S_2)} \text{ and } \Phi_{\mathtt{Bse}(S_1 \cup S_2)} = \Phi_{\mathtt{Bse}(S_1)} \cup \Phi_{\mathtt{Bse}(S_2)}.$$

Signatures The category $\textsc{Sign}_{\mathcal{LSL}}$ is a subcategory of $\textsc{Sig}$ such that each $\Sigma = (S, \Omega)$ in $\textsc{Sign}_{\mathcal{LSL}}$ contains the sort `bool` and such that there exists a signature morphism ι_Σ from $\Sigma_{\mathtt{Bse}(S)}$ to Σ which is the identity on sorts. Note that, because `bool` is a sort of Σ, the signature $\Sigma_{\mathtt{Bse}(\mathtt{bool})}$ is included into $\Sigma_{\mathtt{Bse}(S)}$. In the following we shall use the notation $\iota_\Sigma^{\mathtt{bool}}$ for the corresponding inclusion morphism.

A signature morphism σ from $\Sigma = (S, \Omega)$ to $\Sigma' = (S', \Omega')$ in $\textsc{Sign}_{\mathcal{LSL}}$ is a signature morphism from $\textsc{Sig}$, that is, the identity on the sort `bool` and commutes with ι_Σ and $\iota_{\Sigma'}$ in the following way:

$$\begin{tikzcd}
\Sigma \arrow[rr, "\sigma"] & & \Sigma' \\
\Sigma_{\mathtt{Bse}(S)} \arrow[u, "\iota_\Sigma"] \arrow[rr, "\nu_\sigma"'] & & \Sigma_{\mathtt{Bse}(S')} \arrow[u, "\iota_{\Sigma'}"'] \\
& \Sigma_{\mathtt{Bse}(\mathtt{bool})} \arrow[ul, "\iota_\Sigma^{\mathtt{bool}}"] \arrow[ur, "\iota_{\Sigma'}^{\mathtt{bool}}"'] &
\end{tikzcd}$$

where ν_σ is the same as σ on sorts and the identity on the operation symbols.

Theorem 3.38 *The category* $\textsc{Sign}_{\mathcal{LSL}}$ *is cocomplete.*

Proof. Given a functor $F : J \to \textsc{Sign}_{\mathcal{LSL}}$ then we show that the colimit of F in $\textsc{Sign}_{\mathcal{LSL}}$ is the colimit of $F_\perp : J_\perp \to \textsc{Sig}$ in $\textsc{Sig}$. The difference between the functors $F_\perp$ and F is that $F_\perp$ has in addition to the nodes and edges of F a node $F_\perp(\perp)$ for $\Sigma_{\mathtt{Bse}(\mathtt{bool})}$ and edges from $\Sigma_{\mathtt{Bse}(\mathtt{bool})}$ to each Σ_i given by $\iota_{\Sigma_i}^{\mathtt{bool}}; \iota_{\Sigma_i}$. This ensures that the sort for boolean with its operations is included only once in the colimit of F.

Because colimits are only unique up to isomorphism, we choose $\coprod F_\perp = (S_{cl}, \Omega_{cl})$ such that $\mathtt{bool} \in S_{cl}$ and $\iota_i^{F_\perp}(\mathtt{bool}) = \mathtt{bool}$ for all $i \in J_\perp$.

The category $J_\perp$ is the category freely generated by adding $\perp$ to J and morphisms $f_i : \perp \to i$ for each $i \in J$ where $\perp$ and the f_i do not occur in J. Note that $\perp$ is not necessarily an initial object in $J_\perp$, as there is no guarantee that $f_i = f_j; f$, where f is a morphism from i to j in J.

Then $F_\perp$ is defined by

- $F_\perp(i) = F(i)$ for $i \in J$,
- $F_\perp(\perp) = \Sigma_{\mathtt{Bse}(\mathtt{bool})}$,
- $F_\perp(f : i \to j) = F(f)$ for $f : i \to j \in J$ and

- $F_\perp(f : \perp \to i) = \iota^{\mathtt{bool}}_{F(i)}$ for $i \in J$.

To show that $\coprod F_\perp = \Sigma_{cl} = (S_{cl}, \Omega_{cl})$ is the colimit of F we have to verify

1. that $\coprod F_\perp$ is in $\textsc{Sign}_{\mathcal{CSL}}$, for which we have to provide morphisms $\iota_{\Sigma_{cl}}$ from $\Sigma_{\mathtt{Bse}(S_{cl})}$ to Σ_{cl} and $\iota^{\mathtt{bool}}_{\Sigma_{cl}}$ from $\Sigma_{\mathtt{Bse}(\mathtt{bool})}$ to $\Sigma_{\mathtt{Bse}(S_{cl})}$,
2. that the co-cone morphisms $\iota^F_i = \iota^{F_\perp}_i$, for $i \in J$, are morphisms in $\textsc{Sign}_{\mathcal{CSL}}$ and
3. that for each natural transformation $\mu : F \Rightarrow \Delta(\Sigma, \Omega)$ there exists a unique morphism h_μ from $\coprod F$ to (Σ, Ω).

For 1) we define $\iota_{\Sigma_{cl}}$ from $\Sigma_{\mathtt{Bse}(\Sigma_{cl})}$ to Σ_{cl} as follows: $\iota_{\Sigma_{cl}}$ is the identity on sorts and maps an operation symbol $\omega : w \to s$ to $\iota^{F_\perp}_i(\iota_{\Sigma_i}(\omega_i : w_i \to s_i))$ where $\iota^{F_\perp}_i(\omega_i : w_i \to s_i) = \omega : w \to s$ and $\omega_i : w_i \to s_i$ is in $\Sigma_{\mathtt{Bse}(S_i)}$ for some $i \in J$. $\iota_{\Sigma_{cl}}$ is well-defined because $\Sigma_{\mathtt{Bse}(S_{cl})}$ is the union of $\Sigma_{\mathtt{Bse}(\iota^{F_\perp}_i(S_i))}$.

For 2) we have to show that the following family of diagrams commutes for each $i \in J$:

$$\begin{tikzcd}
\Sigma_i \arrow[r, "\iota^{F_\perp}_i"] & \Sigma_{cl} \\
\Sigma_{\mathtt{Bse}(S_i)} \arrow[u, "\iota_{\Sigma_i}"] \arrow[r, "\nu_{\iota^{F_\perp}_i}"'] & \Sigma_{\mathtt{Bse}(S_{cl})} \arrow[u, "\iota_{\Sigma_{cl}}"']
\end{tikzcd}$$

This follows directly from the definition of $\iota_{\Sigma_{cl}}$ and because $\nu_{\iota^{F_\perp}_i}$ is the identity on operation symbols and the same as $\iota^{F_\perp}_i$ on sorts.

For 3) define h_μ as the unique morphism given by the colimit property of $\coprod F_\perp$ and a natural transformation $\mu_\perp : F_\perp \Rightarrow \Delta\Sigma$. The natural transformation $\mu_\perp$ is defined by $(\mu_\perp)_i = \mu_i$ for $i \in J$ and $(\mu_\perp)_\perp = \iota^{\mathtt{bool}}_\Sigma ; \iota_\Sigma$. We have to show that h_μ is a morphism in $\textsc{Sign}_{\mathcal{CSL}}$, that is, we have to show that the following diagram commutes

$$\begin{tikzcd}
\Sigma_{cl} \arrow[r, "h_\mu"] & \Sigma \\
\Sigma_{\mathtt{Bse}(S_{cl})} \arrow[u, "\iota_{\Sigma_{cl}}"] \arrow[r, "\nu_{h_\mu}"'] & \Sigma_{\mathtt{Bse}(S)} \arrow[u, "\iota_\Sigma"']
\end{tikzcd}$$

where $\Sigma = (S, \Omega)$. The sort parts of the diagram commute because $\iota_{\Sigma_{cl}}$ and ι_Σ are the identity on sorts and ν_h is defined as h on sorts. The operation part is the same because the μ_i are required to be morphisms in $\textsc{Sign}_{\mathcal{CSL}}$.

□

The initial object of $\textsc{Sign}_{\mathcal{CSL}}$ is $\Sigma_{\mathtt{Bse}(\mathtt{bool})}$ because for any other signature Σ there exists a signature morphism $\iota^{\mathtt{bool}}_\Sigma ; \iota_\Sigma$ from $\Sigma_{\mathtt{Bse}(\mathtt{bool})}$ to Σ. This signature morphism is unique in $\textsc{Sign}_{\mathcal{CSL}}$ since if there were another signature morphism σ from $\Sigma_{\mathtt{Bse}(\mathtt{bool})}$ to Σ then, because of the definition of signature morphisms in $\textsc{Sign}_{\mathcal{CSL}}$, we have $\iota^{\mathtt{bool}}_{\Sigma_{\mathtt{Bse}(\mathtt{bool})}} ; \iota_{\Sigma_{\mathtt{Bse}(\mathtt{bool})}} ; \sigma = \iota^{\mathtt{bool}}_\Sigma ; \iota_\Sigma$. Since $\iota^{\mathtt{bool}}_{\Sigma_{\mathtt{Bse}(\mathtt{bool})}}$ and $\iota_{\Sigma_{\mathtt{Bse}(\mathtt{bool})}}$ are the identity this gives us $\sigma = \iota^{\mathtt{bool}}_\Sigma ; \iota_\Sigma$.

Structures Let Σ be a signature from $\textsc{Sign}_{\mathcal{LSL}}$ and let $\mathbb{B}$ be an arbitrary but fixed $\Sigma_{\mathtt{Bse(bool)}}$-algebra satisfying the formulas of $\Phi_{\mathtt{Bse(bool)}}$. The category $\mathsf{Str}_{\mathcal{LSL}}(\Sigma)$ of Σ-structures is the full subcategory of Σ-algebras A such that $A|_{\iota_\Sigma^{\mathtt{bool}};\iota_\Sigma} = \mathbb{B}$ and $A|_{\iota_\Sigma} \models^{\mathcal{EQC}} \Phi_{\mathtt{Bse}(S)}$ where S are the sorts of Σ. The functor $\mathsf{Str}_{\mathcal{LSL}}(\sigma)$ from $\mathsf{Str}_{\mathcal{LSL}}(\Sigma')$ to $\mathsf{Str}_{\mathcal{LSL}}(\Sigma)$ is the same as $\mathsf{Alg}(\sigma)$ for a signature morphism $\sigma : \Sigma \to \Sigma'$ in $\textsc{Sign}_{\mathcal{LSL}}$. For $\mathsf{Str}_{\mathcal{LSL}}(\sigma)$ to be well-defined we have to show that $A'|_\sigma$ is an object of $\mathsf{Str}_{\mathcal{LSL}}(\Sigma)$ for each algebra $A' \in \mathsf{Str}_{\mathcal{LSL}}(\Sigma')$.

That is, we have to show $A'|_{\iota_\Sigma^{\mathtt{bool}};\iota_\Sigma;\sigma} = \mathbb{B}$ and $A'|_{\iota_\Sigma;\sigma} \models^{\mathcal{EQC}} \Phi_{\mathtt{Bse}(S)}$, provided that $A'|_{\iota_{\Sigma'}^{\mathtt{bool}};\iota_{\Sigma'}} = \mathbb{B}$ and $A'|_{\iota_{\Sigma'}} \models^{\mathcal{EQC}} \Phi_{\mathtt{Bse}(S')}$ where S' are the sorts of Σ'.

Because σ is a signature morphism in $\textsc{Sign}_{\mathcal{LSL}}$, we have $\iota_\Sigma;\sigma = \nu_\sigma;\iota_{\Sigma'}$ and thus $A'|_{\iota_\Sigma;\sigma} = A'|_{\nu_\sigma;\iota_{\Sigma'}}$.

The satisfaction condition of $\mathcal{EQC}$ implies that $A'|_{\nu_\sigma;\iota_{\Sigma'}} \models^{\mathcal{EQC}} \Phi_{\mathtt{Bse}(S)}$ is equivalent to

$$A'|_{\iota_{\Sigma'}} \models^{\mathcal{EQC}} \nu_\sigma(\Phi_{\mathtt{Bse}(S)}),$$

which is the equivalent to $A'|_{\iota_{\Sigma'}} \models^{\mathcal{EQC}} \Phi_{\mathtt{Bse}(\nu_\sigma(S))}$ (cf. Fact 3.36).

By definition of $\Phi_{\mathtt{Bse}(S)}$ we have that $\Phi_{\mathtt{Bse}(\nu_\sigma(S))}$ is a subset of $\Phi_{\mathtt{Bse}(S')}$ because $\nu_\sigma(S)$ is a subset of S' and since $A'|_{\iota_{\Sigma'}} \models^{\mathcal{EQC}} \Phi_{\mathtt{Bse}(S')}$ we are done.

In addition we have $\iota_\Sigma^{\mathtt{bool}};\iota_\Sigma;\sigma = \iota_{\Sigma'}^{\mathtt{bool}};\iota_{\Sigma'}$ and thus

$$A'|_{\iota_\Sigma^{\mathtt{bool}};\iota_\Sigma;\sigma} = A'|_{\iota_{\Sigma'}^{\mathtt{bool}};\iota_{\Sigma'}} = \mathbb{B}.$$

Note that it is not enough to require that $A|_{\iota_\Sigma^{\mathtt{bool}};\iota_\Sigma}$ satisfies $\mathtt{Bse(bool)}$ for A to be in $\mathsf{Str}_{\mathcal{LSL}}(\Sigma)$ because in this case the functor $\mathsf{Str}_{\mathcal{LSL}}$ would not preserve the initial object, as the initial object of $\textsc{Sign}_{\mathcal{LSL}}$, $\Sigma_{\mathtt{Bse(bool)}}$, would be mapped to a category of $\Sigma_{\mathtt{Bse(bool)}}$-algebras containing isomorphic algebras. Requiring $A|_{\iota_\Sigma^{\mathtt{bool}};\iota_\Sigma}$ to be $\mathbb{B}$ ensures that $\mathsf{Str}_{\mathcal{LSL}}(\Sigma_{\mathtt{Bse(bool)}})$ contains only one object and one morphism — the identity — and thus is a terminal object of $\textsc{Cat}$.

Theorem 3.39 *The institution $\mathcal{LSL}$ has amalgamation.*

Proof. Consider a functor $F : \mathrm{V} \to \textsc{Sign}_{\mathcal{LSL}}$ with colimit $\Sigma_{cl} = (S_{cl}, \Omega_{cl})$ and co-cone morphisms $\iota_i^F : F(i) \to \Sigma_{cl}$:

$$\begin{tikzcd}
& \Sigma_{cl} & \\
F(1) \arrow[ur, "\iota_1^F"] & & F(2) \arrow[ul, "\iota_2^F"'] \\
& F(0) \arrow[ul, "F(f)"] \arrow[ur, "F(g)"'] &
\end{tikzcd}$$

Assume further a $F(1)$-algebra A_1 and a $F(2)$-algebra A_2 with $A_1|_{F(f)} = A_0 = A_2|_{F(g)}$ then we have to show the existence of a unique Σ_{cl}-algebra A with $A|_{\iota_1^F} = A_1$ and $A|_{\iota_2^F} = A_2$.

Σ_{cl} is the colimit of the functor $F_\perp : J_\perp \to \textsc{Sig}$ in $\textsc{Sig}$:

$$\begin{tikzcd}
& \Sigma_{cl} & \\
F(1) \arrow[ur, "\iota_1^F"] & & F(2) \arrow[ul, "\iota_2^F"'] \\
& F(0) \arrow[ul, "F(f)"'] \arrow[ur, "F(g)"] & \\
& \Sigma_{\texttt{Bse(bool)}} \arrow[u] \arrow[uul, "\iota^{\texttt{bool}}_{F(1)};\iota_{F(1)}"] \arrow[uur, "\iota^{\texttt{bool}}_{F(2)};\iota_{F(2)}"'] &
\end{tikzcd}$$

Note that we have $A_1|_{\iota^{\texttt{bool}}_{F(1)};\iota_{F(2)}} = \mathbb{B} = A_2|_{\iota^{\texttt{bool}}_{F(2)};\iota_{F(2)}}$ because A_1 and A_2 are algebras from $\mathsf{Str}_{\mathcal{LSL}}(F(1))$ and $\mathsf{Str}_{\mathcal{LSL}}(F(2))$, respectively. Since Alg preserves colimits, there exists a unique $\coprod F_\perp$-algebra A with $A|_{\iota_1^{F_\perp}} = A_1$ and $A|_{\iota_2^{F_\perp}} = A_2$. □

Theorem 3.40 *The contravariant functor* $\mathsf{Str}_{\mathcal{LSL}}$ *from* Sign *to* Cat *preserves colimits.*

Proof. Since $\mathcal{LSL}$ has amalgamation (cf. Lemma 3.39) we only have to show that the initial object of $\textsc{Sign}_{\mathcal{LSL}}$ is mapped to the terminal object of Cat.

The initial object in $\textsc{Sign}_{\mathcal{LSL}}$, $\Sigma_{\texttt{Bse(bool)}}$, is mapped to the class of algebras A with

$$A|_{\iota^{\texttt{bool}}_{\Sigma_{\texttt{Bse(bool)}}};\iota_{\Sigma_{\texttt{Bse(bool)}}}} = \mathbb{B}.$$

Since $\iota^{\texttt{bool}}_{\Sigma_{\texttt{Bse(bool)}}}$ and $\iota_{\Sigma_{\texttt{Bse(bool)}}}$ are the identity we have $A = \mathbb{B}$. As a consequence, the category $\mathsf{Str}_{\mathcal{LSL}}(\Sigma_{\texttt{Bse(bool)}})$ has only one element and one morphism (the identity) and is therefore a terminal object in Cat. □

Definition 3.41 (Institution $\mathcal{LSL}$) *The institution $\mathcal{LSL}$ has $\textsc{Sign}_{\mathcal{LSL}}$ as its category of signatures and $\mathsf{Str}_{\mathcal{LSL}}$ as its structure functor. The sentence functor $\mathsf{Sen}_{\mathcal{LSL}}$ is defined by $\mathsf{Sen}_{\mathcal{LSL}}(\Sigma) = \mathsf{EqnC}(\Sigma)$ and $\mathsf{Sen}_{\mathcal{LSL}}(\sigma) = \mathsf{EqnC}(\sigma)$ for every signature Σ and every signature morphism σ in $\textsc{Sign}_{\mathcal{LSL}}$, and the family of satisfaction relations $({\models^{\mathcal{LSL}}}_\Sigma)_{\Sigma \in \textsc{Sign}_{\mathcal{LSL}}}$ is given by $A \models^{\mathcal{LSL}}{}_\Sigma \varphi$ if and only if $A \models^{\mathcal{EQC}}_\Sigma \varphi$ for all signatures $\Sigma \in \textsc{Sign}_{\mathcal{LSL}}$, algebras $A \in \mathsf{Str}_{\mathcal{LSL}}(\Sigma)$ and formulas $\varphi \in \mathsf{Sen}_{\mathcal{LSL}}(\Sigma)$.*

The satisfaction condition holds for $\mathcal{LSL}$ because $\mathsf{Str}_{\mathcal{LSL}}(\Sigma)$ is a subcategory of $\mathsf{Alg}(\Sigma)$, $\mathsf{Sen}_{\mathcal{LSL}}(\Sigma)$ is the same as $\mathsf{EqnC}(\Sigma)$ and $A \models^{\mathcal{LSL}} \varphi$ if and only if $A \models^{\mathcal{EQC}} \varphi$ for every $A \in \mathsf{Str}_{\mathcal{LSL}}(\Sigma)$, $\varphi \in \mathsf{Sen}_{\mathcal{LSL}}(\Sigma)$ and $\Sigma \in \textsc{Sign}_{\mathcal{LSL}}$.

As noted before, in case where p is a term of sort `bool`, we shall write $\forall X\ p$ as an abbreviation of $\forall X\ p = \mathbf{true}$.

The following theorem establishes the relation between the consequence relation in $\mathcal{LSL}$ and the consequence relation in $\mathcal{EQC}$. The idea is that the set

$$\{A \in \mathsf{Str}_{\mathcal{LSL}}(\Sigma) \mid A \models \Phi\}$$

is *almost* the same as

$$\{A \in \mathsf{Alg}(\Sigma) \mid A \models \Phi \cup \iota_\Sigma(\Phi_{\mathtt{Bse}(S)})\}.$$

Almost means that, while the second set is closed under isomorphisms with respect to $\mathsf{Alg}(\Sigma)$, the first is not closed under isomorphisms with respect to $\mathsf{Alg}(\Sigma)$ because $A|_{\iota_\Sigma}$ is fixed to be $\mathbb{B}$. Note however that the first set is still closed under isomorphisms with respect to $\mathsf{Str}_{\mathcal{LSL}}(\Sigma)$.

Theorem 3.42 *Suppose we are given a set of Σ-formulas Φ and a Σ-formula φ where Σ has sorts S. Then*

$$\Phi \models^{\mathcal{LSL}} \varphi \textit{ if and only if } \Phi \cup \iota_\Sigma(\Phi_{\mathtt{Bse}(S)}) \models^{\mathcal{EQC}} \varphi.$$

Proof. ($\Leftarrow$) Assume that $A \in \mathsf{Str}_{\mathcal{LSL}}(\Sigma)$ and $A \models^{\mathcal{LSL}} \Phi$ then, because $A \in \mathsf{Str}_{\mathcal{LSL}}(\Sigma)$, we have that $A|_{\iota_\Sigma} \models^{\mathcal{EQC}} \Phi_{\mathtt{Bse}(S)}$ and $A \models^{\mathcal{EQC}} \Phi$. Since $\Phi \cup \iota_\Sigma(\Phi_{\mathtt{Bse}(S)}) \models^{\mathcal{EQC}} \varphi$ we get $A \models^{\mathcal{EQC}} \varphi$, which implies $A \models^{\mathcal{LSL}} \varphi$.

($\Rightarrow$) Assume that $\Phi \models^{\mathcal{LSL}} \varphi$, $A \in \mathsf{Alg}(\Sigma)$ and $A \models^{\mathcal{EQC}} \Phi \cup \iota_\Sigma(\Phi_{\mathtt{Bse}(S)})$. We have to show that $A \models^{\mathcal{EQC}} \varphi$. To use $\Phi \models^{\mathcal{LSL}} \varphi$, we have to find an algebra A' and in $\mathsf{Str}_{\mathcal{LSL}}(\Sigma)$, such that $A' \models^{\mathcal{LSL}} \Phi$. Then we have $A' \models^{\mathcal{LSL}} \varphi$. Because of the definition of $\models^{\mathcal{LSL}}$ this is equivalent to $A' \models^{\mathcal{EQC}} \varphi$. If we choose A' such that A' is isomorphic to A then $A' \models^{\mathcal{EQC}} \varphi$ implies $A \models^{\mathcal{EQC}} \varphi$ because $\models^{\mathcal{EQC}}$ is closed under isomorphisms (cf. Lemmata 3.13, 3.26 and 3.30).

In the following, we shall abbreviate $\iota_\Sigma^{\mathtt{bool}}; \iota_\Sigma$ by σ.

The carrier sets of A' are defined by $A'(\sigma(\mathtt{bool})) = \mathbb{B}$ and $A'(s) = A(s)$ for every $s \neq \sigma(\mathtt{bool}) \in S$. The natural transformation $\iota : A' \Rightarrow A$ is the identity on all sorts $s \neq \mathtt{bool}$ and maps **true** to $A(\mathbf{true})$ and **false** to $A(\mathbf{false})$ for $\mathtt{bool}$. Note that $\iota_{\mathtt{bool}}$ and therefore ι is an isomorphism because, since A satisfies $\iota_\Sigma(\Phi_{\mathtt{Bse}(S)})$, we have $A(\mathtt{bool}) = \{A(\mathbf{true}), A(\mathbf{false})\}$. Then we define for operations $\omega : w \to s$ in Ω:

$$A'(\omega)(a_1, \ldots, a_n) = \iota^{-1}(A(\omega)(\iota(a_1), \ldots, \iota(a_n))).$$

ι is a homomorphism because

$$\begin{aligned} \iota(A'(\omega)(a_1, \ldots, a_n)) &= \iota(\iota^{-1}(A(\omega)(\iota(a_1), \ldots, \iota(a_n)))) \\ &= A(\omega)(\iota(a_1), \ldots, \iota(a_n)). \end{aligned}$$

It remains to show that A' is in $\mathsf{Str}_{\mathcal{LSL}}(\Sigma)$ and that $A' \models^{\mathcal{LSL}} \Phi$. Since A satisfies $\Phi \cup \iota_\Sigma(\Phi_{\mathtt{Bse(bool)}})$ and $\models^{\mathcal{EQC}}$ is closed under isomorphisms we get $A' \models^{\mathcal{EQC}} \Phi$ and $A' \models^{\mathcal{EQC}} \iota_\Sigma(\Phi_{\mathtt{Bse(bool)}})$, which is equivalent to $A'|_{\iota_\Sigma} \models^{\mathcal{EQC}} \Phi_{\mathtt{Bse(bool)}}$ because of the satisfaction condition of $\mathcal{EQC}$. Since by construction we have $A'|_{\iota_\Sigma^{\mathtt{bool}}; \iota_\Sigma} = \mathbb{B}$, we have $A' \in \mathsf{Str}_{\mathcal{LSL}}(\Phi)$, and by the definition of $\models^{\mathcal{LSL}}$ we have $A' \models^{\mathcal{LSL}} \Phi$. □

4 Specifications of Abstract Datatypes

4.1 Abstract Datatypes

Traditionally, an abstract datatype (Σ, M) is a specification of a datatype in a software system. The signature Σ defines the external interface as a collection of sort and function symbols, and M is a class of Σ-algebras considered admissible implementations of that datatype.

In the context of an arbitrary institution $\mathcal{I}$ an *abstract datatype* is a pair (Σ, M) where Σ is an element of $\textsc{Sign}_{\mathcal{I}}$ and M is a full subcategory of $\mathsf{Str}_{\mathcal{I}}(\Sigma)$. Depending on the institution, Σ may be a many sorted signature with sorts and function symbols, and M a category of Σ-algebras; or it may denote something totally different. The main construction of this thesis is the definition of an institution such that abstract datatypes in this institution are relations between abstract datatypes from another institution (cf. Section 5.3).

The *category of abstract datatypes* $\textsc{Adt}_{\mathcal{I}}$ has as objects abstract datatypes and as morphisms $\sigma : (\Sigma, M) \to (\Sigma', M')$ signature morphisms $\sigma : \Sigma \to \Sigma'$ such that $m'|_\sigma$ is in M for all m' in M'.

A structure m of $\mathsf{Str}_{\mathcal{I}}(\Sigma)$ is a *model* of (Σ, M) if m is in M. In this case we may write $m \models^{\mathcal{I}} (\Sigma, M)$. An abstract datatype is called *monomorphic* if all its models are isomorphic. As before we shall omit the subscript $\mathcal{I}$ if the institution is clear from the context.

Note that we do not require the category of models of an abstract datatype to be closed under isomorphism as done, for example, in Loeckx, Ehrich and Wolf [42]. One reason is that nothing in the definition of an institution guarantees that a class of structures satisfying a set of formulas is closed with respect to isomorphism, though, using the institution $\mathcal{EQ}$, $\mathcal{EQC}$ or $\mathcal{LSL}$ they will. A second reason is that the derive operation introduced in Section 4.2 does not necessarily preserve isomorphism classes, for example, in $\mathcal{EQ}$ this is the case if the signature morphism is not injective on sorts (cf. Sannella and Tarlecki [50]).

Two forgetful functors are associated with the category $\textsc{Adt}$: the covariant functor $\mathsf{Sig}^{\mathcal{A}}$ from $\textsc{Adt}$ to $\textsc{Sign}$ and the contravariant functor $\mathsf{Mod}^{\mathcal{A}}$ from $\textsc{Adt}$ to $\textsc{Cat}$. The functor $\mathsf{Sig}^{\mathcal{A}}$ maps abstract datatypes to their signatures and abstract datatype morphisms to the their signature morphisms while the functor $\mathsf{Mod}^{\mathcal{A}}$ maps abstract datatypes to their category of models and an abstract datatype morphism $\sigma : (\Sigma, M) \to (\Sigma', M')$ to the functor $\_|_\sigma : \mathsf{Str}(\Sigma') \to \mathsf{Str}(\Sigma)$ restricted to M'.

$\mathsf{Sig}^{\mathcal{A}}$ creates colimits, and if $\mathsf{Str}_{\mathcal{I}}$ preserves colimits, then so does $\mathsf{Mod}^{\mathcal{A}}$. The fact that $\mathsf{Sig}^{\mathcal{A}}$ creates colimits implies that one can paste together a collection of abstract datatypes in a unique way if one can paste together their signatures.

FACT 4.1 *The functor* $\mathsf{Sig}^{\mathcal{A}} : \textsc{Adt}_{\mathcal{I}} \to \textsc{Sign}_{\mathcal{I}}$ *creates colimits.*

PROOF. Let $F : J \to \textsc{Adt}$ be a functor such that the functor $F; \mathsf{Sig}^{\mathcal{A}} : J \to \textsc{Sign}$ has a colimit $(\iota^{\mathsf{F};\mathsf{Sig}^{\mathcal{A}}}, \coprod(F; \mathsf{Sig}^{\mathcal{A}}))$ in $\textsc{Sign}$. We have to show that there exists a colimit $(\iota^F, \coprod F)$ in $\textsc{Adt}$ with

$$\mathsf{Sig}^{\mathcal{A}}(\coprod F) = \coprod(F; \mathsf{Sig}^{\mathcal{A}}) \text{ and } \mathsf{Sig}^{\mathcal{A}}(\iota^F) = \iota^{F;\mathsf{Sig}^{\mathcal{A}}}.$$

Let $F(i) = (\Sigma_i, M_i)$ for each $i \in J$ and $(\iota^{F;\mathsf{Sig}^{\mathcal{A}}}, \Sigma_{cl})$ be a colimit of $F; \mathsf{Sig}^{\mathcal{A}}$. A colimit of F is given by the co-cone morphisms ι^F defined by $\mathsf{Sig}^{\mathcal{A}}(\iota^F) = \iota^{F;\mathsf{Sig}^{\mathcal{A}}}$ and the colimit object (Σ_{cl}, M_{cl}) where M_{cl} is the full subcategory of $\mathsf{Str}(\Sigma_{cl})$ with objects

$$\{m \in \mathsf{Str}(\Sigma_{cl}) \mid \forall i \in J\ m|_{\iota_i^F} \in M_i\}.$$

To show that the abstract datatype (Σ_{cl}, M_{cl}) with co-cone morphisms ι^F is indeed a colimit of F we have to show that

1. $\iota_i^F : (\Sigma_i, M_i) \to (\Sigma_{cl}, M_{cl})$ is an $\textsc{Adt}$-morphisms for $i \in J$ and that
2. for each natural transformation $\mu : F \Rightarrow \Delta(\Sigma, M)$ there exists a unique $\textsc{Adt}$-morphism h_μ from (Σ_{cl}, M_{cl}) to (Σ, M) such that

   $$\iota_i^F; h_\mu = \mu_i$$

   for all $i \in J$.

(1) The reduct $m|_{\iota_i^F}$ is in M_i for each $m \in M_{cl}$ and $i \in J$ by the definition of M_{cl}.

(2) Note that $\mu; \mathsf{Sig}^{\mathcal{A}}$ is a natural transformation from $F; \mathsf{Sig}^{\mathcal{A}}$ to $\Delta(\Sigma)$, thus there exists a unique signature morphism h_μ from the colimit Σ_{cl} of $F; \mathsf{Sig}^{\mathcal{A}}$ to Σ with $\iota_i^{F;\mathsf{Sig}^{\mathcal{A}}}; h_\mu = (\mu; \mathsf{Sig}^{\mathcal{A}})_i$ for all $i \in J$.

It remains to show that the signature morphism h_μ is also an $\textsc{Adt}$-morphism, that is, $m|_{h_\mu}$ is in M_{cl} for all $m \in M$. The reduct $m|_{h_\mu}$ is in M_{cl} if $(m|_{h_\mu})|_{\iota_i^{F;\mathsf{Sig}^{\mathcal{A}}}}$ is in M_i for all $i \in J$ by the definition of M_{cl}. Because μ_i is an $\textsc{Adt}$-morphism we have $m|_{\mu_i}$ is in M_i, and because $\mu_i = \iota_i^{F;\mathsf{Sig}^{\mathcal{A}}}; h_\mu$ we get

$$(m|_{h_\mu})|_{\iota_i^{F;\mathsf{Sig}^{\mathcal{A}}}} = m|_{\iota_i^{F;\mathsf{Sig}^{\mathcal{A}}};h_\mu} = m|_{\mu_i}.$$

□

A diagram $F : J \to \textsc{Adt}$ has a colimit if the diagram $F; \mathsf{Sig}^{\mathcal{A}} : J \to \textsc{Sign}_{\mathcal{I}}$ has a colimit. Thus, if every diagram in $\textsc{Sign}$ has a colimit, then every diagram in $\textsc{Adt}$ has a colimit (cf. Theorem 2.11).

COROLLARY 4.2 *The category* $\textsc{Adt}_{\mathcal{I}}$ *of abstract datatypes is (finitely) co-complete if the category of signatures* $\textsc{Sign}_{\mathcal{I}}$ *is (finitely) co-complete.*

The initial object in $\textsc{Adt}$ is $(\Sigma_\perp, \mathsf{Str}(\Sigma_\perp))$ where $\Sigma_\perp$ is the initial object in $\textsc{Sign}$.

Note that it could well be that the colimit of a functor $F : J \to \textsc{Adt}_\mathcal{I}$ exists while $F; \mathsf{Sig}^\mathcal{A}$ has no colimit. However, in the following theorem we prove that $\mathsf{Sig}^\mathcal{A}$ has a right adjoint which implies that $\mathsf{Sig}^\mathcal{A}$ preserves colimits (c.f. Theorem 2.25).

Theorem 4.3 *Let U_{Sig} be the functor from $\textsc{Sign}_\mathcal{I}$ to $\textsc{Adt}_\mathcal{I}$ defined by*

$$U_{\mathsf{Sig}}(\Sigma) = (\Sigma, \{\}) \textit{ and } U_{\mathsf{Sig}}(\sigma) = \sigma.$$

Then U_{Sig} is a right adjoint for $\mathsf{Sig}^\mathcal{A}$.

Proof. The unit of the adjunction is the natural transformation $\eta : \mathsf{Id}_{\textsc{Adt}_\mathcal{I}} \Rightarrow \mathsf{Sig}^\mathcal{A}; U_{\mathsf{Sig}}$ where $\eta_{(\Sigma,M)} : (\Sigma, M) \to (\Sigma, \{\})$ is given by the identity on Σ.

If σ is an $\textsc{Adt}_\mathcal{I}$-morphism from (Σ, M) to $U_{\mathsf{Sig}}(\Sigma')$ for some signature Σ', then the unique morphism $\tilde{\sigma}$ from (Σ, M) to $(\Sigma', \{\})$ is given by the signature morphism σ. Since $\eta_{(\Sigma,M)}$ is the identity on Σ we get $\eta_{(\Sigma,M)}; \mathsf{Sig}^\mathcal{A}(\tilde{\sigma}) = \sigma$. □

Corollary 4.4 *The functor $\mathsf{Sig}^\mathcal{A}$ preserves colimits.*

If the contravariant functor $\mathsf{Str}_\mathcal{I}$ from $\textsc{Sign}_\mathcal{I}$ to $\textsc{Cat}$ preserves colimits, so does the contravariant functor $\mathsf{Mod}^\mathcal{A}$ from $\textsc{Adt}$ to $\textsc{Cat}$. This means that every model m of the colimit of a functor $F : J \to \textsc{Adt}_\mathcal{I}$ is uniquely determined by a family of models $m_i \in \mathsf{Mod}^\mathcal{A}(F(i))$ for each $i \in J$. This is the amalgamation lemma of Ehrig and Mahr [19].

Fact 4.5 *The contravariant functor $\mathsf{Mod}^\mathcal{A}$ from $\textsc{Adt}_\mathcal{I}$ to $\textsc{Cat}$ preserves (finite) colimits if the contravariant functor $\mathsf{Str}_\mathcal{I}$ from $\textsc{Sign}_\mathcal{I}$ to $\textsc{Cat}$ does.*

Proof. Let F from J to $\textsc{Adt}$ be a functor with colimit $(\iota^F, (\Sigma_{cl}, M_{cl}))$. To show that $(\mathsf{Mod}^\mathcal{A}(\iota^F), \mathsf{Mod}^\mathcal{A}((\Sigma_{cl}, M_{cl}))$ is a limit of $F; \mathsf{Mod}^\mathcal{A}$ consider a natural transformation $\mu : \Delta C \Rightarrow F; \mathsf{Mod}^\mathcal{A}$ where C is a category. Let $F(i) = (\Sigma_i, M_i)$. Since, by the definition of abstract datatypes, $\mathsf{Mod}^\mathcal{A}(F(i))$ is a subcategory of $\mathsf{Str}_\mathcal{I}(\mathsf{Sig}^\mathcal{A}(F(i)))$, μ is also a natural transformation from ΔC to $F; \mathsf{Sig}^\mathcal{A}; \mathsf{Str}_\mathcal{I}$. Because $\mathsf{Str}_\mathcal{I}$ and $\mathsf{Sig}^\mathcal{A}$ preserve colimits, the category $\mathsf{Str}_\mathcal{I}(\mathsf{Sig}^\mathcal{A}((\Sigma_{cl}, M_{cl}))) = \mathsf{Str}_\mathcal{I}(\Sigma_{cl})$ is a limit object of $F; \mathsf{Sig}^\mathcal{A}; \mathsf{Str}_\mathcal{I}$ and therefore there exists a unique functor h_μ from C to $\mathsf{Str}_\mathcal{I}(\Sigma_{cl})$ with

$$h_\mu(c)|_{\iota_i^F} = \mu_i(c)$$

for each $c \in C$ and $i \in J$.

It remains to show that h_μ is also a functor from C to M_{cl}, that is, $h_\mu(c)$ is an object of M_{cl} and $h_\mu(f)$ is a morphism of M_{cl} for all objects c and morphisms f in C. Let c be an object of C, then $h_\mu(c)|_{\iota_i^F}$ equals $\mu_i(c)$, which is an element of $\mathsf{Mod}^\mathcal{A}(F(i))$ because of the definition of μ. If f is a morphism in C, then a similar argument shows that $h_\mu(f)$ is a morphism in M_{cl}. □

4.2 The Specification Language $\mathtt{SL}_{\mathcal{I}}$

In this section we are going to define an ASL-like specification language $\mathtt{SL}_{\mathcal{I}}$ based on the operations Σ (signature), I_Φ (impose), D_σ (derive), T_σ (translate), and $+$ (union) (cf. Sannella and Wirsing [53]). The semantics of an $\mathtt{SL}_{\mathcal{I}}$-expression is an abstract datatype in $\textsc{Adt}_{\mathcal{I}}$. A signature Σ, as a specification expression, denotes the abstract datatype with signature Σ and with the category of models the category of Σ-structures.

Impose allows to impose additional requirements on a specification. The semantics of an expression $I_\Phi\textsc{Sp}$ is the abstract datatype with the same signature as the signature of the abstract datatype denoted by $\textsc{Sp}$ and with models all the models of $\textsc{Sp}$ satisfying the set of formulas Φ.

The translate operation can be used to rename symbols in a signature, but also to add new symbols to a signature. If σ is a signature morphism from Σ to Σ', then the expression $T_\sigma\textsc{Sp}$ denotes an abstract datatype of signature Σ' with models all extensions of models of $\textsc{Sp}$ with respect to $\mathsf{Str}_{\mathcal{I}}(\sigma)$.

The derive operation allows to hide parts of a signature. $D_\sigma\textsc{Sp}$ denotes the abstract datatype having as signature the domain of σ and as models the translations of the models of $\textsc{Sp}$ by $\mathsf{Str}_{\mathcal{I}}(\sigma)$.

At last, the union operation is used to combine two specifications. Since for arbitrary institutions the union of signatures is not defined, we have to require that both specifications have the same signature.

The semantics of $\textsc{Sp}_1 + \textsc{Sp}_2$ is the abstract datatype with signature the signature of $\textsc{Sp}_1$, which is the same as the signature of $\textsc{Sp}_2$, and as models the intersection of the category of models of $\textsc{Sp}_1$ and $\textsc{Sp}_2$.

To form the union of two specifications $\textsc{Sp}_1$ and $\textsc{Sp}_2$ of different signatures Σ_1 and Σ_2, one has to provide a signature Σ, signature morphisms $\sigma_1 : \Sigma_1 \to \Sigma$ and $\sigma_2 : \Sigma_2 \to \Sigma$, and write $T_{\sigma_1}\textsc{Sp}_1 + T_{\sigma_2}\textsc{Sp}_2$. In an institution where the union of two signatures is defined, like $\mathcal{EQ}$ (cf. Section 3.1), Σ could be the union of Σ_1 and Σ_2. In this case σ_i is the inclusion of Σ_i into Σ for $i \in \{1,2\}$.

In Chapter 8 we add disjunction ($\textsc{Sp}_1 \vee \textsc{Sp}_2$) to $\mathtt{SL}_{\mathcal{I}}$ having as semantics the union of the category of models of $\textsc{Sp}_1$ and $\textsc{Sp}_2$. This operation will be useful in the construction of new relations (cf. the example in Chapter 8).

The specification language $\mathtt{SL}_{\mathcal{I}}$ is mainly intended for theoretical studies and not as a specification language used in practice. However, the semantics of more practical specification languages like Pluss [8], CASL [45], and the Larch Shared Language [34] can be given in the terms of the constructs of $\mathtt{SL}_{\mathcal{I}}$.

Note that $\mathtt{SL}_{\mathcal{I}}$ does not contain constructs for behavioral abstraction and restriction to minimal models, which, for example, can be found by Sannella and Tarlecki [50] or Farrés-Casals

[21]. We assume that these issues are dealt with on the level of the institution itself by means of appropriate constraints, like generating and partitioned-by constraints (cf. Section 3.3).

In Section 5.9 we use $\mathtt{SL}_\mathcal{I}$ as the basis for the language $\mathtt{RSL}$ used for the construction of relations.

Definition 4.6 (Abstract Syntax of $\mathtt{SL}_\mathcal{I}$) *Given an institution $\mathcal{I}$, signatures Σ, Σ_1, and Σ_2, the abstract syntax of the specification language $\mathtt{SL}_\mathcal{I}$ is given by:*

$$\text{Sp} ::= \Sigma \mid I_\Phi\text{Sp} \mid D_\sigma\text{Sp} \mid T_\sigma\text{Sp} \mid \text{Sp} + \text{Sp}$$

where Φ is a finite set of Σ-formulas and σ is a signature morphism. The signature $\mathsf{Sig}(\text{Sp})$ of a specification expression Sp is defined inductively as follows:

$$\begin{aligned} \mathsf{Sig}(\Sigma) &= \Sigma & \mathsf{Sig}(I_\Phi\text{Sp}) &= \mathsf{Sig}(\text{Sp}) \\ \mathsf{Sig}(D_\sigma\text{Sp}) &= \mathsf{dom}(\sigma) & \mathsf{Sig}(T_\sigma\text{Sp}) &= \mathsf{cod}(\sigma) \\ \mathsf{Sig}(\text{Sp}_1 + \text{Sp}_2) &= (\mathsf{Sig}(\text{Sp}_1) = \mathsf{Sig}(\text{Sp}_2)) \end{aligned}$$

A specification expression Sp from $\mathtt{SL}_\mathcal{I}$ is well-formed *if*

- $\text{Sp} = \Sigma$ *and* $\Sigma \in \text{Sign}_\mathcal{I}$.
- $\text{Sp} = I_\Phi\text{Sp}'$, $\Phi \subseteq \mathsf{Sen}_\mathcal{I}(\mathsf{Sig}(\text{Sp}'))$, *and* Sp' *is well-formed.*
- $\text{Sp} = D_\sigma\text{Sp}'$, σ *is a signature morphism from* Σ *to* $\mathsf{Sig}(\text{Sp}')$, *and* Sp' *is well-formed.*
- $\text{Sp} = T_\sigma\text{Sp}'$, σ *is a signature morphism from* $\mathsf{Sig}(\text{Sp}')$ *to* Σ, *and* Sp' *is well-formed.*
- $\text{Sp} = \text{Sp}_1 + \text{Sp}_2$, $\mathsf{Sig}(\text{Sp}_1) = \mathsf{Sig}(\text{Sp}_2)$, *and* Sp_1 *and* Sp_2 *are well-formed.*

The semantics of a well-formed specification expression in $\mathtt{SL}_\mathcal{I}$ is an abstract datatype in $\text{Adt}_\mathcal{I}$ with the same signature.

Definition 4.7 (Semantics of $\mathtt{SL}_\mathcal{I}$) *Given well-formed specification expressions* Sp, Sp_1, *and* Sp_2, *then the semantics* $[\![\_]\!] : \mathtt{SL}_\mathcal{I} \to \text{Adt}_\mathcal{I}$ *is defined as follows:*

- $[\![\Sigma]\!] = (\Sigma, \mathsf{Str}_\mathcal{I}(\Sigma))$
- $[\![I_\Phi\text{Sp}]\!] = (\mathsf{Sig}(\text{Sp}), \{m \in \mathsf{Mod}^\mathcal{A}([\![\text{Sp}]\!]) \mid m \models^\mathcal{I} \Phi\})$
- $[\![D_\sigma\text{Sp}]\!] = (\mathsf{dom}(\sigma), \{m|_\sigma \mid m \in \mathsf{Mod}^\mathcal{A}([\![\text{Sp}]\!])\})$
- $[\![T_\sigma\text{Sp}]\!] = (\mathsf{cod}(\sigma), \{m \in \mathsf{Str}_\mathcal{I}(\mathsf{cod}(\sigma)) \mid m|_\sigma \in \mathsf{Mod}^\mathcal{A}([\![\text{Sp}]\!])\})$
- $[\![\text{Sp}_1 + \text{Sp}_2]\!] = (\mathsf{Sig}(\text{Sp}_1), \mathsf{Mod}^\mathcal{A}([\![\text{Sp}_1]\!]) \cap \mathsf{Mod}^\mathcal{A}([\![\text{Sp}_2]\!]))$

For any specification expression Sp from $\mathtt{SL}_\mathcal{I}$ we have $\mathsf{Sig}(\text{Sp}) = \mathsf{Sig}^\mathcal{A}([\![\text{Sp}]\!])$.

The objects of the category $\mathtt{SL}_\mathcal{I}$ are specification expressions and the morphisms $\sigma : \text{Sp}_1 \to \text{Sp}_2$ are those signature morphisms σ from $\mathsf{Sig}(\text{Sp}_1)$ to $\mathsf{Sig}(\text{Sp}_2)$ which are also $\text{Adt}_\mathcal{I}$-morphisms σ from $[\![\text{Sp}_1]\!]$ to $[\![\text{Sp}_2]\!]$.

We extend $[\![\_]\!]$ to a functor form $\mathtt{SL}_{\mathcal{I}}$ to $\textsc{Adt}_{\mathcal{I}}$ by mapping $\mathtt{SL}_{\mathcal{I}}$-morphisms σ to their corresponding $\textsc{Adt}_{\mathcal{I}}$-morphisms.

In analogy to $M \models^{\mathcal{I}} \varphi$ we write $\textsc{Sp} \models^{\mathcal{I}} \varphi$ and $\textsc{Sp} \models^{\mathcal{I}} \Phi$ to denote $\mathsf{Mod}^{\mathcal{A}}([\![\textsc{Sp}]\!]) \models^{\mathcal{I}} \varphi$ and $\mathsf{Mod}^{\mathcal{A}}([\![Sp]\!]) \models^{\mathcal{I}} \Phi$, respectively, and further we write $\textsc{Sp}_1 \models^{\mathcal{I}} \textsc{Sp}_2$ to denote $\mathsf{Mod}^{\mathcal{A}}([\![\textsc{Sp}_1]\!]) \subseteq \mathsf{Mod}^{\mathcal{A}}([\![\textsc{Sp}_2]\!])$ for two specification expressions $\textsc{Sp}_1$ and $\textsc{Sp}_2$ of the same signature. We also write $m \models^{\mathcal{I}} \textsc{Sp}$ and $m \in \textsc{Sp}$ for $m \in \mathsf{Mod}^{\mathcal{A}}([\![\textsc{Sp}]\!])$. We have $\textsc{Sp}_1 = \textsc{Sp}_2$ if $\textsc{Sp}_1 \models^{\mathcal{I}} \textsc{Sp}_2$ and $\textsc{Sp}_2 \models^{\mathcal{I}} \textsc{Sp}_1$ or, equivalently, if $[\![\textsc{Sp}_1]\!] = [\![\textsc{Sp}_2]\!]$.

Fact 4.8 *The operations I_Φ, D_σ, T_σ, and $+$ are monotonic with respect to model inclusion, for example, if*

$$\mathsf{Mod}^{\mathcal{A}}([\![\textsc{Sp}]\!]) \subseteq \mathsf{Mod}^{\mathcal{A}}([\![\textsc{Sp}']\!])$$

then

$$\mathsf{Mod}^{\mathcal{A}}([\![I_\Phi\textsc{Sp}]\!]) \subseteq \mathsf{Mod}^{\mathcal{A}}([\![I_\Phi\textsc{Sp}']\!]).$$

Monotonicity allows to conclude from $\textsc{Sp}_1 \models \textsc{Sp}_2$ the facts that $T_\sigma\textsc{Sp}_1 \models T_\sigma\textsc{Sp}_2$, $D_\sigma\textsc{Sp}_1 \models D_\sigma\textsc{Sp}_2$, and $I_\Phi\textsc{Sp}_1 \models I_\Phi\textsc{Sp}_2$. Similar union is monotonic if one argument is fixed, that is, $\textsc{Sp} + \textsc{Sp}_1 \models \textsc{Sp} + \textsc{Sp}_2$ holds if $\textsc{Sp}_1 \models \textsc{Sp}_2$ holds.

Union is associative and commutative, and therefore we may write $\textsc{Sp}_1 + \cdots + \textsc{Sp}_n$ instead of, e.g., $(\cdots(\textsc{Sp}_1 + \textsc{Sp}_2) + \cdots) + \textsc{Sp}_n)$.

What follows are some useful equalities relating the operations of $\mathtt{SL}_{\mathcal{I}}$ to each other.

Fact 4.9 *Let* $\textsc{Sp}$, $\textsc{Sp}_1$ *and* $\textsc{Sp}_2$ *be specification expressions and σ, σ_1, and σ_2 signature morphisms.*

First some special cases:

$$\begin{aligned} I_{\{\}}\textsc{Sp} &= \textsc{Sp} & \textsc{Sp} + \textsc{Sp} &= \textsc{Sp} \\ \textsc{Sp} + \Sigma &= \textsc{Sp} & D_{\mathtt{id}}\textsc{Sp} &= \textsc{Sp} \\ T_{\mathtt{id}}\textsc{Sp} &= \textsc{Sp} \end{aligned}$$

Rules for Impose:

$$\begin{aligned} I_{\Phi_1}I_{\Phi_2}\textsc{Sp} &= I_{\Phi_1\cup\Phi_2}\textsc{Sp} & I_\Phi(\textsc{Sp}_1 + \textsc{Sp}_2) &= I_\Phi\textsc{Sp}_1 + I_\Phi\textsc{Sp}_2 \\ I_\Phi\textsc{Sp} &= I_\Phi\Sigma + \textsc{Sp} & I_\Phi D_\sigma\textsc{Sp} &= D_\sigma I_{\sigma(\Phi)}\textsc{Sp} \end{aligned}$$

Rules for Translate:

$$\begin{aligned} T_{\sigma_1}T_{\sigma_2}\textsc{Sp} &= T_{\sigma_2;\sigma_1}\textsc{Sp} & T_\sigma I_\Phi\textsc{Sp} &= I_{\sigma(\Phi)}T_\sigma\textsc{Sp} \\ T_{\sigma_1}D_{\sigma_2}\textsc{Sp} &= D_{\sigma_2'}T_{\sigma_1'}\textsc{Sp} & T_\sigma(\textsc{Sp}_1 + \textsc{Sp}_2) &= T_\sigma\textsc{Sp}_1 + T_\sigma\textsc{Sp}_2 \\ T_\sigma\mathsf{dom}(\sigma) &= \mathsf{cod}(\sigma) \end{aligned}$$

The equation $T_{\sigma_1}D_{\sigma_2}\textrm{S}\textsc{p} = D_{\sigma'_2}T_{\sigma'_1}\textrm{S}\textsc{p}$ *is true only if the pushout of* σ_1 *and* σ_2 *exists and* $\mathcal{I}$ *has amalgamation. In this case* σ'_1 *and* σ'_2 *are the co-cone morphisms of the following pushout in* $\textrm{S}\textsc{ign}_{\mathcal{I}}$*:*

$$\begin{tikzcd}
 & \Sigma_{cl} & \\
\Sigma_1 \arrow[ur, "\sigma'_2"] & & \Sigma_2 \arrow[ul, "\sigma'_1"'] \\
 & \Sigma \arrow[ul, "\sigma_1"] \arrow[ur, "\sigma_2"'] &
\end{tikzcd}$$

Rules for Derive:

$$\begin{array}{cc}
D_{\sigma_1}D_{\sigma_2}\textrm{S}\textsc{p} = D_{\sigma_1;\sigma_2}\textrm{S}\textsc{p} & D_\sigma(\textrm{S}\textsc{p}_1 + T_\sigma\textrm{S}\textsc{p}_2) = D_\sigma\textrm{S}\textsc{p}_1 + \textrm{S}\textsc{p}_2 \\
D_\sigma\mathsf{cod}(\sigma) \models \mathsf{dom}(\sigma) & D_\sigma I_\Phi\textrm{S}\textsc{p} \models I_{\sigma^{-1}(\Phi)}D_\sigma\textrm{S}\textsc{p} \\
D_\sigma(\textrm{S}\textsc{p}_1 + \textrm{S}\textsc{p}_2) \models D_\sigma\textrm{S}\textsc{p}_1 + D_\sigma\textrm{S}\textsc{p}_2 & D_\sigma T_\sigma\textrm{S}\textsc{p} \models \textrm{S}\textsc{p}
\end{array}$$

$\sigma^{-1}(\Phi)$ *is the set of all* $\mathsf{dom}(\sigma)$*-formulas* φ *such that* $\sigma(\varphi)$ *is in* Φ.

PROOF. As examples we show

1. $D_\sigma\textrm{S}\textsc{p}_1 + \textrm{S}\textsc{p}_2 = D_\sigma(\textrm{S}\textsc{p}_1 + T_\sigma\textrm{S}\textsc{p}_2)$ and
2. $T_{\sigma_1}D_{\sigma_2}\textrm{S}\textsc{p} = D_{\sigma'_2}T_{\sigma'_1}\textrm{S}\textsc{p}$.

(1)($\subseteq$) Let m be in $D_\sigma\textrm{S}\textsc{p}_1 + \textrm{S}\textsc{p}_2$, we have to show that there exists m_1 in $\textrm{S}\textsc{p}_1 + T_\sigma\textrm{S}\textsc{p}_2$ such that $m_1|_\sigma = m$. Since $m \in D_\sigma\textrm{S}\textsc{p}_1$, there exists a model m' of $\textrm{S}\textsc{p}_1$ such that $m'|_\sigma = m$. Set $m_1 = m'$. Then we have to show that $m_1 \in T_\sigma\textrm{S}\textsc{p}_2$, which is implied by $m \models \textrm{S}\textsc{p}_2$ and $m_1|_\sigma = m$.

($\supseteq$) Let m be in $D_\sigma(\textrm{S}\textsc{p}_1 + T_\sigma\textrm{S}\textsc{p}_2)$, which means there exists m_1 with $m_1 \models^{\mathcal{I}} \textrm{S}\textsc{p}_1$, $m_1 \models^{\mathcal{I}} T_\sigma\textrm{S}\textsc{p}_2$, and $m_1|_\sigma = m$. Then we have $m \models D_\sigma\textrm{S}\textsc{p}_1$ and $m \models \textrm{S}\textsc{p}_2$.

(2)($\subseteq$) To show that $T_{\sigma_1}D_{\sigma_2}\textrm{S}\textsc{p} = D_{\sigma'_2}T_{\sigma'_1}\textrm{S}\textsc{p}$, let first m_1 be a model of $T_{\sigma_1}D_{\sigma_2}\textrm{S}\textsc{p}$. Then we have to show that m_1 is a model of $D_{\sigma'_2}T_{\sigma'_1}$, that is, there exists $m \in \mathsf{Str}_{\mathcal{I}}(\Sigma_{cl})$ with $m|_{\sigma'_2}$ being a model of $\textrm{S}\textsc{p}$ and $m|_{\sigma'_1} = m_1$.

Since m_1 is a model of $T_{\sigma_1}D_{\sigma_2}\textrm{S}\textsc{p}$, there exists a model m_2 of $\textrm{S}\textsc{p}$ such that $m_1|_{\sigma_1} = m_2|_{\sigma_2}$. Then m is the amalgamated sum of m_1 and m_2.

($\supseteq$) On the other hand, if m_1 is a model of $D_{\sigma'_2}T_{\sigma'_1}\textrm{S}\textsc{p}$, then we have to show that there exists a model m_2 of $\textrm{S}\textsc{p}$ with $m_1|_{\sigma_1} = m_2|_{\sigma_2}$. Since m_1 is a model of $D_{\sigma'_2}T_{\sigma'_1}\textrm{S}\textsc{p}$, there exists $m \in \mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}(\Sigma_{cl})$ with $m|_{\sigma'_1} = m_1$ and we have

$$m|_{\sigma_1;\sigma'_2} = (m|_{\sigma'_2})|_{\sigma_1} = m|_{\sigma_2;\sigma'_1} = m_1|_{\sigma_2}$$

because $\sigma_1;\sigma'_2 = \sigma_2;\sigma'_1$. Then m_2 is $m|_{\sigma'_2}$. □

Note that $\mathsf{dom}(\sigma) \models^{\mathcal{I}} D_\sigma \mathsf{cod}(\sigma)$ holds only if each $\mathsf{dom}(\sigma)$-structure can be extended to a $\mathsf{cod}(\sigma)$-structure. In $\mathcal{EQ}$ this is for example not possible if the signature morphism is not injective. Consider a signature morphism with $\sigma(s_1) = \sigma(s_2) = s$ and a $\mathsf{dom}(\sigma)$-algebra A with $A(s_1) \neq A(s_2)$, then it is impossible to find a $\mathsf{cod}(\sigma)$-algebra A' with $A'|_\sigma = A$ because $A'|_\sigma(s_1) = A'(s) = A'|_\sigma(s_2)$.

Also note that $D_\sigma \mathrm{S\scriptsize P}_1 + D_\sigma \mathrm{S\scriptsize P}_2$ does not imply $D_\sigma(\mathrm{S\scriptsize P}_1 + \mathrm{S\scriptsize P}_2)$. The problem is that in the first case two extensions $m_1 \models^{\mathcal{I}} \mathrm{S\scriptsize P}_1$ and $m_2 \models^{\mathcal{I}} \mathrm{S\scriptsize P}_2$ of a structure m have to exist while in the second case one extension $\bar{m} \models^{\mathcal{I}} \mathrm{S\scriptsize P}_1 + \mathrm{S\scriptsize P}_2$ has to exist.

Let $\mathrm{S\scriptsize P}_1$ be an extension of `Nat` by a constant $c : \mathbb{N}$ with the constraint $c = 3$ and $\mathrm{S\scriptsize P}_2$ be an extension of `Nat` with $c = 4$. Then $\mathrm{S\scriptsize P}_1 + \mathrm{S\scriptsize P}_2$ has no models while $D_\sigma \mathrm{S\scriptsize P}_1 = \mathtt{Nat} = D_\sigma \mathrm{S\scriptsize P}_2$ and therefore $D_\sigma \mathrm{S\scriptsize P}_1 + D_\sigma \mathrm{S\scriptsize P}_2 = \mathtt{Nat}$.

However, we have the following theorem:

Theorem 4.10 *Provided that the institution $\mathcal{I}$ has amalgamation and that the pushout of $\sigma_1 : \Sigma_0 \to \Sigma_1$ and $\sigma_2 : \Sigma_0 \to \Sigma_2$ exists with co-cone morphisms σ_1' and σ_2', then*

$$D_{\sigma_1;\sigma_2'}(T_{\sigma_2'} \mathrm{S\scriptsize P}_1 + T_{\sigma_1'} \mathrm{S\scriptsize P}_2) = D_{\sigma_1} \mathrm{S\scriptsize P}_1 + D_{\sigma_2} \mathrm{S\scriptsize P}_2.$$

Proof. The problematic direction is to show

$$D_{\sigma_1} \mathrm{S\scriptsize P}_1 + D_{\sigma_2} \mathrm{S\scriptsize P}_2 \models^{\mathcal{I}} D_{\sigma_1;\sigma_2'}(T_{\sigma_2'} \mathrm{S\scriptsize P}_1 + T_{\sigma_1'} \mathrm{S\scriptsize P}_2)$$

because we are given two extensions $m_1 \in \mathsf{Mod}^{\mathcal{A}}([\![\mathrm{S\scriptsize P}_1]\!])$ and $m_2 \in \mathsf{Mod}^{\mathcal{A}}([\![\mathrm{S\scriptsize P}_2]\!])$ of every m in $D_{\sigma_1} \mathrm{S\scriptsize P}_1 + D_{\sigma_2} \mathrm{S\scriptsize P}_2$, but we have to find just one extension m' in $T_{\sigma_2'} \mathrm{S\scriptsize P}_1 + T_{\sigma_1'} \mathrm{S\scriptsize P}_2$.

Since the pushout of σ_1 and σ_2 exists and $\mathcal{I}$ has amalgamation, we set m' to be the amalgamated sum of m_1 and m_2 with respect to m. Then we have $m'|_{\sigma_2'}$ is a model of $\mathrm{S\scriptsize P}_1$ because $m'|_{\sigma_2'} = m_1$ and similar $m'|_{\sigma_1'}$ is a model of $\mathrm{S\scriptsize P}_2$. □

The above theorem is related to the middle distributive law of Goguen and Diaconescu [27]. The difference is that Goguen and Diaconescu define hiding on theories and not on model classes as it is done here. Thus they have to require other properties in addition to amalgamation.

It is well-known, for example Ehrig, Wagner and Thatcher [18], Breu [11], Bergstra, Heering and Klint [7], and Cengarle [14], that using the equalities given in Fact 4.9 any specification expression $\mathrm{S\scriptsize P}$ in $\mathtt{SL}_{\mathcal{I}}$ can be brought into the form $D_\sigma I_\Phi \Sigma$.

Theorem 4.11 (Normal-form) *Given an institution $\mathcal{I}$ with a cocomplete category of signatures $\mathrm{S\scriptsize IGN}_{\mathcal{I}}$ and such that $\mathcal{I}$ has amalgamation. For any well-formed specification expression $\mathrm{S\scriptsize P}$ in $\mathtt{SL}_{\mathcal{I}}$ there exists a well-formed specification expression $D_\sigma I_\Phi \Sigma$ such that $[\![\mathrm{S\scriptsize P}]\!] = [\![D_\sigma I_\Phi \Sigma]\!]$.*

Proof. The theorem follows from the application of the equalities of Fact 4.9. For example, Σ is equivalent to $D_{\mathtt{id}}I_{\{\}}\Sigma$, and

$$\begin{aligned} D_{\sigma_1}I_{\Phi_1}\Sigma_1 + D_{\sigma_2}I_{\Phi_2}\Sigma_2 &= D_{\sigma_1}(I_{\Phi_1}\Sigma_1 + T_{\sigma_1}D_{\sigma_2}I_{\Phi_2}\Sigma_2) \\ &= D_{\sigma_1}(I_{\Phi_1}\Sigma_1 + D_{\sigma_2'}T_{\sigma_1'}I_{\Phi_2}\Sigma_2) \\ &= D_{\sigma_1}D_{\sigma_2'}(T_{\sigma_2'}I_{\Phi_1}\Sigma_1 + T_{\sigma_1'}I_{\Phi_2}\Sigma_2) \\ &= D_{\sigma_1;\sigma_2'}(I_{\sigma_2'(\Phi_1)}\Sigma_{cl} + I_{\sigma_1'(\Phi_2)}\Sigma_{cl}) \\ &= D_{\sigma_1;\sigma_2'}I_{\sigma_2'(\Phi_1)\cup\sigma_1'(\Phi_2)}\Sigma_{cl} \end{aligned}$$

where σ_1' and σ_2' are the co-cone morphisms from Σ_1 and Σ_2 into the pushout signature Σ_{cl}. □

Derived Operations

The category $\mathtt{SL}_{\mathcal{I}}$ is finitely cocomplete if $\textsc{Sign}_{\mathcal{I}}$ is. Let $F : J \to \mathtt{SL}_{\mathcal{I}}$ be a functor such that J is finite, and its collection of objects is given by the set $\{i_1, \ldots, i_n\}$. Then the expression

$$T_{\iota_{i_1}^{F;\mathsf{Sig}}}F(i_1) + \cdots + T_{\iota_{i_n}^{F;\mathsf{Sig}}}F(i_n)$$

is the colimit of F. The co-cone morphisms are the co-cone morphisms of the colimit of $F;\mathsf{Sig}$. Assume that μ is a natural transformation from F to $\Delta\textsc{Sp}$, then $\mu;\mathsf{Sig}$ is a natural transformation from F to $\Delta\mathsf{Sig}(\textsc{Sp})$, and thus there exists a unique signature morphism σ from $\coprod F;\mathsf{Sig}$ to $\mathsf{Sig}(\textsc{Sp})$ with $\iota_i^{F;\mathsf{Sig}};\sigma = \mu_i$ for all $i \in J$. Let m be a model of $\textsc{Sp}$, we have to show that $m|_\sigma$ is a model of $\coprod F$. By the definition of $\coprod F$ this is equivalent to showing that $(m|_\sigma)|_{\iota_i^{F;\mathsf{Sig}}}$ is a model of $F(i)$ for all $i \in J$. This is true because $\iota_i^{F;\mathsf{Sig}};\sigma = \mu_i$ and μ_i is an $\mathtt{SL}_{\mathcal{I}}$-morphism.

Note that $\mathtt{SL}_{\mathcal{I}}$ still has only finite colimits even if $\textsc{Sign}_{\mathcal{I}}$ has also colimits of infinite diagrams because it is impossible to express a union of infinitely many specifications in $\mathtt{SL}_{\mathcal{I}}$.

The existence of finite colimits in $\mathtt{SL}_{\mathcal{I}}$, provided that $\textsc{Sign}_{\mathcal{I}}$ is finitely cocomplete, motivates the following extensions to $\mathtt{SL}_{\mathcal{I}}$

$$\textsc{Sp} ::= \ldots \mid \mathsf{colim}\ F \mid \textsc{Sp}_1 +_{(\sigma_1,\sigma_2)} \textsc{Sp}_2 \mid \textsc{Sp}_1 +_{\textsc{Sp}_0} \textsc{Sp}_2$$

where F is a functor from J to $\mathtt{SL}_{\mathcal{I}}$ with J being a finite category, and $\sigma_1 : \textsc{Sp}_0 \to \textsc{Sp}_1$ and $\sigma_2 : \textsc{Sp}_0 \to \textsc{Sp}_2$ are $\mathtt{SL}_{\mathcal{I}}$-morphisms.

Then $\mathsf{colim}\ F$ is defined as an abbreviation for the specification expression

$$T_{\iota_{i_1}^{F;\mathsf{Sig}}}F(i_1) + \cdots + T_{\iota_{i_n}^{F;\mathsf{Sig}}}F(i_n)$$

where $i_1, \ldots, i_n$ are the objects of J. For the special case where F is a pushout diagram, we write $\textsc{Sp}_1 +_{(\sigma_1,\sigma_2)} \textsc{Sp}_2$ instead of $\mathsf{colim}\ F$. Also we may write $\textsc{Sp}_1 +_{\textsc{Sp}_0} \textsc{Sp}_2$ if σ_1 and σ_2 are the "canonic" $\mathtt{SL}_{\mathcal{I}}$-morphisms from $\textsc{Sp}_0$ to $\textsc{Sp}_1$ and $\textsc{Sp}_2$, for example in $\mathcal{LSL}$ this would be given by the inclusion of signatures.

The following theorem shows that the semantics of $\mathsf{colim}\ F$ is a colimit of $F;[\![\_]\!]$ in $\textsc{Adt}_{\mathcal{I}}$.

Theorem 4.12 *The functor $[\![\_]\!]$ from $\mathtt{SL}_{\mathcal{I}}$ to $\textsc{Adt}_{\mathcal{I}}$ preserves colimits, that is, given a specification expression* $\mathsf{colim}\ F$ *then*

$$[\![\mathsf{colim}\ F]\!] = \coprod F; [\![\_]\!],$$

for functors $F : J \to \mathtt{SL}_{\mathcal{I}}$.

Proof. The theorem is a consequence of the construction of colimits in $\textsc{Adt}_{\mathcal{I}}$, given in the proof of Theorem 4.1. □

If each $F(i)$ has the form $I_{\Phi_i}\Sigma_i$, then the following is a consequence of the definition of $\mathsf{colim}\ F$ and the equalities $I_{\Phi_1}\textsc{Sp} + I_{\Phi_2}\textsc{Sp} = I_{\Phi_1 \cup \Phi_2}\textsc{Sp}$ and $T_\sigma I_\Phi \textsc{Sp} = I_{\sigma(\Phi)} T_\sigma \textsc{Sp}$:

Theorem 4.13 *Given a functor F mapping objects in J to specification expressions of the form $I_{\Phi_i}\Sigma_i$ in $\mathtt{SL}_{\mathcal{I}}$ and morphisms $f : i \to j$ to $\mathtt{SL}_{\mathcal{I}}$-morphisms $F(f)$ from $I_{\Phi_i}\Sigma_i$ to $I_{\Phi_j}\Sigma_j$, then*

$$\mathsf{colim}\ F = I_\Phi \Sigma_{cl}$$

where Σ_{cl} is the colimit of $F;\mathsf{Sig}$ and Φ is the union of all Φ_i translated by the co-cone morphisms $\iota_i^{F;\mathsf{Sig}}$ for $i \in J$.

For institutions with many sorted signatures, like $\mathcal{EQ}$, $\mathcal{EQC}$ and $\mathcal{LSL}$, we use

```
introduces sorts $S$
introduces $\Omega$
asserts $\Phi$
```

as an abbreviation for $I_\Phi\Sigma$ where Σ is the signature with sorts S and operation symbols Ω, and

```
includes Sp[y$_1$ for x$_1$, ..., y$_n$ for x$_n$]
introduces sorts $S$
introduces $\Omega$
asserts $\Phi$
```

as an abbreviation for $I_\Phi T_\sigma \textsc{Sp}$ where σ is a signature morphism from $\mathsf{Sig}(\mathtt{sp}) = (S_1, \Omega_1)$ to $(S_1 \cup S, \Omega_1 \cup \Omega)$ such that $\sigma(x_i) = y_i$ for compatible operation or sort symbols x_i and y_i $(1 \le i \le n)$. We may not mention the set of sorts explicitly if it can be deduced from the family of operations Ω. We are omitting any of the `introduces` or `asserts` part if the corresponding family of operation symbols or set of axioms is empty. These abbreviations are modeled after the syntax of the Larch Shared Language [34]. If we want to give a specification expression a name, for example, `Name` $= I_\Phi T_\sigma \textsc{Sp}$, we write

```
Name : trait
includes Sp
introduces $\Omega$
asserts $\Phi$
```

For specification expressions $\textsc{Sp}$, $\textsc{Sp}_1$ and $\textsc{Sp}_2$ and signature inclusions ι_1 from $\mathsf{Sig}(\textsc{Sp})$ to $\mathsf{Sig}(\textsc{Sp}_1)$ and ι_2 from $\mathsf{Sig}(\textsc{Sp})$ to $\mathsf{Sig}(\textsc{Sp}_2)$, we write $\textsc{Sp}_1 +_{\textsc{Sp}} \textsc{Sp}_2$ instead of $\textsc{Sp}_1 +_{(\iota_1,\iota_2)} \textsc{Sp}_2$.

4.3 Proving Specification Entailment in $\mathtt{SL}_{\mathcal{I}}$

We are interested in the problem of proving that a formula φ is a consequence of the specification $\textsc{Sp}$ and that $\textsc{Sp}'$ is entailed by $\textsc{Sp}$ for specification expressions $\textsc{Sp}$ and $\textsc{Sp}'$ in $\mathtt{SL}_{\mathcal{I}}$. Since proving that φ is a consequence of $\textsc{Sp}$ is the same as proving that the specification expression $I_{\{\varphi\}}\Sigma$ is entailed by $\textsc{Sp}$, we restrict our attention to specification entailment.

Using the normal-form theorem of the previous section, the problem of showing $\textsc{Sp}_1 \models \textsc{Sp}_2$ can be reduced to the problem of showing $D_{\sigma_1} I_{\Phi_1}\Sigma_1 \models D_{\sigma_2} I_{\Phi_2}\Sigma_2$. Consider for now that σ_2 is the identity, the case where σ_2 is not the identity is treated in Section 4.4.

If we are given a sound inference system, that is, a family of relations

$$(\vdash^{\mathcal{I}}_{\Sigma} \subseteq 2^{\mathsf{Sen}_{\mathcal{I}}(\Sigma)} \times \mathsf{Sen}_{\mathcal{I}}(\Sigma))_{\Sigma \in \textsc{Sign}_{\mathcal{I}}}$$

such that $\Phi \vdash^{\mathcal{I}}_{\Sigma} \varphi$, or for short $\Phi \vdash^{\mathcal{I}} \varphi$, implies $\Phi_1 \models^{\mathcal{I}} \varphi$, then to prove $D_{\sigma_1} I_{\Phi_1}\Sigma_1 \models I_{\Phi_2}\Sigma_2$, it suffices to prove $\Phi_1 \vdash^{\mathcal{I}} \sigma_1(\Phi_2)$, which is short for $\Phi_1 \vdash^{\mathcal{I}} \varphi$ for all $\varphi \in \sigma_1(\Phi_2)$.

However, this strategy does not make use of the structure of $\textsc{Sp}_1$ and $\textsc{Sp}_2$. For example, to prove $\textsc{Sp} + \textsc{Sp}_1 \models \textsc{Sp} + \textsc{Sp}_2$, it suffices to prove $\textsc{Sp} + \textsc{Sp}_1 \models \textsc{Sp}_2$, and therefore we need only compute the normal-form of $\textsc{Sp}_2$ instead of $\textsc{Sp} + \textsc{Sp}_2$. In addition, in Chapter 8 we shall extend the specification language $\mathtt{SL}_{\mathcal{I}}$ with disjunction, and for the extended specification language the normal-form theorem does not hold anymore for all specification expressions. Therefore, we introduce a set of inference rules that allow to reduce the goal of proving a specification entailment into a sequence of subgoals which, for example, could be proved using the strategy of computing the normal-form, in case it exists.

A first set of inference rules are extensions of the rules for proving $\textsc{Sp} \vdash \varphi$ using the structure of $\textsc{Sp}$ found by Sannella and Tarlecki [50] or Wirsing [61] to rules for proving specification entailment. For example,

$$\frac{\textsc{Sp} \vdash \sigma(\varphi)}{D_\sigma \textsc{Sp} \vdash \varphi}$$

is extended to

$$\frac{\textsc{Sp} \vdash T_\sigma \textsc{Sp}'}{D_\sigma \textsc{Sp} \vdash \textsc{Sp}'}$$

The first rule is an instance of the second rule if we set $\textsc{Sp}'$ to $I_{\{\varphi\}}\Sigma$.

$$\text{L1}\quad \frac{\textsc{Sp} \vdash \textsc{Sp}'}{I_\Phi \textsc{Sp} \vdash \textsc{Sp}'} \qquad \text{L2}\quad \frac{I_\Phi \Sigma \vdash \textsc{Sp}'}{I_\Phi \textsc{Sp} \vdash \textsc{Sp}'} \qquad \text{L3}\quad \frac{I_{\{\}}\Sigma \vdash \textsc{Sp}'}{\Sigma \vdash \textsc{Sp}'}$$

$$\text{L4}\quad \frac{\textsc{Sp} \vdash \textsc{Sp}'}{T_\sigma \textsc{Sp} \vdash T_\sigma \textsc{Sp}'} \qquad \text{L5}\quad \frac{\textsc{Sp} \vdash T_\sigma \textsc{Sp}'}{D_\sigma \textsc{Sp} \vdash \textsc{Sp}'}$$

$$\text{L6}\quad \frac{\textsc{Sp} \vdash \textsc{Sp}_1}{\textsc{Sp} + \textsc{Sp}' \vdash \textsc{Sp}_1} \qquad \text{L6'}\quad \frac{\textsc{Sp}' \vdash \textsc{Sp}_1}{\textsc{Sp} + \textsc{Sp}' \vdash \textsc{Sp}_1}$$

The next two rules connect the inference system for proving $\textsc{Sp} \vdash \textsc{Sp}'$ with the inference system for proving $\Phi \vdash^{\mathcal{I}} \Phi'$, which depends on the institution $\mathcal{I}$.

$$\text{L7} \quad \frac{\Phi' \subseteq \Phi}{I_{\Phi}\Sigma \vdash I_{\Phi'}\Sigma} \qquad \text{L8} \quad \frac{\textsc{Sp} \vdash I_{\Phi_1}\Sigma \quad \ldots \quad \textsc{Sp} \vdash I_{\Phi_n}\Sigma \quad (\bigcup_{i=1}^{n} \Phi_i) \vdash^{\mathcal{I}} \Phi'}{\textsc{Sp} \vdash I_{\Phi'}\Sigma}$$

Note that the inference system is to be understood modulo the equalities of specification expressions given in Fact 4.9. For example, if we want to prove $D_\sigma\textsc{Sp} \vdash I_\Phi\Sigma$, we have to show $\textsc{Sp} \vdash T_\sigma I_\Phi \Sigma$. Since $T_\sigma I_\Phi\Sigma$ equals $I_{\sigma(\Phi)}\Sigma'$, it suffices showing $\textsc{Sp} \vdash I_{\sigma(\Phi)}\Sigma'$. Thus the consequent can be guaranteed to be of the form $I_\Phi\Sigma$ after the application of the above rules if the consequent was in that form before.

The next set of rules allows the decomposition of the consequent into a set of rules $\textsc{Sp}_i \vdash I_{\Phi_i}\Sigma_i$. Similar rules can be found in Wirsing [61] and Hennicker, Wirsing, and Bidoit [38].

$$\text{R1} \quad \frac{}{\textsc{Sp} \vdash \mathsf{Sig}(\textsc{Sp})} \qquad \text{R2} \quad \frac{\textsc{Sp} \vdash I_\Phi\Sigma \quad \textsc{Sp} \vdash \textsc{Sp}'}{\textsc{Sp} \vdash I_\Phi\textsc{Sp}'}$$

$$\text{R3} \quad \frac{\textsc{Sp} \vdash D_\sigma\textsc{Sp}'' \quad \textsc{Sp}'' \vdash \textsc{Sp}'}{\textsc{Sp} \vdash D_\sigma\textsc{Sp}'} \qquad \text{R5} \quad \frac{D_\sigma\textsc{Sp} \vdash \textsc{Sp}'}{\textsc{Sp} \vdash T_\sigma\textsc{Sp}'}$$

$$\text{R4} \quad \frac{\textsc{Sp} \vdash \textsc{Sp}_1 \quad \textsc{Sp} \vdash \textsc{Sp}_2}{\textsc{Sp} \vdash \textsc{Sp}_1 + \textsc{Sp}_2}$$

The third set of rules allows to exploit similarities between $\textsc{Sp}$ and $\textsc{Sp}'$ in the proofs of $\textsc{Sp} \vdash \textsc{Sp}'$. The rules for monotonicity of impose, union, and derive are derived rules using the above rules. The rule for monotonicity of translate is rule L4.

$$\mathsf{Id} \quad \frac{}{\textsc{Sp} \vdash \textsc{Sp}} \qquad \text{Cut} \quad \frac{\textsc{Sp} \vdash \textsc{Sp}'' \quad \textsc{Sp}'' \vdash \textsc{Sp}'}{\textsc{Sp} \vdash \textsc{Sp}'}$$

$$\text{M1} \quad \frac{\textsc{Sp} \vdash \textsc{Sp}'}{D_\sigma\textsc{Sp} \vdash D_\sigma\textsc{Sp}'} \qquad \text{M2} \quad \frac{\textsc{Sp} \vdash \textsc{Sp}'}{I_\Phi\textsc{Sp} \vdash I_\Phi\textsc{Sp}'}$$

$$\text{M3} \quad \frac{\textsc{Sp} \vdash \textsc{Sp}'}{\textsc{Sp} + \textsc{Sp}'' \vdash \textsc{Sp}' + \textsc{Sp}''} \qquad \text{M3'} \quad \frac{\textsc{Sp} \vdash \textsc{Sp}'}{\textsc{Sp}'' + \textsc{Sp} \vdash \textsc{Sp}'' + \textsc{Sp}}$$

For convenience we also provide rules for the four inequalities of Fact 4.9, although they are derived rules. Rule I3 is a consequence of rules R6, L4, and the monotonicity of derive; I4 can be proved using rule L5; I2 exploits the fact that $\sigma(\sigma^{-1}(\Phi)) \subseteq \Phi$; and I1 is an instance of rule R1.

$$\text{I1} \quad \frac{}{D_\sigma\mathsf{cod}(\sigma) \vdash \mathsf{dom}(\sigma)} \qquad \text{I2} \quad \frac{}{D_\sigma I_\Phi\textsc{Sp} \vdash I_{\sigma^{-1}(\Phi)}D_\sigma\textsc{Sp}}$$

$$\text{I3} \quad \frac{}{D_\sigma(\textsc{Sp}_1 + \textsc{Sp}_2) \vdash D_\sigma\textsc{Sp}_1 + D_\sigma\textsc{Sp}_2} \qquad \text{I4} \quad \frac{}{D_\sigma T_\sigma\textsc{Sp} \vdash \textsc{Sp}}$$

4.4 Problems with Derive

Proving $D_\sigma \textsc{Sp} \vdash \textsc{Sp}'$ poses no problems for this is the same as

$$\forall m\ (\exists m'\ m'|_\sigma = m \wedge m' \vdash \textsc{Sp}) \Rightarrow m \models \textsc{Sp}'$$

which is equivalent to

$$\forall m, m'\ m'|_\sigma = m \wedge m' \models \textsc{Sp} \Rightarrow m \models \textsc{Sp}',$$

and using $m'|_\sigma = m$ we get

$$\forall m'\ m' \models \textsc{Sp} \Rightarrow m'|_\sigma \models \textsc{Sp}'.$$

Note that $m'|_\sigma \models \textsc{Sp}'$ is the same as $m' \models T_\sigma \textsc{Sp}'$, and therefore $D_\sigma \textsc{Sp} \models \textsc{Sp}'$ can be proved by showing $\textsc{Sp} \models T_\sigma \textsc{Sp}'$.

However, if D_σ occurs at the top of the consequent, i.e., $\textsc{Sp} \models D_\sigma \textsc{Sp}'$, then we have to show that

$$\forall m\ m \models \textsc{Sp} \Rightarrow \exists m'\ m' \models \textsc{Sp}' \wedge m'|_\sigma = m.$$

The problem is to prove that all models m of $\textsc{Sp}$ can be extended to a model m' of $\textsc{Sp}'$.

One way to prove this is to provide a refinement $\textsc{Sp}''$ of $\textsc{Sp}'$, that is, we have $\textsc{Sp}'' \models \textsc{Sp}'$, and show that each model of $\textsc{Sp}$ can be extended to a model of $\textsc{Sp}''$. The idea is that it is easier to prove $\textsc{Sp} \models D_\sigma \textsc{Sp}''$ than to prove $\textsc{Sp} \models D_\sigma \textsc{Sp}'$. Thus we have the following rule, which is a consequence of the cut rule and monotonicity of derive:

$$\text{R3} \quad \frac{\textsc{Sp} \vdash D_\sigma \textsc{Sp}'' \quad \textsc{Sp}'' \vdash \textsc{Sp}'}{\textsc{Sp} \vdash D_\sigma \textsc{Sp}'}$$

The problem is to find an appropriate $\textsc{Sp}''$ such that proving $\textsc{Sp} \models D_\sigma \textsc{Sp}''$ is easier than proving $\textsc{Sp} \models D_\sigma \textsc{Sp}'$.

For example, if we choose $\textsc{Sp}'' = \textsc{Sp}'$, then we get the original problem back.

On the other hand, if we choose $\textsc{Sp}'' = \Sigma'$ where Σ' is the codomain of σ, the proof obligations are $\textsc{Sp} \vdash D_\sigma \Sigma'$ and $\Sigma' \vdash \textsc{Sp}'$. Since $\textsc{Sp}' \vdash \Sigma'$ by rule R1, it follows that $\Sigma' \models \textsc{Sp}'$ if and only if $\mathsf{Str}(\Sigma') = \mathsf{Mod}^{\mathcal{A}}(\textsc{Sp}')$.

One way to find $\textsc{Sp}''$ comes from the observation that $D_\sigma \textsc{Sp}'$ corresponds to a, possible higher-order, existential quantifier. The idea is to provide an implementation $\textsc{Sp}_I$ of the hidden symbols of $\textsc{Sp}'$ and prove $T_\sigma \textsc{Sp} + \textsc{Sp}_I \vdash \textsc{Sp}'$. However we have to ensure that the implementation of the hidden symbols can be added to any $\textsc{Sp}$-model. Thus we have to ensure that $T_\sigma \textsc{Sp} + \textsc{Sp}_I$ is a persistent extension of $\textsc{Sp}$, that is, $\textsc{Sp} = D_\sigma(T_\sigma \textsc{Sp} + \textsc{Sp}_I)$. Since $\textsc{Sp} \models D_\sigma(T_\sigma \textsc{Sp} + \textsc{Sp}_I)$ if and only if $\textsc{Sp} \models D_\sigma \textsc{Sp}_I$, we get the following rule, which is an instantiation of rule R3

$$\exists^2 \quad \frac{T_\sigma \textsc{Sp} + \textsc{Sp}_I \vdash \textsc{Sp}' \quad \textsc{Sp} \vdash D_\sigma \textsc{Sp}_I}{\textsc{Sp} \vdash D_\sigma \textsc{Sp}'}$$

A similar rule is used by Wirsing [61] and Hennicker, Wirsing, and Bidoit [38].

Farrés-Casals [21] remarks that the problem with the $\exists^2$-rule is that one has to provide an implementation $\textsc{Sp}_I$ for the hidden symbols for the proof. However, in a lot of cases $\textsc{Sp}'$ can be decomposed in the union of two specifications $\textsc{Sp}_h$ and $\textsc{Sp}_v$ where $\textsc{Sp}_h$ provides an implementation of the hidden symbols and $\textsc{Sp}_v$ defines part of the visible symbols in terms of the hidden symbols. Then $\textsc{Sp}_I$ in the $\exists^2$-rule is the same as $\textsc{Sp}_h$, and we get a somewhat simplified version of the inheriting rule of Farrés-Casals [21].

$$\text{Ih} \quad \frac{T_\sigma \textsc{Sp} + \textsc{Sp}_h \vdash \textsc{Sp}_v \quad \textsc{Sp} \vdash D_\sigma \textsc{Sp}_h}{\textsc{Sp} \vdash D_\sigma(\textsc{Sp}_v + \textsc{Sp}_h)}$$

4.5 Completeness

It is easy to check that the above inference system for $\textsc{Sp} \vdash \textsc{Sp}'$ is sound, that is, $\textsc{Sp} \vdash \textsc{Sp}'$ implies $\textsc{Sp} \models \textsc{Sp}'$. If one restricts entailment to $\textsc{Sp} \vdash I_{\{\varphi\}}\Sigma$, then the inference system is also complete. This assumes that the category of signatures is cocomplete, the institution has amalgamation, and that $\vdash^{\mathcal{I}}$ is complete.

To see this, assume that $\textsc{Sp} \models \varphi$, that is, $m \models \varphi$ for each $m \in \mathsf{Mod}^{\mathcal{A}}(\llbracket \textsc{Sp} \rrbracket)$. Because of Theorem 4.11, there exists a specification expression $D_\sigma I_\Phi \Sigma'$ which has the same class of models as $\textsc{Sp}$. Given a Σ'-structure m' such that $m' \models \Phi$, then $m'|_\sigma$ is a model of $D_\sigma I_\Phi \Sigma'$ and thus a model of $\textsc{Sp}$. Since we have $\textsc{Sp} \models \varphi$, we get that $m'|_\sigma \models \varphi$ which implies $m' \models \sigma(\varphi)$ by the satisfaction condition. Thus we have $\Phi \models \sigma(\varphi)$, and because of completeness of $\vdash^{\mathcal{I}}$ we get $\Phi \vdash^{\mathcal{I}} \varphi$. Because the inference system presented here is given modulo the equalities needed to show that $\textsc{Sp}$ equals $D_\sigma I_\Phi \Sigma'$, we just have to give a derivation for $D_\sigma I_\Phi \Sigma' \vdash I_{\{\varphi\}}\Sigma$ to prove completeness of the inference system.

$$\dfrac{\dfrac{\dfrac{}{I_\Phi \Sigma' \vdash I_\Phi \Sigma'} \quad \Phi \vdash^{\mathcal{I}} \{\sigma(\varphi)\}}{I_\Phi \Sigma' \vdash I_{\{\sigma(\varphi)\}}\Sigma'}\,\text{L8}}{D_\sigma I_\Phi \Sigma' \vdash I_{\{\varphi\}}\Sigma}\,\text{L5}$$

For completeness it is important that we are able to restructure $\textsc{Sp}$, in particular, that we could replace $\textsc{Sp}$ by its normal-form (cf. Wirsing [61]). If we only use the rules L1–L8 and are not allowed to apply the equalities of Fact 4.9 to the antecedent, only to the consequent to ensure that it remains in the form $I_\Phi \Sigma$, then completeness depends on the institution $\mathcal{I}$. In her PhD-thesis Cengarle [14] has shown that the inference system is complete for the institution of first-order logic, and Borzyszkowski [10] extends this proof to any institution that has a cocomplete category of signatures, amalgamation, is closed under conjunction and negation, and has the interpolation property.

An institution is closed under (finite) conjunction if for any (finite) set of formulas Φ of $\mathsf{Sen}_{\mathcal{I}}(\Sigma)$, there exists a formula φ in $\mathsf{Sen}_{\mathcal{I}}(\Sigma)$ such that $m \models \Phi$ if and only if $m \models \varphi$ for all $m \in \mathsf{Str}_{\mathcal{I}}(\Sigma)$. If the institution is compact, that is, for any set of Σ-formulas Φ and a Σ-formula φ with $\Phi \models \varphi$ there exists a finite set $\Psi \subseteq \Phi$ with $\Psi \models \varphi$, then it suffices for completeness to require that the institution is closed under finite conjunction only instead of arbitrary conjunction.

An institution is closed under negation, if for any formula φ in $\mathsf{Sen}_{\mathcal{I}}(\Sigma)$, there exists $\neg\varphi$ such that $m \models \neg\varphi$ if and only if $m \not\models \varphi$ for all $m \in \mathsf{Str}_{\mathcal{I}}(\Sigma)$.

Finally, an institution has the interpolation property if given the pushout Σ_{cl} of $\sigma_1 : \Sigma_0 \to \Sigma_1$ and $\sigma_2 : \Sigma_0 \to \Sigma_2$, a Σ_1-formula φ_1, and a Σ_2-formula φ_2 such that $\{\sigma_1'(\varphi_1)\} \models \sigma_2'(\varphi_2)$, then there exists a Σ_0 formula φ such that $\{\varphi_1\} \models \sigma_1(\varphi)$ and $\{\sigma_2(\varphi)\} \models \varphi_2$ where σ_1' and σ_2' are the co-cone morphisms of the pushout.

For arbitrary specification expressions $\textsc{Sp}'$ the inference system for proving $\textsc{Sp} \vdash \textsc{Sp}'$ is not complete. Consider $\Sigma \models D_\sigma\Sigma'$ where σ is a signature morphism from Σ to Σ'. We have to find a derivation for $\Sigma \vdash D_\sigma\Sigma'$. The only applicable rule is rule R3[1]:

$$\text{R3} \quad \frac{\textsc{Sp} \vdash D_\sigma\textsc{Sp}'' \qquad \textsc{Sp}'' \vdash \textsc{Sp}'}{\textsc{Sp} \vdash D_\sigma\textsc{Sp}'}.$$

Thus we have to find $\textsc{Sp}''$ such that $\Sigma \vdash D_\sigma\textsc{Sp}''$ and $\textsc{Sp}'' \vdash \Sigma'$. The second condition holds trivially (rule R1), but the first condition is even more difficult than the original problem because instead of showing that each Σ-structure can be extended to a Σ'-structure we have to show that each Σ-structure can be extended to a Σ'-structure that is a model of $\textsc{Sp}''$.

The problem is that to decide whether for a given signature morphism σ each Σ-structure can be extended to a Σ'-structure or not, requires knowledge about the particular institution that is not provided by the notion of institutions. For example, in the institution of equational logic a sufficient condition that each Σ-structure can be extended to a Σ'-structure is that σ is injective on sorts and operations, and that for each sort in Σ there exists at least one ground term of that sort. Thus there cannot be a complete, institution independent inference system for $\textsc{Sp} \vdash \textsc{Sp}'$ that does not use, in one way or another, an explicit judgment of the form $\textsc{Sp}_1 \models \textsc{Sp}_2$.

For example the inference system introduced by Wirsing [61] and Hennicker, Wirsing, and Bidoit [38] is complete because they use the following variant of the $\exists^2$-rule:

$$\frac{T_\sigma\textsc{Sp} + \textsc{Sp}_I \vdash \textsc{Sp}' \quad \textsc{Sp} \models D_\sigma\textsc{Sp}_I}{\textsc{Sp} \vdash D_\sigma\textsc{Sp}'}$$

Note that in the second premise of the $\exists^2$-rule $\vdash$ is replaced by $\models$.

To see the completeness of this rule, assume that $\textsc{Sp} \models D_\sigma\textsc{Sp}'$. Then we have to give a derivation for $\textsc{Sp} \vdash D_\sigma\textsc{Sp}'$. Let $\textsc{Sp}_I$ be $\textsc{Sp}'$, then $\textsc{Sp} \models D_\sigma\textsc{Sp}'$ holds trivially because that is our assumption; thus we only have to show $T_\sigma\textsc{Sp} + \textsc{Sp}' \vdash \textsc{Sp}'$ to apply the variant of the

[1] The rules $\exists^2$ and Ih need not be considered because they are instantiations of R3.

$\exists^2$-rule:

$$\dfrac{\dfrac{\overline{\textsc{Sp}' \vdash \textsc{Sp}'}}{T_\sigma\textsc{Sp} + \textsc{Sp}' \vdash \textsc{Sp}'}\;\text{L6'} \qquad\qquad \textsc{Sp} \models D_\sigma\textsc{Sp}'}{\textsc{Sp} \vdash D_\sigma\textsc{Sp}'}$$

Therefore, to trivially complete our inference system, we add the rule

$$\text{T} \quad \frac{\textsc{Sp} \models \textsc{Sp}'}{\textsc{Sp} \vdash \textsc{Sp}'}.$$

5 Relations as Abstract Datatypes

In this chapter we present the main construction of this thesis the institution $\mathcal{R}_\mathcal{I}$, which is based on an exact institution $\mathcal{I}$ with a cocomplete category of signatures. Abstract datatypes in $\mathcal{R}_\mathcal{I}$ denote relations between abstract datatypes from $\mathcal{I}$. First we motivate the construction by an example. Next the institution $\mathcal{R}_\mathcal{I}$ is defined, and we show that $\mathcal{R}_\mathcal{I}$ is exact and has a cocomplete category of signatures. Then we study the relationship between abstract datatypes in $\mathcal{R}_\mathcal{I}$ and abstract datatypes in $\mathcal{I}$ which is later used in Section 7.1 to prove entailment of relations and in Section 9.3 to write Z-style specifcations of dynamic behavior using the Larch Shared Language instead of Z. The operations of the specification language $\mathtt{SL}_\mathcal{I}$ defined in Section 4.2 are then used as a basis for the definition of $\mathtt{RSL}_\mathcal{I}$, a kernel specification language for the specifiation of relations.

5.1 Introduction

In the state as algebra approach the state space of a software system is modeled by an abstract datatype $\mathtt{St} = (\Sigma_\mathtt{St}, M_\mathtt{St})$, and an operation transforming the state of the software system as a relation $R \subseteq M_\mathtt{St} \times M_\mathtt{St}$. As an example consider a counter with an increment relation. The state of the counter is an algebra A with carrier set $A(N) = \mathbb{N}$ and a constant $A(c) \in A(N)$. The set of possible states of a counter is given by the following specification:

```
Counter : trait
   includes Nat
   introduces c : N
```

where `Nat` is:

```
Nat : trait
   introduces
      zero : N
      succ : N → N
      pred : N → N
       +   : N, N → N
       <   : N, N → bool
       ≤   : N, N → bool
       >   : N, N → bool
       ≥   : N, N → bool
   asserts
      N generated freely by zero, succ
   equations forall x,y : N
```

```
zero + y = y;
succ(x) + y = succ(x + y)
pred(succ(x)) = x
x ≠ zero ⇒ succ(pred(x)) = x
¬(s(x) ≤ zero)
zero ≤ x
x ≤ y ⇒ s(x) ≤ s(y)
x ≤ y ∧ x ≠ y ⇒ x < y
¬(x < y) ⇒ x ≥ y
x ≥ y ∨ x ≠ y ⇒ x > y
```

The increment operation increments the value of c by one. It is a relation

$$\mathtt{Inc} \subseteq \mathsf{Mod}^{\mathcal{A}}([\![\mathtt{Counter}]\!]) \times \mathsf{Mod}^{\mathcal{A}}([\![\mathtt{Counter}]\!])$$

with the property

$$\text{if } (A,B) \in \mathtt{Inc} \text{ then } B(c) = A(\mathtt{succ})(A(c))$$

For this equation to be defined, we have to require that $A(N) = B(N)$. In addition we would like that $\mathtt{succ}$ and $+$ are interpreted the same in A and B. That is, if ι is the inclusion of $[\![\mathtt{Nat}]\!]$ into $[\![\mathtt{Counter}]\!]$ then we require that $A|_\iota = B|_\iota$.

In Section 2.3 we have seen that $\{(A,B) \mid A, B \in \mathsf{Mod}^{\mathcal{A}}([\![\mathtt{Counter}]\!]),\ A|_\iota = B|_\iota\}$ is a pullback $\prod F$ of a functor $F : \mathrm{V}^{op} \to \textsc{Set}$ given by the following diagram:

$$\begin{tikzcd}
\mathsf{Mod}^{\mathcal{A}}([\![\mathtt{Counter}]\!]) \arrow[dr, "-|_\iota"'] & & \mathsf{Mod}^{\mathcal{A}}([\![\mathtt{Counter}]\!]) \arrow[dl, "-|_\iota"] \\
& \mathsf{Mod}^{\mathcal{A}}([\![\mathtt{Nat}]\!]) &
\end{tikzcd}$$

The functor F can be given as the composition of the functor $\Gamma : \mathrm{V} \to \textsc{Adt}$:

$$\begin{tikzcd}
[\![\mathtt{Counter}]\!] & & [\![\mathtt{Counter}]\!] \\
& [\![\mathtt{Nat}]\!] \arrow[ul, "\iota"] \arrow[ur, "\iota"'] &
\end{tikzcd}$$

with $\mathsf{Mod}^{\mathcal{A}} : \textsc{Adt}^{op} \to \textsc{Cat}$, ignoring for the moment that $\mathsf{Mod}^{\mathcal{A}}(\Gamma(i))$ yields a category instead of a set for every $i \in \mathrm{V}$.

Thus, the increment relation is a subset of $\prod(\Gamma; \mathsf{Mod}^{\mathcal{A}})$ such that $(A,B) \in \mathtt{Inc}$ if and only if $B(c) = A(\mathtt{succ})(A(c))$.

In each institution that has amalgamation, such as $\mathcal{LSL}$, there exists for each pair (A,B) in $\prod(\Gamma_{\mathtt{Inc}}; \mathsf{Mod}^{\mathcal{A}})$ a unique algebra $A +_C B$ in $\mathsf{Mod}^{\mathcal{A}}(\coprod \Gamma_{\mathtt{Inc}})$, the amalgamated sum of A and B with respect to C, such that $(A +_C B)|_{\iota_1} = A$ and $(A +_C B)|_{\iota_2} = B$. ι_1 and ι_2 from $[\![\mathtt{Counter}]\!]$ to $[\![\mathtt{DeltaCounter}]\!]$ are the co-cone morphisms of the colimit of $\Gamma_{\mathtt{Inc}}$. ι_1 is the inclusion of $[\![\mathtt{Counter}]\!]$ into $[\![\mathtt{DeltaCounter}]\!]$, and ι_2 maps c to c' and all other symbols to themselves.

The pushout $\coprod \Gamma_{\mathtt{Inc}}$ in $\textsc{Adt}$ can be represented by the specification

```
DeltaCounter : trait
   includes Nat
   introduces c, c’ : N
```

Therefore, the relation $\mathtt{Inc}$ is isomorphic to a class of algebras $M_{\mathtt{Inc}}$ from $\mathsf{Mod}^{\mathcal{A}}(\coprod \Gamma_{\mathtt{Inc}})$ such that, if A $\mathtt{Inc}$ B, then $A +_C B \in M_{\mathtt{Inc}}$, and if $D \in M_{\mathtt{Inc}}$, then $D|_{\iota_1}$ $\mathtt{Inc}$ $D|_{\iota_2}$.

For $D \in M_{\mathtt{Inc}}$ we have $D(c') = D(\mathtt{succ})(D(c))$, which is equivalent to writing $D \models c' = \mathtt{succ}(c)$ by the definition of satisfaction in $\mathcal{LSL}$ and since $c' = \mathtt{succ}(c)$ is an equation of signature $\mathsf{Sig}^{\mathcal{A}}(\coprod \Gamma_{\mathtt{Inc}})$.

Thus we can define a binary relation R by providing its type, a functor from V to $\textsc{Adt}$, together with a set of formulas Φ from $\mathsf{Sen}(\mathsf{Sig}^{\mathcal{A}}(\coprod \Gamma))$. Then we have $A\ R\ B$ if and only if $(A, B) \in \prod(\Gamma_{\mathtt{Inc}}; \mathsf{Mod}^{\mathcal{A}})$ and $A +_C B \models \Phi$.

Operations with Input

If we have operations with inputs, like an $\mathtt{Add}$ operation that adds its argument to the counter, we can use the category

$$\mathrm{V}_{\mathtt{in}} = \begin{tikzcd}\mathtt{in} & 1 & 2 \\ & 0 \arrow[ul] \arrow[u] \arrow[ur] & \end{tikzcd}$$

and a functor $\Gamma_{\mathtt{Add}} : \mathrm{V}_{\mathtt{in}} \to \textsc{Adt}$ as the type of the relation instead of a functor from V to $\textsc{Adt}$. We model the input parameter by

```
AddIn : trait
   includes Nat
   introduces n_in : N
```

Then the type of the $\mathtt{Add}$ operation is the functor $\Gamma_{\mathtt{Add}} : \mathrm{V}_{\mathtt{in}} \to \textsc{Adt}$ depicted by

$$\Gamma_{\mathtt{Add}} = \begin{tikzcd}{[\![\mathtt{AddIn}]\!]} & {[\![\mathtt{Counter}]\!]} & {[\![\mathtt{Counter}]\!]} \\ & {[\![\mathtt{Nat}]\!]} \arrow[ul, "\iota_n"'] \arrow[u, "\iota"] \arrow[ur, "\iota"'] & \end{tikzcd}$$

with ι_n being the inclusion of $[\![\mathtt{Nat}]\!]$ into $[\![\mathtt{AddIn}]\!]$. Now $\mathtt{Add}$ is a subset of the limit of $\Gamma_{\mathtt{Add}}; \mathsf{Mod}^{\mathcal{A}}$ such that if $(C, A, B) \in \mathtt{Add}$ then

$$B(c) = A(+)(A(c), C(n_{\mathtt{in}})).$$

Note that the properties of limits guarantee that $C|_{\iota_n} = A|_\iota = B|_\iota$ which implies $C(N) = A(N) = B(N) = \mathbb{N}$ and that the above equation is well-defined.

As before we have that in an institution where Str and thus $\mathsf{Mod}^{\mathcal{A}}$ preserves colimits of shape $\mathrm{V}_{\mathtt{in}}$ we can find for each triple (C, A, B) in $\prod(\Gamma_{\mathtt{Add}}; \mathsf{Mod}^{\mathcal{A}})$ a unique $D \in \mathsf{Mod}^{\mathcal{A}}(\coprod \Gamma_{\mathtt{Add}})$ with $D|_{\iota_{in}} = C$, $D|_{\iota_1} = A$ and $C|_{\iota_2} = B$.

The colimit of $\Gamma_{\mathtt{Add}}$ is given by the specification

```
DeltaCounter1 : trait
   includes Nat
   introduces n_in, c, c' : N
```

and we have that if $(C, A, B) \in \mathtt{Add}$ then $D(c') = D(+)(D(c), D(n_{\mathtt{in}}))$. As before we can define $\mathtt{Add}$ by all triples $(C, A, B) \in \prod(\Gamma_{\mathtt{Add}}; \mathsf{Mod}^{\mathcal{A}})$ such that $D \models c' = c + n_{\mathtt{in}}$.

Arbitrary D

This treatment can be generalized from categories V and $\mathrm{V}_{\mathtt{in}}$ to arbitrary categories D and from the institution $\mathcal{LSL}$ to arbitrary institutions $\mathcal{I}$. We only have to require that the category of signatures of $\mathcal{I}$ has all colimits of shape D and that the structure-functor preserves colimits of shape D.

Take for example the discrete category K_n with objects $\{1, \ldots, n\}$. Let Γ be a functor from K_n to ADT, then $R \subseteq \prod(\Gamma; \mathsf{Mod}^{\mathcal{A}})$ is an n-ary relation

$$R \subseteq \mathsf{Mod}^{\mathcal{A}}(\Gamma(1)) \times \ldots \times \mathsf{Mod}^{\mathcal{A}}(\Gamma(n)).$$

Each morphism $f : d \to d'$ in D adds the requirement that $m_{d'}|_{\Gamma(f)} = m_d$ for each $m \in \prod(\Gamma; \mathsf{Mod}^{\mathcal{A}})$ where $m_d = \pi_d(m)$, $m_{d'} = \pi_{d'}(m)$, and π_d and $\pi_{d'}$ are the projections from $\prod \Gamma; \mathsf{Mod}^{\mathcal{A}}$ to $\mathsf{Mod}^{\mathcal{A}}(\Gamma(d))$ and to $\mathsf{Mod}^{\mathcal{A}}(\Gamma(d'))$, respectively.

Given that $\mathrm{SIGN}_{\mathcal{I}}$ and thus $\mathrm{ADT}_{\mathcal{I}}$ have colimits of shape D and $\mathsf{Str}_{\mathcal{I}}$ preserves them, we can use sets of formulas Φ from $\mathsf{Sen}_{\mathcal{I}}(\mathsf{Sig}^{\mathcal{A}}(\coprod_{\mathrm{D}} \Gamma))$ to define relations $R \subseteq \prod_{\mathrm{D}} \Gamma; \mathsf{Mod}^{\mathcal{A}}$

Note, if we want to use equations to specify relations, we have to make sure that the signatures of the nodes $\mathsf{Sig}^{\mathcal{A}}(\Gamma(i))$ share, at least, a sort. For example, with $\mathrm{D} = \mathrm{K}_2$ equations in $\mathsf{Sen}_{\mathcal{I}}(\coprod \Gamma; \mathsf{Sig}^{\mathcal{A}})$ are of the form $\forall X \;\; s = t$ where both s and t are terms over $\mathsf{Sig}^{\mathcal{A}}(\Gamma(1))$ or $\mathsf{Sig}^{\mathcal{A}}(\Gamma(2))$. Thus the only type of relations $R \subset \mathsf{Mod}^{\mathcal{A}}(\Gamma(1)) \times \mathsf{Mod}^{\mathcal{A}}(\Gamma(2))$ we can define using these equations are relations $R = A \times B$, where A and B are subsets of $\mathsf{Mod}^{\mathcal{A}}(\Gamma(1))$ and $\mathsf{Mod}^{\mathcal{A}}(\Gamma(2))$.

Changing the shape of relations

Consider a relation $R \subseteq A \times B \times C$. We can make R a relation $\bar{R} \subseteq A \times C$ by defining $\bar{R} = \{(a, c) \mid \exists b \in B \;\; (a, b, c) \in R\}$. On the other hand, if we are given a relation $S \subseteq A \times C$, we can make S a relation $\hat{S} \subseteq A \times B \times C$ by defining $\hat{S} = \{(a, b, c) \mid b \in B, \;\; (a, c) \in S\}$.

This can be generalized to relations of shape D and D′ provided there is a functor F from D to D′. In the example above $\mathrm{D} = \mathrm{K}_2$, $\mathrm{D}' = \mathrm{K}_3$, $F(1) = 1$, and $F(2) = 3$.

Given a relation R of type $\Gamma' : D' \to \textsc{Adt}_{\mathcal{I}}$, we define the relation $\bar{R}$ of type $\Gamma = F;\Gamma'$ by

$$\bar{R} = \{\bar{F}(m) \mid m \in R\}$$

where $\bar{F}$ is a functor from $\prod_{\mathrm{D}'} \Gamma';\mathsf{Mod}^{\mathcal{A}}$ to $\prod_{\mathrm{D}} \Gamma;\mathsf{Mod}^{\mathcal{A}}$ given by the universal property of $\prod_{\mathrm{D}} \Gamma;\mathsf{Mod}^{\mathcal{A}}$ with respect to the natural transformation ν from $\Delta(\prod \Gamma';\mathsf{Mod}^{\mathcal{A}})$ to $\Gamma;\mathsf{Mod}^{\mathcal{A}}$ given by $\nu_d = \pi_{F(d)}^{\Gamma';\mathsf{Mod}^{\mathcal{A}}}$:

$$\begin{tikzcd}
\prod \Gamma;\mathsf{Mod}^{\mathcal{A}} \arrow[d, "\pi_d^{\Gamma;\mathsf{Mod}^{\mathcal{A}}}"'] & \prod \Gamma';\mathsf{Mod}^{\mathcal{A}} \arrow[l, "\bar{F}"'] \arrow[dl, "\nu_d=\pi_{F(d)}^{\Gamma';\mathsf{Mod}^{\mathcal{A}}}"] \\
\Gamma(d) = \Gamma'(F(d)) &
\end{tikzcd}$$

In the example above $\pi_i^{\Gamma',\mathsf{Mod}^{\mathcal{A}}}$ maps a triple (a, b, c) to its i-th component and $\pi_j^{\Gamma,\mathsf{Mod}^{\mathcal{A}}}$ a tuple (a, b) to its j-th component where $i \in \{1, 2, 3\}$ and $j \in \{1, 2\}$. Then we get

$$\pi_1^{\Gamma;\mathsf{Mod}^{\mathcal{A}}}(\bar{F}(a, b, c)) = \pi_{F(1)}^{\Gamma';\mathsf{Mod}^{\mathcal{A}}}(a, b, c) = \pi_1^{\Gamma';\mathsf{Mod}^{\mathcal{A}}}(a, b, c) = a$$

and

$$\pi_2^{\Gamma;\mathsf{Mod}^{\mathcal{A}}}(\bar{F}(a, b, c))_2 = \pi_{F(2)}^{\Gamma';\mathsf{Mod}^{\mathcal{A}}}(a, b, c) = \pi_3^{\Gamma';\mathsf{Mod}^{\mathcal{A}}}(a, b, c) = c.$$

Thus $\bar{F}$ maps a triple (a, b, c) to a tuple (a, c).

In the same spirit we define for a relation S of shape Γ a relation of shape Γ' by

$$\hat{S} = \{m' \in \prod \Gamma';\mathsf{Mod}^{\mathcal{A}} \mid \bar{F}(m') \in S\}.$$

5.2 The Category Rel

The category Rel of relations between abstract datatypes from $\mathcal{I}$ has as objects relations (Θ, M) where $\Theta = (\mathrm{D}, \Gamma : \mathrm{D} \to \textsc{Adt})$ is the *type* of the relation, the category D is the *shape* of the relation, and M, a full subcategory of $\prod(\Gamma;\mathsf{Mod}^{\mathcal{A}})$, is the graph of the relation.

A morphism $f = (f^F, f^\mu)$ in Rel from (Θ, M) to (Θ', M') has two components; the first component is a functor $f^F : \mathrm{D} \to \mathrm{D}'$ from the shape category of Θ to the shape category of Θ', and the second component is a natural transformation $f^\mu : \Gamma \Rightarrow f^F;\Gamma'$, which is a D-indexed family of $\textsc{Adt}_{\mathcal{I}}$-morphisms $f_d^F : \Gamma(d) \to \Gamma'(f^F(d))$.

The functor f^F and the natural transformation f^μ give rise to a functor $\_|_f$ from $\prod_{\mathrm{D}'} \Gamma';\mathsf{Mod}^{\mathcal{A}}$ to $\prod_{\mathrm{D}} \Gamma;\mathsf{Mod}^{\mathcal{A}}$ by the universal property of $\prod_{\mathrm{D}} \Gamma;\mathsf{Mod}^{\mathcal{A}}$ and the natural transformation

$\nu_f : \Delta(\prod_{\mathrm{D}'} \Gamma'; \mathsf{Mod}^{\mathcal{A}}) \Rightarrow \Gamma; \mathsf{Mod}^{\mathcal{A}}$ given by $(\nu_f)_d = \pi^{\Gamma';\mathsf{Mod}^{\mathcal{A}}}; \mathsf{Mod}^{\mathcal{A}}(f^\mu_d)$ for $d \in \mathrm{D}$:

$$\begin{tikzcd}
\prod \Gamma; \mathsf{Mod}^{\mathcal{A}} \arrow[d, "\pi_d^{\Gamma;\mathsf{Mod}^{\mathcal{A}}}"'] & \prod \Gamma'; \mathsf{Mod}^{\mathcal{A}} \arrow[l, "\text{-}|_f"'] \arrow[d, "\pi_{f^F(d)}^{\Gamma';\mathsf{Mod}^{\mathcal{A}}}"] \arrow[dl, "(\nu_f)_d"] \\
\mathsf{Mod}^{\mathcal{A}}(\Gamma(d)) & \mathsf{Mod}^{\mathcal{A}}(\Gamma'(f^F(d))) \arrow[l, "\mathsf{Mod}^{\mathcal{A}}(f^\mu_d)"]
\end{tikzcd}$$

Then we require $M'|_f \subseteq M$ for $f = (f^F, f^\mu)$ to be a Rel-morphism, that is, $m'|_f \in M$ for all $m' \in M'$.

As an example take $R = ((\mathrm{K}_2, \Gamma_R), M_R)$ and $S = ((\mathrm{K}_3, \Gamma_S), M_S)$. A Rel-morphism f from R to S is a pair (f^F, f^μ) of a functor $f^F : \mathrm{K}_2 \to \mathrm{K}_3$ and a natural transformation $f^\mu : \Gamma_R \Rightarrow f^F; \Gamma_S$. Let $f^F(1) = 1$ and $f^F(2) = 3$, then f^μ_1 is an Adt-morphism from $\Gamma_R(1)$ to $\Gamma_S(1)$, and f^μ_2 is an Adt-morphism from $\Gamma_R(2)$ to $\Gamma_S(3)$. For f to be a Rel-morphism we have to check that $(m_1|_{f^\mu_1}, m_3|_{f^\mu_2})$ is in M_R for each (m_1, m_2, m_3) in M_S.

The identity $\mathtt{id}_R$ for an object $R = (\Theta, M)$ in Rel is the pair $(\mathsf{Id}_{\mathrm{D}}, \mathtt{id}_\Gamma)$.

Let $f = (f^F, f^\mu)$ be a Rel-morphism from (Θ_1, M_1) to (Θ_2, M_2) and $g = (g^F, g^\mu)$ a Rel-morphism from (Θ_2, M_2) to (Θ_3, M_3). The composition of f and g is the Rel-morphism $(f^F; g^F, f^\mu; (g^\mu; f^F))$:

$$\begin{tikzcd}
\Gamma_1 \arrow[r, Rightarrow, "f^\mu"] & f^F; \Gamma_2 \arrow[r, Rightarrow, "g^\mu; f^F"] & f^F; g^F; \Gamma_3 \\
 & \Gamma_2 \arrow[u, mapsto, "f^F"] \arrow[r, Rightarrow, "g^\mu"'] & g^F; \Gamma_3 \arrow[u, mapsto, "f^F"']
\end{tikzcd}$$

We have to show that $M_3|_{f;g} \subseteq M_1$. This holds because $\text{-}|_{f;g} = \text{-}|_g; \text{-}|_f$. The proof of this will be given in the next section.

Theorem 5.1 *The category* Rel *is (finitely) cocomplete if the category of signatures from the institution $\mathcal{I}$ is (finitely) cocomplete.*

Proof. This follows from fact that Rel is the same as the category $\text{Adt}_{\mathcal{R}_\mathcal{I}}$ of abstract datatypes in the institution $\mathcal{R}_\mathcal{I} = < \text{Sign}_{\mathcal{R}_\mathcal{I}}, \mathsf{Str}_{\mathcal{R}_\mathcal{I}}, \mathsf{Sen}_{\mathcal{R}_\mathcal{I}}, \models^{\mathcal{R}_\mathcal{I}} >$ defined in the next section. Then $\text{Adt}_{\mathcal{R}_\mathcal{I}}$ is (finitely) cocomplete because $\text{Sign}_\mathcal{I}$ and thus $\text{Sign}_{\mathcal{R}_\mathcal{I}}$ are (finitely) cocomplete. □

5.3 The Institution $\mathcal{R}_\mathcal{I}$

In the previous sections we have seen how relations between classes of algebras, given as abstract datatypes, can be defined by a category D, a functor Γ from D to Adt, and a subset of $\prod(\Gamma; \mathsf{Mod}^{\mathcal{A}})$; and how sets of formulas over the signature $\coprod(\Gamma; \mathsf{Sig}^{\mathcal{A}})$ can be used to define these subsets. What we have given are the ingredients of an institution $\mathcal{R}_\mathcal{I}$ based on an institution $\mathcal{I}$.

The category of signatures $\textsc{Sign}_{\mathcal{R}_\mathcal{I}}$ has as objects pairs $(\mathrm{D}, \Gamma : \mathrm{D} \to \textsc{Adt}_\mathcal{I})$, the type of the relation, and morphisms f from $\Theta_1 = (\mathrm{D}_1, \Gamma_1)$ to $\Theta_2 = (\mathrm{D}_2, \Gamma_2)$ are pairs (f^F, f^μ) of a functor f^F from D_1 to D_2 and a natural transformations $f^\mu : \Gamma_1 \Rightarrow f^F; \Gamma_2$. The Θ-structures are elements of $\prod(\Gamma; \mathsf{Mod}^\mathcal{A})$, Θ-formulas are $\coprod(\Gamma; \mathsf{Sig}^\mathcal{A})$-formulas from $\mathcal{I}$, and the satisfaction relation $\models^{\mathcal{R}_\mathcal{I}}$ is defined as $m \models^{\mathcal{R}_\mathcal{I}} \varphi$ if $\overline{m} \models^\mathcal{I} \varphi$. Here $\overline{m}$ refers to the isomorphism between $\prod(\Gamma; \mathsf{Mod}^\mathcal{A})$ and $\mathsf{Mod}^\mathcal{A}(\coprod \Gamma)$, which exists in any exact institution $\mathcal{I}$. That is, if m is an object in $\prod(\Gamma; \mathsf{Mod}^\mathcal{A})$, then $\overline{m}$ is the isomorphic object in $\mathsf{Mod}^\mathcal{A}(\coprod \Gamma)$.

The institution $\mathcal{R}_\mathcal{I}$ can be constructed for any exact base institution $\mathcal{I}$ with a cocomplete category of signatures. Examples of such institutions are $\mathcal{EQ}$, $\mathcal{EQC}$, $\mathcal{LSL}$, $\mathcal{SET}$ (cf. Sections 3 and 9.1), and the institution of first-order logic (cf. [25, 17]). Thus, in the following, we assume that $\mathcal{I} = \langle \textsc{Sign}_\mathcal{I}, \mathsf{Str}_\mathcal{I}, \mathsf{Sen}_\mathcal{I}, \models^\mathcal{I} \rangle$ is an institution with a cocomplete category of signatures $\textsc{Sign}_\mathcal{I}$ and a functor $\mathsf{Str}_\mathcal{I}$ that preserves colimits.

Signatures Instead of directly defining $\textsc{Sign}_{\mathcal{R}_\mathcal{I}}$ as the category with objects $\Theta = (\mathrm{D}, \Gamma)$ and morphisms pairs $f = (f^F, f^\mu)$, as indicated above, we define $\textsc{Sign}_{\mathcal{R}_\mathcal{I}}$ as an instance of the category of diagrams over a category T which is defined by Tarlecki et al. [58].

<span style="font-variant:small-caps">Definition</span> 5.2 ($\mathsf{Fun}(T)$) *Given a category T, then we define the indexed category $\mathsf{Fun}(T) : \textsc{Cat}^{op} \to \textsc{Cat}$ as follows:*
For an object $\mathrm{D} \in \textsc{Cat}$ we set $\mathsf{Fun}(T)_D = T^D$, and for a functor $F : D \to D'$ we define a functor $\mathsf{Fun}(T)_F : T^{D'} \to T^D$ by

- *$\mathsf{Fun}(T)_F(\Gamma') = F; \Gamma'$ for $\Gamma' \in T^{D'}$, that is, $(\mathsf{Fun}(T)_F(\Gamma'))(d) = \Gamma'(F(d))$ for every $d \in \mathrm{D}$, and*
- *$\mathsf{Fun}(T)_F(\mu) = \mu; F$ for a natural transformations $\mu : \Gamma'_1 \Rightarrow \Gamma'_2$ between functors Γ'_1 and Γ'_2 in $T^{D'}$, that is, $(\mathsf{Fun}(T)_F(\mu))_d = \mu_{F(d)}$ for every $d \in \mathrm{D}$.*

Let Id_D be the identity functor on D, then $\mathsf{Fun}(T)_{\mathsf{Id}_D}$ is the identity on T^D because

$$\mathsf{Fun}(T)_{\mathsf{Id}_D}(\Gamma) = \mathsf{Id}_D; \Gamma = \Gamma$$

and

$$\mathsf{Fun}(T)_{\mathsf{Id}_D}(\mu) = \mu; \mathsf{Id}_D = \mu$$

for functors $\Gamma : \mathrm{D} \to T$ and natural transformations $\mu : \Gamma \Rightarrow \Gamma'$.

For two functors $F : D \to D'$ and $G : D' \to D''$ we have $\mathsf{Fun}(T)_{F;G} = \mathsf{Fun}(T)_G; \mathsf{Fun}(T)_F$ because

$$\begin{aligned}\mathsf{Fun}(T)_{F;G}(\Gamma'' : D'' \to T) &= F; G; \Gamma'' \\ &= \mathsf{Fun}(T)_F(G; \Gamma'') \\ &= \mathsf{Fun}(T)_F(\mathsf{Fun}(T)_G(\Gamma'')),\end{aligned}$$

and similar for the composition of natural transformations. Thus $\mathsf{Fun}(T)$ is indeed an indexed category.

Flattening $\mathsf{Fun}(T)$ yields the category $\mathsf{Flat}(\mathsf{Fun}(T))$ of diagrams over T (cf. Section 2.5), which has as objects pairs (D, Γ) — diagrams — where D is a category and Γ a functor from D to T. Morphisms f from (D, Γ) to (D', Γ') in $\mathsf{Flat}(\mathsf{Fun}(T))$ are pairs (f^F, f^μ) where f^F is a functor from D to D′ and f^μ is a natural transformation from Γ to $f^F; \Gamma'$.

The identity on Θ is the pair $(\mathtt{id}^F, \mathtt{id}^\mu)$ where $\mathtt{id}^F$ is the identity functor for D and $\mathtt{id}^\mu_d$ is the identity on $\Gamma(d)$ for all $d \in \mathrm{D}$. The composition of (f^F, f^μ) from Θ_1 to Θ_2 with (g^F, g^μ) from Θ_2 to Θ_3 is defined as

$$f; g = (f^F; g^F, f^\mu; (g^\mu; f^F)),$$

that is, $(f; g)^\mu_d = f^\mu_d; g^\mu_{f^F(d)}$ for all $d \in \mathrm{D}$.

Now we define $\textsc{Sign}_{\mathcal{R}_\mathcal{I}}$ as the category of diagrams over $\textsc{Adt}_\mathcal{I}$.

Definition 5.3 ($\textsc{Sign}_{\mathcal{R}_\mathcal{I}}$) *Let $\mathcal{I}$ be an institution, then define*

$$\textsc{Sign}_{\mathcal{R}_\mathcal{I}} = \mathsf{Flat}(\mathsf{Fun}(\textsc{Adt}_\mathcal{I})).$$

Structures The category of $\Theta = (\mathrm{D}, \Gamma)$-structures — $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta)$ — is given by an arbitrary, but fixed limit of the contravariant functor $\Gamma; \mathsf{Mod}^\mathcal{A} : \mathrm{D}^{op} \to \textsc{Cat}$ in $\textsc{Cat}$. Thus we can view objects in $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta)$ as D-indexed families of structures $m_d \in \mathsf{Mod}^\mathcal{A}(\Gamma(i))$ such that if $f : d \to d'$ is a morphism in J, then $m_{d'}|_{\Gamma(f)} = m_d$.

For a morphism $f = (f^F, f^\mu)$ from (D, Γ) to (D', Γ') in $\textsc{Sign}_{\mathcal{R}_\mathcal{I}}$ the functor $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(f)$, from the limit of $\Gamma'; \mathsf{Mod}^\mathcal{A}$ to the limit of $\Gamma; \mathsf{Mod}^\mathcal{A}$, maps a D′-indexed family of objects $m'_{d'} \in \mathsf{Mod}^\mathcal{A}(\Gamma'(d'))$ to a D-indexed family of objects $m_d = m'_{f^F(d)}|_{f^\mu_{f^F(d)}}$.

Definition 5.4 ($\mathsf{Str}_{\mathcal{R}_\mathcal{I}}$) *Given an institution $\mathcal{I}$. The functor $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}$ maps a signature (D, Γ) to an arbitrary but fixed limit of $\Gamma; \mathsf{Mod}^\mathcal{A}$ in $\textsc{Cat}$ and a signature morphism (f^F, f^μ) from (D, Γ) to (D', Γ') to the unique functor from $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}((\mathrm{D}, \Gamma))$ to $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}((\mathrm{D}', \Gamma'))$ given by the universal property of $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}((\mathrm{D}, \Gamma))$ and the natural transformation ν from $\Delta(\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta_2))$ to $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta_1)$ defined by $\nu_d = \pi'_{f^F(d)}; \mathsf{Mod}^\mathcal{A}(f^\mu_d)$ for all $d \in \mathrm{D}$:*

$$\begin{tikzcd}
\mathsf{Str}_{\mathcal{R}_\mathcal{I}}((\mathrm{D}, \Gamma)) \arrow[d, "\pi_d"'] & \mathsf{Str}_{\mathcal{R}_\mathcal{I}}((\mathrm{D}, \Gamma)) \arrow[l, dashed, "\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(f)"'] \arrow[d, "\pi'_{f^F(d)}"] \arrow[dl, "\nu_d"] \\
\mathsf{Mod}^\mathcal{A}(\Gamma(d)) & \mathsf{Mod}^\mathcal{A}(\Gamma'(f^F(d))) \arrow[l, "\mathsf{Mod}^\mathcal{A}(f^\mu_d)"]
\end{tikzcd}$$

The functors π_d for $d \in \mathrm{D}$ and $\pi'_{d'}$ for $d' \in \mathrm{D}'$ are the cone morphisms of the limit of $\Gamma; \mathsf{Mod}^\mathcal{A}$ and $\Gamma'; \mathsf{Mod}^\mathcal{A}$, respectively.

The definition of $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta)$ as the limit of $\Gamma; \mathsf{Mod}^\mathcal{A}$ is well-defined because $\textsc{Cat}$ is complete. We still have to show that $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}$ is a functor, that is, $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}$ preserves identity and composition.

The identity $(\mathtt{id}^F, \mathtt{id}^\mu)$ in $\textsc{Sign}_{\mathcal{R}_\mathcal{I}}$ is mapped to the identity functor because, on the one hand, $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}((\mathtt{id}^F, \mathtt{id}^\mu))$ is unique as a functor from $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta)$ to $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta)$, and, on the other hand, the identity on $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta)$ is a functor from $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta)$ to $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta)$.

Consider two morphisms f from (D_1, Γ_1) to (D_2, Γ_2) and g from (D_2, Γ_2) to (D_3, Γ_3) in $\textsc{Sign}_{\mathcal{R}_\mathcal{I}}$. $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(f; g)$ is defined as the unique functor from $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}((\mathrm{D}_3, \Gamma_3))$ to $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}((\mathrm{D}_1, \Gamma_1))$ satisfying

$$\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(f; g); \pi_d = \pi''_{g^F(f^F(d))}; \mathsf{Mod}^\mathcal{A}(f^\mu_d; g^\mu_{f^F(d)})$$

for all $d \in \mathrm{D}_1$. However, looking at the following diagram and using the fact that $\mathsf{Mod}^\mathcal{A}$ is a functor, we see that $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(g); \mathsf{Str}_{\mathcal{R}_\mathcal{I}}(f)$ has the same property, and therefore $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(f; g) = \mathsf{Str}_{\mathcal{R}_\mathcal{I}}(g); \mathsf{Str}_{\mathcal{R}_\mathcal{I}}(f)$.

$$\begin{tikzcd}
\prod_{\mathrm{D}} \Gamma; \mathsf{Mod}^\mathcal{A} \arrow[d, "\pi_d"'] & \prod_{\mathrm{D}'} \Gamma'; \mathsf{Mod}^\mathcal{A} \arrow[l, dashed, "\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(f)"'] \arrow[d, "\pi'_{f^F(d)}"] & \prod_{\mathrm{D}''} \Gamma''; \mathsf{Mod}^\mathcal{A} \arrow[l, dashed, "\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(g)"'] \arrow[d, "\pi''_{g^F(f^F(d))}"] \\
\mathsf{Mod}^\mathcal{A}(\Gamma(d)) & \mathsf{Mod}^\mathcal{A}(\Gamma'(f^F(d))) \arrow[l, "\mathsf{Mod}^\mathcal{A}(f^\mu_d)"] & \mathsf{Mod}^\mathcal{A}(\Gamma''(g^F(f^F(d)))) \arrow[l, "\mathsf{Mod}^\mathcal{A}(g^\mu_{f^F(d)})"]
\end{tikzcd}$$

Formulas Formulas in $\mathcal{R}_\mathcal{I}$ are defined by using co-completeness of $\textsc{Sign}_\mathcal{I}$. Co-completeness of $\textsc{Sign}_\mathcal{I}$ ensures that for each type $\Theta = (\mathrm{D}, \Gamma)$ in $\textsc{Sign}_{\mathcal{R}_\mathcal{I}}$ there exists a signature $\coprod_{\mathrm{D}} \Gamma; \mathsf{Sig}^\mathcal{A}$ in $\textsc{Sign}_\mathcal{I}$. Note that, because $\mathsf{Sig}^\mathcal{A}$ preserves and creates colimits, this is the same as $\mathsf{Sig}^\mathcal{A}(\coprod_{\mathrm{D}} \Gamma)$ (cf. Theorem 5.8). Then the set of Θ-formulas in $\mathcal{R}_\mathcal{I}$ is the same as the set of $\coprod \Gamma; \mathsf{Sig}^\mathcal{A}$-formulas in $\mathcal{I}$.

$\mathsf{Sen}_{\mathcal{R}_\mathcal{I}}$ is given as the composition of the functors

- $\mathsf{Flat}(\mathsf{Fun}(\mathsf{Sig}^\mathcal{A})) : \mathsf{Flat}(\mathsf{Fun}(\textsc{Adt}_\mathcal{I})) \to \mathsf{Flat}(\mathsf{Fun}(\textsc{Sign}_\mathcal{I}))$,
- $\mathsf{clim}_{\textsc{Sign}_\mathcal{I}} : \mathsf{Flat}(\mathsf{Fun}(\textsc{Sign}_\mathcal{I})) \to \textsc{Sign}_\mathcal{I}$ and
- $\mathsf{Sen}_\mathcal{I} : \textsc{Sign}_\mathcal{I} \to \textsc{Set}$:

$\mathsf{Flat}(\mathsf{Fun}(\mathsf{Sig}^\mathcal{A}))$ maps a diagram over $\textsc{Adt}_\mathcal{I}$ to a diagram over $\textsc{Sign}_\mathcal{I}$; then $\mathsf{clim}_{\textsc{Sign}_\mathcal{I}}$ maps this diagram over $\textsc{Sign}_\mathcal{I}$ to its colimit in $\textsc{Sign}_\mathcal{I}$, and at last $\mathsf{Sen}_\mathcal{I}$ maps the colimit signature to the set of sentences in $\mathcal{I}$ over this signature.

$$\begin{tikzcd}
\textsc{Sign}_{\mathcal{R}_\mathcal{I}} \arrow[r, "\mathsf{Flat}(\mathsf{Fun}(\mathsf{Sig}^\mathcal{A}))"] \arrow[rrr, bend right, "\mathsf{Sen}_{\mathcal{R}_\mathcal{I}}"'] & \mathsf{Flat}(\mathsf{Fun}(\textsc{Sign}_\mathcal{I})) \arrow[r, "\mathsf{clim}_{\textsc{Sign}_\mathcal{I}}"] & \textsc{Sign}_\mathcal{I} \arrow[r, "\mathsf{Sen}_\mathcal{I}"] & \textsc{Set}
\end{tikzcd}$$

The functor $\mathsf{Flat}(\mathsf{Fun}(\mathsf{Sig}^{\mathcal{A}}))$ is the result of flattening the indexed functor

$$\mathsf{Fun}(\mathsf{Sig}^{\mathcal{A}}) : \mathsf{Fun}(\textsc{Adt}_{\mathcal{I}}) \Rightarrow \mathsf{Fun}(\mathsf{Sig}^{\mathcal{A}}).$$

Given a functor $F : T \to T'$, then F induces an indexed functor $\mathsf{Fun}(F)$ from $\mathsf{Fun}(T)$ to $\mathsf{Fun}(T')$ as follows:

Definition 5.5 *For each category D, $\mathsf{Fun}(F)_D$ is the functor from T^D to $T^{D'}$ given by*

- $\mathsf{Fun}(F)_D(\Gamma : \mathrm{D} \to T) = \Gamma; F$ *and*
- $\mathsf{Fun}(F)_D(\mu : \Gamma \Rightarrow \Gamma') = \mu; F$.

We have to show that $\mathsf{Fun}(F)$ is indeed an indexed functor, that is, $\mathsf{Fun}(F)$ is a natural transformation from $\mathsf{Fun}(T)$ to $\mathsf{Fun}(T')$. Let G be a functor from D to D′, then we have to show that the following diagram commutes:

$$\begin{tikzcd}
\mathsf{Fun}(T)_D \arrow[d, "\mathsf{Fun}(F)_D"'] & \mathsf{Fun}(T)_{D'} \arrow[l, "\mathsf{Fun}(T)_G"'] \arrow[d, "\mathsf{Fun}(F)_{D'}"] \\
\mathsf{Fun}(T')_D & \mathsf{Fun}(T')_{D'} \arrow[l, "\mathsf{Fun}(T')_G"]
\end{tikzcd}$$

For a functor $\Gamma : \mathrm{D} \to T$ we have

$$\begin{aligned}
\mathsf{Fun}(F)_{D'}(\mathsf{Fun}(T)_G(\Gamma)) &= \mathsf{Fun}(F)_{D'}(G;\Gamma) \\
&= (G;\Gamma);F \\
&= G;(\Gamma;F) \\
&= \mathsf{Fun}(T')_G(\Gamma;F) \\
&= \mathsf{Fun}(T')_G(\mathsf{Fun}(F)_D(\Gamma)),
\end{aligned}$$

and similar for natural transformations.

The functor clim_T maps objects $(\mathrm{D}, \Gamma : \mathrm{D} \to T)$ to the colimit of Γ in T.

Definition 5.6 (clim_T) *Given a cocomplete category T, then the functor clim_T from the category $\mathsf{Flat}(\mathsf{Fun}(T))$ defined as:*

- $\mathsf{clim}_T((\mathrm{D}, \Gamma : \mathrm{D} \to T))$ *is an arbitrary, but fixed colimit of Γ in T for every $(\mathrm{D}, \Gamma) \in \mathsf{Flat}(\mathsf{Fun}(T))$, and*
- $\mathsf{clim}_T(f) = \coprod_{\mathrm{D}} (f^{\mu}; (f^F; \iota^{\Gamma'}))$ *for every morphism $f = (f^F, f^{\mu})$ from (D, Γ) to (D', Γ') in $\mathsf{Flat}(\mathsf{Fun}(T))$.*

Because T is cocomplete, the functor clim_T is well defined, and since we have chosen a fixed colimit for each (D, Γ) in $\mathsf{Flat}(\mathsf{Fun}(T))$, the identity $(\mathsf{Id}_F, \mathtt{id}_\Gamma)$ in $\mathsf{Flat}(\mathsf{Fun}(T))$ maps to the identity on $\mathsf{clim}_T((\mathrm{D}, \Gamma))$ in T.

clim_T preserves the composition of morphisms, that is,

$$\mathsf{clim}_T(f;g) = \mathsf{clim}_T(f);\mathsf{clim}_T(g)$$

for all morphisms f from (D_1, Γ_1) to (D_2, Γ_2) and g from (D_2, Γ_2) to (D_3, Γ_3) because $\mathsf{clim}_T(f;g)$ is the unique morphism from $\mathsf{clim}_T(\Theta_1)$ to $\mathsf{clim}_T(\Theta_3)$ making the outer rectangle commute for all $d \in \mathrm{D}_1$ in the following diagram:

$$\begin{tikzcd}
\mathsf{clim}_T(\Theta_1) \arrow[r, "\mathsf{clim}_T(f)"] & \mathsf{clim}_T(\Theta_2) \arrow[r, "\mathsf{clim}_T(g)"] & \mathsf{clim}_T(\Theta_3) \\
\Gamma_1(d) \arrow[u, "\iota^{\Gamma_1}(d)"] \arrow[r, "f^\mu_d"'] & \Gamma_2(f^F(d)) \arrow[u, "\iota^{\Gamma_2}_{f^F(d)}"] \arrow[r, "g^\mu_{g^F(f^F(d))}"'] & \Gamma_3(g^F(f^F(d))) \arrow[u, "\iota^{\Gamma_3}_{g^F(f^F(d))}"']
\end{tikzcd}$$

To define $\mathsf{Sen}_{\mathcal{R}_\mathcal{I}}$ as $\mathsf{Flat}(\mathsf{Fun}(\mathsf{Sig}^\mathcal{A}));\mathsf{clim}_{\mathrm{S{\scriptsize IGN}}_\mathcal{I}};\mathsf{Sen}_\mathcal{I}$, we have to require that $\mathrm{S{\scriptsize IGN}}_\mathcal{I}$ is cocomplete because otherwise the functor $\mathsf{clim}_{\mathrm{S{\scriptsize IGN}}_\mathcal{I}}$ would not be defined.

Definition 5.7 ($\mathsf{Sen}_{\mathcal{R}_\mathcal{I}}$) *Let $\mathcal{I}$ be an institution with a cocomplete category of signatures. Then*

$$\mathsf{Sen}_{\mathcal{R}_\mathcal{I}} = \mathsf{Flat}(\mathsf{Fun}(\mathsf{Sig}^\mathcal{A}));\mathsf{clim}_{\mathrm{S{\scriptsize IGN}}_\mathcal{I}};\mathsf{Sen}_\mathcal{I}.$$

Note that we can always choose the colimits of $\Gamma : \mathrm{D} \to \mathrm{A{\scriptsize DT}}_\mathcal{I}$ and $\Gamma;\mathsf{Sig}^\mathcal{A} : \mathrm{D} \to \mathrm{S{\scriptsize IGN}}_\mathcal{I}$ in such a way that $\mathsf{Sig}^\mathcal{A}(\coprod\Gamma) = \coprod\Gamma;\mathsf{Sig}^\mathcal{A}$ because there is a one-to-one correspondence between the class of colimits of Γ and the class of colimits of $\Gamma;\mathsf{Sig}^\mathcal{A}$. Assume that Γ has as a colimit (Σ, M) then, because $\mathsf{Sig}^\mathcal{A}$ preserves colimits (cf. Theorem 4.4), we have that Σ is a colimit of $\Gamma;\mathsf{Sig}^\mathcal{A}$. On the other hand, if we have given a colimit Σ of $\Gamma;\mathsf{Sig}^\mathcal{A}$ then, because $\mathsf{Sig}^\mathcal{A}$ creates colimits (cf. Theorem 4.1), we can construct a class of Σ-structures M, such that (Σ, M) is a colimit of Γ. This leads to the following theorem:

Theorem 5.8 *Provided that $\mathrm{A{\scriptsize DT}}_\mathcal{I}$ is cocomplete, we have*

$$\mathsf{Flat}(\mathsf{Fun}(\mathsf{Sig}^\mathcal{A}));\mathsf{clim}_{\mathrm{S{\scriptsize IGN}}_\mathcal{I}} = \mathsf{clim}_{\mathrm{A{\scriptsize DT}}_\mathcal{I}};\mathsf{Sig}^\mathcal{A}.$$

Proof. Let (D, Γ) be an object of $\mathsf{Flat}(\mathsf{Fun}(\mathrm{A{\scriptsize DT}}))$, then

$$\mathsf{Flat}(\mathsf{Fun}(\mathsf{Sig}^\mathcal{A}))((\mathrm{D}, \Gamma)) = (\mathrm{D}, \Gamma;\mathsf{Sig}^\mathcal{A})$$

and $\mathsf{clim}_{\mathrm{S{\scriptsize IGN}}_\mathcal{I}}((\mathrm{D}, \Gamma;\mathsf{Sig}^\mathcal{A})) = \coprod_\mathrm{D}\Gamma;\mathsf{Sig}^\mathcal{A}$. For the right-hand side we get:

$$\mathsf{Sig}^\mathcal{A}(\mathsf{clim}_{\mathrm{A{\scriptsize DT}}_\mathcal{I}}((\mathrm{D}, \Gamma))) = \mathsf{Sig}^\mathcal{A}(\coprod_\mathrm{D}\Gamma).$$

Since $\mathsf{Sig}^\mathcal{A}$ preserves and creates colimits, we have

$$\coprod_\mathrm{D}\Gamma;\mathsf{Sig}^\mathcal{A} = \mathsf{Sig}^\mathcal{A}(\coprod_\mathrm{D}\Gamma).$$

□

However, this relationship does not hold, in general, for limits of $\Gamma; \mathsf{Mod}^{\mathcal{A}}$ and colimits of Γ. Consider for example the functor $\Gamma : \mathrm{K}_2 \to \textsc{Adt}_{\mathcal{EQ}}$ in the institution $\mathcal{EQ}$ of equational logic. One possible choice for a limit of $\Gamma; \mathsf{Mod}^{\mathcal{A}}$ is the cartesian product $\mathsf{Mod}^{\mathcal{A}}(\Gamma(1)) \times \mathsf{Mod}^{\mathcal{A}}(\Gamma(2))$. For this limit it is impossible to find an abstract datatype (Σ, M) such that $M = \mathsf{Mod}^{\mathcal{A}}(\Gamma(1)) \times \mathsf{Mod}^{\mathcal{A}}(\Gamma(1))$ since there does not exist a many-sorted signature $\Sigma = (S, \Omega)$ such that Σ-structures are pairs of algebras (A_1, A_2). However, we can find an abstract datatype (Σ', M') such that M' and $\mathsf{Mod}^{\mathcal{A}}(\Gamma(1)) \times \mathsf{Mod}^{\mathcal{A}}(\Gamma(2))$ are isomorphic. This leads to the following theorem:

Theorem 5.9 *The contravariant functor* $\mathsf{Str}_{\mathcal{R}_\mathcal{I}} : \textsc{Sign}^{op}_{\mathcal{R}_\mathcal{I}} \to \textsc{Cat}$ *is natural isomorphic to the contravariant functor* $\mathsf{clim}_{\textsc{Adt}_\mathcal{I}}; \mathsf{Mod}^{\mathcal{A}}$ *provided that the structure functor* $\mathsf{Str}_\mathcal{I}$ *preserves colimits.*

Proof. Since $\mathsf{Str}_\mathcal{I}$ preserves colimits, so does $\mathsf{Mod}^{\mathcal{A}}$ (cf. Theorem 4.5). Thus, the natural isomorphism $\mu : \mathsf{Str}_{\mathcal{R}_\mathcal{I}} \Rightarrow \mathsf{clim}_{\textsc{Adt}_\mathcal{I}}; \mathsf{Mod}^{\mathcal{A}}$ is given for each $\Theta = (\mathrm{D}, \Gamma)$ in $\textsc{Sign}_{\mathcal{R}_\mathcal{I}}$ by the isomorphism between $\mathsf{Mod}^{\mathcal{A}}(\coprod \Gamma)$ and an arbitrary limit $\prod \Gamma; \mathsf{Mod}^{\mathcal{A}}$ of $\Gamma; \mathsf{Mod}^{\mathcal{A}}$. We have to show that the family of all isomorphisms $\mu_\Theta : \mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta) \to \mathsf{Mod}^{\mathcal{A}}(\coprod \Gamma)$ is indeed a natural transformation, that is, the following diagram commutes for all morphisms $f : \Theta \to \Theta'$ in $\textsc{Sign}_{\mathcal{R}_\mathcal{I}}$.

$$\begin{tikzcd}
\mathsf{Mod}^{\mathcal{A}}(\coprod \Gamma) & \mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta) \arrow[l, "\mu_\Theta"'] \\
\mathsf{Mod}^{\mathcal{A}}(\coprod \Gamma') \arrow[u, "\mathsf{Mod}^{\mathcal{A}}(\mathsf{clim}_{\textsc{Adt}_\mathcal{I}}(f))"] & \mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta') \arrow[l, "\mu_{\Theta'}"] \arrow[u, "\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(f)"']
\end{tikzcd}$$

First consider the following family of diagrams for $d \in \mathrm{D}$:

$$\begin{tikzcd}
\mathsf{Mod}^{\mathcal{A}}(\coprod \Gamma) \arrow[rrdd, "\mathsf{Mod}^{\mathcal{A}}(\iota^\Gamma_d)"'] & & & & \mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta) \arrow[llll, "\mu_\Theta"'] \arrow[lldd, "\pi_d^{\Gamma;\mathsf{Mod}^{\mathcal{A}}}"] \\
& & (1) & & \\
& (2) & \mathsf{Mod}(\Gamma(d)) & (4) & \\
& & \mathsf{Mod}^{\mathcal{A}}(\Gamma'(f^F(d))) \arrow[u, "\mathsf{Mod}^{\mathcal{A}}(f^\mu_d)"'] & & \\
& & (3) & & \\
\mathsf{Mod}^{\mathcal{A}}(\coprod \Gamma') \arrow[uuuuu, "\mathsf{Mod}^{\mathcal{A}}(\mathsf{clim}_{\textsc{Adt}_\mathcal{I}}(f))"] \arrow[rruu, "\mathsf{Mod}^{\mathcal{A}}(\iota^{\Gamma'}_{f^F(d)})"] & & & & \mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta') \arrow[llll, "\mu_{\Theta'}"] \arrow[lluu, "\pi^{\Gamma;\mathsf{Mod}^{\mathcal{A}}}_{f^F(d)}"'] \arrow[uuuuu, "\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(f)"']
\end{tikzcd}$$

The diagrams (1) and (3) commute because the functor $\mathsf{Mod}^{\mathcal{A}}$ preserves colimits and $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta)$ is defined as $\prod_{\mathrm{D}} \Theta; \mathsf{Mod}^{\mathcal{A}}$. Diagram (4) commutes because $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(f)$ is defined as the unique morphism making (4) commute, and similar diagram (2) commutes because of the definition of $\mathsf{clim}_{\mathrm{ADT}_\mathcal{I}}(f)$ and because a functor preserves composition.

Note that for this type of diagram, if the subdiagrams (1), (2), (3), and (4) commute for all $d \in \mathrm{D}$, this does not immediately imply that the functors $\mu_{\Theta'}; \mathsf{Mod}^\mathcal{A}(\mathsf{clim}_{\mathrm{ADT}_\mathcal{I}}(f))$ and $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(f); \mu_\Theta$ are the same.

However, because $\mathsf{Mod}^\mathcal{A}(\coprod \Gamma)$ is a limit of $\Gamma; \mathsf{Mod}^\mathcal{A}$, there exists a unique functor

$$H : \mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta') \to \mathsf{Mod}^\mathcal{A}(\coprod \Gamma)$$

which makes the following diagram commute for all $d \in \mathrm{D}$:

$$\begin{tikzcd}
\mathsf{Mod}^\mathcal{A}(\coprod \Gamma) \arrow[d, "\mathsf{Mod}^\mathcal{A}(\iota_d^\Gamma)"'] & \mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta') \arrow[l, "H"'] \arrow[dl, "\pi_{fF(d)}^{\Gamma;\mathsf{Mod}^\mathcal{A}};\mathsf{Mod}^\mathcal{A}(f_d^\mu)"] \\
\mathsf{Mod}^\mathcal{A}(\Gamma(d)) &
\end{tikzcd}$$

Since $\mu_{\Theta'}; \mathsf{Mod}^\mathcal{A}(\mathsf{clim}_{\mathrm{ADT}_\mathcal{I}}(f))$ and $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(f); \mu_\Theta$ have the same property, we get:

$$\mu_{\Theta'}; \mathsf{Mod}^\mathcal{A}(\mathsf{clim}_{\mathrm{ADT}_\mathcal{I}}(f)) = H = \mathsf{Str}_{\mathcal{R}_\mathcal{I}}(f); \mu_\Theta.$$

□

Satisfaction Theorems 5.8 and 5.9 from the previous section allow us to define the satisfaction relation $\models^{\mathcal{R}_\mathcal{I}}$.

On one hand, there exists a unique element $\overline{m} = \mu_\Theta(m)$ in $\mathsf{Mod}^\mathcal{A}(\mathsf{clim}_{\mathrm{ADT}_\mathcal{I}}(\Theta))$ for each m in $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta)$ given by the natural isomorphism μ between $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}$ and $\mathsf{clim}_{\mathrm{ADT}_\mathcal{I}}; \mathsf{Mod}^\mathcal{A}$ (cf. Theorem 5.9), which, by the definition of the category $\mathrm{ADT}_\mathcal{I}$, is also an element of $\mathsf{Str}_\mathcal{I}(\mathsf{Sig}^\mathcal{A}(\mathsf{clim}_{\mathrm{ADT}_\mathcal{I}}(\Theta)))$.

On the other hand, since $\mathsf{Flat}(\mathsf{Fun}(\mathsf{Sig}^\mathcal{A})); \mathsf{clim}_{\mathrm{SIGN}_\mathcal{I}}$ and $\mathsf{clim}_{\mathrm{ADT}_\mathcal{I}}; \mathsf{Sig}^\mathcal{A}$ are equal (cf. Theorem 5.8), each formula $\varphi \in \mathsf{Sen}_{\mathcal{R}_\mathcal{I}}(\Theta)$ is also an element of $\mathsf{Sig}^\mathcal{A}(\mathsf{clim}_{\mathrm{ADT}_\mathcal{I}}(\Theta))$. This allows us to define:

Definition 5.10 ($\models^{\mathcal{R}_\mathcal{I}}$) *Let Θ be a signature in $\mathrm{SIGN}_{\mathcal{R}_\mathcal{I}}$, m a structure in $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta)$, and φ a formula in $\mathsf{Sen}_{\mathcal{R}_\mathcal{I}}(\Theta)$, then*

$$m \models^{\mathcal{R}_\mathcal{I}} \varphi \textit{ if and only if } \overline{m} \models^\mathcal{I} \varphi.$$

For $\mathcal{R}_\mathcal{I}$ to be an institution, it remains to show that the satisfaction condition holds.

Theorem 5.11 *For all signature morphisms $f : \Theta \to \Theta'$ in $\mathrm{SIGN}_{\mathcal{R}_\mathcal{I}}$, Θ'-structures m, and Θ-formulas φ we have:*

$$m \models^{\mathcal{R}_\mathcal{I}} \mathsf{Sen}_{\mathcal{R}_\mathcal{I}}(f)(\varphi) \textit{ if and only if } m|_f \models^{\mathcal{R}_\mathcal{I}} \varphi.$$

Proof. Let f be a morphism from Θ to Θ' in $\mathrm{SIGN}_{\mathcal{R}_\mathcal{I}}$, m a Θ'-structure, and φ a Θ-formula, then we have:

$$\begin{array}{lrll}
 & m & \models^{\mathcal{R}_\mathcal{I}} \mathsf{Sen}_{\mathcal{R}_\mathcal{I}}(f)(\varphi) & \\
\text{iff} & \overline{m} & \models^{\mathcal{I}} \mathsf{Sen}_{\mathcal{I}}(\mathsf{Sig}^{\mathcal{A}}(\mathsf{clim}_{\mathrm{ADT}_\mathcal{I}}(f))) & \mid \text{Def. } \models^{\mathcal{R}_\mathcal{I}} \\
\text{iff} & \overline{m}|_{\mathsf{Sig}^{\mathcal{A}}(\mathsf{clim}_{\mathrm{ADT}_\mathcal{I}}(f))} & \models^{\mathcal{I}} \varphi & \mid \text{Sat. Cond. } \mathcal{I} \\
\text{iff} & \overline{m}|_{\mathsf{clim}_{\mathrm{ADT}_\mathcal{I}}(f)} & \models^{\mathcal{I}} \varphi & \mid \text{Def. } \mathrm{ADT}_\mathcal{I}\text{-morph.} \\
\text{iff} & \overline{m|_f} & \models^{\mathcal{I}} \varphi & \mid * \\
\text{iff} & m|_f & \models^{\mathcal{R}_\mathcal{I}} \varphi & \mid \text{Def. } \models^{\mathcal{R}_\mathcal{I}}
\end{array}$$

Equivalence $(*)$ holds because the naturality of the isomorphism between the functors $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}$ and $\mathsf{clim}_{\mathrm{ADT}_\mathcal{I}};\mathsf{Mod}^{\mathcal{A}}$ implies $\overline{m}|_{\mathsf{clim}_{\mathrm{ADT}_\mathcal{I}}(f)} = \overline{m|_f}$. □

Definition of $\mathcal{R}_\mathcal{I}$ To sum up the previous paragraphs, we define:

Definition 5.12 ($\mathcal{R}_\mathcal{I}$) *Given an institution $\mathcal{I}$ with a cocomplete category of signatures $\mathrm{SIGN}_\mathcal{I}$ and a colimit preserving structure functor $\mathsf{Str}_\mathcal{I}$. The institution*

$$\mathcal{R}_\mathcal{I} = (\mathrm{SIGN}_{\mathcal{R}_\mathcal{I}}, \mathsf{Str}_{\mathcal{R}_\mathcal{I}}, \mathsf{Sen}_{\mathcal{R}_\mathcal{I}}, \models^{\mathcal{R}_\mathcal{I}})$$

is defined as follows:

- $\mathrm{SIGN}_{\mathcal{R}_\mathcal{I}} = \mathsf{Flat}(\mathsf{Fun}(\mathrm{ADT}_\mathcal{I}))$,
- $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta) = \prod_{\mathrm{D}} \Gamma;\mathsf{Mod}^{\mathcal{A}}$ *and* $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(f) = \prod_{\mathrm{D}}(f^F;\pi^{\Gamma';\mathsf{Mod}^{\mathcal{A}}});(f^\mu;\mathsf{Mod}^{\mathcal{A}})$ *for all morphisms $f : \Theta \to \Theta'$ and objects $\Theta = (\mathrm{D},\Gamma)$ and $\Theta' = (\mathrm{D}',\Gamma')$ in* $\mathrm{SIGN}_{\mathcal{R}_\mathcal{I}}$,
- $\mathsf{Sen}_{\mathcal{R}_\mathcal{I}} = \mathsf{Flat}(\mathsf{Fun}(\mathsf{Sig}^{\mathcal{A}}));\mathsf{clim}_{\mathrm{SIGN}_\mathcal{I}};\mathsf{Sen}_\mathcal{I}$, *and*
- *$m \models^{\mathcal{R}_\mathcal{I}} \varphi$ iff $\overline{m} \models^{\mathcal{I}} \varphi$ for all $m \in \mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta)$, $\varphi \in \mathsf{Sen}_{\mathcal{R}_\mathcal{I}}(\Theta)$, and $\Theta \in \mathrm{SIGN}_{\mathcal{R}_\mathcal{I}}$.*

Note that the construction of the functors $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}$ and $\mathsf{Sen}_{\mathcal{R}_\mathcal{I}}$ are very similar. Given an object (D,Γ) then $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}((\mathrm{D},\Gamma))$ is given by $\prod_{\mathrm{D}}\Gamma;\mathsf{Mod}^{\mathcal{A}}$, while $\mathsf{Sen}_{\mathcal{R}_\mathcal{I}}((\mathrm{D},\Gamma))$ is given by $\mathsf{Sen}_\mathcal{I}(\coprod_{\mathrm{D}}\Gamma;\mathsf{Sig}^{\mathcal{A}})$. However, we have given an explicit definition of $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}$ while $\mathsf{Sen}_{\mathcal{R}_\mathcal{I}}$ was given as a composition of functors. And indeed, we can define $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}$ similar to $\mathsf{Sen}_{\mathcal{R}_\mathcal{I}}$ as

$$\mathsf{Str}_{\mathcal{R}_\mathcal{I}} = \mathsf{Flat}(\mathsf{Fun}(\mathsf{Mod}^{\mathcal{A}}));\mathsf{lim}_{\mathrm{CAT}}.$$

Note, however, since $\mathsf{Mod}^{\mathcal{A}}$ is a contravariant functor, $\mathsf{Fun}(\mathsf{Mod}^{\mathcal{A}})$ is a family of contravariant functors indexed by CAT, that is, $\mathsf{Fun}(\mathsf{Mod}^{\mathcal{A}})_{\mathrm{D}}$ is a contravariant functor from the functor category of covariant functors, $\mathsf{Fun}(\mathrm{ADT}_\mathcal{I})_{\mathrm{D}} = \mathrm{ADT}_\mathcal{I}^{\mathrm{D}}$, to the category of contravariant functors from D to CAT, which we denote by $\mathsf{CFun}(\mathrm{ADT}_\mathcal{I})_{\mathrm{D}} = \mathrm{CAT}^{\mathrm{D}^{op}}$. Then flattening a family of contravariant functors yields a covariant functor from $\mathsf{Flat}(\mathsf{Fun}(\mathrm{ADT}_\mathcal{I}))$ to $\mathsf{Flat}(\mathsf{CFun}(\mathrm{CAT})^{op})$. That is, a morphism

$$(f^F : \mathrm{D} \to \mathrm{D}', f^\mu : \Gamma \Rightarrow f^F;\Gamma')$$

in $\mathsf{Flat}(\mathsf{Fun}(\textsc{Adt}_{\mathcal{I}}))$ is mapped to the morphism

$$(f^F : \mathrm{D} \to \mathrm{D}', \mathsf{Mod}^{\mathcal{A}}(f^\mu) : f^F;\Gamma';\mathsf{Mod}^{\mathcal{A}} \Rightarrow \Gamma;\mathsf{Mod}^{\mathcal{A}})$$

in $\mathsf{Flat}(\mathsf{CFun}(\textsc{Cat})^{op})$.

Then $\mathsf{lim}_{\textsc{Cat}}$ maps an object $(\mathrm{D}, \Gamma : \mathrm{D}^{op} \to \textsc{Cat})$ in $\mathsf{Flat}(\mathsf{CFun}(\textsc{Cat})^{op})$ to $\prod_{\mathrm{D}} \Gamma$ in $\textsc{Cat}$ and a morphism $(f^F : \mathrm{D} \to \mathrm{D}', f^\mu : f^F;\Gamma' \Rightarrow \Gamma)$ in $\mathsf{Flat}(\mathsf{CFun}(\textsc{Cat})^{op})$ to the unique morphism $\prod_{\mathrm{D}'}(f^F;\pi^{\Gamma'});f^\mu$ from $\prod_{\mathrm{D}} \Gamma'$ to $\prod_{\mathrm{D}} \Gamma$ given by the limit property of $\prod_{\mathrm{D}} \Gamma$.

Note the similarity between the category of many-sorted signatures $\textsc{Sig}$ (cf. Section 3.1) and the category $\textsc{Sign}_{\mathcal{R}_{\mathcal{I}}}$. Both can be defined by flattening an indexed category, $\mathsf{MS} : \textsc{Set}^{op} \to \textsc{Cat}$ for $\textsc{Sig}$ (cf. Section 2.5) and $\mathsf{Fun}(\textsc{Adt}_{\mathcal{I}}) : \textsc{Cat}^{op} \to \textsc{Cat}$ for $\textsc{Sign}_{\mathcal{R}_{\mathcal{I}}}$. For a set S, MS_S is the set of functions from S^+ to $\textsc{Set}$, and for a category D, $\mathsf{Fun}(\textsc{Adt}_{\mathcal{I}})_{\mathrm{D}}$ is the set of functors from D to $\textsc{Adt}_{\mathcal{I}}$. We shall see in the next section that also the construction of colimits is very similar.

Abstract datatypes in $\mathcal{R}_{\mathcal{I}}$, objects in $\textsc{Adt}_{\mathcal{R}_{\mathcal{I}}}$, are pairs (Θ, M) where M is a full subcategory of $\mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}(\Theta) = \prod \Gamma;\mathsf{Mod}^{\mathcal{A}}$. Morphisms f from (Θ_1, M_1) to (Θ_2, M_2) in $\textsc{Adt}_{\mathcal{R}_{\mathcal{I}}}$ are type morphisms $f = \Theta_f$ from Θ_1 to Θ_2 such that $\mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}(f)(M_2) \subseteq M_1$. Thus we have the following fact:

Fact 5.13 *The category of relations between abstract datatypes* $\textsc{Rel}$*, defined in Section 5.2, is the same as the category of abstract datatypes* $\textsc{Adt}_{\mathcal{R}_{\mathcal{I}}}$ *in* $\mathcal{R}_{\mathcal{I}}$ *for an institution* $\mathcal{I}$ *with a cocomplete category of signatures and a structure preserving structure functor.*

Note that the definition of $\textsc{Rel}$ does not require $\textsc{Sign}_{\mathcal{I}}$ to be cocomplete and $\mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}$ to preserve colimits. This is only needed for the definition of formulas and satisfaction in $\mathcal{R}_{\mathcal{I}}$.

Example In the example of the counter the increment operation is the abstract datatype $(\Theta_{\mathtt{Inc}}, M)$ in the institution $\mathcal{R}_{\mathcal{LSL}}$ with $\mathcal{LSL}$ as the base institution where $\Theta_{\mathtt{Inc}} \in \textsc{Sign}_{\mathcal{R}_{\mathcal{LSL}}}$ is the type $(\mathrm{V}, \Gamma_{\mathtt{Inc}})$. $\Gamma_{\mathtt{Inc}}$ can be depicted by the following diagram

$$\begin{tikzcd} & {[\![C]\!]} & & {[\![C]\!]} \\ \Gamma_{\mathtt{Inc}} = & & {[\![\mathtt{Nat}]\!]} \arrow[ul, hook] \arrow[ur, hook'] & \end{tikzcd}$$

where

```
C : trait
includes Nat
introduces c: N
```

and M is given by all pairs (A, B) in

$$\prod \Gamma_{\mathtt{Inc}};\mathsf{Mod}^{\mathcal{A}} = \mathsf{Mod}^{\mathcal{A}}([\![C]\!]) \times_{\mathsf{Mod}^{\mathcal{A}}([\![\mathtt{Nat}]\!])} \mathsf{Mod}^{\mathcal{A}}([\![C]\!]),$$

with $B(c) = A(\mathtt{succ})(A(c))$.

To fix a colimit of $\Gamma_{\mathtt{Inc}};\mathsf{Sig}^{\mathcal{A}}$ for the definition of $\mathsf{Sen}_{\mathcal{R}_{\mathcal{LSL}}}(\Theta_{\mathtt{Inc}})$ we use the convention that the components of the state after an operation is performed are decorated with a prime. Then the formulas in $\mathsf{Sen}_{\mathcal{R}_{\mathcal{LSL}}}(\Theta_{\mathtt{Inc}})$ are the formulas in $\mathsf{Sen}_{\mathcal{LSL}}(\coprod \Gamma_{\mathtt{Inc}};\mathsf{Sig}^{\mathcal{A}})$ where $\coprod \Gamma_{\mathtt{Inc}};\mathsf{Sig}^{\mathcal{A}}$ is the signature

$$(\{N\}, \{\mathtt{zero} : N, \mathtt{succ} : N \to N, c : N, c' : N\}).$$

Using the formulas in $\mathsf{Sen}_{\mathcal{R}_{\mathcal{LSL}}}(\Gamma_{\mathtt{Inc}})$, M can be characterized by all pairs (A, B) from $\mathsf{Str}_{\mathcal{R}_{\mathcal{LSL}}}(\Gamma_{\mathtt{Inc}})$ such that

$$(A, B) \models^{\mathcal{R}_{\mathcal{LSL}}} c' = \mathtt{succ}(c),$$

which is, by the definition of satisfaction in $\mathcal{R}_{\mathcal{LSL}}$, equivalent to

$$A +_C B \models^{\mathcal{LSL}} c' = \mathtt{succ}(c),$$

where $A +_C B$ is the amalgamated of A and B with respect to $C = A|_\iota = B|_\iota$.

With the specification language $\mathtt{SL}_{\mathcal{I}}$ introduced in Section 4.2 and instantiated with $\mathcal{I} = \mathcal{R}_{\mathcal{LSL}}$, we can write

$$\mathtt{Inc} = I_{\{c'=\mathtt{succ}(s)\}}\Theta_{\mathtt{Inc}}$$

to define the increment operation.

5.4 Colimits in $\mathrm{SIGN}_{\mathcal{R}_{\mathcal{I}}}$

Since $\mathrm{SIGN}_{\mathcal{R}_{\mathcal{I}}}$ is defined as $\mathsf{Flat}(\mathsf{Fun}(\mathrm{ADT}_{\mathcal{I}}))$, we can use Theorem 2.28 to prove that $\mathrm{SIGN}_{\mathcal{R}_{\mathcal{I}}}$ is cocomplete. To apply this theorem, we have to check that

1. CAT is cocomplete,
2. $\mathrm{ADT}_{\mathcal{I}}^{\mathrm{D}}$ is cocomplete for each category D, and
3. the functor $\mathsf{Fun}(\mathrm{ADT}_{\mathcal{I}})_F : \mathrm{ADT}_{\mathcal{I}}^{\mathrm{D}'} \to \mathrm{D}$ has a left adjoint for all functors $F : \mathrm{D} \to \mathrm{D}'$.

CAT is cocomplete and since in the construction of $\mathcal{R}_{\mathcal{I}}$ we have assumed that $\mathrm{SIGN}_{\mathcal{I}}$ is cocomplete, so is $\mathrm{ADT}_{\mathcal{I}}$ and thus $\mathrm{ADT}_{\mathcal{I}}^{\mathrm{D}}$ is cocomplete for all categories D.

Thus it remains to show that $\mathsf{Fun}(\mathrm{ADT}_{\mathcal{I}})_F$ has a left adjoint for every functor $F : \mathrm{D} \to \mathrm{D}'$. Given a functor $\Gamma : \mathrm{D} \to \mathrm{ADT}_{\mathcal{I}}$, we are looking for a functor $L^F(\Gamma) : \mathrm{D}' \to \mathrm{ADT}_{\mathcal{I}}$ such that for each functor $\Gamma'' : \mathrm{D}' \to \mathrm{ADT}_{\mathcal{I}}$ and natural transformation $\mu : \Gamma \Rightarrow F;\Gamma''$ there exists a unique natural transformation $\overline{\mu} : L^F(\Gamma) \Rightarrow \Gamma''$:

$$\frac{\Gamma \xRightarrow{\mu} F;\Gamma''}{L^F(\Gamma) \xRightarrow{\overline{\mu}} \Gamma''}.$$

As observed by Tarlecki, Burstall and Goguen [58], finding for each $\Gamma : \mathrm{D} \to \textsc{Adt}_\mathcal{I}$ a functor $L^F(\Gamma) : \mathrm{D}' \to \textsc{Adt}_\mathcal{I}$ with the above properties is the same as finding the left Kan extension of Γ along $F : \mathrm{D} \to \mathrm{D}'$.

Definition 5.14 (Left Kan Extension) *Given categories T, D, and D' and a functor $F : \mathrm{D} \to \mathrm{D}'$. Let Γ be an object of T^{D}, that is, a functor from* D *to T, then the left Kan extension of Γ along F is a pair (Γ', η) consisting of a functor Γ' from* D′ *to T and a natural transformation η from Γ to $F;\Gamma'$ such that for all natural transformations $\nu : \Gamma \Rightarrow F;\Gamma''$ in $T^{\mathrm{D}'}$ there exists a unique natural transformation $\bar{\nu} : \Gamma' \Rightarrow \Gamma''$ such that the following diagram commutes in T^{D}:*

$$\begin{tikzcd}
\Gamma \arrow[r, Rightarrow, "\eta"] \arrow[dr, Rightarrow, "\nu"'] & F;\Gamma' \arrow[d, Rightarrow, "F;\bar{\nu}"] \\
 & F;\Gamma''
\end{tikzcd}$$

Note that if each Γ in T^{D} has a left Kan extension Γ' along F, then this determines a functor $L^F : T^{\mathrm{D}} \to T^{\mathrm{D}'}$ left adjoint to $\mathsf{Fun}(T)_F$ by $L^F(\Gamma) = \Gamma'$ and $L^F(\nu : \Gamma \Rightarrow \Gamma'') = \overline{\nu;\eta''}$ where η'' is the natural transformation associated to the left Kan extension of Γ'' along F:

$$\frac{\Gamma \xRightarrow{\nu} \Gamma'' \xRightarrow{\eta''} F; L^F(\Gamma'')}{L^F(\Gamma) \xRightarrow{L^F(\nu)} L^F(\Gamma'')}.$$

As an example consider the following categories

$$\mathrm{D} = \begin{tikzcd}
1 & & 2 \\
 & 0 \arrow[ul, "f"] \arrow[ur, "g"'] &
\end{tikzcd} \quad \text{and } \mathrm{D}' = \begin{tikzcd}
 & 3 & \\
1 \arrow[ur, "f'"] & & 2 \arrow[ul, "g'"'] \\
 & 0 \arrow[ul, "f"] \arrow[ur, "g"'] &
\end{tikzcd},$$

Let F be the inclusion of D into D′ and Γ an arbitrary functor from D to T.

The problem is to find $\Gamma'(0)$, $\Gamma'(1)$, $\Gamma'(2)$, and $\Gamma'(3)$. While it seems natural to choose $\Gamma'(0) = \Gamma(0)$, $\Gamma'(1) = \Gamma(1)$, and $\Gamma'(2) = \Gamma(2)$, the question remains what to choose for $\Gamma'(3)$. Intuitively $\Gamma'(3)$ is the “smallest” object in T which “includes” $\Gamma'(1)$ and $\Gamma'(2)$, that is, $\Gamma'(3)$ is the coproduct of $\Gamma'(1)$ and $\Gamma'(2)$.

To see that this indeed defines the left Kan extension of Γ along F, consider a functor $\Gamma'' : D' \to T$ and a natural transformation $\mu : \Gamma \Rightarrow F;\Gamma''$. Then we have to be able to extend μ to a natural transformation $\overline{\mu}$ from Γ' to Γ''. Again it is natural to choose $\overline{\mu}_0 = \mu_0$, $\overline{\mu}_1 = \mu_1$ and $\overline{\mu}_2 = \mu_2$. Now $\overline{\mu}_3 : \Gamma'(3) \to \Gamma''(3)$ is given by the universal property of the coproduct $\Gamma'(3) = \Gamma'(1) \uplus \Gamma'(2)$ and because $\mu_1;\Gamma''(f')$ is a morphism from $\Gamma'(1)$ to $\Gamma''(3)$ and $\mu_2;\Gamma''(g')$ is a morphism from $\Gamma'(2)$ to $\Gamma''(3)$.

Theorem 1 on page 233 of Mac Lane's book [43] provides a construction of the right Kan extension. The notion of right Kan extension is a dual to the notion of left Kan extension. Thus by reversing the arrows in the construction of the right Kan extension we get a left Kan extension.

Theorem 5.15 *(Mac Lane [43]) Under the assumptions of Definition 5.14 the left Kan extension of* Γ *along* F *exists provided that* T *is cocomplete.*

Proof. Here, we only provide the construction of the left Kan extension; the proof that this is indeed a left Kan extension can be found in [43]. Let Γ be a functor from D to T, then we have to provide a functor Γ' from D' to T. For each $d' \in \mathrm{D}'$ we take $\Gamma'(d')$ as the "smallest element" including $\Gamma(d)$ if there exists a morphism f from $F(d)$ to d' in D'. Thus $\Gamma'(d')$ is the colimit of a functor $F_{d'}$ from an appropriate category to T. The appropriate category is the comma category $(F \downarrow d')$. Objects of $(F \downarrow d')$ are pairs (d, f) consisting of an object d of D and an arrow f from $F(d)$ to d' in D'. Morphisms from (d_1, f_1) to (d_2, f_2) are arrows $g : d_1 \to d_2$ in D such that the following diagram commutes in D':

$$\begin{tikzcd}
F(d_1) \arrow[rr, "F(g)"] \arrow[dr, "f_1"'] & & F(d_2) \arrow[dl, "f_2"] \\
& d' &
\end{tikzcd}$$

Then $F_{d'} : (F \downarrow d') \to T$ is given by

$$F_{d'}((d, f)) = \Gamma(d) \text{ and } F_{d'}(g) = \Gamma(g)$$

for objects (d, f) and morphisms $g : (d_1, f_1) \to (d_2, f_2)$ in $(F \downarrow d')$.

Now we can define $\Gamma'(d')$ as $\coprod F_{d'}$ and $\Gamma'(g : d_1 \to d_2)$ as $\coprod \nu$ where ν is the natural transformation from F_{d_1} to $\Delta(\coprod F_{d_2})$ given by $\nu_{(d,f)} = \iota^{F_{d_2}}_{(d,f;g)}$.

Note that if (d, f) is an object of $(F \downarrow d_1)$ and g is a morphism from d_1 to d_2, then $(d, f; g)$ is an object of $(F \downarrow d_2)$ and $F_{d_1}((d, f)) = \Gamma(d) = F_{d_2}((d, f; g))$.

The colimit of $F_{d'}$ exists because T is cocomplete.

□

As an example of the construction of the left Kan extension, we define $\Gamma' : \mathrm{D}' \to T$ from the previous example.

The category $(F \downarrow 0)$ has one object $(0, \mathtt{id})$ and the identity as the only arrow. The functor F_0 maps $(0, \mathtt{id})$ to $\Gamma(0)$. The colimit of F_0 is $\Gamma(0)$ and thus $\Gamma'(0) = \Gamma(0)$.

The category $(F \downarrow 1)$ has two objects $(0, f)$ and $(1, \mathtt{id})$ and one arrow f from $(0, f)$ to $(1, \mathtt{id})$ other than the identity. Then $\Gamma'(1)$ is $\Gamma(1)$ because the colimit of F_1 is $\Gamma(1)$. Similar we get $\Gamma'(2) = \Gamma(2)$.

$(F \downarrow 3)$ is the category freely generated by

$$\begin{tikzcd}
(1, f') & (2, g') \\
(0, f; f') \arrow[u, "f"] & (0, g; g') \arrow[u, "g"']
\end{tikzcd}$$

and F_3 can be depicted by

$$\begin{tikzcd}
\Gamma(1) & \Gamma(2) \\
\Gamma(0) \arrow[u, "\Gamma(f)"] & \Gamma(0) \arrow[u, "\Gamma(g)"']
\end{tikzcd}$$

which has the coproduct of $\Gamma(1)$ and $\Gamma(2)$ as its colimit. Note that $(0, f; f')$ and $(0, g; g')$ are two different objects in $(F \downarrow 3)$, and therefore the colimit of F_3 is the coproduct of $\Gamma(1)$ and $\Gamma(2)$ and not the pushout of $\Gamma(1)$ and $\Gamma(2)$ with respect to $\Gamma(0)$.

The morphism $\Gamma'(f)$ is $\Gamma(f)$, $\Gamma'(g) = \Gamma(g)$, $\Gamma'(f')$ is the injection ι_1 of $\Gamma(1)$ into the coproduct $\Gamma(1) \uplus \Gamma(2)$, and $\Gamma'(g')$ the injection ι_2 of $\Gamma(2)$ into $\Gamma(1) \uplus \Gamma(2)$. Thus Γ' can be depicted as:

$$\begin{tikzcd}
& \Gamma(1) \uplus \Gamma(2) & \\
\Gamma(1) \arrow[ur, "\iota_1"] & & \Gamma(2) \arrow[ul, "\iota_2"'] \\
& \Gamma(0) \arrow[ul, "\Gamma(f)"] \arrow[ur, "\Gamma(g)"'] &
\end{tikzcd}$$

In the proof of the existence of a left Kan extension (cf. Theorem 5.15) we had to require cocompleteness of T because we needed the colimit of the functor $F_{d'} : (F \downarrow d') \to T$. However, provided that D and D' are finite categories, it suffices that T is finitely cocomplete because $(F \downarrow d)'$ will be always finite if D and D$'$ are finite.

COROLLARY 5.16 *Under the assumptions of Definition 5.14 the left Kan extension of* Γ *along* F *exists provided that* T *is finitely cocomplete and that* D *and* D' *are finite.*

Let T be a (finitely) cocomplete category, then Theorem 5.15 implies the existence of a left adjoint functor for $\mathsf{Fun}(T)_F$ for every functor F from a (finite) category D to a (finite) category D$'$. Together with Theorem 2.28 we get:

THEOREM 5.17 *The category* $\mathsf{Flat}(\mathsf{Fun}(T))$ *is (finitely) cocomplete if* T *is (finitely) cocomplete.*

Since $\mathrm{A\scriptstyle{DT}}_{\mathcal{I}}$ is cocomplete, we get as a direct consequence:

Corollary 5.18 *The category* $\textsc{Sign}_{\mathcal{R}_{\mathcal{I}}}$ *is cocomplete.*

The initial object in $\textsc{Sign}_{\mathcal{R}_{\mathcal{I}}}$ is $(\mathbf{0}, \Gamma_\perp)$ where $\Gamma_\perp$ is the unique functor from $\mathbf{0}$ to $\textsc{Adt}$.

In the following we study types of functors $F : J \to \textsc{Sign}_{\mathcal{R}_{\mathcal{I}}}$ which do not require the existence of left adjoint functors to construct their colimit. An example of these type of diagrams can be found in the definition of the sequential composition of relations from $\textsc{Adt}_{\mathcal{R}_{\mathcal{I}}}$ (cf. Section 6.3).

Instead of proving the next theorem for $\textsc{Sign}_{\mathcal{R}_{\mathcal{I}}} = \mathsf{Flat}(\mathsf{Fun}(\textsc{Adt}_{\mathcal{I}}))$, we give a proof for $\mathsf{Flat}(\mathsf{Fun}(T))$ where T is any category. Note that for this theorem we do not have to require that T is (finitely) cocomplete.

Theorem 5.19 *Given a functor F from J to $\mathsf{Flat}(\mathsf{Fun}(T))$. Let $F(i) = (\mathrm{D}_i, \Gamma_i)$ and $F(f) = (f^F, f^\mu)$ such that f^μ is the identity on Γ_i, that is, $\Gamma_i = f^F; \Gamma_j$ for all objects i and morphisms $f : i \to j$ in J. Let further $F_{\textsc{Cat}}$ be the functor from J to $\textsc{Cat}$ defined by $F_{\textsc{Cat}}(i) = \mathrm{D}_i$ and $F_{\textsc{Cat}}(f) = f^F$ for every i and f in J. Then the colimit (D_{cl}, Γ_{cl}) of F is given by the colimit D_{cl} of $F_{\textsc{Cat}}$ and the unique functor Γ_{cl} from D_{cl} to T given by the colimit property of D_{cl} w.r.t. the natural transformation $\nu : F_{\textsc{Cat}} \Rightarrow \Delta T$ with $\nu_i = \Gamma_i$ for every $i \in J$.*

Proof. First note that ν is indeed a natural transformation because

$$\nu_i = \Gamma_i = f^F; \Gamma_j = F_{\textsc{Cat}}(f); \nu_j$$

for all morphisms $f : i \to j$ in J:

$$\begin{tikzcd}
D_i \arrow[r, "\nu_i=\Gamma_i"] \arrow[d, "F_{\textsc{Cat}}(f)=f^F"'] & T \arrow[d, equal, "(\Delta T)(f)=\mathtt{id}_T"] \\
D_j \arrow[r, "\nu_j=\Gamma_j"'] & T
\end{tikzcd}$$

Using the representation of colimits in $\textsc{Cat}$ from Theorem 2.3 we can write the functor Γ_{cl} as

$$\Gamma_{cl}([(i,d)]_\equiv) = \Gamma_i(d) \text{ and } \Gamma_{cl}([(i, f : d \to d')]_\equiv) = \Gamma_i(f).$$

The co-cone morphisms ι_i^F are given by $\iota_i^F = (\iota_i^{F_{\textsc{Cat}}}, \mathtt{id})$ where $\iota_i^{F_{\textsc{Cat}}}(d) = [(i,d)]_\equiv$ for all $i \in J$ and $d \in \mathrm{D}_i$.

To show that $(\iota^F, \coprod F)$ is a colimit in $\mathsf{Flat}(\mathsf{Fun}(T))$, consider an object (D', Γ') and a natural transformation ϵ from F to $\Delta(\mathrm{D}', \Gamma')$. We have to find a unique morphism $h = (h^F, h^\mu)$ from $\coprod F$ to (D', Γ') such that the following diagram commutes for all $i \in J$:

$$\begin{tikzcd}
\coprod F \arrow[r, dashed, "(h^F{,}h^\mu)"] & (\mathrm{D}', \Gamma') \\
(\mathrm{D}_i, \Gamma_i) \arrow[u, "(\iota^{F_{\textsc{Cat}}}{,}\mathtt{id})"] \arrow[ur, "\epsilon_i=(\epsilon_i{}^F{,}\epsilon_i{}^\mu)"'] &
\end{tikzcd}$$

h^F is given by the colimit property of $\coprod F_{\textsc{Cat}}$ with respect to the natural transformation η from $F_{\textsc{Cat}}$ to $\Delta D'$ given by $\eta_i = {\epsilon_i}^F$ and is unique with the property $\iota^{F_{\textsc{Cat}}}; h^F = {\epsilon_i}^F$. Using the explicit representation of $\coprod F_{\textsc{Cat}}$ the functor h^F can be written as:

$$h^F([(i,d)]_\equiv) = {\epsilon_i}^F(d) \text{ and } h^F([(i, f : d \to d')]_\equiv) = {\epsilon_i}^F(f).$$

Define the natural transformation h^μ by $(h^\mu)_{[(i,d)]_\equiv} = ({\epsilon_i}^\mu)_d$ for $[(i,d)]_\equiv \in \coprod F_{\textsc{Cat}}$. We have to show that h^μ is well defined, that h^μ is a natural transformation from Γ_{cl} to $F_{h^\mu}; \Gamma'$, and that $\iota^F; h = \epsilon$.

For proving the well-definedness of h^μ consider $(i,d) \equiv (j,d')$ and $f : i \to j$ in J with $d' = F_{\textsc{Cat}}(f)(d)$:[1]

$$\begin{aligned}
(h^\mu)_{[(j,d')]_\equiv} &= ({\epsilon_j}^\mu)_{d'} && \mid \text{Def. of } h^\mu \\
&= ({\epsilon_j}^\mu)_{F_{\textsc{Cat}}(f)(d)} && \mid F_{\textsc{Cat}}(f)(d) = d' \\
&= (F_{\textsc{Cat}}(f); {\epsilon_j}^\mu)_d && \mid \text{Def. of } F_{\textsc{Cat}}(f)({\epsilon_i}^\mu) \\
&= ({\epsilon_i}^\mu)_d && \mid * \\
&= (h^\mu)_{[(i,d)]_\equiv} && \mid \text{Def. of } h^\mu
\end{aligned}$$

The equation (*) holds because ϵ is a natural transformation from F to $\Delta(D', \Gamma')$ and thus $\epsilon_i = F(f); \epsilon_j$, which implies ${\epsilon_i}^\mu = f^\mu; F_{\textsc{Cat}}(f)(\epsilon_j)$ by the definition of composition in $\mathsf{Flat}(\mathsf{Fun}(T))$. Then ${\epsilon_i}^\mu = F_{\textsc{Cat}}(f)(\epsilon_j)$ because f^μ is the identity natural transformation.

To show that h^μ is a natural transformation from Γ_{cl} to $h^F; \Gamma'$, consider a morphism $[(i, f : d \to d')]_\equiv : [(i,d)]_\equiv \to [(i,d')]_\equiv$ in $\coprod F_{\textsc{Cat}}$. We have to show that the following diagram commutes

$$\begin{tikzcd}
\Gamma_{cl}([(i,d)]_\equiv) \arrow[r, "(h^\mu)_{[(i,d)]_\equiv}"] \arrow[d, "\Gamma_{cl}([(i,f)]_\equiv)"'] & \Gamma'(h^F([(i,d)]_\equiv)) \arrow[d, "\Gamma'(h^F)([(i,f)]_\equiv)"] \\
\Gamma_{cl}([(i,d')]_\equiv) \arrow[r, "(h^\mu)_{[(i,d')]_\equiv}"'] & \Gamma'(h^F([(i,d')]_\equiv))
\end{tikzcd}$$

However, by the definition of $\coprod F$ and h^μ this is the same as

$$\begin{tikzcd}
\Gamma_i(d) \arrow[r, "({\epsilon_i}^\mu)_d"] \arrow[d, "\Gamma_i(f)"'] & \Gamma'({\epsilon_i}^F(d)) \arrow[d, "\Gamma'({\epsilon_i}^F(f))"] \\
\Gamma_i(d') \arrow[r, "({\epsilon_i}^\mu)_{d'}"'] & \Gamma'({\epsilon_i}^F(d'))
\end{tikzcd}$$

for all $i \in J$, which commutes since ${\epsilon_i}^\mu$ is a natural transformation from Γ_i to ${\epsilon_i}^F; \Gamma'$.

Now we have to show that $\iota^F; h = \epsilon$, that is, we have to show

$$\iota^{F_{\textsc{Cat}}}; h^F = {\epsilon_i}^F \text{ and } \iota_i^{F_{\textsc{Cat}}}(h^\mu) = {\epsilon_i}^\mu$$

[1] To show the well-definedness of a function $h : A/{\equiv_R} \to B$ defined by $h([a]_{\equiv_R}) = g(a)$, it suffices to show that $g(a) = g(a')$ if $a \; R \; a'$ because equality is an equivalence relation.

for each $i \in J$. The first equation holds because of the definition of h^F and for the second equation consider $d \in \mathrm{D}_i$ and

$$\begin{aligned}(\iota_i^{F_{\textsc{Cat}}}(h^\mu))_d &= (h^\mu)_{(\iota_i^{F_{\textsc{Cat}}}(d))} \\ &= (h^\mu)_{[(i,d)]_\equiv} && \mid \text{Def. } \iota_i^{F_{\textsc{Cat}}} \\ &= (\epsilon_i{}^\mu)_d && \mid \text{Def. of } h^\mu.\end{aligned}$$

□

Note that the construction of the above theorem does not work for arbitrary diagrams since, in general, ν, as defined above, is not a natural transformation because instead of $\nu_i(d) = \Gamma_i(d)$ being equal to $(F_{\textsc{Cat}}(f); \nu_j)(d) = \Gamma_j(f^F(d))$ for all $d \in \mathrm{D}_i$ we only know that $\Gamma_i(d)$ and $\Gamma_j(f^F(d))$ are related by the T-morphism f_d^μ.

5.5 Preservation of Colimits

In this section we prove that $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}$ preserves colimits, that is, $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\coprod F)$ is a limit of $F; \mathsf{Str}_{\mathcal{R}_\mathcal{I}}$ for functors $F : J \to \textsc{Sign}_{\mathcal{R}_\mathcal{I}}$. Because of Theorem 5.9, we know that $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}$ is natural isomorphic to $\mathsf{clim}_{\textsc{Adt}_\mathcal{I}}; \mathsf{Mod}^\mathcal{A}$, and Theorem 2.20 allows us to deduce from that fact and the fact that $\mathsf{clim}_{\textsc{Adt}_\mathcal{I}}; \mathsf{Mod}^\mathcal{A}$ preserves colimits that $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}$ preserves colimits.

We have that $\mathsf{Str}_\mathcal{I}$ preserves colimits, and thus $\mathsf{Mod}^\mathcal{A}$ preserves colimits. Because the composition of (co)limit preserving functors is again a (co)limit preserving functor, we are done if we can show that $\mathsf{clim}_{\textsc{Adt}_\mathcal{I}}$ preserves colimits. To this end we prove that $\mathsf{clim}_{\textsc{Adt}_\mathcal{I}}$ has a right adjoint functor $U : \textsc{Adt}_\mathcal{I} \to \textsc{Sign}_{\mathcal{R}_\mathcal{I}}$.

Lemma 5.20 *Let U be the functor from $\textsc{Adt}_\mathcal{I}$ to $\textsc{Sign}_{\mathcal{R}_\mathcal{I}}$ defined by*

$$U((\Sigma, M)) = (\mathbf{1}, \Gamma_{(\Sigma,M)}) \text{ and } U(\sigma) = (\mathsf{Id}_\mathbf{1}, \sigma^\mu)$$

for each abstract datatype (Σ, M) and morphism σ in $\textsc{Adt}_\mathcal{I}$ where $\Gamma_{(\Sigma,M)}$ maps the only object of $\mathbf{1}$ to (Σ, M) and $\sigma_1^\mu = \sigma$. Then U is a right adjoint for $\mathsf{clim}_{\textsc{Sign}_{\mathcal{R}_\mathcal{I}}}$.

Proof. The co-unit $\varepsilon : \mathsf{Id}_{\textsc{Adt}_\mathcal{I}} \Rightarrow U; \mathsf{clim}_{\textsc{Adt}_\mathcal{I}}$ of the adjunction is given by $\varepsilon_{(\Sigma,M)} = \mathtt{id}_{(\Sigma,M)}$ for every (Σ, M) in $\textsc{Adt}$. And for a morphism σ from $\mathsf{clim}_{\textsc{Sign}_\mathcal{I}}(\Theta)$ to (Σ, M) the morphism $\tilde{\sigma}$ from Θ to $U((\Sigma, M))$ is given by $(\tilde{\sigma}^F, \tilde{\sigma}^\mu)$ where $\tilde{\sigma}^F$ is the unique morphism from D to $\mathbf{1}$ because $\mathbf{1}$ is the terminal object in $\textsc{Cat}$ and $\tilde{\sigma}_d^\mu = \iota_d^\Gamma; \sigma$ for each object d in D. □

Theorem 2.25 and the previous lemma imply:

Corollary 5.21 *The functor $\mathsf{clim}_{\textsc{Sign}_\mathcal{I}} : \textsc{Sign}_{\mathcal{R}_\mathcal{I}} \to \textsc{Adt}_\mathcal{I}$ preserves colimits.*

Altogether we get the main theorem of this section:

Theorem 5.22 *The contravariant functor $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}$ from $\textsc{Sign}_{\mathcal{R}_\mathcal{I}}$ to $\textsc{Cat}$ preserves colimits.*

5.6 Relation between $\textsc{Adt}_{\mathcal{R}_\mathcal{I}}$ and $\textsc{Adt}_\mathcal{I}$

There is a strong relationship between objects (Θ_R, M_R) in $\textsc{Adt}_{\mathcal{R}_\mathcal{I}}$ and objects (Σ, M) in $\textsc{Adt}_\mathcal{I}$ resulting from the preservation of colimits of $\mathsf{Str}_\mathcal{I}$ and the fact that $\textsc{Sign}_\mathcal{I}$ and thus $\textsc{Adt}_\mathcal{I}$ are cocomplete.

Definition 5.23 *The functor* $\mathsf{G}^\mathcal{A} : \textsc{Adt}_{\mathcal{R}_\mathcal{I}} \to \textsc{Adt}_\mathcal{I}$ *is defined by*

$$\mathsf{G}^\mathcal{A}((\Theta_R, M_R)) = (\coprod_{\mathrm{D}_R} \Gamma_R ; \mathsf{Sig}^\mathcal{A}, \{\overline{m} \mid m \in M_R\})$$

$$\mathsf{G}^\mathcal{A}(f) = \mathsf{clim}_{\textsc{Adt}_\mathcal{I}}(f)$$

where $\Theta_R = (\mathrm{D}_R, \Gamma_R)$ *and* f *is an* $\textsc{Adt}_{\mathcal{R}_\mathcal{I}}$*-morphism from* $R = (\Theta_R, M_R)$ *to* $S = (\Theta_S, M_S)$*, that is,* f *is an* $\textsc{Sign}_{\mathcal{R}_\mathcal{I}}$*-morphism from* Θ_R *to* Θ_S *such that* $M_S|_f \subseteq M_R$*.*

For $\mathsf{G}^\mathcal{A}$ to be well-defined we have to check that $\mathsf{G}^\mathcal{A}(f)$ is indeed an $\textsc{Adt}_\mathcal{I}$-morphism from $\mathsf{G}^\mathcal{A}(R)$ to $\mathsf{G}^\mathcal{A}(S)$, that is,

$$\mathsf{Mod}^\mathcal{A}(\mathsf{G}^\mathcal{A}(S))|_{\mathsf{G}^\mathcal{A}(f)} \subseteq \mathsf{Mod}^\mathcal{A}(\mathsf{G}^\mathcal{A}(R)).$$

Let $\overline{m}_S$ be a model of $\mathsf{Mod}^\mathcal{A}(\mathsf{G}^\mathcal{A}(S))$, that is, m_S is an object of M_S. Then we have to show that $\overline{m}_S|_{\mathsf{clim}_{\textsc{Adt}_\mathcal{I}}(f)}$ is in $\mathsf{Mod}^\mathcal{A}(\mathsf{G}^\mathcal{A}(R))$. Since the isomorphism between $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}$ and $\mathsf{clim}_{\textsc{Adt}_\mathcal{I}} ; \mathsf{Mod}^\mathcal{A}$ is natural, this is equivalent to showing that $\overline{m_S|_f}$ is in $\mathsf{Mod}^\mathcal{A}(\mathsf{G}^\mathcal{A}(R))$. This holds by definition of $\mathsf{G}^\mathcal{A}$ and because f is an $\textsc{Adt}_{\mathcal{R}_\mathcal{I}}$-morphism, and therefore $m_S|_f$ is in M_R.

In the example of the counter $\mathsf{G}^\mathcal{A}((\Theta_{\mathtt{Inc}}, M_{\mathtt{Inc}}))$ is the abstract datatype (Σ_I, M_I) in the institution $\mathcal{LSL}$ where

$$\Sigma_I = \coprod_{\mathrm{V}} \Gamma_{\mathtt{Inc}} ; \mathsf{Sig}^\mathcal{A} = \Sigma_{\mathtt{Counter}} +_{\Sigma_{\mathtt{Nat}}} \Sigma_{\mathtt{Counter}}$$

and

$$M_I = \{A +_C B \mid (A, B) \in M_{\mathtt{Inc}}\}.$$

Conversely, if we are given M_I, we can get $M_{\mathtt{Inc}}$ by

$$M_{\mathtt{Inc}} = \{(D|_{\iota_1}, D|_{\iota_2}) \mid D \in M_I\}$$

where ι_1 and ι_2 are the co-cone morphism of the colimit of $\Gamma_{\mathtt{Inc}} ; \mathsf{Sig}^\mathcal{A}$.

Fact 5.24 *For every* R *in* $\textsc{Adt}_{\mathcal{R}_\mathcal{I}}$ *there exists an isomorphism between the model category of* R *and the model category of* $\mathsf{G}^\mathcal{A}(R)$*.*

Proof. This is obvious since $\mathsf{G}^\mathcal{A}((\Theta, M))$ is defined as $(\coprod \Gamma ; \mathsf{Sig}^\mathcal{A}, \overline{M})$ where $\overline{M}$ is the category with objects all $\overline{m}$ such that $m \in M$. □

Note however, though the model categories of R and $\mathsf{G}^{\mathcal{A}}(R)$ are isomorphic, there is no isomorphism between $\textsc{Adt}_{\mathcal{R}_{\mathcal{I}}}$ and $\textsc{Adt}_{\mathcal{I}}$ because it is impossible to recover the type of R from $\mathsf{G}^{\mathcal{A}}(R)$.

Still, it is possible to define for each abstract datatype (Σ, M) a relation $U((\Sigma, M))$ in $\textsc{Adt}_{\mathcal{R}_{\mathcal{I}}}$ which is universal in the sense that if σ is an $\textsc{Adt}$-morphism from $\mathsf{G}^{\mathcal{A}}((\Theta, M'))$ to (Σ, M) for some object (Θ, M') in $\textsc{Adt}_{\mathcal{R}_{\mathcal{I}}}$, then there exists a unique morphism $\tilde{\sigma}$ from (Θ, M') to $U((\Sigma, M))$ with $\mathsf{G}^{\mathcal{A}}(\tilde{\sigma}) = \sigma$.

Let (Σ, M) be an abstract datatype in $\textsc{Adt}_{\mathcal{I}}$, then

$$U((\Sigma, M)) = ((\mathbf{1}, \Gamma_\Sigma), M'')$$

where $\Gamma_\Sigma : \mathbf{1} \to \textsc{Adt}_{\mathcal{I}}$ maps 1 to $(\Sigma, \{\})$ and $\overline{M''}$ is the same as M.

Further, given an $\textsc{Adt}$-morphism σ from (Σ, M) to (Σ', M'), then σ extends to an $\textsc{Adt}_{\mathcal{R}_{\mathcal{I}}}$-morphism $f = (f^F, f^\mu)$ from $U((\Sigma, M))$ to $U((\Sigma', M'))$ by

$$f^F(1) = 1 \text{ and } f^\mu_1 = \sigma.$$

THEOREM 5.25 *U as defined above is a functor from* $\textsc{Adt}$ *to* $\textsc{Adt}_{\mathcal{R}_{\mathcal{I}}}$ *right adjoint for* $\mathsf{G}^{\mathcal{A}}$.

PROOF. The co-unit $\varepsilon : \mathsf{Id}_{\textsc{Adt}_{\mathcal{I}}} \Rightarrow U; \mathsf{G}^{\mathcal{A}}$ of the adjunction is given by $\varepsilon_{(\Sigma,M)} = \mathtt{id}_{(\Sigma,M)}$ for abstract datatypes (Σ, M). For a morphism σ from $\mathsf{G}^{\mathcal{A}}((\Theta, M'))$ to (Σ, M) the morphism $\tilde{\sigma}$ from (Θ, M') to $U((\Sigma, M))$ is given by $(\tilde{\sigma}^F, \tilde{\sigma}^\mu)$ where $\tilde{\sigma}^F$ is the unique morphism from D to $\mathbf{1}$ since $\mathbf{1}$ is the terminal object in $\textsc{Cat}$ and $\tilde{\sigma}^\mu_d = \iota_d^{\Gamma;\mathsf{Sig}^{\mathcal{A}}}; \sigma$ for each object d in D. The morphism $\iota_d^{\Gamma;\mathsf{Sig}^{\mathcal{A}}}$ is the co-cone morphism from $\mathsf{Sig}^{\mathcal{A}}(\Gamma(d))$ to $\coprod \Gamma; \mathsf{Sig}^{\mathcal{A}}$, the signature of $\mathsf{G}^{\mathcal{A}}((\Theta, M))$, where $\Theta = (\mathrm{D}, \Gamma)$.

Let $U((\Sigma, M)) = ((\mathbf{1}, \Gamma_\Sigma), M'')$. Note that by the definition of U we have $\overline{M''} = M$.

For $\tilde{\sigma}$ to be an $\textsc{Adt}_{\mathcal{R}_{\mathcal{I}}}$-morphism it remains to show that $M''|_{\tilde{\sigma}} \subseteq M'$. Note that, because $\tilde{\sigma}$ is unique with the property $\mathsf{G}^{\mathcal{A}}(\tilde{\sigma}); \varepsilon_{(\Sigma,M)} = \sigma$ and $\varepsilon_{(\Sigma,M)}$ is the identity, we have $\mathsf{G}^{\mathcal{A}}(\tilde{\sigma}) = \sigma$. Using the fact that the isomorphism between $\mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}$ and $\mathsf{clim}_{\textsc{Adt}_{\mathcal{I}}}; \mathsf{Mod}^{\mathcal{A}}$ is natural and noting that $\mathsf{G}^{\mathcal{A}}(\tilde{\sigma}) = \mathsf{clim}_{\textsc{Adt}_{\mathcal{I}}}(\tilde{\sigma}) = \sigma$, we get$\overline{M''|_{\tilde{\sigma}}} = \overline{M''}|_\sigma = M|_\sigma$. Since σ is an $\textsc{Adt}$-morphism, we have $M|_\sigma \subseteq \overline{M'}$ and thus $\overline{M''|_{\tilde{\sigma}}} \subseteq \overline{M'}$, which implies $M''|_{\tilde{\sigma}} \subseteq M'$. □

The existence of a right adjoint for $\mathsf{G}^{\mathcal{A}}$ implies:

COROLLARY 5.26 *The functor* $\mathsf{G}^{\mathcal{A}}$ *from* $\textsc{Adt}_{\mathcal{R}_{\mathcal{I}}}$ *to* $\textsc{Adt}_{\mathcal{I}}$ *preserves colimits.*

The following theorem is the core theorem for proving properties of relations (cf. Chapter 7). It allows us to prove $(\Theta_R, M_R) \models^{\mathcal{R}_{\mathcal{I}}} \varphi$ by proving $\mathsf{G}^{\mathcal{A}}((\Theta_R, M_R)) \models^{\mathcal{I}} \varphi$ and is a consequence of the isomorphism of the model categories of R and $\mathsf{G}^{\mathcal{A}}(R)$.

THEOREM 5.27 *Let $R = (\Theta, M)$ be a relation in* $\textsc{Adt}_{\mathcal{R}_{\mathcal{I}}}$ *and φ a formula in* $\mathsf{Sen}_{\mathcal{R}_{\mathcal{I}}}(\Theta)$, *then*

$$R \models^{\mathcal{R}_{\mathcal{I}}} \varphi \text{ if and only if } \mathsf{G}^{\mathcal{A}}(R) \models^{\mathcal{I}} \varphi.$$

Note though that this does not imply that if we are given a sound and complete inference system $\vdash^{\mathcal{I}}$ for proving $\Phi_1 \models^{\mathcal{I}} \Phi_2$, then we get a sound and complete inference system for proving $\Phi_1 \models^{\mathcal{R}_{\mathcal{I}}} \Phi_2$. We can only conclude that if $\Phi_1 \vdash^{\mathcal{I}} \Phi_2$, then $\Phi_1 \models^{\mathcal{R}_{\mathcal{I}}} \Phi_2$. The reason is that the category of all Θ_R-structures that satisfy Φ_1 with respect to $\models^{\mathcal{R}_{\mathcal{I}}}$ is only isomorphic to the subcategory of all Σ-structures that satisfy Φ_1 with respect to $\models^{\mathcal{I}}$ where Σ is the signature of $\mathsf{G}^{\mathcal{A}}(R)$ (cf. Section 7).

5.7 Operations on relations

One advantage of representing relations as abstract datatypes of the institution $\mathcal{R}_{\mathcal{I}}$ is to use the language $\mathtt{SL}_{\mathcal{I}'}$ (cf. Section 4.2) where $\mathcal{I}'$ is $\mathtt{RSL}_{\mathcal{I}}$ with its operations impose, translate, derive, and union to define relations.

A relation $R = \Theta$ is the universal relation. For example, if $\Theta = (\mathrm{K}_n, \Gamma)$, then

$$\begin{aligned}(m_1, \dots, m_n) \in R \quad &\text{iff} \quad (m_1, \dots, m_n) \in \mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}(\Theta)\\ &\text{iff} \quad (m_1, \dots, m_n) \in \mathsf{Str}_{\mathcal{I}}(\Gamma(1)) \times \dots \times \mathsf{Str}_{\mathcal{I}}(\Gamma(n)).\end{aligned}$$

Impose Let $R = (\Theta_R, M_R)$ be a relation in $\mathrm{ADT}_{\mathcal{R}_{\mathcal{I}}}$. The operation $I_\Phi R$ allows to restrict the tuples in M_R to those satisfying Φ:

$$(m_1, \dots, m_n) \in I_\Phi R$$
$$\text{if and only if}$$
$$(m_1, \dots, m_n) \in R \text{ and } (m_1, \dots, m_n) \models^{\mathcal{R}_{\mathcal{I}}} \Phi$$

For example, the increment operation of the counter can be defined using the impose operation as

$$\mathtt{Inc} = I_{\{c'=\mathtt{succ}(c)\}}\Theta_{\mathtt{Inc}}$$

and the decrement operation as

$$\mathtt{Dec} = I_{\{\mathtt{succ}(c')=c,\, c\neq\mathtt{zero}\}}\Theta_{\mathtt{Inc}}.$$

Union The union operation can be used to combine several relations of the same type such that

$$(m_1, \dots, m_n) \in R_1 + R_2$$
$$\text{if and only if}$$
$$(m_1, \dots, m_n) \in R_1 \text{ and } (m_1, \dots, m_n) \in R_2$$

for relations $R_1 = (\Theta, M_{R_1})$ and $R_2 = (\Theta, M_{R_2})$.

Translate The translate operation allows to change the type of a relation, for example, so that the union of relations is defined. It can be used to add new state components and nodes/edges to the type of a relation.

Given a relation R of type Θ_R and a $\textsc{Sign}_{\mathcal{R}_\mathcal{I}}$-morphism from Θ_R to some type Θ_S, then $T_f R$ yields a relation of type Θ_S as follows:

$$(m_1, \dots, m_n) \in T_f R \quad \text{if and only if} \quad (m_1, \dots, m_n)|_f \in R.$$

For example, assume that we want to add a new component d of sort N to the state of the counter, that is, the new state is given by the following specification:

```
New : trait
   includes Counter
   introduces d : N
```

Further, we would like to extend the increment relation `Inc` to an increment relation `Inc`′ on the new state, that is, `Inc`′ is of the type

$$\Theta_{\texttt{Inc}'} = \begin{tikzcd} \llbracket \texttt{New} \rrbracket & & \llbracket \texttt{New} \rrbracket \\ & \llbracket \texttt{Env} \rrbracket \arrow[ul] \arrow[ur] & \end{tikzcd}$$

Let ι_C denote the inclusion of $\llbracket \texttt{Counter} \rrbracket$ into $\llbracket \texttt{New} \rrbracket$ and let $f = (\mathsf{Id}_\mathrm{V}, f^\mu)$ be the $\textsc{Sign}_{\mathcal{R}_{\mathcal{LSL}}}$-morphism from $\Theta_{\texttt{Inc}}$ to $\Theta_{\texttt{Inc}'}$ such that $f_0^\mu : \llbracket \texttt{Env} \rrbracket \to \llbracket \texttt{Env} \rrbracket$ is the identity, $f_1^\mu : \llbracket \texttt{Counter} \rrbracket \to \llbracket \texttt{New} \rrbracket = \iota_C$, and $f_2^\mu : \llbracket \texttt{Counter} \rrbracket \to \llbracket \texttt{New} \rrbracket = \iota_C$. Then `Inc`′ is T_f`Inc`, and we have:

$$\mathsf{Mod}^\mathcal{A}(\texttt{Inc}') = \{(A, B) \in \mathsf{Mod}^\mathcal{A}(\llbracket \texttt{New} \rrbracket) \times \mathsf{Mod}^\mathcal{A}(\llbracket \texttt{New} \rrbracket) \mid (A|_{\iota_C}, B|_{\iota_C}) \in \llbracket \texttt{Inc} \rrbracket\}.$$

Note that `Inc`′ does not fix the relationship between $A(d)$ and $B(d)$. Thus any values for $A(d)$ and $B(d)$ are possible in case of A `Inc`′ B. To establish a relationship between these two values, we can use, for example, the impose operation as follows:

$$I_{\{d' = \texttt{succ}(\texttt{succ}(d))\}} \texttt{Inc}'$$

This ensures that $B(d) = A(\texttt{succ})(A(\texttt{succ})(d))$ for all $(A, B) \in \texttt{Inc2}$.

Derive By hiding state components the derive operation can be used to adapt the state space of relations so that, for example, the union of relations is defined. Hiding nodes by the derive operation is used in the definition of sequential composition (cf. Section 6.3), in the computation of the precondition of a state transformation (cf. Section 7.2), and can be used to define a relation with the help of auxiliary symbols, like the tracking map between sorts in Section 6.1.

If R is a relation of type Θ_R and f a $\textsc{Sign}_{\mathcal{R}_\mathcal{I}}$-morphism from a type Θ_S to Θ_R, then $D_f R$ yields a relation of type Θ_S as follows:

$$(m_1, \dots, m_n) \in D_f R$$
$$\text{if and only if}$$
$$\exists (m'_1, \dots, m'_k) \in R \text{ with } (m'_1, \dots, m'_k)|_f = (m_1, \dots, m_n)$$

One use of the derive operation is to define the identity relation. Suppose, we are interested in specifying the identity relation for some abstract datatype `St`, that is, we want to define a relation $\mathsf{Id}_{\mathtt{St}}$ of type

$$\Theta = \begin{tikzcd} \llbracket \mathtt{St} \rrbracket & & \llbracket \mathtt{St} \rrbracket \\ & \llbracket \mathtt{Env} \rrbracket \arrow[ul] \arrow[ur] & \end{tikzcd}$$

such that $(m_1, m_2) \in \mathsf{Id}_{\mathtt{St}}$ if and only if $m_1 = m_2$ for structures m_1 and m_2 from $\mathsf{Mod}^{\mathcal{A}}(\mathtt{St})$.

Consider the type

$$\Theta' = \begin{tikzcd} \llbracket \mathtt{St} \rrbracket \\ \llbracket \mathtt{Env} \rrbracket \arrow[u] \end{tikzcd}$$

and the morphism $\iota = (\iota^F, \iota^\mu)$ from Θ to Θ' mapping each `St` in Θ to `St` in Θ' and which is the identity on `Env`. That is, ι^F maps 0 of V to 0 of **2** and 1 and 2 of V to 1 of **2**. ι^μ_0 is the identity on $\llbracket \mathtt{Env} \rrbracket$, and $\iota^\mu_1 = \iota^\mu_2$ is the identity on $\llbracket \mathtt{St} \rrbracket$. Then $\mathsf{Id}_{\mathtt{St}}$ is given by $\mathsf{Id}_{\mathtt{St}} = D_\iota \Theta'$.

To show that $\mathsf{Id}_{\mathtt{St}}$ really is the identity, let (A, B) in $\mathsf{Id}_{\mathtt{St}}$, that is, there exists some C in $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta')$ with $C|_\iota = (A, B)$. By the definition of $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\iota)$ we have

$$\begin{aligned} C|_\iota &= (C|_{\iota^\mu_{\iota^F(1)}}, C|_{\iota^\mu_{\iota^F(2)}}) & \\ &= (C|_{\iota^\mu_1}, C|_{\iota^\mu_1}) & \mid \iota^F(1) = 1 = \iota^F(2) \\ &= (C, C) & \mid \iota^\mu_i = \mathtt{id} \end{aligned}$$

Thus we have $(A, B) = (C, C)$ and therefore $A = B = C$.

On the other hand, it is obvious that (A, A) is in $\mathsf{Id}_{\mathtt{St}}$ for each $A \in \mathsf{Mod}^{\mathcal{A}}(\mathtt{St})$.

Colimit The colimit construction can be used to combine different relations on the same state sharing some common part or to combine relations on different states with the possibility of sharing state components. If $F : J \to \mathtt{SL}_{\mathcal{R}_\mathcal{I}}$ is a diagram of relations such that the colimit of $F; \mathsf{Sig}$ exists, then

$$(m_1, \ldots, m_n) \in \mathsf{colim}\ F \text{ iff } (m_1, \ldots, m_n)|_{\iota_i} \in F(i) \text{ for all } i \in J$$

where ι_i is the co-cone morphism from $F(i)$ to the colimit of $F; \mathsf{Sig}$ for all $i \in J$.

As an example we define the relation `Inc2` on the state space

$$C' = \llbracket \mathtt{Counter} \rrbracket +_{\llbracket \mathtt{Nat} \rrbracket} \llbracket \mathtt{Counter} \rrbracket$$

incrementing both copies of the counter simultaneously as $\mathtt{Inc} +_{\mathsf{Id}_{\mathtt{Nat}}} \mathtt{Inc}$ where $\mathsf{Id}_{\mathtt{Nat}}$ is the identity relation on `Nat`, which has type

$$\begin{tikzcd}
\Theta_{\mathtt{Nat}} = & \llbracket \mathtt{Nat} \rrbracket & & \llbracket \mathtt{Nat} \rrbracket \\
& & \llbracket \mathtt{Nat} \rrbracket \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Let $\iota = (\mathsf{Id}_{\mathrm{V}}, \iota^{\mu})$ be the $\textsc{Sign}_{\mathcal{R}_{\mathcal{I}}}$-morphism from $\Theta_{\mathtt{Nat}}$ to $\Theta_{\mathtt{Inc}}$ where ι_0^{μ} is the identity on `Nat` and $\iota_1^{\mu} = \iota_2^{\mu}$ is the inclusion of `Nat` into `Counter`. The pushout $\Theta_{\mathtt{Inc}} +_{(\iota,\iota)} \Theta_{\mathtt{Inc}}$ is the type

$$\begin{tikzcd}
\Theta_{\mathtt{Inc2}} = & \llbracket \mathtt{Counter} \rrbracket +_{\llbracket \mathtt{Nat} \rrbracket} \llbracket \mathtt{Counter} \rrbracket & & \llbracket \mathtt{Counter} \rrbracket +_{\llbracket \mathtt{Nat} \rrbracket} \llbracket \mathtt{Counter} \rrbracket \\
& & \llbracket \mathtt{Nat} \rrbracket \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Let

```
PO : trait
   include Counter[c1 for c]
           Counter[c2 for c]
```

be the pushout $\llbracket \mathtt{Counter} \rrbracket +_{\llbracket \mathtt{Nat} \rrbracket} \llbracket \mathtt{Counter} \rrbracket$. Then the co-cone morphism ι_1 from $\Theta_{\mathtt{Inc}}$ to $\Theta_{\mathtt{Inc2}}$ is $(\mathsf{Id}_{\mathrm{V}}, \iota_1{}^{\mu})$ where $(\iota_1{}^{\mu})_0$ is the identity morphism on $\llbracket \mathtt{Nat} \rrbracket$ and $(\iota_1{}^{\mu})_1 = (\iota_1{}^{\mu})_2$ maps $c : N$ to $c_1 : N$ and is the identity otherwise. The second co-cone morphism ι_2 from $\Theta_{\mathtt{Inc}}$ to $\Theta_{\mathtt{Inc2}}$ is $(\mathsf{Id}_{\mathrm{V}}, \iota_2{}^{\mu})$ where $(\iota_2{}^{\mu})_0$ is the identity morphism on $\llbracket \mathtt{Nat} \rrbracket$ and $(\iota_2{}^{\mu})_1 = (\iota_2{}^{\mu})_2$ maps $c : N$ to $c_2 : N$.

Using the definition of the `colim` operator from Section 4.2 and the equalities of Fact 4.9 we get

$$\begin{aligned}
\mathtt{Inc2} &= \mathtt{Inc} +_{\mathsf{Id}_{\mathtt{Nat}}} \mathtt{Inc} \\
&= T_{\iota_1'} \mathtt{Inc} + T_{\iota_2'} \mathtt{Inc} \\
&= T_{\iota_1'} I_{\{c'=\mathtt{succ}(c)\}} \Theta_{\mathtt{Inc}} + T_{\iota_2'} I_{\{c'=\mathtt{succ}(c)\}} \Theta_{\mathtt{Inc}} \\
&= I_{\iota_1'(\{c'=\mathtt{succ}(c)\})} \Theta_{\mathtt{Inc2}} + I_{\iota_2'(\{c'=\mathtt{succ}(c)\})} \Theta_{\mathtt{Inc2}} \\
&= I_{\{c_1'=\mathtt{succ}(c_1)\}} I_{\{c_2'=\mathtt{succ}(c_2)\}} \Theta_{\mathtt{Inc2}} \\
&= I_{\{c_1'=\mathtt{succ}(c_1)\} \cup \{c_2'=\mathtt{succ}(c_2)\})} \Theta_{\mathtt{Inc2}} \\
&= I_{\{c_1'=\mathtt{succ}(c_1),\, c_2'=\mathtt{succ}(c_2)\}} \Theta_{\mathtt{Inc2}}
\end{aligned}$$

5.8 Finite Diagrams

In the previous sections we have assumed that $\textsc{Sign}_{\mathcal{I}}$ is cocomplete. However, in many interesting institutions $\textsc{Sign}_{\mathcal{I}}$ is only finitely cocomplete. For example, if we restrict the signatures of the institutions $\mathcal{EQ}$ and $\mathcal{EQC}$ to finite many-sorted signatures, that is, instead of $\textsc{Sig}$ we use the category of finite many-sorted signatures $\textsc{FSig}$. $\textsc{FSig}$ is a full subcategory of $\textsc{Sig}$, but only finitely cocomplete (cf. Section 3.1).

To use an institution $\mathcal{I}$ with a finitely cocomplete category of signatures for the construction of $\mathcal{R}_{\mathcal{I}}$, we have to restrict the objects in $\textsc{Sign}_{\mathcal{R}_{\mathcal{I}}}$ to pairs (D, Γ) where D is a finite category because the definition of $\mathsf{Sen}_{\mathcal{R}_{\mathcal{I}}}$ depends on taking colimits of shape D in $\textsc{Sign}_{\mathcal{I}}$. Since all diagrams are finite, $\mathsf{Str}_{\mathcal{I}}$ needs to preserve only finite colimits instead of arbitrary colimits. The definitions of $\mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}$, $\mathsf{Sen}_{\mathcal{R}_{\mathcal{I}}}$, and $\models^{\mathcal{R}_{\mathcal{I}}}$ are, in principle, unaffected.

Formally, instead of defining $\textsc{Sign}_{\mathcal{R}_{\mathcal{I}}}$ as $\mathsf{Flat}(\mathsf{Fun}(\textsc{Adt}_{\mathcal{I}}))$, we define

$$\textsc{Sign}_{\mathcal{R}_{\mathcal{I}}} = \mathsf{Flat}(\mathsf{FFun}(\textsc{Adt}_{\mathcal{I}}))$$

where $\mathsf{FFun}(T) : \textsc{FCat}^{op} \to \textsc{Cat}$ is the indexed category $\mathsf{FFun}(T)_{\mathrm{D}} = T^{\mathrm{D}}$ for finite categories D and $\mathsf{FFun}(T)(\Gamma') = F; \Gamma$ for functors $\mathsf{F} : \mathrm{D} \to \mathrm{D}'$ and $\Gamma' : \mathrm{D}' \to T$. Similar to $\mathsf{Fun}(G)$ we define $\mathsf{FFun}(G)_{\mathrm{D}}(\Gamma) = \Gamma; G$ and $\mathsf{FFun}(G)_{\mathrm{D}}(\mu) = \mu; G$ for finite categories D, functors $\Gamma : \mathrm{D} \to T$, and natural transformations $\mu : \Gamma_1 \Rightarrow \Gamma_2$.

Note that $\mathsf{Flat}(\mathsf{FFun}(T))$ is a full subcategory of $\mathsf{Flat}(\mathsf{Fun}(T))$, thus the new structure functor $\mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}$ is the old structure functor $\mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}$ with its domain restricted to $\mathsf{Flat}(\mathsf{FFun}(T))$.

Similarly, we get a functor $\mathsf{clim}_T : \mathsf{Flat}(\mathsf{FFun}(T)) \to T$ by restricting the domain of the functor $\mathsf{clim}_T : \mathsf{Flat}(\mathsf{Fun}(T)) \to T$. Note that for $\mathsf{clim}_T : \mathsf{Flat}(\mathsf{FFun}(T)) \to T$ to be well-defined we only have to require that T is finitely cocomplete. Then $\mathsf{Sen}_{\mathcal{R}_{\mathcal{I}}}$ is again defined as $\mathsf{Flat}(\mathsf{FFun}(\mathsf{Sig}^{\mathcal{A}})); \mathsf{clim}_{\textsc{Sign}_{\mathcal{I}}}; \mathsf{Sen}_{\mathcal{I}}$.

The results that $\mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}$ is natural isomorphic to $\mathsf{clim}_{\textsc{Adt}_{\mathcal{I}}}; \mathsf{Mod}^{\mathcal{A}}$ and that

$$\mathsf{Flat}(\mathsf{FFun}(\mathsf{Sig}^{\mathcal{A}})); \mathsf{clim}_{\textsc{Sign}_{\mathcal{I}}} = \mathsf{clim}_{\textsc{Adt}_{\mathcal{I}}}; \mathsf{Sig}^{\mathcal{A}}$$

carry over from Theorems 5.8 and 5.9. This allows again to define $m \models^{\mathcal{R}_{\mathcal{I}}} \varphi$ by $\overline{m} \models^{\mathcal{I}} \varphi$ for all types $\Theta \in \textsc{Sign}_{\mathcal{R}_{\mathcal{I}}}$, $m \in \mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}(\Theta)$, and $\varphi \in \mathsf{Sen}_{\mathcal{R}_{\mathcal{I}}}(\Theta)$.

Theorem 5.17 ensures that $\textsc{Sign}_{\mathcal{R}_{\mathcal{I}}}$ is also finitely cocomplete because $\textsc{Sign}_{\mathcal{I}}$ and thus $\textsc{Adt}_{\mathcal{I}}$ are finitely cocomplete.

Similarly, $\mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}$ preserves colimits because $\mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}$ is isomorphic to the composition $\mathsf{clim}_{\textsc{Adt}_{\mathcal{I}}}$ followed by $\mathsf{Mod}^{\mathcal{A}}$, $\mathsf{Mod}^{\mathcal{A}}$ preserves finite colimits if $\mathsf{Str}_{\mathcal{I}}$ preserves them, and $\mathsf{clim}_{\textsc{Adt}_{\mathcal{I}}}$ preserves colimits and thus also finite colimits.

5.9 The Language $\mathtt{RSL}_{\mathcal{I}}$

In the following we present the language $\mathtt{RSL}_{\mathcal{I}}$ for the definition of relations and introduce some useful abbreviations. At first glance $\mathtt{RSL}_{\mathcal{I}}$ expressions are the same as $\mathtt{SL}_{\mathcal{I}'}$ expressions introduced in Section 4.2 where $\mathcal{I}'$ is the institution $\mathcal{R}_{\mathcal{I}}$. However, the difference is that instead of functors $\Gamma : \mathrm{D} \to \textsc{Adt}_{\mathcal{I}}$ functors $\Gamma : \mathrm{D} \to \mathtt{SL}_{\mathcal{I}}$ are used, where $\mathtt{SL}_{\mathcal{I}}$ is the specification language for the base institution $\mathcal{I}$ of $\mathcal{R}_{\mathcal{I}}$. The idea is that not only the relations are defined by expressions but also the type of the relations. In addition this allows to define the translation of expressions in $\mathtt{RSL}_{\mathcal{I}}$ to expressions in $\mathtt{SL}_{\mathcal{I}}$ given in Section 7.1, which would be impossible otherwise.

To this end we define the category $\textsc{Type}_{\mathcal{I}}$. $\textsc{Type}_{\mathcal{I}}$ is similar to $\textsc{Sign}_{\mathcal{R}_{\mathcal{I}}}$, however, using pairs $(\mathrm{D}, \Gamma : \mathrm{D} \to \mathtt{SL}_{\mathcal{I}})$ instead of pairs $(\mathrm{D}, \Gamma : \mathrm{D} \to \textsc{Adt}_{\mathcal{I}})$. The morphisms f from $\Theta = (\mathrm{D}, \Gamma)$ to $\Theta' = (\mathrm{D}', \Gamma')$ in $\textsc{Type}_{\mathcal{I}}$ are pairs (f^F, f^μ) where f^F is a functor from D, and f^μ is a natural transformation from Γ to $f^F;\Gamma'$.

Note that the category $\textsc{Type}_{\mathcal{I}}$ is the same as the flattened indexed category $\mathsf{FFun}(\mathtt{SL}_{\mathcal{I}})$. Since $\mathtt{SL}_{\mathcal{I}}$ is finitely cocomplete, this implies that $\textsc{Type}_{\mathcal{I}} = \mathsf{Flat}(\mathsf{FFun}(\mathtt{SL}_{\mathcal{I}}))$ is also finitely cocomplete (cf. Section 5.8).

The semantics functor $[\![\_]\!] : \mathtt{SL}_{\mathcal{I}} \to \textsc{Adt}_{\mathcal{I}}$ (cf Section 4.2) is extended to a functor $[\![\_]\!]$ from $\textsc{Type}_{\mathcal{I}}$ to $\textsc{Sign}_{\mathcal{R}_{\mathcal{I}}}$ that maps an object $(\mathrm{D}, \Gamma : \mathrm{D} \to \mathtt{SL}_{\mathcal{I}})$ in $\textsc{Type}_{\mathcal{I}}$ to an object $(\mathrm{D}, \Gamma; [\![\_]\!] : \mathrm{D} \to \textsc{Adt}_{\mathcal{I}})$ in $\textsc{Sign}_{\mathcal{R}_{\mathcal{I}}}$ and similar a $\textsc{Type}_{\mathcal{I}}$-morphism f^μ to a $\textsc{Sign}_{\mathcal{R}_{\mathcal{I}}}$-morphism $(f^F, f^\mu; [\![\_]\!])$.

Definition 5.28 (Abstract Syntax of $\mathtt{RSL}_{\mathcal{I}}$)
Given the institution $\mathcal{R}_{\mathcal{I}}$ based on the institution $\mathcal{I}$, types Θ and Θ' in $\textsc{Type}_{\mathcal{I}}$*, and a* $\textsc{Type}_{\mathcal{I}}$*-morphism $f : \Theta \to \Theta'$, then the abstract syntax of the language* $\mathtt{RSL}_{\mathcal{I}}$ *is given by*

$$R ::= \Theta \mid I_\Phi R \mid T_f R \mid D_f R \mid R + R$$

where Φ is a set of formulas in the institution $\mathcal{R}_{\mathcal{I}}$.

The type $\mathsf{type}(R)$ *of an expression R in* $\mathtt{RSL}_{\mathcal{I}}$ *is an object of* $\textsc{Type}_{\mathcal{I}}$ *and is defined inductively on the structure of* $\mathtt{RSL}_{\mathcal{I}}$ *as follows:*

- $\mathsf{type}(\Theta) = \Theta$
- $\mathsf{type}(I_\Phi R) = \mathsf{type}(R)$
- $\mathsf{type}(T_f R) = \mathsf{cod}(f)$
- $\mathsf{type}(D_f R') = \mathsf{dom}(f)$
- $\mathsf{type}(R_1 + R_2) = \mathsf{type}(R_1) = \mathsf{type}(R_2)$.

An expression R from $\mathtt{RSL}_{\mathcal{I}}$ *is* well-formed *if*

- $R = I_\Phi R'$, $\Phi \subseteq \mathsf{Sen}_{\mathcal{R}_{\mathcal{I}}}([\![\mathsf{type}(R')]\!])$*, and R' is well-formed.*
- $R = T_f R'$*, f is a morphism from* $\mathsf{type}(R')$ *to* $\mathsf{type}(R)$ *in* $\textsc{Type}_{\mathcal{I}}$*, and R' is well-formed.*
- $R = D_f R'$*, f is a morphism from* $\mathsf{type}(R)$ *to* $\mathsf{type}(R')$ *in* $\textsc{Type}_{\mathcal{I}}$*, and R' is well-formed.*
- $R = R_1 + R_2$, $\mathsf{type}(R_1) = \mathsf{type}(R_2)$*, and R_1 and R_2 are well-formed.*

The semantics of a well-formed expression in $\mathtt{RSL}_{\mathcal{I}}$ is an object of $\textsc{Adt}_{\mathcal{R}_{\mathcal{I}}}$ and is defined in two steps: first an expression in $\mathtt{RSL}_{\mathcal{I}}$ is converted to an expression in $\mathtt{SL}_{\mathcal{R}_{\mathcal{I}}}$, and then the semantics of this expression yields a relation in $\textsc{Adt}_{\mathcal{R}_{\mathcal{I}}}$.

Let Θ be a type in $\textsc{Type}_\mathcal{I}$, R a well-formed expression in $\mathtt{RSL}_\mathcal{I}$, f a $\textsc{Type}_\mathcal{I}$-morphism from $\mathsf{type}(R)$ to Θ', and g a $\textsc{Type}_\mathcal{I}$-morphisms from Θ' to $\mathsf{type}(R)$ for some Θ' in $\textsc{Type}_\mathcal{I}$, then

- $[\![\Theta]\!]_{\mathtt{SL}_{\mathcal{R}_\mathcal{I}}} = [\![\Theta]\!]$
- $[\![I_\Phi R]\!]_{\mathtt{SL}_{\mathcal{R}_\mathcal{I}}} = I_\Phi \, [\![R]\!]_{\mathtt{SL}_{\mathcal{R}_\mathcal{I}}}$
- $[\![T_f R]\!]_{\mathtt{SL}_{\mathcal{R}_\mathcal{I}}} = T_{[\![f]\!]} \, [\![R]\!]_{\mathtt{SL}_{\mathcal{R}_\mathcal{I}}}$
- $[\![D_g R]\!]_{\mathtt{SL}_{\mathcal{R}_\mathcal{I}}} = D_{[\![g]\!]} \, [\![R]\!]_{\mathtt{SL}_{\mathcal{R}_\mathcal{I}}}$
- $[\![R_1 + R_2]\!]_{\mathtt{SL}_{\mathcal{R}_\mathcal{I}}} = [\![R_1]\!]_{\mathtt{SL}_{\mathcal{R}_\mathcal{I}}} + [\![R_2]\!]_{\mathtt{SL}_{\mathcal{R}_\mathcal{I}}}$.

Finally, the semantics of an expression R in $\mathtt{RSL}_\mathcal{I}$ is given by

$$[\![R]\!]_\mathcal{R} = \left[\!\!\left[[\![R]\!]_{\mathtt{SL}_{\mathcal{R}_\mathcal{I}}} \right]\!\!\right].$$

Similar as with $\mathtt{SL}_\mathcal{I}$ (c.f. Section 4.2), we make $\mathtt{RSL}_\mathcal{I}$ a category by defining the morphsims f from R_1 to R_2 to be the $\textsc{Type}_\mathcal{I}$ morphisms (f^F, f^μ) having the property that $[\![f]\!] = (f^F, f^\mu; [\![\_]\!])$ is an $\textsc{Adt}_{\mathcal{R}_\mathcal{I}}$-morphism from $[\![R_1]\!]_\mathcal{R}$ to $[\![R_2]\!]_\mathcal{R}$.

$\textsc{Fact}$ 5.29 *For any expression R in $\mathtt{RSL}_\mathcal{I}$ we have*

$$\mathsf{Sig}^\mathcal{A}([\![R]\!]_\mathcal{R}) = [\![\mathsf{type}(R)]\!] \text{ and } \mathsf{Mod}^\mathcal{A}([\![R]\!]_\mathcal{R}) \subseteq \mathsf{Str}_{\mathcal{R}_\mathcal{I}}([\![\mathsf{type}(R)]\!]).$$

We say that a type $\Theta = (\mathrm{D}, \Gamma)$ is included in a type $\Theta' = (\mathrm{D}', \Theta')$ if Γ is a sub-diagram of Γ'. However, we also talk about inclusion if there are obvious inclusion morphisms between corresponding nodes. For example, given the following diagrams

$$\begin{tikzcd}
 & \Gamma(1) & & \Gamma(2) \\
\Gamma = & & \Gamma(0) \arrow[ul, "\Gamma(f_1)"] \arrow[ur, "\Gamma(f_2)"'] &
\end{tikzcd}$$

and

$$\begin{tikzcd}
 & & \Gamma'(3) & \\
\Gamma' = & \Gamma'(1) \arrow[ur, "\Gamma(f_3)"] & & \Gamma'(2) \arrow[ul, "\Gamma'(f_4)"'] \\
 & & \Gamma'(0) \arrow[ul, "\Gamma'(f_1)"] \arrow[ur, "\Gamma'(f_2)"'] &
\end{tikzcd}$$

then Γ is included in Γ' if $\Gamma(0) = \Gamma'(0)$, $\Gamma(1) = \Gamma'(1)$, $\Gamma(2) = \Gamma'(2)$, $\Gamma(f_1) = \Gamma'(f_1)$, and $\Gamma(f_2) = \Gamma'(f_2)$, or if there are inclusion morphisms ι_0 from $\Gamma(0)$ to $\Gamma'(0)$, ι_1 from $\Gamma(1)$ to $\Gamma'(1)$, and ι_2 from $\Gamma(2)$ to $\Gamma(2)$.

Formally, we require that there exists a unique morphism $\iota = (\iota^F, \iota^\mu)$ from Θ to Θ' such that $\iota^F : \mathrm{D} \to \mathrm{D}'$ is faithful and $\iota^\mu_d : \Gamma(d) \to \Gamma'(\iota^F(d))$ is "canonic" for all $d \in \mathrm{D}$.

Abbreviations

Since $\textsc{Type}_{\mathcal{I}}$ is finitely cocomplete, so is $\mathtt{RSL}_{\mathcal{I}}$, and, similar as with $\mathtt{SL}_{\mathcal{I}}$, we extend $\mathtt{RSL}_{\mathcal{I}}$ by a colimit construct

$$R ::= \ldots \mid \mathsf{colim}\ F \mid R_1 +_{(\sigma_1,\sigma_2)} R_2 \mid R_1 +_{R_0} R_2$$

where F is a functor from a finite category J to $\mathtt{RSL}_{\mathcal{I}}$. Then $\mathsf{colim}\ F$ is an abbreviation for

$$T_{\iota_{i_1}^{F;\mathsf{type}}}(F(i_1)) + \cdots + T_{\iota_{i_n}^{F;\mathsf{type}}}(F(i_n))$$

where $\iota_{i_k}^{F;\mathsf{type}}$ are the co-cone morphisms of the colimit of $F;\mathsf{type}$. Similarly, we use $R_1 +_{(f_1,f_2)} R_2$ for the pushout of R_1 with R_2 with respect to f_1 and f_2, and $R_1 +_{R_0} R_2$ if f_1 and f_2 are the unique inclusion of R_0 in R_1 and R_2.

In Chapter 6 we mainly have to deal with relations whose types have the form

$$\begin{tikzcd}
\textsc{Sp}_1 & \cdots & \textsc{Sp}_n \\
& \mathtt{Env} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

where the morphisms from $\mathtt{Env}$ to $\textsc{Sp}_i$ are *canonic*, for example, in $\mathcal{LSL}$ this means that $\mathtt{Env}$ is included into $\textsc{Sp}_i$ and that the morphisms are the corresponding inclusions of $\mathtt{Env}$ into $\textsc{Sp}_i$. We shall write $<\textsc{Sp}_1 \times \ldots \times \textsc{Sp}_n>_{\mathtt{Env}}$ for types of this form.

Then $[e : E]$ in $<[e : E] \times \textsc{Sp}>_{\mathtt{Env}}$ is an abbreviation for

```
includes Env
introduces e : E
```

For example, the type of the `Add` operation from page 83 can be now written as

$$<[n_{\mathtt{in}} : \mathtt{Nat}] \times \mathtt{Counter} \times \mathtt{Counter}>_{\mathtt{Nat}}.$$

If R is a relation defined by $I_\Phi\Theta$ then we may also write

```
R : Θ
asserts Φ
```

And

```
R : Θ
includes R'
asserts Φ
```

is the same as $I_\Phi T_\iota R'$ where ι is the canonic morphism from the type of R' to Θ.

6 Abstract Machines

The motivation to study relations between classes of algebras is to model state-transitions in software systems. An example of such software systems are abstract machines. An abstract machines offers a set of operations to its clients. Depending on the operations that have been requested before, the same operation may yield different results. Thus abstract machines have memory. A typical abstract machine is the abstract machine of a stack. It provides the operations `Push` and `Pop`. The `Push` operation pushes its argument on the stack, and the `Pop` operation removes the topmost element of the stack and returns it. Depending on what `Push` and `Pop` operations have been performed before, different results are returned by the `Pop` operation. Thus an abstract machine consists of a set of states, a set of initial states, a subset of the set of states, and a set of services. A service is a relation between states with possible input and output values.

Abstract machines are often called abstract datatypes with states or simply abstract datatypes. However, to not confuse abstract datatypes with state with the notion of abstract datatypes introduced in Chapter 4, we stick to the term abstract machines.

In the framework of this thesis, the signature of an abstract machine is a 4-tuple $(\mathtt{Env}, I, O, \tau)$ where the environment `Env` is an $\mathtt{SL}_{\mathcal{I}}$-expression, I is the name of the set of initial states, O is a set of names for the operations, and τ a function assigning to an operation name *op* in O a type $\tau(op)$ from $\textsc{Type}_{\mathcal{I}}$. We require that $\tau(op)$ has the form

$$\begin{tikzcd}
I_1 & \cdots & I_n & & O_1 & \cdots & O_m \\
 & & & \mathtt{Env} \arrow[lllu, "f_1"] \arrow[lu, "f_n"'] \arrow[ru, "g_1"] \arrow[rrru, "g_m"'] & & &
\end{tikzcd}$$

We shall also write $\tau(op) = <I_1 \times \ldots \times I_n> \to <O_1 \times \ldots \times O_m>$ assuming that the morphisms $f_i : \mathtt{Env} \to I_i$ and $g_j : \mathtt{Env} \to O_j$ are "canonic", which, for example, in $\mathcal{LSL}$ means that f_i and g_j are inclusions.

Since abstract machines encapsulate the state, the signature does not contain an $\mathtt{SL}_{\mathcal{I}}$-expression for the state, and similar the type of an operation does not contain the state.

For the example, the signature of the stack is $(\mathtt{Env}, \mathtt{Empty}, \{\mathtt{Push}, \mathtt{Pop}\}, \tau)$ where $\tau(\mathtt{Push}) = <[e : E]> \to <>$, $\tau(\mathtt{Pop}) = <> \to <[e : E]>$, and `Env` is an abstract datatype containing at least the sort E.

Whenever convenient, we may also write

$$(\mathtt{Env}, \mathtt{Empty}, \{\mathtt{Push} : <[e : E]> \to <>, \mathtt{Pop} : <> \to <[e : E]>\})$$

for the the signature $(\mathtt{Env}, \mathtt{Empty}, \{\mathtt{Push}, \mathtt{Pop}\}, \tau)$.

An abstract machine $\mathcal{M} = (\mathtt{St}_{\mathcal{M}}, I_{\mathcal{M}}, \mathcal{M}_O)$ of signature $(\mathtt{Env}, I, O, \tau)$ consists of an abstract datatype $\mathtt{St}_{\mathcal{M}}$ from $\textsc{Adt}_{\mathcal{I}}$ such that $\mathtt{Env}$ is included in $\mathtt{St}_{\mathcal{M}}$, defining the state space; a relation $I_{\mathcal{M}}$ of type $<\mathtt{St}_{\mathcal{M}}>_{\mathtt{Env}}$ and a set of relations $\mathcal{M}_O = \{op_{\mathcal{M}} \mid op \in O\}$ from $\textsc{Adt}_{\mathcal{R}_{\mathcal{I}}}$ such that if $\tau(op) = <I_1 \times \ldots \times I_n> \rightarrow <O_1 \times \ldots \times O_m>$, then the type of $op_{\mathcal{M}}$ is

$$<I_1 \times \ldots I_n \times \mathtt{St}_{\mathcal{M}} \times \mathtt{St}_{\mathcal{M}} \times O_1 \times \ldots O_m>_{\mathtt{Env}}.$$

The initial states $I_{\mathcal{M}}$ are given as a unary relation of type $<\mathtt{St}_{\mathcal{M}}>_{\mathtt{Env}}$ instead of an abstract datatype from $\textsc{Adt}_{\mathcal{I}}$ because this ensures that the class of initial states are models of $\mathtt{St}_{\mathcal{M}}$, and this simplifies writing the condition of simulation in Section 7.2.

The abstract machine of a stack is given by

$$\mathcal{S} = (\mathtt{Stack}_{\mathcal{S}}, \mathtt{Empty}_{\mathcal{S}}, \{\mathtt{Push}_{\mathcal{S}}, \mathtt{Pop}_{\mathcal{S}}\})$$

where $\mathtt{Stack}_{\mathcal{S}}$ is the abstract datatype:

```
Stack_S : trait
includes Env
introduces s : Stack
```

and $\mathtt{Empty}_{\mathcal{S}}$ is the abstract datatype

```
Empty_S : <Stack_S>
asserts s = empty
```

We assume that the following specification is part of the environment $\mathtt{Env}$:

```
Stack : trait
introduces empty : Stack
           push : E, Stack → Stack
           pop : Stack → Stack
           top : Stack → E
asserts
  Stack freely generated by empty, push
  ∀ e:E, s:Stack  top(push(e,s)) = e
  ∀ e:E, s:Stack  pop(push(e,s)) = s
```

The $\mathtt{Push}_{\mathcal{S}}$ operation is defined as

```
Push_S : <[e:E] × Stack_S × Stack_S>
asserts s' = push(e,s)
```

and the type of the operation $\mathtt{Pop}_{\mathcal{S}}$ is $<\mathtt{Stack}_{\mathcal{S}} \times \mathtt{Stack}_{\mathcal{S}} \times [e : E]>_{\mathtt{Env}}$ and the operation is given by

```
Pop_S : <[e:E] × Stack_S × Stack_S>
asserts s' = pop(s)
        e = top(s)
```

6.1 State as Algebra Approach

In the state as algebra approach the state of a software system is modeled by an algebra. This corresponds to using one of the institutions $\mathcal{EQ}$, $\mathcal{EQC}$, or $\mathcal{LSL}$ as the base institution $\mathcal{I}$ for $\mathcal{R}_{\mathcal{I}}$. The state components in the state as algebra approach are constants (null-ary functions), functions, and sorts.

Constants

Let us first consider constants as state components. Suppose we are given the following state space:

```
State : trait
 includes Env
 introduces c : S
```

and a type $\Theta = <\texttt{State} \times \texttt{State}>_{\texttt{Env}}$. The formulas that can be used to define relations of type Θ are the formulas from $\mathcal{LSL}$ over the signature

$$\begin{aligned} \Sigma_{cl} &= \mathsf{Sig}(\texttt{State}) +_{\mathsf{Sig}(\texttt{Env})} \mathsf{Sig}(\texttt{State}) \\ &= \mathsf{Sig}(\texttt{Env}) \cup \{c : S,\ c' : S\}. \end{aligned}$$

For example, a relation $R = I_{\{c'=t[c]\}}\Theta$ where t is a term of signature Σ_{cl} not containing an occurrence of c', defines for each algebra A a unique algebra B by $B|_{\mathsf{Sig}(\texttt{Env})} = A|_{\mathsf{Sig}(\texttt{Env})}$ and $B(c) = A(t)$ such that $A\ R\ B$. An example for this type of relation is the `Inc` relation on a counter from Section 5.1

$$\texttt{Inc} = I_{\{c'=\texttt{succ}(c)\}}\Theta_{\texttt{Inc}}.$$

However, there are also other kinds of relations, like, for example the decrement operation on a counter, which has the form:

$$\texttt{Dec} = I_{\{\texttt{succ}(c')=c\}}\Theta_{\texttt{Inc}}.$$

Note that the decrement operation is undefined for an algebra A if $A(c) = A(\texttt{zero})$ as there is no $n \in A(\texttt{Nat})$ such that $A(\texttt{succ})(n) = A(\texttt{zero})$.

If one wants to describe that the value of a state component $c : S$ does not change one can define the relation $R = I_{\{c=c'\}}\Theta$. Then $A\ R\ B$ implies $A(c) = B(c)$. A more general way to define the identity on (part of) the state was shown in Section 5.7; the method presented there has also the advantage that it is independent of the base institution.

Functions

Consider the state space of a dictionary, which we model by a function `map` from keys to values and a function `dom` from keys to `bool`. The intention is that only in the case where `dom`(x) yields **true** the value of `map`(x) is valid.

```
Dict : trait
  includes Env
  introduces
     map : Key → Value
     dom : Key → Bool
```

Then type Θ of the add operation, adding a new key, value pair to the dictionary, is

$$\Theta = <[k : \mathtt{Key}] \times [v : \mathtt{Value}] \times \mathtt{Dict} \times \mathtt{Dict}>_{\mathtt{Env}}$$

A possible way to define the add operation seems to be as

$$\mathtt{Add} = I_{\{\mathtt{map}'(k)=v,\ \mathtt{dom}'(k)=\mathbf{true}\}}\Theta.$$

However, this does not define the expected relation. Consider the relation defined by the above add operation. Add is the set of 4-tuples (A, B, C, D) such that $A + B + C + D \models^{\mathcal{LSL}} \{\mathtt{map}'(k) = v,\ \mathtt{dom}'(k) = \mathbf{true}\}$. Thus the only restrictions on $\mathtt{map}'$ and $\mathtt{dom}'$ are $D(\mathtt{map})(A(k)) = B(v)$ and $D(\mathtt{dom})(A(k)) = D(\mathbf{true})$. Nothing else is assumed, in particular it need not hold that $D(\mathtt{map})(a)$ is the same as $C(\mathtt{map})(a)$ if $a \neq A(k)$. This is in contrast, for example, to the Abstract State Machine approach of Gurevich, where $D(\mathtt{map})(a) = C(\mathtt{map})(a)$ for $a \neq A(k)$ would be implied.

Therefore, to get the intended meaning of the add operation, we have to write $\mathtt{Add} = I_\Phi\Theta$ with Φ being the set:

```
    ∀ x:Key  map'(x) = (if x = k then v else map(x))
    ∀ x:Key  dom'(x) ⇔ x = k ∨ dom(x)
```

Note also that it is important that the sorts Key and Value are defined in the environment Env because otherwise, $A(\mathtt{Key})$, $C(\mathtt{Key})$ and $D(\mathtt{Key})$ need not be the same set and thus a term of the form $D(\mathtt{map})(A(k))$ would be meaningless since $A(k) \in A(\mathtt{Key})$ may not be an element of $D(\mathtt{Key})$ as required by $D(\mathtt{map}) : D(\mathtt{Key}) \to D(\mathtt{Value})$.

Our choice to model dom and map as functions depends strongly on the base institution. For example, dom could have been a predicate instead of a boolean function in an institution with predicates, like first-order logic [25], or one could have omitted dom altogether and used a partial function map instead of a total one, in the institution of partial algebras [48]. And using $\mathcal{SET}$ (cf. Section 9.1) as the base institution, map could be a set of key, value pairs.

Sorts

Similar to constants and functions, sorts can be used as state components. Consider for example a state with a sort OID of object identifies, with a constant nil of sort OID:

```
State : trait
  includes Env
  introduces sorts OID
  introduces nil : OID
```

We want to define a relation R that adds a new element to $\mathtt{OID}$. However, how can two sorts be related? In an algebraic institution no operations are defined on sorts, thus we cannot write $\mathtt{OID}' = \mathtt{OID} \cup \{o\}$. Another way to relate the sorts $\mathtt{OID}$ and $\mathtt{OID}'$ is by a function $f : \mathtt{OID} \to \mathtt{OID}'$, called tracking map in the D-oids approach of Astesiano and Zucca [4]. The problem with this approach is that in $\mathsf{Sig}(\mathtt{State}) +_{\mathtt{Env}} \mathsf{Sig}(\mathtt{State})$, which is the same as

```
Sp_cl : trait
includes Env
introduces sorts OID, OID'
introduces nil : OID
           nil' : OID'
```

there are no functions between the sorts of the pre- and the post-state. However this can be helped by the use of the derive operation and a specification expression of type Θ', where Θ' is

$$\begin{tikzcd}
 & {[f : \mathtt{OID} \to \mathtt{OID}',\ o : \mathtt{OID}']} & \\
\mathtt{State} \arrow[ur] & & \mathtt{State} \arrow[ul] \\
 & \mathtt{Env} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

The notation $[f : \mathtt{OID} \to \mathtt{OID}', o : OID']$ is an abbreviation for

```
includes SP_cl
introduces
  f: OID →  OID'
  o: OID'
```

Then adding a new element to $\mathtt{OID}$ is the following relation $R = D_\iota I_\Phi \Theta'$, where Φ is:

```
    f(nil) = nil'
    f(x) = f(y) ⇒ x = y % f is injective
    f(x) ≠ o            % o is a new object identifier.
```

The type-morphism ι is the inclusion $\Theta = <\mathtt{State} \times \mathtt{State}>_{\mathtt{Env}}$ into Θ'.

An alternative is to define $\mathtt{OID}'$ with the help of a generating constraint as $R' = D_\iota I_{\Phi'} \Theta'$ with $\Phi' = \{\mathtt{OID}'\ \mathsf{generated\ freely\ by}\ \{f,\ o\}\}$. The derive operation D_ι forgets the node defining f and o.

Another problem is to access the newly created object identifier o. Since $\mathtt{OID}$ and $\mathtt{OID}'$ are state components, we cannot use the following type for the new operation:

$$\begin{tikzcd}
\mathtt{State} & \mathtt{State} & {[o : \mathtt{OID}]} \\
 & \mathtt{Env} \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}$$

as the sort $\mathtt{OID}$ and the constant o have nothing to do with the sort $\mathtt{OID}$ in $\mathtt{State}$ and the constant o used in the type Θ'. A solution is to allow the output parameter to be an extension of the state instead of the environment. This gives the output parameter direct access to the state components, which, in this case, is desirable. Thus the type of the new operation is

$$\begin{tikzcd}
\mathtt{State} & & \mathtt{State} \arrow[r] & {[o : \mathtt{OID}]} \\
& \mathtt{Env} \arrow[ul] \arrow[ur] & &
\end{tikzcd}$$

Then we have to adapt Θ' to

$$\begin{tikzcd}
& {[f : \mathtt{OID} \to \mathtt{OID}',\ o : \mathtt{OID}']} & & {[o : \mathtt{OID}]} \\
\mathtt{State} \arrow[ur] & & \mathtt{State} \arrow[ul] \arrow[ur] & \\
& \mathtt{Env} \arrow[ul] \arrow[ur] & &
\end{tikzcd}$$

Note that still o in the output parameter and o in the hidden part of R are not identified, but the colimit $\coprod \Theta'$ is:

```
∐ Θ' : trait
includes Env
introduces sorts OID, OID'
introduces nil: OID
           nil' : OID'
           f: OID → OID'
           o: OID'
           o': OID'
```

contains both constants o as o and o'. Note that the constant o' is of sort $\mathtt{OID}'$ because the sort $\mathtt{OID}$ of the post-state and the parameters are the same since the post-state is included into the output parameter. Then we can define R by $D_\iota I_\Phi \Theta'$ with Φ being:

```
  OID' generated freely by f, o
  o = o'
```

6.2 Models of the State

In the following we use different approaches to model the state of an abstract machine. Though they are described using $\mathcal{LSL}$ as the base institution, the approaches are not limited to algebraic institutions.

One way to model the state is by providing a possible implementation of the state, that is, we give a *concrete model of the state*. A characteristic of the concrete model of the state is the existence of states which cannot be distinguished by the operations of the abstract machine. In this case the specification is biased towards an implementation of the specification (cf. Jones [40]). Note that a concrete model of the state does not prescribe the representation of the state space used by the final implementation of the abstract machine (cf. Section 7.3).

An *abstract model of the state*, or a *two tiered model*, uses a functional abstract datatype modeling the state space and as a component a constant of the sort of the functional abstract datatype. This approach is, for example, used in the Larch approach to specifications [34]. The Larch Shared Language is used to define functional abstract datatypes, which are then used in one of the Larch Interface Languages tailored to a specific programming language to specify an abstract datatype with state.

A third approach uses as state components possible observations on the state. In general, it is not possible to model all abstract machines with this technique because not all abstract machines can be defined in a finite way by the effect the state changing operations have on the observations. An example of this is the abstract machine of a stack as noted by Schoett [55]. The observers of a stack are `isEmpty` and `top`. However, just with these two state components it is impossible to describe the intended behavior of the `Pop` operation.

As an example of the different styles, we shall give the specification of a birthday book from the Z reference manual of Spivey [57] in each of the three styles. The task is to model a birthday book recording the names of people and their birthday. We define three operations on the birthday book: `AddBirthday` adds a name and a date to the birthday book assuming that not already a birthday for that name is recorded, `FindBirthday` returns the date of birth associated to a name in the birthday book, and `Remind` returns a set of names for persons whose birthday is recorded in the birthday book for a given date.

Thus the signature of the birthday book is

$$(\mathtt{Env}, \mathtt{InitBirthday}, \{\mathtt{AddBirthday}, \mathtt{FindBirthday}, \mathtt{Remind}\}, \tau),$$

where τ is:

$$\begin{aligned} \tau(\mathtt{AddBirthday}) &= <[n : \mathtt{Name}] \times [d : \mathtt{Date}]> \rightarrow <> \\ \tau(\mathtt{FindBirthday}) &= <[n : \mathtt{Name}]> \rightarrow <[d : \mathtt{Date}]> \\ \tau(\mathtt{Remind}) &= <[d : \mathtt{Date}]> \rightarrow <[s : \mathtt{NameSet}]>. \end{aligned}$$

The environment `Env` contains at least sorts `Name`, `Date` and the sort `NameSet`, of finite sets of sort `Name`.

Concrete State

In the concrete model of the state, the state of the birthday book is modeled by a finite set of names `known`, containing the names for which the birthday book contains an entry and a finite map `birthday` between names and dates. `known` is modeled as a constant of sort `NameSet`, defined in the specification `Set` and `birthdate` as a constant of sort `NameToDate`, defined in the specification `FiniteMap`. The environment `Env` is given by the specification:

```
Env : trait
includes Set[Name for E, NameSet for C]
         FiniteMap[Name for F, Date for T, NameToDate for M]
introduces sorts Name, Date
```

The specifications for `Set` and `FiniteMap` can be found in the appendix A.

The state of the birthday book is given by

```
BirthdayBook_BC : trait
   includes Env
   introduces
      known: NameSet
      birthday: NameToDate
   asserts
      ∀ n : Name  n ∈ known ⇔ defined(birthday,n)
```

The state invariant requires that if n is in `known` then the association with n as its key is valid in `birthday`. Note that, because in $\mathcal{LSL}$ all functions are total, $\mathtt{apply}(\mathtt{birthday}, n)$ always has a value, however, the value is not specified, if $\mathtt{defined}(\mathtt{birthday}, n)$ does not hold.

Of the three versions of the birthday book example presented here, this version is the closest to the original Z specification of Spivey [57].

The initial birthday book contains no entries.

```
InitBirthday_BC : <BirthdayBook_BC>
   asserts
         known = {}
```

The operation `AddBirthday` adds a new association to `birthday` provided that n is not in `known`. The type $\Theta_{\mathtt{AB}}$ of the `AddBirthday` operation is:

$$<[n : \mathtt{Name}] \times [d : \mathtt{Date}] \times \mathtt{BirthdayBook}_{\mathcal{BC}} \times \mathtt{BirthdayBook}_{\mathcal{BC}}>_{\mathtt{Env}}$$

Then $\mathtt{AddBirthday}_{\mathcal{BC}} = I_\Phi \Theta_{\mathtt{AB}}$ with Φ the set

```
not(n ∈ known);
birthday' = update(birthday,n,d)
```

As in Spivey [57] we have not defined the effect of `AddBirthday` to $\mathtt{known}'$. In particular we would like to show that n is an element of `known` *after* executing the `AddBirthday` operation. This should be a consequence of the definition of `AddBirthday` and the state invariant of `BirthdayBook`. Thus we would like to show

$$\mathtt{AddBirthday} \models^{\mathcal{R}_\mathcal{I}} n \in \mathtt{known}'$$

In Chapter 7 we shall develop methods to do this.

`FindBirthday` returns the birthday d of a person with name n. The type $\Theta_{\mathtt{FB}}$ of `FindBirthday` is:

$$<[n : \mathtt{Name}] \times \mathtt{BirthdayBook}_{\mathcal{BC}} \times \mathtt{BirthdayBook}_{\mathcal{BC}} \times [d : \mathtt{Date}]>_{\mathtt{Env}}$$

Then $\texttt{FindBirthday}_{\mathcal{BC}} = I_\Phi \Theta_{\texttt{FB}}$ with Φ given as:

```
n ∈ known
d = apply(birthday,n)
birthday' = birthday
known' = known
```

For a given date d, the remind operation returns the set of names of persons whose birthday is recorded in the birthday book and who have their birthday at that date. The type of the remind operation is Θ_R:

$$<[d : \texttt{Date}] \times \texttt{BirthdayBook}_{\mathcal{BC}} \times \texttt{BirthdayBook}_{\mathcal{BC}} \times [s : \texttt{NameSet}]>_{\texttt{Env}}$$

And $\texttt{Remind}_{\mathcal{BC}} = I_\Phi \Theta_R$ with Φ being the set

```
known' = known
birthday' = birthday
∀ n : Name  n ∈ s ⇔ known(n) ∧ apply(birthday,n) = d
```

Abstract State

In the abstract state approach, the state space is modeled by a sort BBook defined in the environment, and the state consists of a constant bdb of this sort. The specification BirthdayBookSpec defines the sort BBook with the functions initBirthday, addBirthday, known, and findBirthday. These functions are later used to define the operations AddBirthday, FindBirthday, and Remind.

```
BirthdayBookSpec : trait
   introduces
      initBirthday : BBook
      addBirthday : BBook, Name, Date → BBook

      known : BBook, Name → Bool
      findBirthday : BBook, Name → Date

   asserts
      BBook generated by initBirthday, addBirthday
      forall b:BBook, n,n1:Name, d,d1:Date
         ¬known(initBirthday,n);
         known(addBirthday(b,n1,d),n) ⇔
            n = n1 ∨ known(b,n);
         findBirthday(addBirthday(b,n1,d),n) =
            (if n1 = n
            then d
            else findBirthday(b,n));
```

Note that it is possible to add several dates for the birthday of one person to the birthday book. However, only the birthday added last will be found using the operation findBirthday.

We require that Env includes BirthdayBookSpec:

```
Env : trait
   includes BirthdayBookSpec,
            Set[Name for E, NameSet for C]
```

Then the state space of the birthday book is given by:

```
BirthdayBook$_{\mathcal{BA}}$ : trait
   includes Env
   introduces bdb:  BBook
```

The initial state InitBirthday$_{\mathcal{BA}}$ initializes bdb to the constant initBirthday of type BBook.

```
InitBirthday$_{\mathcal{BA}}$ : <BirthdayBook$_{\mathcal{BA}}$>
   asserts bdb = initBirthday
```

The operations on BirthdayBook may, but need not, rely on the operations provided by BirthdayBookSpec. For example, the AddBirthday-operation is given by:

```
AddBirthday$_{\mathcal{BA}}$:<[n:Name]×[d:Date]×BirthdayBook$_{\mathcal{BA}}$×BirthdayBook$_{\mathcal{BA}}$>$_{\text{Env}}$
   asserts
      known(bdb,n)
      bdb' = addBirthday(bdb,n,d)
```

and FindBirthday:

```
FindBirthday$_{\mathcal{BA}}$:<[n:Name]×BirthdayBook$_{\mathcal{BA}}$×BirthdayBook$_{\mathcal{BA}}$×[d:Date]>$_{\text{Env}}$
   asserts
      known(bdb,n)
      d = findBirthday(bdb,n)
      bdb' = bdb
```

Both, AddBirthday and FindBirthday have used in their definition the corresponding functions addBirthday and findBirthday from the specification BirthdayBookSpec.

```
Remind$_{\mathcal{BA}}$:<[d:Date]×BirthdayBook$_{\mathcal{BA}}$×BirthdayBook$_{\mathcal{BA}}$×[s:NameSet]>$_{\text{Env}}$
 asserts
  ∀ n:Name  n ∈ s ⇔ findBirthday(bdb,n) = d ∧ known(bdb,n)
      bdb' = bdb
```

The abstract state approach is similar to the *two tiered approach* of Larch. First we have given a functional abstract datatype of a birthday book in an algebraic institution $\mathcal{LSL}$. Then we have used this trait to specify an abstract data type with state using formulas relating the components of the pre- and post-states of an operation. But instead of using a Larch Interface Language tailored to a specific programming language, we use the specification language RSL$_{\mathcal{LSL}}$ as a kind of *abstract interface language* that makes no references to a specific programming language.

Using Observers

The basic observations one can make about a birthday book are whether a name is in the birthday book or not and what the birth date is for a given name. Thus the state has two components, a function known : Name → Bool and birthday : Name → Date.

```
BirthdayBook_BO : trait
   includes Env
   introduces
      known: Name → Bool
      birthday : Name → Date
```

In the initial state of a birthday book no names are known, that is,

```
InitBirthday_BO : <BirthdayBook_BO>
   asserts
     ∀ n : Name  known(n) = false
```

The effect of the operations are described by their effect on the observers. Thus the AddBirthday operation is given by:

```
AddBirthday_BO:<[n:Name]×[d:Date]×BirthdayBook_BO×BirthdayBook_BO>_Env
   asserts
     ¬ known(n)
    ∀ n' : Name  known'(n') ⇔ n' = n ∨ known(n')
    ∀ n' : Name  birthday'(n') = if n' = n then d else birthday(n')
```

The FindBirthday operation does not change the observers and returns for a given person with name n its birthday d, if known(n) is true.

```
FindBirthday_BO:<[n:Name]×BirthdayBook_BO×BirthdayBook_BO×[d:Date]>_Env
   asserts
      known(n);
      d = birthday(n)
      ∀ n' : Name  known'(n') = known(n')
      ∀ n' : Name  birthday'(n') = birthday(n')
```

Instead of explicitly writing the formulas that the state of the birthday book does not change, we can define a relation IDBirthdayBook$_{\mathcal{BO}}$ that expresses this:

```
IDBirthdayBook:<BirthdayBook_BO×BirthdayBook_BO>
   asserts
   ∀ n' : Name  known'(n') = known(n')
   ∀ n' : Name  birthday'(n') = birthday(n')
```

then we can write

```
FindBirthday_BO:<[n:Name]×BirthdayBook_BO×BirthdayBook_BO×[d:Date]>_Env
   includes IDBirthdayBook
   asserts
      known(n);
      d = birthday(n)
```

to stand for

$$I_{\Phi}T_{\sigma}\texttt{IDBirthdayBook}_{\mathcal{BO}}$$

where σ is the inclusion of the type of IDBirthdayBook$_{\mathcal{BO}}$ into the type of FindBirthday$_{\mathcal{BO}}$, and Φ is the set $\{\texttt{known}(n),\ d = \texttt{birthday}(n)\}$.

Another way to define $\mathtt{IDBirthdayBook}_{\mathcal{BO}}$ is by using the derive operation with a morphism σ from $\Theta_{\mathtt{IDB}}$ to $\Theta_{\mathtt{IDB}'} = $ $<\mathtt{BirthdayBook}_{\mathcal{BO}}>_{\mathtt{Env}}$, mapping the environment component of $\Theta_{\mathtt{IDB}}$ to the environment component of $\Theta_{\mathtt{IDB}'}$ and both state components to the single state component of $\Theta_{\mathtt{IDB}'}$ (cf. Section 5.7). Then $\mathtt{IDBirthdayBook}_{\mathcal{BO}} = D_\sigma \Theta_{\mathtt{IDB}'}$. To see that the definition of $\mathtt{IDBirthdayBook}_{\mathcal{BO}}$ really does what it is supposed to do, we prove

$$\mathtt{IDBirthdayBook}_{\mathcal{BO}} \models^{\mathcal{R}_{\mathcal{LSL}}} \forall n' : \mathtt{Name}\ \mathtt{known}'(n') = \mathtt{known}(n')$$

The proof that

$$\forall n' : \mathtt{Name}\ \mathtt{birthday}'(n') = \mathtt{birthday}(n')$$

follows from $\mathtt{IDBirthdayBook}_{\mathcal{BO}}$ is analog.

Using the rules for proving properties of specification expressions $\mathtt{SL}_{\mathcal{I}}$ from Section 4.3, it suffices to prove

$$\Theta_{\mathtt{IDB}'} \models^{\mathcal{R}_{\mathcal{LSL}}} \overline{\sigma}(\forall n' : \mathtt{Name}\ \mathtt{known}'(n') = \mathtt{known}(n'))$$

Since the state components of $\Theta_{\mathtt{IDB}}$ are mapped to the same state component of $\Theta_{\mathtt{IDB}'}$, we get that $\sigma(\mathtt{known}') = \mathtt{known} = \sigma(\mathtt{known})$ and therefore it suffices to show

$$\Theta_{\mathtt{IDB}'} \models^{\mathcal{R}_{\mathcal{LSL}}} \forall n' : \mathtt{Name}\ \mathtt{known}(n') = \mathtt{known}(n'),$$

which holds trivially.

The `Remind` operation is defined as follows:

```
Remind_BO:<[d:Date]×BirthdayBook_BO×BirthdayBook_BO×[s:NameSet]>_Env
   includes
      IDBirthdayBook
   asserts
      ∀ n : Name  n ∈ s ⇔ birthday(n) = d ∧ known(n)
      ∀ n : Name  known(n) = known(n)
      ∀ n : Name  birthday'(n) = birthday(n)
```

6.3 Extensions to $\mathtt{RSL}_{\mathcal{I}}$

In the following we shall introduce operations on relations that are useful for defining the operations of abstract machines, and that can be explained in terms of derive, translate, and union.

Sequential Composition

Consider two relations $R \subseteq A \times B$ and $S \subseteq B \times C$. In the first step of the construction we extend R and S to three place relations in the following way:

$$(a,b,c) \in \bar{R} \text{ iff } (a,b) \in R \text{ for every } c \in C \text{ and}$$
$$(a,b,c) \in \bar{S} \text{ iff } (b,c) \in S \text{ for every } a \in A.$$

In the next step we form the intersection of $\bar{R}$ and $\bar{S}$, which yields

$$\begin{aligned}\bar{R} \cap \bar{S} &= \{(a,b,c) \mid (a,b,c) \in \bar{R},\ (a,b,c) \in \bar{S}\}\\ &= \{(a,b,c) \mid (a,b) \in R,\ (b,c) \in S\}\end{aligned}$$

In the last step we forget about the middle component of $\bar{R} \cap \bar{S}$ and get a binary relation

$$\begin{aligned}(\bar{R} \hat{\cap} \bar{S}) &= \{(a,c) \mid \exists b.(a,b,c) \in \bar{R} \cap \bar{S}\}\\ &= \{(a,c) \mid \exists b.(a,b) \in R,\ (b,c) \in S\},\end{aligned}$$

which is the sequential composition $R;S$ of the relation R with S.

Now let R be a relation of type $<\textsc{Sp}_1 \times \textsc{Sp}_2>_{\mathtt{Env}}$ and S a relation of type $<\textsc{Sp}_2 \times \textsc{Sp}_3>_{\mathtt{Env}}$ in $\mathtt{RSL}_{\mathcal{I}}$. In a first step we form the pushout of S and R with respect to the inclusions of $<\textsc{Sp}_2>_{\mathtt{Env}}$ into R and S.

$$\begin{tikzcd}
R & & S \\
& <\textsc{Sp}_2>_{\mathtt{Env}} \arrow[ul, "s_1"] \arrow[ur, "s_2"'] &
\end{tikzcd}$$

where s_1 and s_2 are the inclusions of the relation $<\textsc{Sp}_2>_{\mathtt{Env}}$ into R and S.

To construct the pushout in $\mathtt{RSL}_{\mathcal{I}}$ we first take the pushout of the types of R and S, translate R and S by the co-cone morphism and form their union.

Theorem 5.19 can be used to form the pushout of the following diagram in $\textsc{Type}_{\mathcal{I}}$:

$$\begin{tikzcd}
\textsc{Sp}_1 & & \textsc{Sp}_2 & & \textsc{Sp}_2 & & \textsc{Sp}_3 \\
& \mathtt{Env} \arrow[ul] \arrow[ur] & & & & \mathtt{Env} \arrow[ul] \arrow[ur] & \\
& & & \textsc{Sp}_2 \arrow[uul, dashed, "(s_1{}^{\mu})_1=\mathtt{id}"'] \arrow[uur, dashed, "(s_2{}^{\mu})_1=\mathtt{id}"] & & & \\
& & & \mathtt{Env} \arrow[u] \arrow[uull, dashed, "(s_1{}^{\mu})_0=\mathtt{id}"] \arrow[uurr, dashed, "(s_2{}^{\mu})_0=\mathtt{id}"'] & & &
\end{tikzcd}$$

which is

$$\Theta_{cl} = \begin{tikzcd}
\textsc{Sp}_1 & \textsc{Sp}_2 & \textsc{Sp}_3 \\
& \mathtt{Env} \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}$$

with co-cone morphisms s_1'

$$\begin{tikzcd}
& & \textsc{Sp}_1 & \textsc{Sp}_2 & \textsc{Sp}_3 \\
\textsc{Sp}_1 \arrow[urr, dashed, "\mathtt{id}"] & & \textsc{Sp}_2 \arrow[ur, dashed] & \mathtt{Env} \arrow[ul] \arrow[u] \arrow[ur] & \\
& \mathtt{Env} \arrow[ul] \arrow[ur] \arrow[urr, dashed, "\mathtt{id}"'] & & &
\end{tikzcd}$$

and s'_2

$$\begin{tikzcd}
\textsc{Sp}_1 & \textsc{Sp}_2 & \textsc{Sp}_3 & & \\
& \mathtt{Env} \arrow[ul] \arrow[u] \arrow[ur] & \textsc{Sp}_2 \arrow[ul, dashed] & & \textsc{Sp}_3 \arrow[ull, dashed, "\mathtt{id}"'] \\
& & & \mathtt{Env} \arrow[ul] \arrow[ur] \arrow[ull, dashed, "\mathtt{id}"] &
\end{tikzcd}$$

Then the pushout of R and S with respect to morphism s_1 and s_2 in $\mathtt{RSL}_{\mathcal{I}}$ is $T_{s'_1}R + T_{s'_2}S$.

In the next step, we have to forget the $\textsc{Sp}_2$ component of the pushout. This is done with the help of the derive operation. Then we get

$$R; S = D_\iota(R +_{(<\textsc{Sp}_2>_{\mathtt{Env}}, s_1, s_2)} S) = D_\iota(T_{s'_1}R + T_{s'_2}S),$$

where ι is the inclusion of $<\textsc{Sp}_1 \times \textsc{Sp}_3>_{\mathtt{Env}}$ into $<\textsc{Sp}_1 \times \textsc{Sp}_2 \times \textsc{Sp}_3>_{\mathtt{Env}}$.

$$\begin{tikzcd}
\textsc{Sp}_1 & \textsc{Sp}_2 & \textsc{Sp}_3 \\
\textsc{Sp}_1 \arrow[u, dashed, "\mathtt{id}"] & \mathtt{Env} \arrow[ul] \arrow[u] \arrow[ur] & \textsc{Sp}_3 \arrow[u, dashed, "\mathtt{id}"'] \\
& \mathtt{Env} \arrow[ul] \arrow[u, dashed, "\mathtt{id}"] \arrow[ur] &
\end{tikzcd}$$

Definition 6.1 (Sequential Composition)
Let R be a relation of type $<\textsc{Sp}_1 \times \textsc{Sp}_2>_{\mathtt{Env}}$ and S a relation of type $<\textsc{Sp}_2 \times \textsc{Sp}_3>_{\mathtt{Env}}$. Further, let s_1 and s_2 be the obvious inclusion morphisms of $<\textsc{Sp}_2>_{\mathtt{Env}}$ into the type of R and S. The type $\Theta = <\textsc{Sp}_1 \times \textsc{Sp}_2 \times \textsc{Sp}_3>$ is the colimit of s_1 and s_2. Let further s be the morphism from $<\textsc{Sp}_1 \times \textsc{Sp}_3>$ into Θ mapping $\textsc{Sp}_1$ to the first component and $\textsc{Sp}_3$ to the last component of Θ.

Then the sequential composition of R with S, *$R; S$, is a relation of type $<\textsc{Sp}_1 \times \textsc{Sp}_3>$ and defined by*

$$R; S = D_s(R +_{(<\textsc{Sp}_2>_{\mathtt{Env}}, s_1, s_2)} S).$$

This definition can be extended to arbitrary relations R of type

$$<\textsc{Sp}_1 \times \ldots \times \textsc{Sp}_n \times \textsc{Sp}'_1 \times \ldots \times \textsc{Sp}'_m>_{\mathtt{Env}}$$

and S of type

$$<\textsc{Sp}'_1 \times \ldots \times \textsc{Sp}'_m \times \textsc{Sp}''_1 \times \ldots \times \textsc{Sp}''_k>_{\mathtt{Env}}.$$

Let Θ be $<\mathrm{SP}'_1 \times \ldots \times \mathrm{SP}'_m>_{\mathtt{Env}}$ and $s_1 = ({s_1}^F, \mathtt{id})$ a morphism from Θ to the type of R and $s_2 = ({s_2}^F, \mathtt{id})$ a morphism from Θ to the type of S. The result type of the composition of R with S is

$$\Theta' = <\mathrm{SP}_1 \times \ldots \times \mathrm{SP}_n \times \mathrm{SP}''_1 \times \ldots \times \mathrm{SP}''_k>_{\mathtt{Env}}.$$

Let s be the inclusion of Θ' into the pushout of the type of R and S with respect to s_1 and s_2, then we have

$$R; S = D_s(R +_{(\Theta, s_1, s_2)} S).$$

Instantiation

If R is a binary relation $R \subseteq A \times B$ and S a subset of A then the instantiation of R with S, $R(S)$, is the set $\{b \mid \exists a.\ (a,b) \in R,\ a \in S\}$. One can extend this to arbitrary relations R and S, where the type of S is a subtype of R. For example, let $R \subseteq A \times B \times C$ be a relation and $S \subseteq A \times B$. Then

$$R(S) = \{c \mid \exists a, b.\ (a,b,c) \in R,\ (a,b) \in S\}.$$

Definition 6.2 (Instantiation of Relations)
Let R be a relation of type $\Theta_R = <\mathrm{SP}_1 \times \ldots \times \mathrm{SP}_n>_{\mathtt{Env}}$ and S a relation of type $\Theta_S = <\mathrm{SP}_1 \times \cdots \times \mathrm{SP}_m>_{\mathtt{Env}}$ for $m \leq n$. The instantiation of R with S, $R(S)$, is a relation of type $\Theta_{R(S)} = <\mathrm{SP}_{m+1} \times \cdots \times \mathrm{SP}_n>_{\mathtt{Env}}$ defined by

$$R(S) = D_\iota(R + T_{\iota_S} S),$$

where ι is the inclusion of $\Theta_{R(S)}$ into Θ_R and ι_s the inclusion of Θ_S into Θ_R.

The section about refinement (cf. Section 7.2) contains an example of the use of instantiation.

7 Proving Entailment of Relations

We are interested in methods for showing that a relation R entails another relation S, which, for example, is needed to prove the correctness of refinement (cf. Section 7.2). Applying the results of Section 4.3, the goal of proving $R \models^{\mathcal{R}_\mathcal{I}} S$, where R and S are expressions in $\mathsf{SL}_{\mathcal{R}_\mathcal{I}}$, can be reduced to subgoals $R_i \models^{\mathcal{R}_\mathcal{I}} S_i$ for $1 \leq i \leq n$, which, in a lot of cases, can be further reduced to $\Phi_j \models^{\mathcal{R}_\mathcal{I}} \Phi'_j$ for $1 \leq j \leq m$. Thus the problem is how to prove $\Phi \models^{\mathcal{R}_\mathcal{I}} \Phi'$.

In Section 5.3 we have defined $m \models^{\mathcal{R}_\mathcal{I}} \varphi$ iff $\overline{m} \models^{\mathcal{I}} \varphi$. This might lead to the conclusion that $\Phi \models^{\mathcal{R}_\mathcal{I}} \Phi'$ if and only if $\Phi \models^{\mathcal{I}} \Phi'$. However, while $\Phi \models^{\mathcal{I}} \Phi'$ implies $\Phi \models^{\mathcal{R}_\mathcal{I}} \Phi'$, the converse is not always true.

Theorem 7.1 *Let Φ and Φ' be subsets of* $\mathsf{Sen}_{\mathcal{R}_\mathcal{I}}(\Theta)$ *for $\Theta \in \mathrm{SIGN}_{\mathcal{R}_\mathcal{I}}$, then $\Phi \models^{\mathcal{I}} \Phi'$ implies $\Phi \models^{\mathcal{R}_\mathcal{I}} \Phi'$.*

Proof. Let m be a Θ-structure. We have to show that $m \models^{\mathcal{R}_\mathcal{I}} \Phi$ implies $m \models^{\mathcal{R}_\mathcal{I}} \Phi'$. By the definition of $\models^{\mathcal{R}_\mathcal{I}}$ we have $m \models^{\mathcal{R}_\mathcal{I}} \Phi$ if and only if $\overline{m} \models^{\mathcal{I}} \Phi$. Because $\Phi \models^{\mathcal{I}} \Phi'$, we have $\overline{m} \models^{\mathcal{I}} \Phi'$ and thus $m \models^{\mathcal{R}_\mathcal{I}} \Phi'$. □

Let Θ be (D, Γ), the problem with the inverse direction is that the class of Θ-structures in $\mathcal{R}_\mathcal{I}$ that satisfy Φ is usually only isomorphic to a subclass of $\coprod \Gamma; \mathsf{Sig}^{\mathcal{A}}$-structures in $\mathcal{I}$ that satisfy Φ because $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta)$ is only isomorphic to a subcategory of $\mathsf{Str}_\mathcal{I}(\coprod \Gamma; \mathsf{Sig}^{\mathcal{A}})$. However, if $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta)$ is isomorphic to $\mathsf{Str}_\mathcal{I}(\coprod \Gamma; \mathsf{Sig}^{\mathcal{A}})$ then we get:

Theorem 7.2 *Let Φ and Φ' be subsets of* $\mathsf{Sen}_{\mathcal{R}_\mathcal{I}}(\Theta)$ *for $\Theta = (\mathrm{D}, \Gamma)$ in $\mathrm{SIGN}_{\mathcal{R}_\mathcal{I}}$, and let* $\mathsf{Mod}^{\mathcal{A}}(\mathsf{clim}_{\mathrm{ADT}_\mathcal{I}}(\Theta))$ *be the same as* $\mathsf{Str}_\mathcal{I}(\coprod \Gamma; \mathsf{Sig}^{\mathcal{A}})$. *Then $\Phi \models^{\mathcal{R}_\mathcal{I}} \Phi'$ implies $\Phi \models^{\mathcal{I}} \Phi'$.*

Proof. Assume that m is in $\mathsf{Str}_\mathcal{I}(\coprod \Gamma; \mathsf{Sig}^{\mathcal{A}})$ with $m \models^{\mathcal{I}} \Phi$, then m is in $\mathsf{Mod}^{\mathcal{A}}(\mathsf{clim}_{\mathrm{ADT}_\mathcal{I}}(\Theta))$ because we have assumed that $\mathsf{Str}_\mathcal{I}(\coprod \Gamma; \mathsf{Sig}^{\mathcal{A}})$ is the same as $\mathsf{Mod}^{\mathcal{A}}(\mathsf{clim}_{\mathrm{ADT}_\mathcal{I}}(\Theta))$. Since $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}$ and $\mathsf{clim}_{\mathrm{ADT}_\mathcal{I}}; \mathsf{Mod}^{\mathcal{A}}$ are isomorphic, there exists m' in $\mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta)$ such that $\overline{m}' = m$. By the definition of $\models^{\mathcal{R}_\mathcal{I}}$ we have $m \models^{\mathcal{I}} \Phi$ if and only if $m' \models^{\mathcal{R}_\mathcal{I}} \Phi$. Because $\Phi \models^{\mathcal{R}_\mathcal{I}} \Phi'$, we have $m' \models^{\mathcal{R}_\mathcal{I}} \Phi'$ and therefore $m \models^{\mathcal{I}} \Phi'$. □

However, in general, the condition $\mathsf{Mod}^{\mathcal{A}}(\mathsf{clim}_{\mathrm{ADT}_\mathcal{I}}\Theta) = \mathsf{Str}_\mathcal{I}(\coprod \Gamma; \mathsf{Sig}^{\mathcal{A}})$ does not hold as $\mathsf{clim}_{\mathrm{ADT}_\mathcal{I}}\Theta$ contains restrictions given by the datatypes defined in the environment and possible state invariants. Take the birthday book with the AddBirthday operation from Section 6.2 as an example.

```
Env : trait
includes Set[Name for E, NameSet for C]
         FiniteMap[Name for F, Date for T, NameToDate for M]
introduces sorts Name, Date
```

```
BirthdayBook : trait
   includes Env
   introduces
      known:  NameSet
      birthday:  NameToDate
   asserts
      ∀ n:Name  n ∈ known ⇔ defined(birthday,n)
```

$\mathtt{AddBirthday} = I_\Phi \Theta_{\mathtt{AB}}$ with

$$\Phi = \{n \notin \mathtt{known},\ \mathtt{birthday}' = \mathtt{update}(\mathtt{birthday}, n, d)\}$$

and

$$\Theta_{\mathtt{AB}} = <[n : \mathtt{Name}] \times [d : \mathtt{Date}] \times \mathtt{BirthdayBook} \times \mathtt{BirthdayBook}>_{\mathtt{Env}}$$

We would expect the formula $n \notin \mathtt{known} \Rightarrow n \in \mathtt{known}'$ to follow from Φ in $\mathcal{R}_\mathcal{I}$ since otherwise the state invariant

$$\forall n : \mathtt{Name} \quad n \in \mathtt{known}' \Leftrightarrow \mathtt{defined}(\mathtt{birthday}', n)$$

would not hold for the state after the AddBirthday operation; however, it is easy to see that the above formula is not a consequence of Φ in $\mathcal{I}$. The problem is that we need, in addition to the formulas in Φ, the state invariant, which is part of the type Θ_{AB}, and the definition of `defined`, which is part of the `FiniteMap` trait included in the environment, to prove $n \notin \mathtt{known} \Rightarrow n \in \mathtt{known}'$.

7.1 Translation of RSL$_\mathcal{I}$-Expressions

Thus, to prove $\Phi \models^{\mathcal{R}_\mathcal{I}} \Phi'$, we have to prove that Φ' holds in all models that satisfy Φ and are models of $\coprod \Gamma$. If we use the specification language $\mathtt{SL}_\mathcal{I}$ to specify the abstract datatypes $\Gamma(d)$ for $d \in \mathrm{D}$, that is, if we provide a functor Γ' from D to $\mathtt{SL}_\mathcal{I}$ such that $[\![\Gamma'(d)]\!] = \Gamma(d)$, then we can reduce the task of proving $\Phi \models^{\mathcal{R}_\mathcal{I}} \Phi'$ to the task of proving that the $\mathtt{SL}_\mathcal{I}$-expression $I_{\{\Phi'\}}\Sigma_{cl}$ is entailed by $I_\Phi \Sigma_{cl} + \mathsf{colim}\ \Gamma'$ in $\mathcal{I}$ where Σ_{cl} is $\coprod \Gamma; \mathsf{Sig}^\mathcal{A}$.

Theorem 7.3 *Let* Γ *be a functor from* D *to* $\mathtt{SL}_\mathcal{I}$ *then* $\Theta = (\mathrm{D}, \Gamma; [\![\_]\!])$ *is an object of* $\textsc{Sign}_{\mathcal{R}_\mathcal{I}}$. *Let* Φ *and* Φ' *be sets of* Θ*-formulas then*

$$\Phi \models^{\mathcal{R}_\mathcal{I}} \Phi' \quad \textit{iff} \quad I_\Phi \Sigma_{cl} + \mathsf{colim}\ \Gamma \models^\mathcal{I} I_{\Phi'} \Sigma_{cl},$$

where $\Sigma_{cl} = \coprod_{\mathrm{D}} \Gamma; \mathsf{Sig}$

We do not directly prove this theorem, but use a more general approach. To an $\mathtt{RSL}_\mathcal{I}$-expression R we associate an $\mathtt{SL}_\mathcal{I}$-expressions $\mathtt{tr}(R)$ such that $\mathsf{G}^\mathcal{A}([\![R]\!]_\mathcal{R}) = [\![\mathtt{tr}(R)]\!]$. For two $\mathtt{RSL}_\mathcal{I}$-expressions R and S this implies that $R \models^{\mathcal{R}_\mathcal{I}} S$ if and only if $\mathtt{tr}(R) \models^\mathcal{I} \mathtt{tr}(S)$.

Definition 7.4 *Let R and S be well-formed expressions in $\mathtt{RSL}_{\mathcal{I}}$ of type Θ_R and $\Theta_S = (\mathrm{D}_S, \Gamma_S)$, respectively, Γ a functor from D to $\mathtt{SL}_{\mathcal{I}}$, Φ a set of $(\mathrm{D}, \Gamma; [\![\_]\!])$-formulas and $f : \Theta_R \to \Theta_S$ an arrow in* Type$_{\mathcal{I}}$*. Then*

- $\mathtt{tr}((\mathrm{D}, \Gamma)) = \mathsf{colim}\ \Gamma$
- $\mathtt{tr}(I_\Phi R) = I_\Phi \mathtt{tr}(R)$
- $\mathtt{tr}(T_f R) = T_{\mathtt{tr}(f)} \mathtt{tr}(R) + \mathsf{colim}\ \Gamma_S$
- $\mathtt{tr}(D_f S) = D_{\mathtt{tr}(f)} \mathtt{tr}(S)$
- $\mathtt{tr}(R + S) = \mathtt{tr}(R) + \mathtt{tr}(S)$

and $\mathtt{tr}$ *maps a morphism f in $\mathtt{RSL}_{\mathcal{I}}$ to a morphism $\mathsf{clim}_{\mathtt{SL}_{\mathcal{I}}}(f)$ in $\mathtt{SL}_{\mathcal{I}}$.*

Theorem 7.5 (Correctness of $\mathtt{tr}$) *Let R be an expression in $\mathtt{RSL}_{\mathcal{I}}$ then*

$$\mathsf{G}^{\mathcal{A}}([\![R]\!]_{\mathcal{R}}) = [\![\mathtt{tr}(R)]\!] .$$

Proof. The proof is done by induction over the structure of R.

$$\begin{aligned} \mathsf{G}^{\mathcal{A}}([\![\Theta]\!]_{\mathcal{R}}) &= \mathsf{G}^{\mathcal{A}}(([\![\Theta]\!], \mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}([\![\Theta]\!]))) && \mid \text{Def. } [\![\_]\!]_{\mathcal{R}} \\ &= (\coprod [\![\Gamma]\!] ; \mathsf{Sig}^{\mathcal{A}}, \overline{\mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}([\![\Theta]\!])}) \end{aligned}$$

because of the definition of $\mathsf{G}^{\mathcal{A}}$ and because $\Theta = (\mathrm{D}, \Gamma)$,

$$= (\coprod [\![\Gamma]\!] ; \mathsf{Sig}^{\mathcal{A}}, \mathsf{Mod}^{\mathcal{A}}(\mathsf{clim}_{\mathrm{ADT}_{\mathcal{I}}}([\![\Theta]\!])))$$

because $\mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}$ is natural isomorphic to $\mathsf{clim}_{\mathrm{ADT}_{\mathcal{I}}}; \mathsf{Mod}^{\mathcal{A}}$

$$= \mathsf{clim}_{\mathrm{ADT}_{\mathcal{I}}}([\![\Theta]\!])$$

because $(\mathsf{Sig}^{\mathcal{A}}(\mathrm{Sp}), \mathsf{Mod}^{\mathcal{A}}(\mathrm{Sp}))$ is the same as Sp,

$$\begin{aligned} &= \coprod_{\mathrm{D}} [\![\Gamma]\!] && \mid \text{Def. } \mathsf{clim}_{\mathrm{ADT}_{\mathcal{I}}} \\ &= [\![\mathsf{colim}\ \Gamma]\!] && \mid \text{Theorem 4.12} \\ &= [\![\mathtt{tr}(\Theta)]\!] && \mid \text{Def. } \mathtt{tr}, \end{aligned}$$

where $\Theta = (\mathrm{D}, \Gamma)$.

Impose Let $\llbracket R\rrbracket_{\mathcal{R}} = (\Theta_R, M_R)$ and $\Sigma_R = \coprod \Theta_R; \mathsf{Sig}^{\mathcal{A}}$.

$$\begin{aligned}
\mathsf{G}^{\mathcal{A}}(\llbracket I_\Phi R\rrbracket_{\mathcal{R}}) &= \mathsf{G}^{\mathcal{A}}((\Theta_R, \{m \mid m \models^{\mathcal{R}_{\mathcal{I}}} \Phi,\ m \in M_R\})) && \mid \text{Def. } \llbracket\_\rrbracket_{\mathcal{R}} \\
&= (\Sigma_R, \overline{\{m \mid m \models^{\mathcal{R}_{\mathcal{I}}} \Phi,\ m \in M_R\}}) && \mid \text{Def. } \mathsf{G}^{\mathcal{A}} \\
&= (\Sigma_R, \{\overline{m} \mid \overline{m} \models^{\mathcal{I}} \Phi,\ \overline{m} \in \overline{M}_R\})
\end{aligned}$$

because $m \models^{\mathcal{R}_{\mathcal{I}}} \Phi$ if and only $\overline{m} \models^{\mathcal{I}} \Phi$,

$$= (\Sigma_R, \{\overline{m} \mid \overline{m} \models^{\mathcal{I}} \Phi,\ \overline{m} \in \mathsf{Mod}^{\mathcal{A}}(\mathsf{G}^{\mathcal{A}}(\llbracket R\rrbracket_{\mathcal{R}}))\})$$

because $\mathsf{G}^{\mathcal{A}}((\Theta, M_R))$ is the same as $(\Sigma_R, \overline{M}_R)$,

$$= (\Sigma_R, \{\overline{m} \mid \overline{m} \models^{\mathcal{I}} \Phi,\ \overline{m} \in \mathsf{Mod}^{\mathcal{A}}(\llbracket \mathtt{tr}(R)\rrbracket)\})$$

because of the induction hypotheses,

$$\begin{aligned}
&= \llbracket I_\Phi \mathtt{tr}(R)\rrbracket && \mid \text{Def. } \llbracket\_\rrbracket \\
&= \llbracket \mathtt{tr}(I_\Phi R)\rrbracket && \mid \text{Def. } \mathtt{tr}.
\end{aligned}$$

Union Let $\llbracket R\rrbracket_{\mathcal{R}} = (\Theta, M_R)$, $\llbracket S\rrbracket_{\mathcal{R}} = (\Theta, M_S)$ and $\Sigma_{cl} = \coprod \Theta; \mathsf{Sig}^{\mathcal{A}}$.

$$\begin{aligned}
\mathsf{G}^{\mathcal{A}}(\llbracket R + S\rrbracket_{\mathcal{R}}) &= \mathsf{G}^{\mathcal{A}}((\Theta, M_R \cap M_S)) && \mid \text{Def. } \llbracket\_\rrbracket_{\mathcal{R}} \\
&= (\Sigma_{cl}, \overline{M}_R \cap \overline{M}_S) && \mid \text{Def. } \mathsf{G}^{\mathcal{A}} \\
&= (\Sigma_{cl}, \mathsf{Mod}^{\mathcal{A}}(\mathsf{G}^{\mathcal{A}}(\llbracket R\rrbracket_{\mathcal{R}})) \cap \mathsf{Mod}^{\mathcal{A}}(\mathsf{G}^{\mathcal{A}}(\llbracket S\rrbracket_{\mathcal{R}})))
\end{aligned}$$

because $\mathsf{G}^{\mathcal{A}}((\Theta, M_R)) = (\Sigma_{cl}, \overline{M}_R)$ and $\mathsf{G}^{\mathcal{A}}((\Theta, M_S)) = (\Sigma_{cl}, \overline{M}_S)$

$$= (\Sigma_{cl}, \mathsf{Mod}^{\mathcal{A}}(\llbracket \mathtt{tr}(R)\rrbracket) \cap \mathsf{Mod}^{\mathcal{A}}(\llbracket \mathtt{tr}(S)\rrbracket))$$

because of the induction hypotheses we have $\mathsf{G}^{\mathcal{A}}(\llbracket R\rrbracket_{\mathcal{R}})$ is the same as $\llbracket \mathtt{tr}(R)\rrbracket$ and $\mathsf{G}^{\mathcal{A}}(\llbracket S\rrbracket_{\mathcal{R}})$ is the same as $\llbracket \mathtt{tr}(S)\rrbracket$,

$$\begin{aligned}
&= \llbracket \mathtt{tr}(R) + \mathtt{tr}(S)\rrbracket && \mid \text{Def. } \llbracket\_\rrbracket \\
&= \llbracket \mathtt{tr}(R + S)\rrbracket && \mid \text{Def. } \mathtt{tr}.
\end{aligned}$$

Derive Let $\llbracket R\rrbracket_{\mathcal{R}} = (\Theta_R, M_R)$, $\llbracket f\rrbracket : \Theta_S \to \Theta_R$, $\Sigma_S = \coprod \Theta_S; \mathsf{Sig}^{\mathcal{A}}$ and $\Sigma_R = \coprod \Theta_R; \mathsf{Sig}^{\mathcal{A}}$.

$$\begin{aligned}
\mathsf{G}^{\mathcal{A}}(\llbracket D_f R\rrbracket_{\mathcal{R}}) &= \mathsf{G}^{\mathcal{A}}((\Theta_S, \{m|_{\llbracket f\rrbracket} \mid m \in M_R\})) && \mid \text{Def. } \llbracket\_\rrbracket_{\mathcal{R}} \\
&= (\Sigma_S, \overline{\{m|_{\llbracket f\rrbracket_{\mathcal{R}}} \mid m \in M_R\}}) && \mid \text{Def. } \mathsf{G}^{\mathcal{A}} \\
&= (\Sigma_S, \{\overline{m}|_{\mathsf{clim}_{\mathrm{ADT}_{\mathcal{I}}}(\llbracket f\rrbracket_{\mathcal{R}})} \mid \overline{m} \in \overline{M}_R\})
\end{aligned}$$

because $\mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}$ is natural isomorphic to $\mathsf{clim}_{\mathrm{ADT}_{\mathcal{I}}}; \mathsf{Mod}^{\mathcal{A}}$,

$$= (\Sigma_S, \{\overline{m}|_{\mathsf{clim}_{\mathrm{ADT}_{\mathcal{I}}}(\llbracket f \rrbracket_{\mathcal{R}})} \mid \overline{m} \in \mathsf{Mod}^{\mathcal{A}}(\mathsf{G}^{\mathcal{A}}(\llbracket R \rrbracket_{\mathcal{R}}))\})$$

because $\mathsf{G}^{\mathcal{A}}(R) = (\Sigma_R, \overline{M}_R)$,

$$= (\Sigma_S, \{\overline{m}|_{\llbracket \mathtt{tr}(f) \rrbracket} \mid \overline{m} \in \mathsf{Mod}^{\mathcal{A}}(\llbracket \mathtt{tr}(R)) \rrbracket)\})$$

because of the induction hypotheses,

$$\begin{aligned} &= \left\llbracket D_{\mathtt{tr}(f)} \mathtt{tr}(R) \right\rrbracket && \mid \text{Def. } \llbracket \_ \rrbracket \\ &= \llbracket \mathtt{tr}(D_f R) \rrbracket && \mid \text{Def. } \mathtt{tr} \end{aligned}$$

Translate Let $\llbracket R \rrbracket_{\mathcal{R}} = (\Theta_R, M_R)$, $\llbracket f \rrbracket : \Theta_R \to \Theta_S$, $\Theta_S = (\mathrm{D}_S, \Gamma_S)$, $\Sigma_R = \coprod \Theta_R; \mathsf{Sig}^{\mathcal{A}}$ and $\Sigma_S = \coprod \Theta_S; \mathsf{Sig}^{\mathcal{A}}$.

$$\begin{aligned} \mathsf{G}^{\mathcal{A}}(\llbracket T_f R \rrbracket_{\mathcal{R}}) &= \mathsf{G}^{\mathcal{A}}((\Theta_S, \{m \mid m|_{\llbracket f \rrbracket_{\mathcal{R}}} \in M_R,\ m \in \mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}(\Theta_S)\})) \\ &= (\Sigma_S, \overline{\{m \mid m|_{\llbracket f \rrbracket_{\mathcal{R}}} \in M_R,\ m \in \mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}(\Theta_S)\}}) \\ &= (\Sigma_S, \{\overline{m} \mid \overline{m}|_{\mathsf{clim}_{\mathrm{ADT}_{\mathcal{I}}}(\llbracket f \rrbracket_{\mathcal{R}})]} \in \overline{M}_R,\ \overline{m} \in \mathsf{Mod}^{\mathcal{A}}(\mathsf{clim}_{\mathrm{ADT}_{\mathcal{I}}}(\Theta_S))\}) \\ &= (\Sigma_S, \{\overline{m} \mid \overline{m}|_{\llbracket \mathtt{tr}(f) \rrbracket} \in \mathsf{Mod}^{\mathcal{A}}(\mathsf{G}^{\mathcal{A}}(\llbracket R \rrbracket_{\mathcal{R}})),\ \overline{m} \in \mathsf{Mod}^{\mathcal{A}}(\coprod \Gamma_S)\}) \\ &= (\Sigma_S, \{\overline{m} \mid \overline{m}|_{\llbracket \mathtt{tr}(f) \rrbracket} \in \mathsf{Mod}^{\mathcal{A}}(\llbracket \mathtt{tr}(R) \rrbracket),\ \overline{m} \in \mathsf{Mod}^{\mathcal{A}}(\coprod \llbracket \mathsf{type}(S) \rrbracket)\}) \\ &= (\Sigma_S, \{\overline{m} \mid \overline{m}|_{\llbracket \mathtt{tr}(f) \rrbracket} \in \mathsf{Mod}^{\mathcal{A}}(\llbracket \mathtt{tr}(R) \rrbracket)\} \cap \mathsf{Mod}^{\mathcal{A}}(\llbracket \mathsf{colim\ type}(S) \rrbracket)) \\ &= (\Sigma_S, \mathsf{Mod}^{\mathcal{A}}(\left\llbracket T_{\mathtt{tr}(f)} \mathtt{tr}(R) \right\rrbracket) \cap \mathsf{Mod}^{\mathcal{A}}(\llbracket \mathsf{colim\ type}(S) \rrbracket)) \\ &= \left\llbracket T_{\mathtt{tr}(f)} \mathtt{tr}(R)) + \mathsf{colim\ type}(S) \right\rrbracket \\ &= \llbracket \mathtt{tr}(T_f R) \rrbracket \end{aligned}$$

□

Since $\mathsf{G}^{\mathcal{A}}$ preserves colimits, $\mathtt{tr}$ commutes with the construction of colimits, that is, the abstract datatypes denoted by $\mathtt{tr}(\mathsf{colim}\ F)$ and $\mathsf{colim}\ F; \mathtt{tr}$ are the same.

THEOREM 7.6 *Given a functor F from a finite category J to* $\mathtt{RSL}_{\mathcal{I}}$*, then*

$$\llbracket \mathtt{tr}(\mathsf{colim}\ F) \rrbracket = \llbracket \mathsf{colim}\ F; \mathtt{tr} \rrbracket\,.$$

Proof.

$$\begin{aligned}
[\![\mathtt{tr}(\mathsf{colim}\ F)]\!] &= \mathsf{G}^{\mathcal{A}}([\![\mathsf{colim}\ F]\!]_{\mathcal{R}}) && \mid \text{Thm. 7.5}\\
&= \mathsf{G}^{\mathcal{A}}(\coprod_J F; [\![\_]\!]_{\mathcal{R}}) && \mid \text{Thm. 4.12}\\
&= \coprod_J F; [\![\_]\!]_{\mathcal{R}}; \mathsf{G}^{\mathcal{A}} && \mid \text{Thm. 5.26}\\
&= \coprod_J F; \mathtt{tr}; [\![\_]\!] && \mid \text{Thm. 7.5}\\
&= [\![\mathsf{colim}\ F; \mathtt{tr}]\!] && \mid \text{Thm. 4.12}
\end{aligned}$$

□

Corollary 7.7 *Let R and S be expressions in $\mathtt{RSL}_{\mathcal{I}}$ then*

$$R \models^{\mathcal{R}_{\mathcal{I}}} S \textit{ iff } \mathtt{tr}(R) \models^{\mathcal{I}} \mathtt{tr}(S)$$

Proof. By Fact 5.24 we have $R \models^{\mathcal{R}_{\mathcal{I}}} S$ if and only if $\mathsf{G}^{\mathcal{A}}(R) \models^{\mathcal{I}} \mathsf{G}^{\mathcal{A}}(S)$ and by Theorem 7.5 we get $\mathtt{tr}(R) \models^{\mathcal{I}} \mathtt{tr}(S)$. □

Now we can finish the proof of Theorem 7.3.

Proof of Theorem 7.3. We have

$$\Phi \models^{\mathcal{R}_{\mathcal{I}}} \Phi' \text{ if and only if } I_{\Phi}\Theta \models^{\mathcal{R}_{\mathcal{I}}} I'_{\Phi}\Theta.$$

By Corollary 7.7 we have $I_{\Phi}\Theta \models^{\mathcal{R}_{\mathcal{I}}} I'_{\Phi}\Theta$ if and only if

$$\mathtt{tr}(I_{\Phi}\Theta) \models^{\mathcal{I}} \mathtt{tr}(I'_{\Phi}\Theta).$$

Because $\mathtt{tr}(I_{\Phi}\Theta) = I_{\Phi}\mathsf{colim}\ \Gamma$ and $\mathtt{tr}(I_{\Phi'}\Theta) = I_{\Phi'}\mathsf{colim}\ \Gamma$ we are done if we can prove

$$I_{\Phi}\mathsf{colim}\ \Gamma \models^{\mathcal{I}} I_{\Phi'}\mathsf{colim}\ \Gamma \quad \text{iff} \quad I_{\Phi}\Sigma_{cl} + \mathsf{colim}\ \Gamma \models^{\mathcal{I}} I_{\Phi'}\Sigma_{cl}.$$

Note that R and S are $\mathtt{RSL}_{\mathcal{I}}$-expressions and not $\mathtt{SL}_{\mathcal{R}_{\mathcal{I}}}$-expressions. However, the semantics of $\mathtt{RSL}_{\mathcal{I}}$-expressions are defined by translating them to $\mathtt{SL}_{\mathcal{R}_{\mathcal{I}}}$-expressions and therefore we can apply all the techniques developed in Section 4.3 to entailment of $\mathtt{RSL}_{\mathcal{I}}$ expressions. Then we are done by the following derivations:

(⇒)

$$\dfrac{\dfrac{I_{\Phi}\mathsf{colim}\ \Gamma \models^{\mathcal{I}} I_{\Phi'}\mathsf{colim}\ \Gamma}{I_{\Phi}\Sigma_{cl} + \mathsf{colim}\ \Gamma \models^{\mathcal{I}} I_{\Phi'}\Sigma_{cl} + \mathsf{colim}\ \Gamma} \qquad \dfrac{\dfrac{}{I_{\Phi'}\Sigma_{cl} \models^{\mathcal{I}} I_{\Phi'}\Sigma_{cl}}\,\text{Id}}{I_{\Phi'}\Sigma_{cl} + \mathsf{colim}\ \Gamma \models^{\mathcal{I}} I_{\Phi'}\Sigma_{cl}}\,\text{L6'}}{I_{\Phi}\Sigma_{cl} + \mathsf{colim}\ \Gamma \models^{\mathcal{I}} I_{\Phi'}\Sigma_{cl}}\,\text{Cut}$$

(⇐)

$$\dfrac{\dfrac{I_\Phi\Sigma_{cl} + \mathsf{colim}\ \Gamma \models^{\mathcal{I}} I_{\Phi'}\Sigma_{cl} \qquad \dfrac{\dfrac{}{\mathsf{colim}\ \Gamma \models^{\mathcal{I}} \mathsf{colim}\ \Gamma}\,\text{Id}}{I_\Phi\Sigma_{cl} + \mathsf{colim}\ \Gamma \models^{\mathcal{I}} \mathsf{colim}\ \Gamma}\,\text{L6'}}{I_\Phi\Sigma_{cl} + \mathsf{colim}\ \Gamma \models^{\mathcal{I}} I_{\Phi'}\Sigma_{cl} + \mathsf{colim}\ \Gamma}\,\text{R4}}{I_\Phi\mathsf{colim}\ \Gamma \models^{\mathcal{I}} I_{\Phi'}\mathsf{colim}\ \Gamma}$$

□

The translation of `AddBirthday` yields:

$$\begin{aligned} \mathtt{tr}(\mathtt{AddBirthday}) &= \mathtt{tr}(I_\Phi\Theta_{AB}) \\ &= I_\Phi(\mathtt{tr}(\Theta_{AB})) \\ &= I_\Phi(\mathsf{colim}\ \Gamma_{AB}), \end{aligned}$$

where $\mathsf{colim}\ \Gamma_{\mathtt{AB}}$ is

```
colim ΓAB : trait
includes Env
introduces
   n: Name
   d: Name
   known: NameSet
   birthday: NameToDate
   known': NameSet
   birthday': NameToDate
   asserts
   ∀ n:Name  n ∈ known ⇔ defined(birthday,n)
   ∀ n:Name  n ∈ known' ⇔ defined(birthday',n)
```

and

$$\Phi = \{n \notin \mathtt{known},\ \mathtt{birthday}' = \mathtt{update}(\mathtt{birthday}, n, d)\}$$

To show that `AddBirthday` $\models^{\mathcal{R}_\mathcal{I}} I_{\{n\notin\mathtt{known}\Rightarrow n\in\mathtt{known}'\}}\Theta_{\mathtt{AB}}$ holds, we have to show

$$I_\Phi(\mathsf{colim}\ \Gamma_{\mathtt{AB}}) \models^{\mathcal{I}} n \notin \mathtt{known} \Rightarrow n \in \mathtt{known}',$$

which is implied by $\forall n : \mathtt{Name}\ n \in \mathtt{known}' \Leftrightarrow \mathtt{defined}(\mathtt{birthday}', n)$, $n \notin \mathtt{known}$ and $\mathtt{defined}(\mathtt{birthday}', n)$ and the definition of `defined` in the trait `FiniteMap`, included in the environment.

7.2 Refinement

Usually an abstract datatype denotes the class of possible realizations of a software system (cf. [51, 42, 52]). By making design decisions, the class of possible realizations is refined until one realization or a class of isomorphic realizations remain. Given a specification $\textsc{Sp}$, each design decision introduces a new specification $\textsc{Sp}'$, a *refinement* of $\textsc{Sp}$, such that $\mathsf{Mod}^{\mathcal{A}}(\textsc{Sp}') \subseteq \mathsf{Mod}^{\mathcal{A}}(\textsc{Sp})$. In general, we have several refinement steps $\textsc{Sp}_1, \textsc{Sp}_2, \ldots, \textsc{Sp}_n$ with $\mathsf{Mod}^{\mathcal{A}}(\textsc{Sp}) \supseteq \mathsf{Mod}^{\mathcal{A}}(\textsc{Sp}_1) \supseteq \ldots \supseteq \mathsf{Mod}^{\mathcal{A}}(\textsc{Sp}_n)$. To prove that $\textsc{Sp}_i$ is a refinement of $\textsc{Sp}_{i-1}$, we have to show that $\textsc{Sp}_i \models \textsc{Sp}_{i-1}$. Of course, for a specification to be sensible, we require that $\mathsf{Mod}^{\mathcal{A}}(\textsc{Sp})$ is not empty.

A software component m is *correct* with respect to a specification $\textsc{Sp}$ if m is a model of $\textsc{Sp}$. Let $p[m]$ be a program that uses the software component m. Then p is *correct* with respect to specifications $\textsc{Sp}'$ and $\textsc{Sp}$ if whenever m is a model of $\textsc{Sp}'$, then $p[m]$ is a model of $\textsc{Sp}$.

In contrast, an abstract datatype in the institution $\mathcal{R}_{\mathcal{I}}$ denotes *one* relation and not a class of relations. Therefore the above notion of refinement does not apply directly to objects from $\textsc{Adt}_{\mathcal{R}_{\mathcal{I}}}$. To apply the above notion of refinement, we use a function spec which given a relation R yields a set of relations $\mathsf{spec}(R)$. Then a relation S *refines* a relation R if $\mathsf{spec}(S) \subseteq \mathsf{spec}(R)$.

In principle, the choice of the set $\mathsf{spec}(R)$ can be arbitrary because if p is proved correct with respect to $\mathsf{spec}(S)$ and $\mathsf{spec}(R)$, then for any relation S' in $\mathsf{spec}(S)$ the result of $p[S']$ is in $\mathsf{spec}(R)$ regardless how spec is defined. For example, an extreme case is to define $\mathsf{spec}(R)$ as $\{R\}$.

The usual definition for $\mathsf{spec}(R)$ is as the set of all relations R' such that R can be substituted by R' in any context where R can be used. If R' can substitute R in all contexts, we call R' a *simulation* of R and write $R' \sqsubseteq R$. We get different sets $\mathsf{spec}(R)$ depending on the meaning of "context" and "R can be substituted by R'". For example, compare the definition of Hehner [37], where $R' \sqsubseteq R$ if R' is a subset of R, with the definition of $\sqsubseteq$ below. Usually one requires $\sqsubseteq$ to be a partial order, that is, $\sqsubseteq$ is reflexive and transitive.

Note that, if $\mathsf{spec}(R)$ is defined as $\{R' \mid R' \sqsubseteq R\}$ where $\sqsubseteq$ is a partial order, then proving that S is a refinement of R, that is $\mathsf{spec}(S)$ is a subset of $\mathsf{spec}(R)$, can be simplified to proving $S \sqsubseteq R$, and, similarly, it suffices to show that $p[S] \sqsubseteq R$ to show that a program p is correct with respect to $\mathsf{spec}(S)$ and $\mathsf{spec}(R)$.

The problem in defining $\sqsubseteq$ is to determine what the context where a relation can be used in is and what does it mean that R' can substitute R in that context. In the following we use the most common approach [39, 36, 46, 40, 57], where a context is a program p which is viewed as partial, non-deterministic function from some input values I to some output values O. A program p' *implements* a program p, denoted by $p' \leq p$, if p' is defined on the same input values as p and produces correct output values, that is, if o is a possible output value of p' for a given input value then o is also a possible output value of p for the same input value. Then $R' \sqsubseteq R$ holds if for all programs $p[R]$ that use R, R' can be used instead of R, and $p[R']$ is an implementation of $p[R]$. Note that $\leq$, and therefore $\sqsubseteq$, are reflexive and transitive and that $\sqsubseteq \subseteq \leq$ because R itself can be viewed as the program $p[R] \equiv R$.

In the literature, e.g. [39, 36, 46], it is shown that $\sqsubseteq$ is the same as $\leq$ for a suitable class of programming languages. Suitable means, for example, that a program $p[R]$ can use a relation

R as a component only as a partial, non-deterministic function f_R, that is, if R is a relation $R \subseteq I \times O$, then f_R is a partial, non-deterministic function from I to O such that f_R is defined for all values i in the domain of R and if $f_R(i)$ returns the value o, then $(i, o) \in R$. However, it is not possible for p to have access to *all* possible outcomes of f_R, that is, the set $R(i) = \{o \mid (i, o) \in R\}$.

As a counter-example consider the program

$$p[R] \equiv \text{if } R(1) = \{1, 2\} \text{ then } 1 \text{ else } 2.$$

Let $R' = \{(1, 1)\}$ and $R = \{(1, 1), (1, 2)\}$, although $R' \sqsubseteq R$, we have $p[R'] \not\leq p[R]$ since $p[R']$ yields the relation $\{(i, 2) \mid i \in I\}$ while $p[R]$ yields the relation $\{(i, 1) \mid i \in I\}$.

Thus, for the rest of this section we assume that relations are interpreted as partial, non-deterministic functions and that $\sqsubseteq = \leq$. Then $R' \sqsubseteq R$ if $\mathsf{dom}(R) \subseteq \mathsf{dom}(R')$ and $R'(i) \subseteq R(i)$ for all $i \in \mathsf{dom}(R)$ where $\mathsf{dom}(R)$ is the set of all input values of R for which there exists an output value, that is, $\mathsf{dom}(R) = \{i \mid \exists o\ (i, o) \in R\}$. Let S be a set, then $R|_S$ denotes the restriction of the domain of R to S, that is, $R|_S = \{(i, o) \mid \exists o\ (i, o) \in R,\ i \in S\}$. Then we have $R' \sqsubseteq R$ if and only if

$$\mathsf{dom}(R) \subseteq \mathsf{dom}(R') \text{ and } R'|_{\mathsf{dom}(R)} \subseteq R.$$

In the framework of this thesis we are dealing with relations $R = (\Theta_R, M_R)$ of type $\Theta_R = {<}I_1 \times \ldots \times I_n \times O_1 \times \ldots \times O_n{>}_{\mathtt{Env}}$ where $\Theta_i = {<}I_1 \times \ldots \times I_n{>}$ is the input type of R and $\Theta_o = {<}O_1 \times \ldots \times O_m{>}_{\mathtt{Env}}$ is the output type of R. Let f_i be the injection of Θ_i in Θ_R and f_o the injection of Θ_o in Θ_R. Then we define

$$\mathsf{dom}(R) = D_{f_i} R.$$

As an example consider the `Push` operation on stacks. `Push` has the type

$${<}[e : E] \times \mathtt{Stack} \times \mathtt{Stack}{>}_E.$$

The morphism f_i from ${<}[e : E] \times \mathtt{Stack}{>}_E$ to $\mathsf{type}(\mathtt{Push})$ maps $[e : E]$ to $[e : E]$ and `Stack` to the first `Stack` component. Then the domain of `Push` is a relation of type $\Theta_i = {<}[e : E] \times \mathtt{Stack}{>}_{\mathtt{Env}}$ given by

$$\mathsf{dom}(\mathtt{Push}) = D_{f_i} \mathtt{Push}.$$

Let S be of type Θ_i then we define

$$R|_S = T_{f_i} S + R.$$

$T_{f_i} S$ describes the relation $\{m \in \mathsf{Str}_{\mathcal{R}_\mathcal{I}}(\Theta_R) \mid m|_{f_i} \in \mathsf{Mod}^{\mathcal{A}}(S)\}$, that is, the domain of $T_{f_i} S$ is S and the output values are all possible output values. Then the union of $T_{f_i} S$ with R is the relation R with its domain restricted to S.

Now we can formulate $R' \sqsubseteq R$ as an entailment of relations as follows:

Definition 7.8 (Simulation) *Given relations R' and R of type*

$$\Theta_R = <I_1 \times \ldots \times I_n \times O_1 \times \ldots \times O_m>_{\texttt{Env}}.$$

R' simulates R, denoted by $R' \sqsubseteq R$, if

1. $\mathsf{dom}(R) \models^{\mathcal{R}_\mathcal{I}} \mathsf{dom}(R')$
2. $R'|_{\mathsf{dom}(R)} \models^{\mathcal{R}_\mathcal{I}} R$

We prove that simulation is a preorder as an example of the use of the inference system of Section 4.3.

Theorem 7.9 *Simulation $\sqsubseteq$ is a preorder.*

Proof. We have to show that $\sqsubseteq$ is reflexive and transitive.

For reflexivity we have to show $\mathsf{dom}(R) \models^{\mathcal{R}_\mathcal{I}} \mathsf{dom}(R)$ and $R|_{\mathsf{dom}(R)} \models^{\mathcal{R}_\mathcal{I}} R$. The first condition is trivial and the second condition also if rewritten as $T_{f_i}D_{f_i}R + R \models^{\mathcal{R}_\mathcal{I}} R$.

For transitivity we have to show that $S' \sqsubseteq R$ if $S' \sqsubseteq S$ and $S \sqsubseteq R$. Thus we have to show $D_fR \models^{\mathcal{R}_\mathcal{I}} D_fS'$ and $T_fD_fR+S' \models^{\mathcal{R}_\mathcal{I}} R$. Since $S' \sqsubseteq S$ we have $D_fS \models^{\mathcal{R}_\mathcal{I}} D_fS'$ and therefore

$$\frac{D_fR \vdash D_fS \quad D_fS \vdash D_fS'}{D_fR \vdash D_fS'}\ \text{Cut}$$

Because $S' \sqsubseteq S$, we have $T_fD_fS + S' \models^{\mathcal{R}_\mathcal{I}} S'$, and then we get

$$\frac{\dfrac{T_fD_fS + S' \vdash S}{T_fD_fR + T_fD_fS + S' \vdash T_fD_fR + S}\ \text{M3'} \qquad T_fD_fR + S \vdash R}{T_fD_fR + T_fD_fS + S' \vdash R}\ \text{Cut}$$

and

$$\frac{D_fR \vdash D_fS}{T_fD_fR \vdash T_fD_fS}\ \text{L4}$$

Using the following derived rule (*)

$$\frac{\textsc{Sp}_1 + \textsc{Sp}_2 + \textsc{Sp}_4 \vdash \textsc{Sp}_3 \quad \textsc{Sp}_1 \vdash \textsc{Sp}_2}{\textsc{Sp}_1 + \textsc{Sp}_4 \vdash \textsc{Sp}_3}$$

we get

$$\frac{T_fD_fR + T_fD_fS + S' \vdash R \qquad T_fD_fR \vdash T_fD_fS}{T_fD_fR + S' \vdash R}\ (*)$$

The derivation for rule (*) is:

$$\dfrac{\dfrac{\dfrac{\mathrm{Sp}_1 \vdash \mathrm{Sp}_2}{\mathrm{Sp}_1 + \mathrm{Sp}_4 + \mathrm{Sp}_1 \vdash \mathrm{Sp}_1 + \mathrm{Sp}_4 + \mathrm{Sp}_2}\,\text{M3'} \qquad \mathrm{Sp}_1 + \mathrm{Sp}_2 + \mathrm{Sp}_4 \vdash \mathrm{Sp}_3}{\mathrm{Sp}_1 + \mathrm{Sp}_4 + \mathrm{Sp}_1 \vdash \mathrm{Sp}_3}\,\text{Cut}}{\mathrm{Sp}_1 + \mathrm{Sp}_4 \vdash \mathrm{Sp}_3}$$

□

7.3 Data-Refinement

The following section extends the notion of refinement of relations to refinement of abstract machines. Consider an abstract machine $\mathcal{A} = (\mathtt{St}_\mathcal{A}, I_\mathcal{A}, \mathcal{A}_O)$ of signature $\Sigma = (\mathtt{Env}, I, O, \tau)$. Similar to the previous section, we have the problem that, to talk about the refinement of $\mathcal{A}$, we have to define the class $\mathsf{spec}(\mathcal{A})$ of abstract machines for a given abstract machine $\mathcal{A}$. One way to define $\mathsf{spec}(\mathcal{A})$ is to extend the definition of $\sqsubseteq$ to abstract machines. For an abstract machine $\mathcal{C} = (\mathtt{St}_\mathcal{C}, I_\mathcal{C}, \mathcal{C}_O)$ we have $\mathcal{C} \sqsubseteq \mathcal{A}$ if and only if $\mathtt{St}_\mathcal{C} = \mathtt{St}_\mathcal{A}$, $I_\mathcal{C} = I_\mathcal{A}$, and $op_\mathcal{C} \sqsubseteq op_\mathcal{A}$ for all $o \in O$. Then $\mathsf{spec}(\mathcal{A}) = \{\mathcal{C} \mid \mathcal{C} \sqsubseteq \mathcal{A}\}$.

However, this definition does not take into account the fact that the state of an abstract machine is only accessible by the operations of that abstract machine. That is, there may be abstract machines $\mathcal{C}$ and $\mathcal{A}$ for which $\mathcal{C} \sqsubseteq \mathcal{A}$ does not hold, but which cannot be distinguished by programs p, that is, $p[\mathcal{C}] \leq p[\mathcal{A}]$ holds for all programs p. As in the previous section $p[\mathcal{A}]$ and $p[\mathcal{C}]$ are programs whose semantics are non-deterministic, partial functions between input and output. The differences to the previous section are that the programs may refer to complete abstract machines instead of single relations only and that $\sqsubseteq$ now compares abstract machines while $\leq$ still compares non-deterministic, partial functions.

Therefore we can weaken the above definition and get:

Definition 7.10 (Simulation II)
Let $\mathcal{A}$ and $\mathcal{C}$ be two abstract machines of signature $(\mathtt{Env}, I, O, \tau)$, then $\mathcal{C}$ simulates $\mathcal{A}$, denoted by $\mathcal{C} \sqsubseteq \mathcal{A}$, if there exists a relation $\mathtt{Sim}$ of type $<\mathtt{St}_\mathcal{A} \times \mathtt{St}_\mathcal{C}>_\mathtt{Env}$ such that $I_\mathcal{C} \models^{\mathcal{R}_\mathcal{I}} \mathtt{Sim}(I_\mathcal{A})$ and for all operations $op \in O$:

1. $\mathtt{Sim}_i(\mathsf{dom}(op_\mathcal{A})) \models^{\mathcal{R}_\mathcal{I}} \mathsf{dom}(op_\mathcal{C})$
2. $(\mathtt{Sim}_i; op_\mathcal{C})|_{\mathsf{dom}(op_\mathcal{A})} \models^{\mathcal{R}_\mathcal{I}} op_\mathcal{A}; \mathtt{Sim}_o$

where $\mathtt{Sim}_i$ is the extension of $\mathtt{Sim}$ to a relation of type

$$<I_1 \times \ldots \times I_n \times \mathtt{St}_\mathcal{A} \times \mathtt{St}_\mathcal{C} \times I_1 \times \ldots \times I_n>$$

by the identity on the inputs. If $\tau(op) = <I_1 \times \ldots \times I_n> \to <O_1 \times \ldots \times O_m>$, then $\mathtt{Sim}_i$ is the colimit of $\mathsf{Id}_{I_1}, \ldots, \mathsf{Id}_{I_n}$, and $\mathtt{Sim}$ with respect to the identity on the environment. Similarly, $\mathtt{Sim}_o$ is the colimit of $\mathtt{Sim}$ and $\mathsf{Id}_{O_1} \ldots \mathsf{Id}_{O_m}$.

The second condition can be depicted by:

$$\begin{tikzcd}
I_1 \times \ldots \times I_n \times \mathsf{dom}(op_{\mathcal{A}}) \arrow[r, "op_{\mathcal{A}}"] \arrow[d, "\mathtt{Sim}_i"'] & \mathtt{St}_{\mathcal{A}} \times O_1 \times \ldots \times O_m \arrow[d, "\mathtt{Sim}_o"] \\
I_1 \times \ldots \times I_n \times \mathtt{St}_{\mathcal{C}} \arrow[r, "op_{\mathcal{C}}"'] & \mathtt{St}_{\mathcal{C}} \times O_1 \times \ldots \times O_m
\end{tikzcd}$$

Note that `Sim` is a relation between abstract and concrete state values instead of a function from the concrete state space to the abstract state as, for example, in the seminal work of Hoare [39]. However, later work by Nipkow [46], Schoett [55], and He, Hoare and Sanders [36] have shown that $\mathcal{A}$ may be biased towards a particular implementation (cf. Section 6.2), that is, some states in $\mathtt{St}_{\mathcal{A}}$ cannot be distinguished by the operations in $\mathcal{A}$. In this case one concrete state may be related to several indistinguishable abstract states.

Note also that not every concrete state value needs to be related to an abstract state value since some of the states of $\mathtt{St}_{\mathcal{C}}$ may not be reachable from the initial states and the operations of $\mathcal{C}$.

In general, choosing the empty relation for `Sim` will not work because `Sim` has to relate at least the initial states of $\mathcal{A}$ to the initial states of $\mathcal{C}$.

It was shown by He et al [36] and Nipkow [46] that simulation is sound and complete with respect to implementation, that is, $\mathcal{C} \sqsubseteq \mathcal{A}$ if and only if $p[\mathcal{C}] \leq p[\mathcal{A}]$ for all programs p provided that the programming language ensures the encapsulation principle.

7.4 An Example of Data-Refinement

As a non-trivial example of doing proofs using the methods introduced in this chapter and Chapter 4, we show that the abstract machine $\mathcal{SI}$ of a stack is a correct refinement of the abstract machine $\mathcal{S}$ introduced in Chapter 6.

The abstract machine $\mathcal{SI}$ uses the abstract machine $\mathcal{C}$ of a counter and an abstract machine $\mathcal{A}$ of an array to implement its operations.

Counter The abstract machine $\mathcal{C}$ of a counter has the following signature:

$$(\mathtt{Env}, \mathtt{Zero}, \{\mathtt{Inc} : <> \rightarrow <>, \mathtt{Dec} : <> \rightarrow <>\}),$$

The elements of the state space $\mathtt{St}_{\mathcal{C}}$ of the counter have a state component c of type N. We assume that the specification `Nat` (cf. Section 5.1) is part of the environment `Env`.

```
StC : trait
   includes Env
   introduces c : N
```

In the initial states $\mathtt{Zero}_{\mathcal{C}}$, c is set to the constant `zero` of sort N.

```
Zero : <St_C>
   includes Counter
   asserts c = zero
```

The increment operation is given by

$$\mathtt{Inc}_{\mathcal{C}} = I_{\{\mathtt{succ}(c)=c'\}}{<}\mathtt{St}_{\mathcal{C}} \times \mathtt{St}_{\mathcal{C}}{>}_{\mathtt{Env}}.$$

The decrement operation decrements c only if c is not `zero` and is defined by

$$\mathtt{Dec}_{\mathcal{C}} = I_{\{c\neq\mathtt{zero},\ \mathtt{succ}(c')=c\}}{<}\mathtt{St}_{\mathcal{C}} \times \mathtt{St}_{\mathcal{C}}{>}_{\mathtt{Env}}.$$

Array The abstract machine $\mathcal{A}$ of an array has an operation to update the array at a position i with a new value v and an operation to return the value of the array at a given position. Thus the signature of $\mathcal{A}$ is

$$\begin{aligned}(\mathtt{Env}, \mathtt{Empty}, \{\ & \mathtt{Update} : {<}[i : N] \times [v : E]{>} \to {<>},\\ & \mathtt{Get} : {<}[i : N]{>} \to {<}[v : E]{>}\ \}),\end{aligned}$$

We assume that `Env` contains a sort E and that `Nat` is part of `Env`. The state $\mathtt{St}_{\mathcal{A}}$ of $\mathcal{A}$ is similar to the state of the dictionary defined in Section 6.1. $\mathtt{St}_{\mathcal{A}}$ has two functions `map` and `dom` as components. If $\mathtt{dom}(i)$ is true, $\mathtt{map}(i)$ yields the value of the array at position i. In the empty array, the function `dom` always yields false.

```
St_A : trait
   includes Env
   introduces
      map : N → E
      dom : N → Bool
```

```
Empty_A : trait
   includes St_A
   asserts ∀ n : N   ¬ dom(n)
```

The update operation changes the function `map` such that $\mathtt{map}(i) = v$ is true and for all other $j \neq i$ $\mathtt{map}(i)$ remains the same.

```
Update_A : <[i:N] × [v:E] × St_A × St_A>
asserts
   ∀ n: N  map'(n) = if n = i then v else map(n)
   ∀ n: N  dom'(n) ⇔ n = i ∨ dom(n)
```

The get operation returns the value v for an index value i

```
Get_A : <[i:N] × St_A × St_A × [v:E]>
asserts
   dom(i)
   ∀ n:N  map'(n) = map(n)
   ∀ n:N  dom'(n) = dom(n)
   v = map(i)
```

Stack Implementation In Chapter 6 we have given the definition of an abstract machine of a stack $\mathcal{S}$. Here we define another abstract machine for a stack $\mathcal{SI}$, based on the abstract machine $\mathcal{C}$ of a counter and $\mathcal{A}$ of an array.

The state space of $\mathcal{SI}$ is the pushout of the state spaces for the counter and the array with respect to the environment $\texttt{Env}$ and inclusions $\iota_1 : \texttt{Env} \to \texttt{St}_\mathcal{C}$ and $\iota_2 : \texttt{Env} \to \texttt{St}_\mathcal{A}$ such that all values smaller than the value of the counter have a value associated by the array.

$$\texttt{St}_\mathcal{SI} = I_{\{\forall n : N \; n < c \Rightarrow \mathsf{dom}(n)\}}(\texttt{St}_\mathcal{C} +_{(\iota_1,\iota_2)} \texttt{St}_\mathcal{A}).$$

For simplicity we assume that the environment of $\mathcal{C}$ and $\mathcal{A}$ and are the same. If this is not the case, one can always use the translate operation to adapt the environments.

The initial states $\texttt{Empty}_\mathcal{SI}$ of the stack are basicly given by the pushout of the initial states of the counter and the array with respect to the inclusions $e_1 : \texttt{<Env>}_\texttt{Env} \to \texttt{St}_\mathcal{C}$ and $e_2 : \texttt{<Env>}_\texttt{Env} \to \texttt{St}_\mathcal{A}$. However, the result type of $\texttt{Zero}_\mathcal{C} +_{(e_1,e_2)} \texttt{Empty}_\mathcal{A}$ is $\texttt{<St}_\mathcal{C} +_{(\iota_1,\iota_2)} \texttt{St}_\mathcal{A}\texttt{>}_\texttt{Env}$, which does not include that state-invariant $\{\forall n : N \; n < c \Rightarrow \mathsf{dom}(n)\}$. Thus we have to translate the pushout of $\texttt{Zero}_\mathcal{C}$ with $\texttt{Empty}_\mathcal{A}$ by σ_e, the inclusion of $\texttt{<St}_\mathcal{C} +_{(\iota_1,\iota_2)} \texttt{St}_\mathcal{A}\texttt{>}_\texttt{Env}$ into $\texttt{<St}_\mathcal{SI}\texttt{>}_\texttt{Env}$, yielding

$$\texttt{Empty}_\mathcal{SI} = T_{\sigma_e}(\texttt{Zero}_\mathcal{C} +_{(e_1,e_2)} \texttt{Empty}_\mathcal{A}).$$

Similarly, the push operation can be given as the pushout of the increment operation on counters and the update operation on arrays:

$$\texttt{Push}_\mathcal{SI} = T_{\sigma_{ps}}(\texttt{Inc}_\mathcal{C} +_{(ps_1,ps_2)} \texttt{Update}_\mathcal{A}).$$

The parameter $[i : N]$ of the update operation is given by the pre-state of the increment operation. Thus the $\textsc{Type}_\mathcal{I}$-morphism ps_1 has domain $\texttt{<}[i : N] \times \texttt{Env} \times \texttt{Env}\texttt{>}_\texttt{Env}$ and codomain $\texttt{<St}_\mathcal{C} \times \texttt{St}_\mathcal{C}\texttt{>}$, mapping $[i : N]$ and the first $\texttt{Env}$ to the first $\texttt{St}_\mathcal{C}$. The morphism ps_2 has the same domain as ps_1 and codomain $\texttt{<}[i : N] \times [e : E] \times \texttt{St}_\mathcal{A} \times \texttt{St}_\mathcal{A}\texttt{>}$, mapping $[i : N]$ to $[i : N]$, the first $\texttt{Env}$ to the first $\texttt{St}_\mathcal{A}$ and the second $\texttt{Env}$ to the second $\texttt{St}_\mathcal{A}$. Thus we get the following pushout diagram:

$$\begin{tikzcd}
 & \texttt{<}[e : E] \times \texttt{St}_\mathcal{C} +_{(\iota_1,\iota_2)} \texttt{St}_\mathcal{A} \times \texttt{St}_\mathcal{C} +_{(\iota_1,\iota_2)} \texttt{St}_\mathcal{A}\texttt{>}_\texttt{Env} & \\
\texttt{<St}_\mathcal{C} \times \texttt{St}_\mathcal{C}\texttt{>}_\texttt{Env} \arrow[ur, dashed, "ps'_2"] & & \texttt{<}[i : N] \times [e : E] \times \texttt{St}_\mathcal{A} \times \texttt{St}_\mathcal{A}\texttt{>}_\texttt{Env} \arrow[ul, dashed, "ps'_1"'] \\
 & \texttt{<}[i : N] \times \texttt{Env} \times \texttt{Env}\texttt{>}_\texttt{Env} \arrow[ul, "ps_1"] \arrow[ur, "ps_2"'] &
\end{tikzcd}$$

Note that ps_1 maps $[i : N]$ and the first $\texttt{Env}$ to the same $\texttt{St}_\mathcal{C}$ implies that the co-cone morphism ps'_1 identifies the argument $[i : N]$ of the update operation with the $c : N$ component of the counter. This means that the first input to the update operation is provided by the pre-state of increment operation.

As with $\texttt{Empty}_{\mathcal{SI}}$, we have to add the state-invariant to the pre- and post-states of the push operation by translating $\texttt{Inc}_{\mathcal{C}} +_{(ps_1,ps_2)} \texttt{Update}_{\mathcal{A}}$ by σ_{ps}, where σ_{ps} is the inclusion of $<[e:E] \times \texttt{St}_{\mathcal{C}} +_{(\iota_1,\iota_2)} \texttt{St}_{\mathcal{A}} \times \texttt{St}_{\mathcal{C}} +_{(\iota_1,\iota_2)} \texttt{St}_{\mathcal{A}}>_{\texttt{Env}}$ into $<[e:E] \times \texttt{St}_{\mathcal{SI}} \times \texttt{St}_{\mathcal{SI}}>_{\texttt{Env}}$.

The normal-form of the push operation is

```
Push_SI : <[v:E] × St_SI × St_SI>
asserts
   c' = succ(c)
   ∀ n:N  map'(n) = (if n = i then v else map(n))
   ∀ n:N  dom'(n) ⇔ n = i ∨ dom(n)
```

The pop operation removes the top element from the stack and returns it in v. The pop operation is defined using the pushout of the decrement and the get operations. This time the $c : N$ component of the post-state of the counter is identified with the $i : N$ argument of the get operation:

$$\texttt{Pop}_{\mathcal{SI}} = T_{\sigma_{po}}(\texttt{Dec}_{\mathcal{C}} +_{(po_1,po_2)} \texttt{Get}_{\mathcal{A}}).$$

The morphisms po_1 and po_2 form the following pushout diagram:

$$\begin{tikzcd}
 & <\texttt{St}_{\mathcal{C}} +_{(\iota_1,\iota_2)} \texttt{St}_{\mathcal{A}} \times \texttt{St}_{\mathcal{C}} +_{(\iota_1,\iota_2)} \texttt{St}_{\mathcal{A}} \times [v:E]>_{\texttt{Env}} & \\
<\texttt{St}_{\mathcal{C}} \times \texttt{St}_{\mathcal{C}}>_{\texttt{Env}} \arrow[ur, dashed, "po'_2"] & & <[i:N] \times \texttt{St}_{\mathcal{A}} \times \texttt{St}_{\mathcal{A}} \times [e:E]>_{\texttt{Env}} \arrow[ul, dashed, "po'_1"'] \\
 & <[i:N] \times \texttt{Env} \times \texttt{Env}>_{\texttt{Env}} \arrow[ul, "po_1"] \arrow[ur, "po_2"'] &
\end{tikzcd}$$

po_1 maps $[i:N]$ and the second $\texttt{Env}$ to the second $\texttt{St}_{\mathcal{C}}$ and po_2 maps $[i:N]$ to $[i:N]$ and the first $\texttt{Env}$ to the first $\texttt{St}_{\mathcal{A}}$ and the second $\texttt{Env}$ to the second $\texttt{St}_{\mathcal{A}}$.

σ_{po} is again the inclusion of

$$<\texttt{St}_{\mathcal{C}} +_{(\iota_1,\iota_2)} \texttt{St}_{\mathcal{A}} \times \texttt{St}_{\mathcal{C}} +_{(\iota_1,\iota_2)} \texttt{St}_{\mathcal{A}} \times [v:E]>_{\texttt{Env}}$$

into

$$<\texttt{St}_{\mathcal{SI}} \times \texttt{St}_{\mathcal{SI}} \times [e:E]>_{\texttt{Env}}.$$

The normal-form of $\texttt{Pop}_{\mathcal{SI}}$ is

```
Pop_SI : <St_SI × St_SI × [v:E]>
asserts
   c ≠ zero
   succ(c') = c
   dom(c')
   ∀ n:N  map'(n) = map(n)
   ∀ n:N  dom'(n) = dom(n)
   v = map(c')
```

Proof of Refinement

To show that $\mathcal{SI}$ is a refinement of $\mathcal{S}$, we have to provide a simulation `Sim` of type $<\mathtt{St}_{\mathcal{S}} \times \mathtt{St}_{\mathcal{SI}}>_{\mathtt{Env}}$. The relation `Sim` is defined with the help of a function $\alpha : N \to \mathtt{Stack}$, where $\alpha(n)$ is the stack consisting of the first $n-1$ entries of `map`. Then the relation between $\mathtt{St}_{\mathcal{S}}$ and $\mathtt{St}_{\mathcal{SI}}$ is established by the equation $s = \alpha(c)$.

Formally `Sim` is defined as $D_a(\mathtt{Alpha} + \mathtt{Sim}')$ where `Alpha` defines the function α and $\mathtt{Sim}'$ relates c to s:

```
Alpha : $\Theta_{\mathtt{Alpha}}$
asserts
   $\alpha$(zero) = empty
   $\forall$ n:Nat  succ(n) $\leq$ c $\Rightarrow$ $\alpha$(succ(n)) = push(map(n),$\alpha$(n))

Sim' : $\Theta_{\mathtt{Alpha}}$
asserts
   s = $\alpha$(c)
```

The type $\Theta_{\mathtt{Alpha}}$ is

$$\begin{tikzcd}
 & {[\alpha : N \to \mathtt{Stack}]} & \\
\Theta_{\mathtt{Alpha}} = \mathtt{St}_{\mathcal{SI}} \arrow[ur] & & \mathtt{St}_{\mathcal{S}} \arrow[ul] \\
 & \mathtt{Env} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

with a being the inclusion of $<\mathtt{St}_{\mathcal{S}} \times \mathtt{St}_{\mathcal{SI}}>_{\mathtt{Env}}$ into Θ.

To facilitate the following refinement proofs we use the following lemma:

LEMMA 7.11

$$<\mathtt{St}_{\mathcal{S}} \times \mathtt{St}_{\mathcal{SI}}>_{\mathtt{Env}} \models^{\mathcal{R}_{\mathcal{I}}} D_a\mathtt{Alpha}$$

PROOF. We have to show that each $<\mathtt{St}_{\mathcal{S}} \times \mathtt{St}_{\mathcal{SI}}>_{\mathtt{Env}}$-structure (m_1, m_2) can be extended to an element m of $\mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}(\Theta)$ such that $m|_a = (m_1, m_2)$ and $m \models^{\mathcal{R}_{\mathcal{I}}} \mathit{Alpha}$. What we have to do is to define the appropriate function $m(\alpha) : m_1(N) \to m_1(N)$. Note that because $(m_1, m_2) \in \mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}(<\mathtt{St}_{\mathcal{S}} \times \mathtt{St}_{\mathcal{SI}}>_{\mathtt{Env}})$ and N is a sort in `Env` we have $m_1(N) = m_2(N)$. Provided that `Env` restricts the interpretations of the sort N to be isomorphic to the natural numbers, the equations used in `Alpha` can be viewed as a functional program defining $m(\alpha)$ for each pair (m_1, m_2) in $\mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}(<\mathtt{St}_{\mathcal{S}} \times \mathtt{St}_{\mathcal{SI}}>_{\mathtt{Env}})$. □

Initial States

For the initial states of $\mathcal{S}$ and $\mathcal{SI}$ we have to show

$$\mathtt{Empty}_{\mathcal{SI}} \vdash^{\mathcal{R}_{\mathcal{I}}} \mathtt{Sim}(\mathtt{Empty}_{\mathcal{S}})$$

This means that given a model m of $\mathtt{Empty}_{\mathcal{SI}}$, we have to find a pair (m_1, m_2) such that $m_1 = m$, m_2 is model of $\mathtt{Empty}_{\mathcal{S}}$ and (m_1, m_2) is a model of $\mathtt{Sim}$.

However, we prove a stronger result:

1. for each model m of $\mathtt{Empty}_{\mathcal{SI}}$ there exists a pair (m_1, m_2) such that $m_1 = m$ and m_2 is a model of $\mathtt{Empty}_{\mathcal{S}}$ and
2. for all models m of $\mathtt{Empty}_{\mathcal{SI}}$ and models m_2 of $\mathtt{Empty}_{\mathcal{S}}$, the pair (m, m_2) is a model of $\mathtt{Sim}$.

The first condition is formally written as

$$\mathtt{Empty}_{\mathcal{SI}} \vdash^{\mathcal{R}_{\mathcal{I}}} D_f T_{f'} \mathtt{Empty}_{\mathcal{S}},$$

where f is the inclusion of $<\mathtt{St}_{\mathcal{SI}}>$ into $<\mathtt{St}_{\mathcal{S}} \times \mathtt{St}_{\mathcal{SI}}>$ and f' is the inclusion of $<\mathtt{St}_{\mathcal{S}}>$ into $<\mathtt{St}_{\mathcal{S}} \times \mathtt{St}_{\mathcal{SI}}>$. The second condition is

$$T_f \mathtt{Empty}_{\mathcal{SI}} + T_{f'} \mathtt{Empty}_{\mathcal{S}} \vdash^{\mathcal{R}_{\mathcal{I}}} \mathtt{Sim}.$$

This decomposition is an instance of Rule R3 because the original proof obligation can be rewritten as

$$\mathtt{Empty}_{\mathcal{SI}} \vdash^{\mathcal{R}_{\mathcal{I}}} D_f(\mathtt{Sim} + T_{f'} \mathtt{Empty}_{\mathcal{S}}),$$

using the definition of instantiation.

To show that $\mathtt{Empty}_{\mathcal{SI}} \vdash^{\mathcal{R}_{\mathcal{I}}} D_f T_{f'} \mathtt{Empty}_{\mathcal{S}}$, we observe that f and f' are the co-cone morphisms of the following pushout:

$$\begin{tikzcd}
 & <\mathtt{St}_{\mathcal{S}} \times \mathtt{St}_{\mathcal{SI}}>_{\mathtt{Env}} & \\
<\mathtt{St}_{\mathcal{S}}>_{\mathtt{Env}} \arrow[ur, dashed, "f'"] & & <\mathtt{St}_{\mathcal{SI}}>_{\mathtt{Env}} \arrow[ul, dashed, "f"'] \\
 & <>_{\mathtt{Env}} \arrow[ul, "g"] \arrow[ur, "g'"'] &
\end{tikzcd},$$

where $<>_{\mathtt{Env}}$ denotes the type $(\mathbf{1}, \Theta_{\mathtt{Env}} : \mathbf{1} \to \mathtt{SL}_{\mathcal{I}})$ with $\Theta_{\mathtt{Env}}(1) = \mathtt{Env}$. Using the equalities of Fact 4.9 we can rewrite our proof obligation as $\mathtt{Empty}_{\mathcal{SI}} \vdash^{\mathcal{R}_{\mathcal{I}}} T_{g'} D_g \mathtt{Empty}_{\mathcal{S}}$ and using Rule R5 we have to show

$$D_{g'} \mathtt{Empty}_{\mathcal{SI}} \vdash^{\mathcal{R}_{\mathcal{I}}} D_g \mathtt{Empty}_{\mathcal{S}}.$$

This means that given a model m of $\mathtt{Empty}_{\mathcal{SI}}$, we have to show that $m|_{g'}$, call it $m_{\mathtt{Env}}$, can be extended to some model m' of $\mathtt{Empty}_{\mathcal{S}}$ such that $m'|_{g'} = m_{\mathtt{Env}}$. We are done, if we can show that any model $m_{\mathtt{Env}}$ of $\mathtt{Env}$ can be extended to a model m' of $\mathtt{Empty}_{\mathcal{S}}$ such that $m'|_g = m_{\mathtt{Env}}$, that is, if we can show

$$<>_{\mathtt{Env}} \vdash^{\mathcal{R}_{\mathcal{I}}} D_g \mathtt{Empty}_{\mathcal{S}}$$

because we have $D_{g'}\mathtt{Empty}_{\mathcal{SI}} \vdash^{\mathcal{R}_\mathcal{I}} \mathord{<>}_{\mathtt{Env}}$ by Rule R1. By the definition of $\mathtt{Empty}_\mathcal{S}$ we thus have to show:

$$\mathord{<>}_{\mathtt{Env}} \vdash^{\mathcal{R}_\mathcal{I}} D_g I_{\{s=\mathtt{empty}\}}\mathtt{St}_\mathcal{S}.$$

Let $m_{\mathtt{Env}}$ be a model of $\mathord{<>}_{\mathtt{Env}}$ then define m' by $m'|_g = m_{\mathtt{Env}}$ and $m'(s) = m'(\mathtt{empty}) = m_{\mathtt{Env}}(\mathtt{empty})$. Then m' satisfies $s = \mathtt{empty}$ and thus m' is a model of $\mathtt{Empty}_\mathcal{S}$. Therefore we have $\mathord{<>}_{\mathtt{Env}} \models^{\mathcal{R}_\mathcal{I}} D_g I_{\{s=\mathtt{empty}\}}\mathtt{St}_\mathcal{S}$. Since we have trivially completed our inference system (cf. Section 4.5) we get $\mathord{<>}_{\mathtt{Env}} \vdash^{\mathcal{R}_\mathcal{I}} D_g I_{\{s=\mathtt{empty}\}}\mathtt{St}_\mathcal{S}$.

In the next step we have to show that $T_f\mathtt{Empty}_{\mathcal{SI}} + T_{f'}\mathtt{Empty}_\mathcal{S} \vdash^{\mathcal{R}_\mathcal{I}} \mathtt{Sim}$ holds. Given the definition of $\mathtt{Sim}$ as $D_a(\mathtt{Alpha} + \mathtt{Sim}')$, we can use again Rule R3 to decompose our proof obligation into

1. $T_f\mathtt{Empty}_{\mathcal{SI}} + T_{f'}\mathtt{Empty}_\mathcal{S} \vdash^{\mathcal{R}_\mathcal{I}} D_a\mathtt{Alpha}$ and
2. $T_a(T_f\mathtt{Empty}_{\mathcal{SI}} + T_{f'}\mathtt{Empty}_\mathcal{S}) + \mathtt{Alpha} \vdash^{\mathcal{R}_\mathcal{I}} \mathtt{Sim}'$.

1. is a consequence of Lemma 7.11 and for 2. we note that

- $T_f\mathtt{Empty}_{\mathcal{SI}} + T_{f'}\mathtt{Empty}_\mathcal{S} \vdash^{\mathcal{R}_\mathcal{I}} I_{\{s=\mathtt{empty}\}}\Theta$,
- $T_f\mathtt{Empty}_{\mathcal{SI}} + T_{f'}\mathtt{Empty}_\mathcal{S} \vdash^{\mathcal{R}_\mathcal{I}} I_{\{c=\mathtt{zero}\}}\Theta$ and
- $T_f\mathtt{Empty}_{\mathcal{SI}} + T_{f'}\mathtt{Empty}_\mathcal{S} \vdash^{\mathcal{R}_\mathcal{I}} I_{\{\alpha(\mathtt{zero})=\mathtt{empty}\}}\Theta$

because the formula $s = \mathtt{empty}$ is included in the definition of $\mathtt{Empty}_{\mathcal{SI}}$, the formula $c = \mathtt{zero}$ is included in the definition of $\mathtt{Zero}_\mathcal{C}$ and $\alpha(\mathtt{zero}) = \mathtt{empty}$ is part of the definition of $\mathtt{Alpha}$. Since $\mathtt{Sim}' = I_{\{s=\alpha(c)\}}\Theta_{\mathtt{Alpha}}$ and using Rule L8, we are done if we can show that

$$\{s = \mathtt{empty},\ c = \mathtt{zero},\ \alpha(\mathtt{zero}) = \mathtt{empty}\} \vdash^{\mathcal{R}_\mathcal{I}} \{s = \alpha(c)\}.$$

Using Theorem 7.1 it suffices if we can show

$$\{s = \mathtt{empty},\ c = \mathtt{zero},\ \alpha(\mathtt{zero}) = \mathtt{empty}\} \models^{\mathcal{LSL}} \{s = \alpha(c)\},$$

which is easy to see.

Operations

Next we have to show for the operations $\mathtt{Push}$ and $\mathtt{Pop}$:

$$\mathtt{Sim}_i(\mathsf{dom}(\mathsf{Push}_\mathcal{S})) \vdash^{\mathcal{R}_\mathcal{I}} \mathsf{dom}(\mathsf{Push}_{\mathcal{SI}})$$
$$(\mathtt{Sim}_i; \mathsf{Push}_{\mathcal{SI}})|_{\mathsf{dom}(\mathsf{Push}_\mathcal{S})} \vdash^{\mathcal{R}_\mathcal{I}} \mathsf{Push}_\mathcal{S}; \mathtt{Sim}_o$$

and

$$\mathtt{Sim}_i(\mathsf{dom}(\mathsf{Pop}_\mathcal{S})) \vdash^{\mathcal{R}_\mathcal{I}} \mathsf{dom}(\mathsf{Pop}_{\mathcal{SI}})$$
$$(\mathtt{Sim}_i; \mathsf{Pop}_{\mathcal{SI}})|_{\mathsf{dom}(\mathsf{Pop}_\mathcal{S})} \vdash^{\mathcal{R}_\mathcal{I}} \mathsf{Pop}_\mathcal{S}; \mathtt{Sim}_o$$

As an example, we give the proof for $\mathtt{Pop}$.

Entailment of Preconditions

Step 1 First we have to prove

$$\mathtt{Sim}_i(\mathsf{dom}(\mathtt{Pop}_{\mathcal{S}})) \vdash^{\mathcal{R_I}} \mathsf{dom}(\mathtt{Pop}_{\mathcal{SI}}).$$

By applying the definition of dom this is the same as

$$\mathtt{Sim}_i(\mathsf{dom}(\mathtt{Pop}_{\mathcal{S}})) \vdash^{\mathcal{R_I}} D_{po_i}\mathtt{Pop}_{\mathcal{SI}},$$

where po_i maps $<\mathtt{St}_{\mathcal{SI}}>$ to the first component of $<\mathtt{St}_{\mathcal{SI}} \times \mathtt{St}_{\mathcal{SI}} \times [v : E]>$.

That is, for each model m of $\textsc{Sp} = \mathtt{Sim}_i(\mathsf{dom}(\mathtt{Pop}_{\mathcal{S}}))$ we have to show that there exists a triple of structures (m_1, m_2, m_3), such that $m_1 = m$ and (m_1, m_2, m_3) is a model of $\mathtt{Pop}_{\mathcal{S}}$.

To this end we define a relation $\textsc{Sp}_1$ such that each $<\mathtt{St}_{\mathcal{SI}}>_{\mathtt{Env}}$-structure m can be extended to a model (m, m_2, m_3) of $\textsc{Sp}_1$, that is, we prove

$$<\mathtt{St}_{\mathcal{SI}}> \vdash^{\mathcal{R_I}} D_{po_i}\textsc{Sp}_1,$$

and show that each model of $T_{po_i}\textsc{Sp} + \textsc{Sp}_1$ is also a model of $\mathtt{Pop}_{\mathcal{S}}$, that is,

$$T_{po_i}\textsc{Sp} + \textsc{Sp}_1 \vdash^{\mathcal{R_I}} \mathtt{Pop}_{\mathcal{SI}}.$$

This procedure is justified by Rule R3, if we let $\textsc{Sp}''$ in Rule R3 be $T_{po_i}\textsc{Sp} + \textsc{Sp}_1$. Then $\textsc{Sp} \vdash^{\mathcal{R_I}} D_{po_i}(T_{po_i}\textsc{Sp} + \textsc{Sp}_1)$ is implied by $<\mathtt{St}_{\mathcal{SI}}> \vdash^{\mathcal{R_I}} D_{po_1}\textsc{Sp}_1$ using monotonicity of union and the equalities of Fact 4.9.

$\textsc{Sp}_1$ is given by

```
Sp₁ : <St_SI × St_SI × [v:E]>
asserts
   c' = pred(c)
   v = map(c)
   ∀ n : N  map'(n) = map(n)
   ∀ n : N  dom'(n) ⇔ dom(n)
```

Now let m be a $<\mathtt{St}_{\mathcal{SI}}>$ then we define a $<\mathtt{St}_{\mathcal{SI}}>$-structure m_2 by

- $m_2|_{\mathtt{Env}} = m|_{\mathtt{Env}}$.
- $m_2(\mathtt{map}) = m(\mathtt{map})$,
- $m_2(\mathsf{dom}) = m(\mathsf{dom})$,
- $m_2(c) = m(pred)(m(c))$ and

and a $<[v : E]>$-structure m_3 by

- $m_3|_{\mathtt{Env}} = m|_{\mathtt{Env}}$ and
- $m_3(v) = m(\mathtt{map})(m(c))$,

It is obvious that (m, m_2, m_3) is a model of $\textsc{Sp}_1$ and thus we have

$$<\mathtt{St}_{\mathcal{SI}}> \models^{\mathcal{R}_\mathcal{I}} D_{po_i}\textsc{Sp}_1,$$

which implies

$$<\mathtt{St}_{\mathcal{SI}}> \vdash^{\mathcal{R}_\mathcal{I}} D_{po_i}\textsc{Sp}_1.$$

Step 2 In the second step we have to prove

$$T_{po_i}\textsc{Sp} + \textsc{Sp}_1 \vdash^{\mathcal{R}_\mathcal{I}} \mathtt{Pop}_{\mathcal{SI}}.$$

Bringing $\mathtt{Pop}_{\mathcal{SI}}$ into its normal-form we get $I_{\Phi_1}<\mathtt{St}_{\mathcal{SI}} \times \mathtt{St}_{\mathcal{SI}} \times [v : E]>$, where Φ_1 is the set

```
{  dom(c)
   c ≠ zero
   succ(c') = c
   ∀ n:N  map'(n) = map(n)
   ∀ n:N  dom'(n) = dom(n)
   v = map(c) }
```

The normal-form of $T_{po_i}\textsc{Sp} + \textsc{Sp}_1$ is $D_\sigma I_{\Phi_2}\Theta'$, where Φ_2 is the set

```
{  α(zero) = empty
   n ≤ c ⇒ α(succ(n)) = push(map(n), α(n))
   s = α(c)
   s ≠ empty
   s' = pop(s)
   v' = top(s)
   c'  = pred(c)
   ∀ n:N  map'(n) = map(n)
   ∀ n:N  dom'(n) = dom(n)
   v = map(c) }
```

and σ is the inclusion of $<\mathtt{St}_{\mathcal{SI}} \times \mathtt{St}_{\mathcal{SI}} \times [v : E]>$ into Θ', with Θ' being the type:

$$\begin{tikzcd}
 & {[\alpha : N \to \mathtt{Stack}]} & & & \\
\mathtt{St}_{\mathcal{S}} \arrow[ur] & \mathtt{St}_{\mathcal{S}} & \mathtt{St}_{\mathcal{SI}} \arrow[ul] & \mathtt{St}_{\mathcal{SI}} & {[v : E]} \\
 & & \mathtt{Env} \arrow[ull] \arrow[ul] \arrow[u] \arrow[ur] \arrow[urr] & &
\end{tikzcd}$$

Note that we have two components v and v'; v' denotes the result of the operation $\mathtt{Pop}_{\mathcal{S}}$ of the abstract stack, which is hidden by the dom operation and thus is distinct from the result v of the operation $\mathtt{Pop}_{\mathcal{SI}}$ of the stack implementation.

Now we have to prove:

$$D_\sigma I_{\Phi_1}\Theta' \vdash^{\mathcal{R}_\mathcal{I}} I_{\Phi_2}<\mathtt{St}_{\mathcal{SI}} \times \mathtt{St}_{\mathcal{SI}} \times [v : E]>,$$

which is, by Rule L5, equivalent to

$$I_{\Phi_1}\Theta' \vdash^{\mathcal{R}_\mathcal{I}} I_{\sigma(\Phi_2)}\Theta'.$$

Because of Rule L7 we have nothing to do for formulas in Φ_2 that also occur in Φ_1. The only formulas in Φ_2 not already in Φ_1 are $\mathsf{dom}(c)$, $c \neq \mathtt{zero}$ and $\mathtt{succ}(c') = c$.

Thus it remains to show that $\mathsf{dom}(c)$, $c \neq \mathtt{zero}$ and $\mathtt{succ}(c') = c$ are consequences of Φ_1 in the institution $\mathcal{R}_{\mathcal{LSL}}$. However, to prove this, we cannot apply Theorem 7.1 and simply prove

$$\Phi_1 \models^\mathcal{I} \{\mathsf{dom}(c), c \neq \mathtt{zero}, \mathtt{succ}(c') = c\},$$

as we did in the proof for $\mathtt{Empty}_{\mathcal{SI}} \vdash^{\mathcal{R}_\mathcal{I}} \mathtt{Empty}_\mathcal{S}; \mathtt{Sim}$ given in the previous section. The reason is that we need the state invariant $\forall n : N\ n \leq c \Rightarrow \mathsf{dom}(n)$ given in $\mathtt{St}_{\mathcal{SI}}$ and the definition of predecessor, given in the environment $\mathtt{Env}$. Thus we have to apply Theorem 7.3 and prove

$$I_{\Phi_1}\Sigma + \mathsf{colim}\ \Theta' \models^\mathcal{I} I_{\{\mathsf{dom}(c),\ c\neq\mathtt{zero},\ \mathtt{succ}(c')=c\}}\Sigma,$$

where Σ is the signature of $\mathtt{SL}_{\mathcal{LSL}}$-expression $\mathsf{colim}\ \Theta'$. Then $\mathsf{dom}(c)$ is a consequence of $\forall n : N\ n \leq c \Rightarrow \mathsf{dom}(n)$, $c \neq \mathtt{zero}$ is a consequence of $\alpha(\mathtt{zero}) = \mathtt{Empty}$, $s \neq \mathtt{Empty}$ and $\alpha(c) = s$, and $\mathtt{succ}(c') = c$ is a consequence of $\forall n : N\ n \neq \mathtt{zero} \Rightarrow \mathtt{succ}(\mathtt{pred}(n)) = n$ and $c' = \mathtt{pred}(c)$.

Entailment of Postconditions We have to show:

$$(\mathtt{Sim}_i; \mathtt{Pop}_{\mathcal{SI}})|_{\mathsf{dom}(\mathtt{Pop}_\mathcal{S})} \vdash^{\mathcal{R}_\mathcal{I}} \mathtt{Pop}_\mathcal{S}; \mathtt{Sim}_o$$

Let $\textsc{Sp} = \mathtt{Sim}_i; \mathtt{Pop}_{\mathcal{SI}}$. Using the definition of sequential composition, we have to show:

$$\textsc{Sp}|_{\mathsf{dom}(\mathtt{Pop}_\mathcal{S})} \vdash^{\mathcal{R}_\mathcal{I}} D_{f_3}(\mathtt{Pop}_\mathcal{S} +_{(f_1,f_2)} \mathtt{Sim}_o),$$

where f_1 and f_2 are $\textsc{Type}_\mathcal{I}$-morphisms from $<\mathtt{St}_\mathcal{S} \times [v : E]>$ to the type of $\mathtt{Pop}_\mathcal{S}$ and the type of $\mathtt{Sim}_o$, respectively, and f_3 is the inclusion of

$$<\mathtt{St}_\mathcal{S} \times \mathtt{St}_{\mathcal{SI}} \times [v : E]>_{\mathtt{Env}}$$

into the pushout of the type of $\mathtt{Pop}_\mathcal{S}$ and the type of $\mathtt{Sim}_o$ with respect to f_1 and f_2, which is $\Theta = <\mathtt{St}_\mathcal{S} \times \mathtt{St}_\mathcal{S} \times [v : E] \times \mathtt{St}_{\mathcal{SI}} \times [v : E]>_{\mathtt{Env}}$.

$$\begin{tikzcd}
 & \Theta & \\
\mathsf{type}(\mathtt{Pop}_\mathcal{S}) \arrow[ur, "f_2'"] & & \mathsf{type}(\mathtt{Sim}_o) \arrow[ul, "f_1'"'] \\
 & <\mathtt{St}_\mathcal{S} \times [v : E]>_{\mathtt{Env}} \arrow[ul, "f_1"] \arrow[ur, "f_2"'] &
\end{tikzcd}$$

We can apply Rule R3 if we can show

1. $\textsc{Sp}|_{\mathsf{dom}(\mathtt{Pop}_\mathcal{S})} \vdash^{\mathcal{R}_\mathcal{I}} D_{f_3}T_{f_2'}\mathtt{Pop}_\mathcal{S}$ and
2. $T_{f_3}(\textsc{Sp}|_{\mathsf{dom}(\mathtt{Pop}_\mathcal{S})}) + T_{f_2'}\mathtt{Pop}_\mathcal{S} \vdash^{\mathcal{R}_\mathcal{I}} T_{f_1'}\mathtt{Sim}_o$.

Step 1 First we show:

$$\textsc{Sp}|_{\mathsf{dom}(\mathtt{Pop}_{\mathcal{SI}})} \vdash^{\mathcal{R}_\mathcal{I}} D_{f_3}T_{f_2'}\mathtt{Pop}_\mathcal{S}$$

That is, given a model (m_1, m_2, m_3) of $\textsc{Sp}|_{\mathsf{dom}(\mathtt{Pop}_\mathcal{S})}$, we have to find structures m_2' and m_3', such that (m_1, m_2', m_3') is a model of $\mathtt{Pop}_\mathcal{S}$. Since we have restricted the domain of $\textsc{Sp}|_{\mathsf{dom}(\mathtt{Pop}_\mathcal{S})}$ to $\mathsf{dom}(\mathtt{Pop}_\mathcal{S})$, we know that if (m_1, m_2, m_3) is a model of $\textsc{Sp}|_{\mathsf{dom}(\mathtt{Pop}_\mathcal{S})}$ then m_1 is in $\mathsf{dom}(\mathtt{Pop}_\mathcal{S})$ and thus there exists models m_2' and m_3' such that (m_1, m_2', m_3') is a model of $\mathtt{Pop}_\mathcal{S}$.

We prove this formally by observing that $R|_S \vdash^{\mathcal{R}_\mathcal{I}} T_f S$ because $R|_S$ is defined as $R + T_f S$. Thus we have

$$\textsc{Sp}|_{\mathsf{dom}(\mathtt{Pop}_\mathcal{S})} \vdash^{\mathcal{R}_\mathcal{I}} T_f(\mathsf{dom}(\mathtt{Pop}_\mathcal{S})),$$

where f maps $\mathtt{St}_\mathcal{S}$ in $<\mathtt{St}_\mathcal{S}>_{\mathtt{Env}}$ to the first component of $<\mathtt{St}_\mathcal{S} \times \mathtt{St}_{\mathcal{SI}} \times [v : E]>_{\mathtt{Env}}$. Using the definition of dom we get

$$\textsc{Sp}|_{\mathsf{dom}(\mathtt{Pop}_\mathcal{S})} \vdash^{\mathcal{R}_\mathcal{I}} T_f D_g \mathtt{Pop}_\mathcal{S},$$

where g maps $\mathtt{St}_\mathcal{S}$ of $<\mathtt{St}_\mathcal{S}>_{\mathtt{Env}}$ to the first component of $<\mathtt{St}_\mathcal{S} \times \mathtt{St}_\mathcal{S} \times [v : E]>_{\mathtt{Env}}$.

Note that f_3 and f_2' are the co-cone morphisms of the pushout of $<\mathtt{St}_\mathcal{S} \times \mathtt{St}_\mathcal{S} \times [v : E]>_{\mathtt{Env}}$ with $<\mathtt{St}_\mathcal{S} \times \mathtt{St}_{\mathcal{SI}} \times [v : E]>_{\mathtt{Env}}$ with respect to the morphisms f and g.

$$\begin{tikzcd}
 & \Theta & \\
\mathsf{type}(\mathtt{Pop}_\mathcal{S}) \arrow[ur, "f_2'"] & & \mathsf{type}(\mathtt{Pop}_\mathcal{S}; \mathtt{Sim}_o) \arrow[ul, "f_3"'] \\
 & <\mathtt{St}_\mathcal{S}>_{\mathtt{Env}} \arrow[ul, "g"] \arrow[ur, "f"'] &
\end{tikzcd}$$

Using the equalities of Fact 4.9 we have $T_f D_g \mathtt{Pop}_\mathcal{S} = D_{f_3} T_{f_2'} \mathtt{Pop}_\mathcal{S}$ and thus we get

$$\textsc{Sp}|_{\mathsf{dom}(\mathtt{Pop}_\mathcal{S})} \vdash^{\mathcal{R}_\mathcal{I}} D_{f_3}T_{f_2'}\mathtt{Pop}_\mathcal{S}.$$

Step 2 Thus it remains to show

$$T_{f_3}(\textsc{Sp}|_{\mathsf{dom}(\mathtt{Pop}_\mathcal{S})}) + T_{f_2'}\mathtt{Pop}_\mathcal{S} \vdash^{\mathcal{R}_\mathcal{I}} T_{f_1'}\mathtt{Sim}_o.$$

Because $\textsc{Sp}|_{\mathsf{dom}(\mathtt{Pop}_\mathcal{S})} \vdash^{\mathcal{R}_\mathcal{I}} \textsc{Sp}$ we get

$$T_{f_3}(\textsc{Sp}|_{\mathsf{dom}(\mathtt{Pop}_\mathcal{S})}) + T_{f_2'}\mathtt{Pop}_\mathcal{S} \vdash^{\mathcal{R}_\mathcal{I}} T_{f_3}\textsc{Sp} + T_{f_2'}\mathtt{Pop}_\mathcal{S}$$

by monotonicity of translate and union. By the Cut rule, we are done, if we can prove

$$T_{f_3}\textsc{Sp} + T_{f_2'}\mathtt{Pop}_\mathcal{S} \vdash^{\mathcal{R}_\mathcal{I}} T_{f_1'}\mathtt{Sim}_o.$$

$\mathtt{Sim}_o$ is defined as $D_a(\mathtt{Alpha} + \mathtt{Sim}') +_{(\iota_1,\iota_2)} \mathsf{Id}_{<[v:E]>}$, where ι_1 is the inclusion of $<>_{\mathtt{Env}}$ into the type of $\mathtt{Sim}$, ι_2 the inclusion of $<>_{\mathtt{Env}}$ into the type of $\mathsf{Id}_{[v:E]}$ and $\mathsf{Id}_{[v:E]}$ is defined as $I_{\{v=v'\}}{<}[v : E] \times [v : E]{>}_{\mathtt{Env}}$.

Using the equalities of Fact 4.9 we get that

$$T_{f_1'}(D_a(\mathtt{Alpha} + \mathtt{Sim}') +_{(\iota_1,\iota_2)} \mathsf{Id}_{[v:E]}),$$

denotes the same relation as

$$D_{a'}(T_{h_1}\mathtt{Alpha} + T_{h_1}\mathtt{Sim}' + T_{h_2}\mathsf{Id}_{[v:E]}),$$

where a' is the inclusion of Θ into Θ', which is

$$\begin{tikzcd}
 & & {[\alpha : N \to \mathtt{Stack}]} & & \\
\mathtt{St}_{\mathcal{S}} & \mathtt{St}_{\mathcal{S}} \arrow[ur] & {[v : E]} & \mathtt{St}_{\mathcal{SI}} \arrow[ul] & {[v : E]} \\
 & & \mathtt{Env} \arrow[ull] \arrow[ul] \arrow[u] \arrow[ur] \arrow[urr] & &
\end{tikzcd}$$

h_1 is the inclusion of $\Theta_{\mathtt{Alpha}}$ into Θ' and h_2 is the inclusion of ${<}[v : E] \times [v : E]{>}$ into Θ'.

To apply Rule R3 we have to show

1. $T_{f_3}\mathrm{S}\textsc{p} + T_{f_2'}\mathtt{Pop}_{\mathcal{S}} \vdash^{\mathcal{R}_{\mathcal{I}}} D_{a'}T_{h_1}\mathtt{Alpha}$ and
2. $T_{a'}(T_{f_3}\mathrm{S}\textsc{p} + T_{f_2'}\mathtt{Pop}_{\mathcal{S}}) + T_{h_1}\mathtt{Alpha} \vdash^{\mathcal{R}_{\mathcal{I}}} T_{h_1}\mathtt{Sim}' + T_{h_2}\mathsf{Id}_{[v:E]}$.

Step 3 We have ${<}\mathtt{St}_{\mathcal{S}} \times \mathtt{St}_{\mathcal{SI}}{>}_{\mathtt{Env}} \vdash^{\mathcal{R}_{\mathcal{I}}} D_a\mathtt{Alpha}$ (cf. Lemma 7.11) and h_1 and a' are the co-cone morphisms of the pushout of a and f_1'. Thus we get

$$T_{f_1'}{<}\mathtt{St}_{\mathcal{S}} \times \mathtt{St}_{\mathcal{SI}}{>}_{\mathtt{Env}} = \Theta \vdash^{\mathcal{R}_{\mathcal{I}}} T_{f_1'}D_a\mathtt{Alpha} = D_{a'}T_{h_1}\mathtt{Alpha}$$

Since we have $T_{f_3}\mathrm{S}\textsc{p} + T_{f_2'}\mathtt{Pop}_{\mathcal{S}} \models \Theta$, we get

$$T_{f_3}\mathrm{S}\textsc{p} + T_{f_2'}\mathtt{Pop}_{\mathcal{S}} \vdash^{\mathcal{R}_{\mathcal{I}}} D_{a'}T_{h_1}\mathtt{Alpha}.$$

Step 4 In the next step we show

$$T_{a'}(T_{f_3}\mathrm{S}\textsc{p} + T_{f_2'}\mathtt{Pop}_{\mathcal{S}}) + T_{h_1}\mathtt{Alpha} \vdash^{\mathcal{R}_{\mathcal{I}}} T_{h_1}\mathtt{Sim}' + T_{h_2}\mathsf{Id}_{<[v:E]>_{\mathtt{Env}}}.$$

Using the equalities

$$\mathtt{Sim}' = I_{\{s=\alpha(c)\}}{<}\mathtt{St}_{\mathcal{S}} \times \mathtt{St}_{\mathcal{SI}}{>}_{\mathtt{Env}}$$

and

$$\mathsf{Id}_{<v:E>_{\texttt{Env}}} = I_{\{v'=v\}}{<}[v:E] \times [v:E]{>}_{\texttt{Env}},$$

and using the equalities of Fact 4.9 to form the normal-form of the left and right hand side, we have to show:

$$D_\sigma I_\Phi \Theta''' \vdash^{\mathcal{R}_\mathcal{I}} I_{\{s'=\alpha(c'),\, v'=v\}}\Theta'.$$

Note that s and c are renamed to s' and c' by the $\texttt{SL}_\mathcal{I}$-morphism

$$\mathsf{clim}_{\texttt{SL}_\mathcal{I}}(h_1) : \mathsf{clim}_{\texttt{SL}_\mathcal{I}}(\mathsf{type}(\mathsf{Pop}_\mathcal{S})) \to \mathsf{clim}_{\texttt{SL}_\mathcal{I}}(\Theta')$$

because s is from the post-state of the operation $\texttt{Pop}_\mathcal{S}$ and c is from the post-state of the operation $\texttt{Pop}_{\mathcal{SI}}$.

Θ''' is the type

$$\begin{tikzcd}
{[\alpha : N \to \texttt{Stack}]} & & & {[\alpha : N \to \texttt{Stack}]} & & \\
\texttt{St}_\mathcal{S} \arrow[u] & \texttt{St}_\mathcal{SI} \arrow[ul] & \texttt{St}_\mathcal{S} \arrow[ur] & {[v : E]} & \texttt{St}_\mathcal{SI} \arrow[ul] & {[v : E]} \\
& & \texttt{Env} \arrow[ull] \arrow[ul] \arrow[u] \arrow[ur] \arrow[urr] \arrow[urrr] & & &
\end{tikzcd}$$

and σ the inclusion of Θ' into Θ'''.

Let s correspond to the first occurrence of $\texttt{St}_\mathcal{S}$; $\texttt{map}$, $\texttt{dom}$ and c to the first occurrence of $\texttt{St}_\mathcal{SI}$; α_1 to the first occurrence of $[\alpha : N \to \texttt{Stack}]$; α_2 to the second occurrence of $[\alpha : N \to \texttt{Stack}]$; s' to the second occurrence of $\texttt{St}_\mathcal{S}$ and $\texttt{map}'$, $\texttt{dom}'$ and c' to the second occurrence of $\texttt{St}_\mathcal{SI}$. Then the set Φ is given by the following list of formulas:

```
% Alpha: Def α1
α1(zero) = empty
∀ n:Nat  n < c  ⇒ α1(succ(n)) = push(map(n),α1(n))

% Sim
s = α1(c)

% PopSI = Get + Dec
dom(c)
c ≠ zero
succ(c') = c
∀ n:Nat  dom'(n) = dom(n)
∀ n:Nat  map'(n) = map(n)
v' = map(c')

% PopS
s ≠ empty
s' = pop(s)
```

```
v = top(s)

% Alpha: Def α2
α2(zero) = empty
∀ n:Nat  n < c' ⇒ α2(succ(n)) = push(map'(n),α2(n))
```

To prove $D_\sigma I_\Phi \Theta''' \vdash^{\mathcal{R}_\mathcal{I}} I_{\{s'=\alpha(c'),\, v'=v\}}\Theta'$, we can use Rule L5 and show

$$I_\Phi \Theta''' \vdash^{\mathcal{R}_\mathcal{I}} I_{\{s'=\alpha_2(c'),\, v'=v\}}\Theta'''.$$

Step 5 Note that we cannot prove $\Phi \models^{\mathcal{I}} \{v' = v,\ s' = \alpha_2(c')\}$ because we need properties of natural numbers and stacks for the proof, which are not present in Φ. Therefore we cannot use Theorem 7.1 to prove $\Phi \vdash^{\mathcal{R}_\mathcal{I}} \{v' = v,\ s' = \alpha_2(c')\}$, instead we have to use Theorem 7.7 and show

$$I_\Phi \Sigma_{cl} + \mathsf{colim}\ \Theta''' \models^{\mathcal{I}} I_{\{v'=v,\, s'=\alpha_2(c')\}}\Sigma_{cl},$$

where Σ_{cl} is the signature of the colimit of Θ'''.

Now we have $v' = \mathtt{map}(c')$ and $v = \mathtt{top}(s)$ and $s = \alpha_1(c)$ and $\mathtt{succ}(c') = c$ thus $v = \mathtt{top}(\alpha_1(c))$. Since we have $c \leq c$ because $\forall n : N\ n \leq n$ is derivable from $\mathsf{colim}\ \Theta'''$, we get $v = \mathtt{top}(\mathtt{push}(\mathtt{map}(c'), \alpha_1(c')))$ which yields $v = \mathtt{map}(c')$ because

$$\forall e : E,\ s : \mathtt{Stack}\ \mathtt{top}(\mathtt{push}(e, s)) = e$$

is derivable from $\mathsf{colim}\ \Theta'''$.

To prove $s' = \alpha_2(c')$ we need the lemma

$$\forall n : N\ n \leq c' \Rightarrow \alpha_1(n) = \alpha_2(n),$$

which can be shown by induction on n as follows:

- $n = \mathtt{zero}$: We have $\alpha_1(\mathtt{zero}) = \mathtt{empty} = \alpha_2(\mathtt{zero})$.
- $n = \mathtt{succ}(n')$: Assume that $\mathtt{succ}(n') \leq c'$ then

  $$\alpha_1(n) = \mathtt{push}(\mathtt{map}(n), \alpha_1(n')).$$

  Since $n' \leq \mathtt{succ}(n') \leq c'$ we can apply the induction hypotheses and with $n \leq c$ we get $\alpha_1(n) = \mathtt{push}(\mathtt{map}(n'), \alpha_2(n'))$ and because $\forall m : N\ \mathtt{map}'(m) = \mathtt{map}(m)$, we get $\alpha_1(n) = \mathtt{push}(\mathtt{map}'(n'), \alpha_2(n'))$, which is $\alpha_2(n)$, by the definition of α_2.

Note that the justification to use induction over $n : N$ comes from the constraint that N is generated by $\mathtt{zero}$ and $\mathtt{succ}$, derivable from $\mathsf{colim}\ \Theta'''$.

Now we have

$$\begin{aligned} s' &= \mathtt{pop}(s) && \mid s = \mathtt{pop}(s) \in \Phi \\ &= \mathtt{pop}(\alpha_1(c)) && \mid s = \alpha_1(c) \in \Phi \\ &= \mathtt{pop}(\alpha_1(\mathtt{succ}(c'))) && \mid \mathtt{succ}(c') = c \in \Phi \\ &= \mathtt{pop}(\mathtt{push}(\mathtt{map}(c'), \alpha_1(c'))) && \mid c \leq c \\ &= \alpha_1(c') && \mid c' \leq c' \\ &= \alpha_2(c') \end{aligned}$$

7.5 Conclusion

The proof that the abstract machine of a stack $\mathcal{SI}$ is an implementation of $\mathcal{S}$ is surprisingly quite complex.

One of the reasons is that we have to make explicit the relationship between the type of the relations by using $\textsc{Type}_{\mathcal{I}}$-morphisms. This is due to the fact that we assumed that everything which is not related by morphisms is different even if the components have the same name. This requires to make explicit even simple inclusions of one type into another and to adapt the type of relations by using the translate operation.

In practice though it should be possible to hide a lot of these morphisms and the corresponding translate operations from the user.

Another reason for the complexity of the proofs is the use of the derive operation in the definition of the relations that occur on the right hand side of a turnstile. For example, expanding the right hand side of

$$(\mathtt{Sim}_i; \mathtt{Pop}_{\mathcal{SI}})|_{\mathsf{dom}(\mathtt{Pop}_{\mathcal{S}})} \vdash^{\mathcal{R}_{\mathcal{I}}} \mathtt{Pop}_{\mathcal{S}}; \mathtt{Sim}_o$$

yields

$$D_{f_3}(T_{f_2'}\mathtt{Pop}_{\mathcal{S}} + D_{a'}(T_{h_1}\mathtt{Alpha} + T_{h_1}\mathtt{Sim}' + T_{h_2}\mathsf{Id}_{[v:E]})),$$

which contains two derive operations. To get rid of the derive operations we have to use two times the Rule R3, which introduces two proof obligations involving again derive: $\ldots \vdash^{\mathcal{R}_{\mathcal{I}}} D_{f_3}T_{f_2'}\mathtt{Pop}_{\mathcal{S}}$ and $\ldots \vdash^{\mathcal{R}_{\mathcal{I}}} D_{a'}T_{h_1}\mathtt{Alpha}$.

On some occasions, it helps to view the derive operation as an existential quantifier, and then use the base institution to deal with that quantifier. For example, let

$$\mathtt{Pop}_{\mathcal{S}} = I_{\{s'=\mathtt{pop}(s),\ v=\mathtt{top}(s)\}}\Theta_{\mathtt{Pop}_{\mathcal{S}}}$$

where $\Theta_{\mathtt{Pop}_{\mathcal{S}}} = (D_{\mathtt{Pop}_{\mathcal{S}}}, \Gamma_{\mathtt{Pop}_{\mathcal{S}}})$. $D_{f_3}T_{f_2'}\mathtt{Pop}_{\mathcal{S}}$ corresponds to

$$I_{\{\exists s':\mathtt{Stack}, v:E\ \ s'=\mathtt{pop}(s)\wedge v=\mathtt{top}(s)\}}\Theta_{\mathtt{Pop}_{\mathcal{S}}}.$$

Then for any $\mathtt{RSL}_{\mathcal{I}}$-expression R we have $R \models^{\mathcal{R}_{\mathcal{I}}} D_{f_3} T_{f_2'} \mathsf{Pop}_{\mathcal{S}}$ if and only if

$$R \models^{\mathcal{R}_{\mathcal{I}}} I_{\{\exists s' : \mathtt{Stack}, v : E \;\; s' = \mathtt{pop}(s) \wedge v = \mathtt{top}(s)\}} \Theta_{\mathtt{Pop}_{\mathcal{S}}},$$

and because of Theorem 7.5 if and only if

$$\mathtt{tr}(R) \models^{\mathcal{I}} I_{\{\exists s' : \mathtt{Stack}, v : E \;\; s' = \mathtt{pop}(s) \wedge v = \mathtt{top}(s)\}} \mathsf{colim}\; \Gamma_{\mathtt{Pop}_{\mathcal{S}}}.$$

Now we can use a theorem prover for the institution $\mathcal{I}$ to proof this entailment and thus deal with the derive operation.

However, this correspondence does not hold for all logics, for example, the above formula is not admissible in $\mathcal{LSL}$. Also $D_{a'} T_{h_1} \mathtt{Alpha}$ requires a second order existential quantifier as it states the existence of an operation $\alpha : N \to \mathtt{Stack}$.

8 Disjunction

A useful operation to define new relations is the disjunction of relations. Disjunction allows to define separate relations for each subset of input values and then define a relation on all input values by the disjunction of the component relations. For example in Spivey [57], the `AddBirthday` operation (cf. Section 6.2) is extended to yield an error message if n is already in `known`.

Let us recall the definition of the state space of the birthday book

```
BirthdayBook$_{\mathcal{BC}}$ : trait
   includes Env
   introduces
      known: NameSet
      birthday: NameToDate
   asserts
      ∀ n:Name  n ∈ known ⇔ defined(birthday,n)
```

and the definition of the `AddBirthday` operation

```
AddBirthday$_{\mathcal{BC}}$: Θ$_{\mathtt{AB}}$
   not(n ∈ known)
   birthday' = update(birthday,n,d)
```

where the type $\Theta_{\mathtt{AB}}$ is:

$$<[n : \mathtt{Name}] \times [d : \mathtt{Date}] \times \mathtt{BirthdayBook} \times \mathtt{BirthdayBook}>_{\mathtt{Env}}$$

We define a new operation $\mathtt{Success}_{\mathcal{BC}}$ of type:

$$\Theta_R = <[\mathtt{result} : \mathtt{Report}]>_{\mathtt{Env}}$$

```
Success$_{\mathcal{BC}}$ : Θ$_{\mathtt{R}}$
   result = ok
```

We assume that `Env` contains a sort `Report` which is freely generated by constants `ok` and `alreadyKnown`. Given the type $\Theta_{\mathtt{RAB}}$ as

$$<[n : \mathtt{Name}] \times [d : \mathtt{Date}] \times \mathtt{BirthdayBook} \times \mathtt{BirthdayBook} \times [\mathtt{result} : \mathtt{Report}]>_{\mathtt{Env}}$$

with inclusions f of Θ_{AB} into Θ_{RAB} and g of Θ_R into Θ_{RAB} then

$$T_f\mathtt{AddBirthday} + T_g\mathtt{Success}$$

is a relation that adds a name to the birthday book and has `ok` as the value of `result`. The relation $\mathtt{AlreadyKnown}_{\mathcal{BC}}$ is defined by

```
AlreadyKnown_BC : <[n:Name]×BirthdayBook×BirthdayBook×[result:Report]>
   asserts
      known' = known
      birthday' = birthday
      n ∈ known
      result = alreadyKnown
```

$\texttt{AlreadyKnown}_{\mathcal{BC}}$ does not change `known` and `birthday` and the value of `result` is `alreadyKnown`. Then a robust version of the $\texttt{AddBirthday}_{\mathcal{BC}}$ is

$$\texttt{RAddBirthday}_{\mathcal{BC}} = (T_f\texttt{AddBirthday}_{\mathcal{BC}} + T_g\texttt{Success}_{\mathcal{BC}}) \vee T_h\texttt{AlreadyKnown}_{\mathcal{BC}}$$

with h being the inclusion of the type of $\texttt{AlreadyKnown}_{\mathcal{BC}}$ into Θ_{RAB}. The operation $\texttt{RAddBirthday}_{\mathcal{BC}}$ adds a birthday for a person which is not already in the birthday-book and reports `ok`, or does not change `known` and `birthday` and reports $\texttt{alreadyKnown}_{\mathcal{BC}}$.

8.1 Adding Disjunction to $\texttt{SL}_{\mathcal{I}}$

Similar to the union of two specification expressions $\textsc{Sp}_1$ and $\textsc{Sp}_2$, which denotes the intersection of their model classes, we can define the disjunction of $\textsc{Sp}_1$ and $\textsc{Sp}_2$ in any institution $\mathcal{I}$, having the union of their model classes as the semantics. If $\textsc{Sp}_1$ is given as $I_{\Phi_1}\Sigma$ and $\textsc{Sp}_2$ as $I_{\Phi_2}\Sigma$ then m is a model of $\textsc{Sp}_1 \vee \textsc{Sp}_2$ if and only if $m \models \Phi_1$ or $m \models \Phi_2$.

Disjunction as an operation to form abstract datatypes has not been treated in the literature (cf. Sannella and Wirsing [54], Sannella and Tarlecki [50], Loeckx, Ehrich and Wolf [42] and Farrés-Casals [21]). One reason for this is that in contrast to union, disjunction does not have a semantics on the presentation level independent from the choice of the institution. For example, for union we have

$$I_{\Phi_1}\Sigma + I_{\Phi_2}\Sigma = I_{\Phi_1\cup\Phi_2}\Sigma,$$

while for disjunction

$$I_{\Phi_1}\Sigma \vee I_{\Phi_2}\Sigma = I_{\Phi}\Sigma$$

only holds in institutions, where one can express the disjunction of two sets of formulas again by a set of formulas (cf. Section 8.2).

The abstract syntax of specification expressions of section 4.2 is extended by

$$\textsc{Sp} ::= \ \dots \ | \ \textsc{Sp} \vee \textsc{Sp},$$

where $\textsc{Sp}_1 \vee \textsc{Sp}_2$ is well-formed if $\mathsf{Sig}(\textsc{Sp}_1) = \mathsf{Sig}(\textsc{Sp}_2)$ and $\textsc{Sp}_1$ and $\textsc{Sp}_2$ are well-formed. The signature of $\textsc{Sp}_1 \vee \textsc{Sp}_2$ is the signature of $\textsc{Sp}_1$ and the semantics of disjunction is

$$[\![\textsc{Sp}_1 \vee \textsc{Sp}_1]\!] = (\Sigma, M_1 \cup M_2)$$

with $[\![\textsc{Sp}_1]\!] = (\Sigma, M_1)$ and $[\![\textsc{Sp}_2]\!] = (\Sigma, M_2)$.

8.2 Proving Properties with Disjunction

One problem with proving entailment $\mathrm{SP}_1 \models^{\mathcal{I}} \mathrm{SP}_2$ for specification expressions in $\mathtt{SL}_{\mathcal{I}}$ with disjunction is that the normal-form result of Section 4.2 does not hold anymore.

In the case of union we could use the equation $D_\sigma \mathrm{SP}_1 + \mathrm{SP}_2 = D_\sigma(\mathrm{SP}_1 + T_\sigma \mathrm{SP}_2)$ to pull D_σ out from the union (cf. Section 4.2). However for disjunction only

$$D_\sigma(\mathrm{SP}_1 \vee T_\sigma \mathrm{SP}_2) \models^{\mathcal{I}} D_\sigma \mathrm{SP}_1 \vee \mathrm{SP}_2$$

is valid, while

$$D_\sigma \mathrm{SP}_1 \vee \mathrm{SP}_2 \models^{\mathcal{I}} D_\sigma(\mathrm{SP}_1 \vee T_\sigma \mathrm{SP}_2)$$

does not hold in general. Let σ be a signature morphism from Σ to Σ' and let $m \models^{\mathcal{I}} D_\sigma \mathrm{SP}_1 \vee \mathrm{SP}_2$, thus either there exists a Σ'-structure m', which is a model of SP_1 and satisfies $m'|_\sigma = m$ or m is a model of SP_2. Assume that m is a model of SP_2. Then, in general, m cannot be extended to a Σ'-structure m' with $m'|_\sigma = m$. For example, in $\mathcal{LSL}$ this is the case if σ is not injective on sorts and function symbols.

In $\mathcal{R}_{\mathcal{I}}$ this happens if σ identifies parts of the shape of a relation. Consider the relation

$$R = D_f\text{<}[c : N]\text{>} \vee I_{\{c'=0\}}\text{<}[c : N] \times [c : N]\text{>},$$

where f is a morphism from $<[c : N] \times [c : N]>$ to $<[c : N]>$ identifying both states. Then $(m_1, m_2) \in R$ if $m_1 = m_2$ or $m_2(c) = 0$. However

$$S = D_f\text{<}[c : N]\text{>} \vee T_f I_{\{c'=0\}}\text{<}[c : N] \times [c : N]\text{>},$$

is the same as $D_f<[c : N]>$ because $\Theta \vee \mathrm{SP}$ has the same models as Θ (cf. the equalities on page 161). Thus S and R do not describe the same relation.

Another problem with computing the normal-form is to find a set of formulas Φ that expresses the union of two model classes defined by sets of formulas Φ_1 and Φ_2 in an arbitrary institution. That is, Φ has the property

$$\mathsf{Mod}^{\mathcal{A}}([\![I_\Phi \mathrm{SP}]\!]) = \mathsf{Mod}^{\mathcal{A}}([\![I_{\Phi_1} \mathrm{SP} \vee I_{\Phi_2} \mathrm{SP}]\!]).$$

In some institutions, like the institution of first-order logic, the disjunction of two (finite) sets can be expressed as one formula.

Definition 8.1 *Given two sets of Σ-formulas Φ_1 and Φ_2. In the case, where a Σ-formula φ in an institution $\mathcal{I}$ exists such that for each Σ-structure m we have $m \models^{\mathcal{I}} \varphi$ if and only if $m \models \Phi_1$ or $m \models \Phi_2$, we call φ the disjunction of Φ_1 and Φ_2 and write $\varphi =$ '$\Phi_1 \vee \Phi_2$'.*

Theorem 8.2 *Given an institution $\mathcal{I}$ and a signature Σ in $\mathrm{SIGN}_{\mathcal{I}}$. If '$\Phi_1 \vee \Phi_2$' exists for sets of Σ-formulas Φ_1 and Φ_2 then for a specification expression* SP *in $\mathtt{SL}_{\mathcal{I}}$ with signature Σ we have*

$$I_{\Phi_1}\mathrm{SP} \vee I_{\Phi_2}\mathrm{SP} = I_{\{`\Phi_1 \vee \Phi_2'\}}\mathrm{SP}.$$

PROOF. ($\subseteq$) Assume $m \models^{\mathcal{I}} I_{\Phi_1}\textsc{Sp}$, that is, $m \models^{\mathcal{I}} \Phi_1$ and $m \models^{\mathcal{I}} \textsc{Sp}$. Because of the definition of '$\Phi_1 \vee \Phi_2$', $m \models^{\mathcal{I}} \Phi_1$ implies $m \models^{\mathcal{I}}$ '$\Phi_1 \vee \Phi_2$' and therefore $m \models^{\mathcal{I}} I_{`\Phi_1 \vee \Phi_2'}\textsc{Sp}$. A similar argument holds if we assume $m \models^{\mathcal{I}} I_{\Phi_2}\textsc{Sp}$.

($\supseteq$) If $m \models^{\mathcal{I}} I_{`\Phi_1 \vee \Phi_2'}\textsc{Sp}$ then $m \models^{\mathcal{I}}$ '$\Phi_1 \vee \Phi_2$' and $m \models^{\mathcal{I}} \textsc{Sp}$. Because of the definition of '$\Phi_1 \vee \Phi_2$' we have $m \models^{\mathcal{I}} \Phi_1$ or $m \models \Phi_2$. Assume now that $m \models^{\mathcal{I}} \Phi_1$ then $m \models^{\mathcal{I}} I_{\Phi_1}\textsc{Sp} \vee I_{\Phi_2}\textsc{Sp}$ and similar if $m \models^{\mathcal{I}} \Phi_2$. □

In first-order logic the disjunction of two sets of formulas

$$\Phi_1 = \{\forall X\ F_1, \forall Y\ F_2\} \text{ and } \Phi_2 = \{\forall Z\ F_3\}$$

is

$$`\Phi_1 \vee \Phi_2' = (\forall X\ F_1 \wedge \forall Y\ F_2) \vee \forall Z\ F_3.$$

In $\mathcal{LSL}$ the situation is more complex because formulas in $\mathcal{LSL}$ can only be equations or constraints but not arbitrary first-order formulas, in particular, they cannot contain quantified sub-formulas. However, since each signature Σ in $\textsc{Sign}_{\mathcal{LSL}}$ contains the sort `Bool` with its operations and the equality symbol, and further each structure in $\mathsf{Str}_{\mathcal{LSL}}(\Sigma)$ is required to satisfy the property of booleans, we get the following theorems:

LEMMA 8.3 *Given a finite set of Σ-formulas $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ such that each φ_i has the form $\forall X\ s_i = t_i$. Define '$\wedge\Phi$' as the equation $\forall X\ (s_1 \equiv t_1 \wedge \cdots \wedge s_n \equiv t_n) = \mathbf{true}$ and let A be a Σ-algebra in $\mathsf{Str}_{\mathcal{LSL}}(\Sigma)$ then*

$$(\forall \varphi \in \Phi\ A \models \varphi) \text{ if and only if } A \models `\wedge\Phi'.$$

PROOF.

$$\begin{array}{llll} A \models `\wedge\Phi' & \text{iff} & A \models \forall X\ (s_1 \equiv t_1 \wedge \cdots \wedge s_n \equiv t_n) = \mathbf{true} & \mid \text{Def. } `\wedge\Phi' \\ & \text{iff} & \forall 1 \le i \le n\ A \models \forall X\ (s_i \equiv t_i) = \mathbf{true} & \mid \text{Lemma 3.32} \\ & \text{iff} & \forall 1 \le i \le n\ A \models \forall X\ s_i = t_i & \mid \text{Lemma 3.35} \end{array}$$

□

Note that it is important that each equation $\forall X\ s_i = t_i$ in Φ has the same set of variables X. An alternative could have been to allow each equation its own set of variables X_i and to define X in '$\wedge\Phi$' as the union of all X_i for $1 \le i \le n$. However, this does not work in the presence of empty carrier-sets as the following example shows. Let Σ be the signature with two sorts S_1 and S_2 and with two constants c_1 and c_2 of sort S_2. Let Φ be the set consisting of the equations $\forall x_1 : S_1\ x_1 = x_1$ and $c_1 = c_2$. Let A be a Σ-algebra such that $A(S_1) = \{\}$ and $A(c_1) \neq A(c_2)$. Then we have $A \models \forall x_1 : S_1\ x_1 = x_1 \wedge c_1 = c_2$ because there cannot be a variable assignment ρ from X to A because that would imply the existence of a function from a non-empty set, $\{x_1\}$, to the empty set. On the other hand, we have $A \models \forall x_1 : S_1\ x_1 = x_1$ but also $A \not\models c_1 = c_2$. However, if we change Φ to be the set containing the equations $\forall x_1 : S_1\ x_1 = x_1$ and $\forall x_1 : S_1\ c_1 = c_2$ then we have $A \models \forall x_1 : S_1\ c_1 = c_2$.

Theorem 8.4 *Given finite sets of Σ-formulas Φ_1 and Φ_2 such that*

1. *Φ_1 and Φ_2 do not contain constraints and*
2. *Φ_i has the form $\{\forall X^i\ s^i_1 = t^i_1, \ldots, \forall X^i\ s^i_{n_i} = t^i_{n_i}\}$ for $i \in \{1, 2\}$.*

Let X be the co-product of X^1 and X^2 with injections $\iota_1 : X^1 \Rightarrow X$ and $\iota_2 : X^2 \Rightarrow X$ and let '$\wedge\Phi_1$' be $\forall X^1\ c_1 = \mathbf{true}$ and '$\wedge\Phi_2$' be $\forall X^2\ c_2 = \mathbf{true}$.

Then the disjunction '$\Phi_1 \vee \Phi_2$' of Φ_1 and Φ_2 is given by

$$\forall X\ \iota_1(c_1) \vee \iota_2(c_2) = \mathbf{true}.$$

Proof. We have to show that for each Σ-algebra A in $\mathsf{Str}_{\mathcal{LSL}}(\Sigma)$ we have

$$A \models \Phi_1 \text{ or } A \models \Phi_2 \text{ if and only if } A \models \text{`}\Phi_1 \vee \Phi_2\text{'}.$$

$$\begin{array}{rll l}
A \models \text{`}\Phi_1 \vee \Phi_2\text{'} & \text{iff} & A \models \forall X\ \iota_1(c_1) \vee \iota_2(c_2) = \mathbf{true} & \mid \text{Def. `}\Phi_1 \vee \Phi_2\text{'} \\
& \text{iff} & A \models \forall X^1\ c_1 = \mathbf{true} \text{ or } A \models \forall X^2\ c_2 = \mathbf{true} & \mid \text{Lemma 3.33} \\
& \text{iff} & A \models \text{`}\wedge\Phi_1\text{'} \text{ or } A \models \text{`}\wedge\Phi_2\text{'} & \\
& \text{iff} & A \models \Phi_1 \text{ or } A \models \Phi_2 & \mid \text{Lemma 8.3}
\end{array}$$

□

If either Φ_1 or Φ_2 contains constraints then the disjunction '$\Phi_1 \vee \Phi_2$' is not defined since a constraint cannot be used as a sub formula in $\mathcal{LSL}$.

The Inference Rules

To the inference rules of Section 4.3 we add the following rules:

$$\text{R6}\quad \frac{\textsc{Sp} \vdash \textsc{Sp}_1}{\textsc{Sp} \vdash \textsc{Sp}_1 \vee \textsc{Sp}_2} \qquad \text{R6'}\quad \frac{\textsc{Sp} \vdash \textsc{Sp}_2}{\textsc{Sp} \vdash \textsc{Sp}_1 \vee \textsc{Sp}_2},$$

$$\text{L9}\quad \frac{\textsc{Sp}_1 \vdash \textsc{Sp} \quad \textsc{Sp}_2 \vdash \textsc{Sp}}{\textsc{Sp}_1 \vee \textsc{Sp}_2 \vdash \textsc{Sp}}$$

and the rules for monotonicity of $\vee$:

$$\text{M4}\quad \frac{\textsc{Sp} \vdash \textsc{Sp}'}{\textsc{Sp} \vee \textsc{Sp}'' \vdash \textsc{Sp}' \vee \textsc{Sp}''} \qquad \text{M4'}\quad \frac{\textsc{Sp} \vdash \textsc{Sp}'}{\textsc{Sp}'' \vee \textsc{Sp} \vdash \textsc{Sp}'' \vee \textsc{Sp}'}.$$

Disjunction is associative and commutative and the following equalities hold:

$$\begin{array}{rr}
\textsc{Sp} \vee \Sigma = \Sigma & D_\sigma(\textsc{Sp}_1 \vee \textsc{Sp}_2) = D_\sigma\textsc{Sp}_1 \vee D_\sigma\textsc{Sp}_2 \\
\textsc{Sp} \vee \textsc{Sp} = \textsc{Sp} & I_\Phi(\textsc{Sp}_1 \vee \textsc{Sp}_2) = I_\Phi\textsc{Sp}_1 \vee I_\Phi\textsc{Sp}_2 \\
T_\sigma(\textsc{Sp}_1 \vee \textsc{Sp}_2) = T_\sigma\textsc{Sp}_1 \vee T_\sigma\textsc{Sp}_2 &
\end{array}$$

and

$$I_{\Phi_1}\textsc{Sp} \vee I_{\Phi_2}\textsc{Sp} = I_{\text{`}\Phi_1 \vee \Phi_2\text{'}}\textsc{Sp}$$

if '$\Phi_1 \vee \Phi_2$' exists.

8.3 Adding Disjunction to $\mathtt{RSL}_{\mathcal{I}}$

The language $\mathtt{RSL}_{\mathcal{I}}$ is extended by disjunction as follows:

$$R ::= \ \ldots \mid R \vee R.$$

Then $R \vee S$ is well-formed if R and S are well-formed and have the same type. The type of $R \vee S$ is the type of R. The semantics of disjunction is defined by reduction to expressions in $\mathtt{SL}_{\mathcal{R}_{\mathcal{I}}}$:

$$[\![R \vee S]\!]_{\mathtt{SL}_{\mathcal{R}_{\mathcal{I}}}} = [\![R]\!]_{\mathtt{SL}_{\mathcal{R}_{\mathcal{I}}}} \vee [\![S]\!]_{\mathtt{SL}_{\mathcal{R}_{\mathcal{I}}}}.$$

We also extend the translation of $\mathtt{RSL}$ expressions to $\mathtt{SL}_{\mathcal{I}}$ expressions (cf. Chapter 8) to disjunction:

$$\mathtt{tr}(R \vee S) = \mathtt{tr}(R) \vee \mathtt{tr}(S).$$

Then we have to show that $\mathsf{G}^{\mathcal{A}}([\![R \vee S]\!]_{\mathcal{R}}) = [\![\mathtt{tr}(R) \vee \mathtt{tr}(S)]\!]$. Let $[\![R]\!]_{\mathcal{R}} = (\Theta, M_R)$ and $[\![S]\!]_{\mathcal{R}} = (\Theta, M_S)$, where $\Theta = (\mathrm{D}, \Gamma)$, and let

$$\mathsf{G}^{\mathcal{A}}([\![R]\!]_{\mathcal{R}}) = (\coprod \Gamma; \mathsf{Sig}^{\mathcal{A}}, \overline{M}_R) \text{ and } \mathsf{G}^{\mathcal{A}}([\![S]\!]_{\mathcal{R}}) = (\coprod \Gamma; \mathsf{Sig}^{\mathcal{A}}, \overline{M}_S).$$

The induction hypothesis gives us further $\mathsf{G}^{\mathcal{A}}([\![R]\!]_{\mathcal{R}}) = [\![\mathtt{tr}(R)]\!]$ and $\mathsf{G}^{\mathcal{A}}([\![S]\!]_{\mathcal{R}}) = [\![\mathtt{tr}(S)]\!]$. Then we have:

$$\begin{aligned}
\mathsf{G}^{\mathcal{A}}([\![R \vee S]\!]_{\mathcal{R}}) &= \mathsf{G}^{\mathcal{A}}((\Theta, M_R \cup M_S)) && \mid \text{Def. } \vee \\
&= (\coprod \Gamma; \mathsf{Sig}^{\mathcal{A}}, \overline{M}_R \cup \overline{M}_S) && \mid \text{Def. } \mathsf{G}^{\mathcal{A}} \\
&= (\coprod \Gamma; \mathsf{Sig}^{\mathcal{A}}, \mathsf{Mod}^{\mathcal{A}}(\mathsf{G}^{\mathcal{A}}([\![R]\!]_{\mathcal{R}})) \cup \mathsf{Mod}^{\mathcal{A}}(\mathsf{G}^{\mathcal{A}}([\![S]\!]_{\mathcal{R}}))) \\
&= [\![\mathtt{tr}(R) \vee \mathtt{tr}(S)]\!]
\end{aligned}$$

The Birthday-Book Example

As an example, the robust version of the $\mathtt{AddBirthdayBook}_{\mathcal{BC}}$ relation from the beginning of this chapter is translated to $\mathtt{SL}_{\mathcal{LSL}}$.

$$\begin{aligned}
&\mathtt{tr}(\mathtt{RAddBirthday}_{\mathcal{BC}}) \\
&\qquad = (T_{\overline{f}}\mathtt{tr}(\mathtt{AddBirthday}_{\mathcal{BC}}) + T_{\overline{g}}\mathtt{tr}(\mathtt{Success}_{\mathcal{BC}})) \vee T_{\overline{h}}\mathtt{tr}(\mathtt{AlreadyKnown}_{\mathcal{BC}})
\end{aligned}$$

And we have $\mathtt{tr}(\mathtt{AddBirthday}_{\mathcal{BC}})$:

```
includes colim Θ_AB
asserts
   not(n ∈ known)
   birthday' = update(birthday,n,d)
```

And colim $\Theta_{\mathtt{AB}}$ is

```
includes
   BirthdayBook_BC,
   BirthdayBook_BC[known’ for known, birthday’ for birthday]
introduces
   n : Name
   d : Date
```

Then $\mathtt{tr}(\mathtt{Success}_{\mathcal{BC}})$ yields

```
includes colim Θ_R
asserts
   result = ok
```

with colim Θ_{R} being

```
includes Env
introduces
   result : Result
```

And $\mathtt{tr}(\mathtt{AlreadyKnown}_{\mathcal{BC}})$ is

```
includes colim <[n:Name]×BirthdayBook_BC×BirthdayBook_BC×[result: Report]>
asserts
   known’ = known
   birthday’ = birthday
   n ∈ known
   result = alreadyKnown
```

and colim $<[n : \mathtt{Name}] \times \mathtt{BirthdayBook}_{\mathcal{BC}} \times \mathtt{BirthdayBook}_{\mathcal{BC}} \times [\mathtt{result} : \mathtt{Report}]>$ is

```
includes
   BirthdayBook_BC,
   BirthdayBook_BC[known’ for known, birthday’ for birthday]
introduces
   result :  Result
   n :  Name
```

Then we get as $\mathtt{tr}(\mathtt{RAddBirthday}_{\mathcal{BC}})$:

```
includes colim Θ_RAB
asserts
   not(n ∈ known)
   birthday’ = update(birthday,n,d)
   result = ok

 ∨

includes colim Θ_RAB
asserts
   known’ = known
   birthday’ = birthday
   n ∈ known
   result = alreadyKnown
```

with colim $\Theta_{\mathtt{RAB}}$ being:

```
includes
   BirthdayBook_𝓑𝓒
   BirthdayBook_𝓑𝓒[known’ for known, birthday’ for birthday]
introduces
   n :  Name
   d : Date
   result : Result
```

If we use $I_{\Phi_1}(\mathsf{colim}\ \Theta_{\mathtt{RAB}}) \vee I_{\Phi_2}(\mathsf{colim}\ \Theta_{\mathtt{RAB}}) = I_{`\Phi_1 \vee \Phi_2'}(\mathsf{colim}\ \Theta_{\mathtt{RAB}})$, we get $\mathtt{tr}(\mathtt{RAddBirthday})$ equals

```
includes colim Θ_RAB
asserts
   (not(n ∈ known)
      ∧ birthday’ = update(birthday,n,d)
      ∧ result = ok)
∨
   (n ∈ known
      ∧ known’ = known
      ∧ birthday’ = birthday
      ∧ result = alreadyKnown)
```

9 Z Specifications

This chapter compares in more detail the approach presented in this thesis for the specification of sequential system with the approach used by the model-oriented specification language Z (cf. The Z Base Standard [12]). To this end we first define the institution $\mathcal{SET}$, which formalizes the logical system underlying Z. We show that the category of signatures $\textsc{Sign}_{\mathcal{SET}}$ of $\mathcal{SET}$ is cocomplete and that the covariant structure functor $\mathsf{Str}_{\mathcal{SET}} : \textsc{Sign}^{op}_{\mathcal{SET}} \to \textsc{Cat}$ preserves colimits. Thus $\mathcal{SET}$ can be used as a base institution to construct $\mathcal{R}_{\mathcal{SET}}$.

As already noted by Spivey [56], schemata are closely related to signatures and schema-types to abstract datatypes of $\mathcal{SET}$. We shall investigate this relationship further and show a correspondence between the operations on abstract datatypes given in Section 4.2 and the schema-expressions.

This correspondence and the relationship between $\textsc{Adt}_{\mathcal{R}_{\mathcal{SET}}}$ and $\textsc{Adt}_{\mathcal{SET}}$, elaborated in Section 5.6, allows to write Z-style specifications of sequential systems using any logical system, for which there is an exact institution with a (finitely) cocomplete category of signatures. In particular, we give an example specification using $\mathcal{LSL}$.

9.1 The Institution $\mathcal{SET}$

Signatures

A signature Σ in $\textsc{Sign}_{\mathcal{SET}}$ consists of a set of names for *given-sets* G and a set of identifiers O. Each identifier id in O is associated with a type $\tau(id)$ built from the names of given-sets and the constructors: cartesian product, power-set and schema-type.

Note that $\mathcal{SET}$ has no type constructors for function types. Instead, a function from T_1 to T_2 is identified with its graph and is of type $\mathcal{P}(T_1 \times T_2)$. This allows functions to be treated as sets and admits higher-order functions, as functions may take as argument the graph of a function and also return the graph of a function.

Definition 9.1 (Signatures) *A signature Σ in $\textsc{Sign}_{\mathcal{SET}}$ is a tuple (G, O, τ) where G and O are finite subsets of a recursive enumerable set of names F. The function τ maps names in O to types in $\mathcal{T}(G)$ where $\mathcal{T}(G)$ is inductively defined by:*

- $G \subseteq \mathcal{T}(G)$
- *(product)* $T_1 \times \cdots \times T_n \in \mathcal{T}(G)$ *for* $T_i \in \mathcal{T}(G)$, $1 \leq i \leq n$
- *(power-set)* $\mathcal{P}(T) \in \mathcal{T}(G)$ *for* $T \in \mathcal{T}(G)$

- *(schema-type)* $<x_1 : T_1, \ldots, x_n : T_n> \in \mathcal{T}(G)$ *for* $T_i \in \mathcal{T}(G)$ *and* $x_i \in F$ *and* $x_i \neq x_j$ *for* $1 \leq i, j \leq n$.

G and O need not be disjoint. If a name is used in an expression that occurs both in G and in O then it is interpreted as an identifier from O and not as a given-set from G; this means that the identifiers in O *hide* the identifiers in G.

The function $\mathcal{T}$, mapping a given-set G to $\mathcal{T}(G)$, is extended to a functor from Set to Set by the canonical extension of a function $f : G \to G'$ to a function $\mathcal{T}(f) : T(G) \to T(G')$ given by the definition of $\mathcal{T}(G)$, that is:

- $\mathcal{T}(f)(g) = g$ for $g \in G$,
- $\mathcal{T}(f)(T_1 \times \ldots \times T_n) = \mathcal{T}(f)(T_1) \times \ldots \times \mathcal{T}(f)(T_n)$ for $T_1, \ldots, T_n \in \mathcal{T}(G)$,
- $\mathcal{T}(f)(\mathcal{P}(T)) = \mathcal{P}(\mathcal{T}(f)(T))$ for $T \in \mathcal{T}(G)$,
- $\mathcal{T}(f)(<x_1 : T_1, \ldots, x_n : T_n>) = <x_1 : \mathcal{T}(f)(T_1), \ldots, x_n : \mathcal{T}(f)(T_n)>$ for $T_1, \ldots, T_n \in \mathcal{T}(G)$.

A signature morphism $\sigma : (G, O, \tau) \to (G', O', \tau')$ is a pair of maps between the given-sets and the set of identifiers such that if *id* is the name of a given-set from $G \setminus O$ then it has to be mapped to $G' \setminus O'$. This requirement is important for the proof of Lemma 9.14, which in turn is needed in the proof of the satisfaction condition (cf. Theorem 9.15).

Definition 9.2 (Signature-Morphisms) *A signature morphism* σ *from a signature* (G, O, τ) *to signature* (G', O', τ') *is a pair of functions* $\sigma_G : G \to G'$ *and* $\sigma_O : O \to O'$ *such that* σ_G *and* σ_O *are compatible with* τ *and* τ'*, that is* $\tau; T(\sigma_G) = \sigma_O; \tau'$*. We require that if id is in* $G \setminus O$ *then* $\sigma_O(id)$ *is in* $G \setminus O$*.*

Definition 9.3 (Sign$_{\mathcal{SET}}$) *The category* Sign$_{\mathcal{SET}}$ *has as objects signatures* $\Sigma = (G, O, \tau)$ *and as morphisms signature morphisms* $\sigma = (\sigma_G, \sigma_O)$ *as defined above.*

The colimit of a functor $F : J \to$ Sign$_{\mathcal{SET}}$ is given by the colimits of the set of given-set names and the set of identifiers. Note that Sign$_{\mathcal{SET}}$ is only finitely cocomplete because we have assumed that the set of given-set names and the set of identifiers are finite.

Theorem 9.4 *The category* Sign$_{\mathcal{SET}}$ *is finitely cocomplete.*

Proof L. et F be a functor from J to Sign$_{\mathcal{SET}}$ and let $F(i) = \Sigma_i = (G_i, O_i, \tau_i)$ and $F(f) = \sigma_f = (\sigma_G^f, \sigma_O^f)$. Define $F_G : J \to$ Set by $F_G(i) = G_i$ and $F_G(f) = \sigma_G^f$, and define $F_O : J \to$ Set by $F_O(i) = O_i$ and $F_O(f) = \sigma_O^f$. Then $\coprod F = (G, O, \tau)$, where G is the colimit of F_G, O the colimit of F_O and τ is given as the unique function satisfying $\iota^{F_O}; \tau = \tau_i; T(\iota^{F_G})$ for all $i \in J$ given by the colimit property of F_O.

$$\begin{tikzcd}
O \arrow[r, "\tau"] & T(G) \\
O_i \arrow[u, "\iota_i^{F_O}"] \arrow[r, "\tau_i"'] & T(G_i) \arrow[u, "T(\iota_i^{F_G})"']
\end{tikzcd}$$

The co-cone morphisms $\iota_i^F = ((\iota_G^F)_i, (\iota_O^F)_i)$ from Σ_i to $\coprod F$ are given by the co-cone morphisms $\iota_i^{F_G}$ from G_i to G and $\iota_i^{F_O}$ from O_i to O. Note that, by the above diagram, $(\iota_G^F)_i$ and $(\iota_O^F)_i$ are compatible and therefore ι_i^F is a signature morphism.

Given a natural transformation $\mu : F \Rightarrow \Delta(G', O', \tau')$ then the unique function $\bar{\mu} = (\bar{\mu}_G, \bar{\mu}_O)$ from $\coprod F$ to (G', O', τ') with the property $\iota_i^F; \bar{\mu} = \mu_i$ for all $i \in J$ is given by the unique functions $\bar{\mu}_G$ and $\bar{\mu}_O$ given by the colimit property of G and O. For $\bar{\mu}$ to be a signature morphisms we have to show that $\bar{\mu}_O; \tau' = \tau; T(\bar{\mu}_G)$. We show this by using the colimit property of O in SET for the following J-indexed families of diagrams

$$\begin{tikzcd}
O \arrow[r, "\tau"] & T(G) \arrow[r, "T(\bar{\mu}_G)"] & T(G') \\
O_i \arrow[u, "(\iota_O^F)_i"] \arrow[r, "\tau_i"'] & T(G_i) \arrow[u, "T((\iota_G^F)_i)"'] \arrow[ur, "T((\mu_G)_i)"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
O \arrow[r, "\bar{\mu}_O"] & O' \arrow[r, "\tau'"] & T(G') \\
& O_i \arrow[ul, "(\iota_O^F)_i"] \arrow[u, "(\mu_O)_i"'] \arrow[r, "\tau_i"'] & T(G_i) \arrow[u, "T((\mu_G)_i)"']
\end{tikzcd}$$

In the first family of diagrams the diagrams commute because the right triangle commutes since $\bar{\mu}_G$ is defined as the unique function that satisfies $(\iota_G^F)_i; \bar{\mu}_G = (\mu_G)_i$, and the left rectangle commutes because ι^F is a signature morphism. The diagrams in the second family commute because the μ_i are signature morphisms (right rectangle) and $\bar{\mu}_O$ is defined as the unique function satisfying $(\iota_O^F)_i; \bar{\mu}_O = (\mu_O)_i$ (left triangle). Then $\tau; T(\bar{\mu}_G) = \bar{\mu}_O; \tau'$ because of the uniqueness of the function from O to $T(G')$ making the outer quadrangles of the diagrams commute. □

The initial object in $\text{SIGN}_{\mathcal{SET}}$ is $(\{\}, \{\}, \tau)$ where τ is the unique function from the empty set to $T(\{\}) = \{\}$.

Structures

Given a signature $\Sigma = (G, O, \tau)$ a Σ-structure A interprets each given-set in G by a set from SET and each identifier *id* in O by a value of the set corresponding to the type of *id*.

DEFINITION 9.5 (Σ-STRUCTURES) *For a given signature $\Sigma = (G, O, \tau)$ the category $\mathsf{Str}_{\mathcal{SET}}(\Sigma)$ of Σ-structures has as objects pairs (A_G, A_O), where A_G is a functor from the set G, viewed as a discrete category, to* SET *and A_O is the set $\{(o_1, v_1), \ldots, (o_n, v_n)\}$ for $O = \{o_1, \ldots, o_n\}$ and $v_i \in \bar{A}_G(\tau(o_i))$, where the functor $\bar{A}_G : \mathcal{T}(G) \to$ SET is given by:*

- $\bar{A}_G(T) = A_G(T)$ *for $T = g$ and $g \in G$*
- $\bar{A}_G(T_1 \times \cdots \times T_n) = (\bar{A}_G(T_1) \times \cdots \times \bar{A}_G(T_n))$ *for $T_1 \times \cdots \times T_n \in \mathcal{T}(G)$*
- $\bar{A}_G(\mathcal{P}(T)) = 2^{\bar{A}_G(T)}$ *for $\mathcal{P}(T) \in \mathcal{T}(G)$*
- $\bar{A}_G(<x_1 : T_1, \ldots, x_n : T_n>)$
$$= \{\{(x_1, v_1), \ldots, (x_n, v_n)\} \mid v_i \in \bar{A}_G(T_i),\ i \in 1 \ldots n\}$$
for $<x_1 : T_1, \ldots, x_n : T_n> \in \mathcal{T}(G)$.

A morphism h from a Σ-structure A to a Σ-structure B is a family of functions between the interpretations of the given-sets which is compatible with the interpretations of the identifiers in O.

Definition 9.6 (Σ-homomorphism) *A Σ-homomorphism h from a structure $A = (A_G, A_O)$ to a structure $B = (B_G, B_O)$ is a natural transformation $h : A_G \Rightarrow B_G$ for which $\bar{h}_{\tau(o)}(v_A) = v_B$ for all $o \in O$, $(o, v_A) \in A_O$ and $(o, v_B) \in B_O$ holds, where $\bar{h}$ is the extension of $h : A_G \Rightarrow B_G$ to $\bar{h} : \bar{A}_G \Rightarrow \bar{B}_G$ given by:*

- $\bar{h}_T(v) = h_T(v)$ *for* $T \in G$
- $\bar{h}_T((v_1, \ldots, v_n)) = (\bar{h}_{T_1}(v_1), \ldots, \bar{h}_{T_n}(v_n))$ *for* $T = T_1 \times \cdots \times T_n \in \mathcal{T}(G)$
- $\bar{h}_{\mathcal{P}(T)}(S) = \{\bar{h}_T(v) \mid v \in S\}$ *for* $\mathcal{P}(T) \in \mathcal{T}(G)$
- $\bar{h}_T(\{(x_1, v_1), \ldots, (x_n, v_n)\}) = \{(x_1, \bar{h}_{T_1}(v_1)), \ldots, (x_n, \bar{h}_{T_n}(v_n))\}$ *for* $T = <x_1 : T_1, \ldots, x_n : T_n> \in \mathcal{T}(G)$

Definition 9.7 (σ-reduct) *Given a signature morphism σ from $\Sigma = (G, O, \tau)$ to $\Sigma' = (G', O', \tau')$ in $\mathrm{Sign}_{\mathcal{SET}}$ and a structure $A = (A_G, A_O)$ in $\mathsf{Str}_{\mathcal{SET}}(\Sigma')$ the σ-reduct of A, written $A|_\sigma$, is the structure $B = (B_G, B_O)$ given by:*

- $B_G = \sigma_G; A_G$
- $B_O = \{(o, v) \mid (\sigma_O(o), v) \in A_O,\ o \in O\}$

For a Σ'-homomorphism $h : A \to B$ the σ-reduct is defined as $h|_\sigma = \sigma_G; h$.

Definition 9.8 ($\mathsf{Str}_{\mathcal{SET}}$) *The contravariant functor $\mathsf{Str}_{\mathcal{SET}}$ from $\mathrm{Sign}_{\mathcal{SET}}$ to Cat assigns to each signature Σ the category having as objects Σ-structures and as morphisms Σ-homomorphisms, and to each $\mathrm{Sign}_{\mathcal{SET}}$-morphism σ from Σ to Σ' a functor from the category $\mathsf{Str}_{\mathcal{SET}}(\Sigma')$ to the category $\mathsf{Str}_{\mathcal{SET}}(\Sigma)$ mapping a Σ-structure A and a Σ-homomorphism to their σ-reduct.*

Theorem 9.9 *The functor $\mathsf{Str}_{\mathcal{SET}}$ preserves finite colimits.*

The functor $\mathsf{Str}_{\mathcal{SET}}$ maps the initial object $(\{\}, \{\}, \tau)$ in $\mathrm{Sign}_{\mathcal{SET}}$ to the category with one element $(A_G, \{\})$ where A_G is the unique functor from the category with no objects to the category Set. If $\mathsf{Str}_{\mathcal{SET}}$ in addition preserves pushouts, then $\mathsf{Str}_{\mathcal{SET}}$ preserves finite colimits. The functor $\mathsf{Str}_{\mathcal{SET}}$ preserves pushouts if the institution $\mathcal{SET}$ has amalgamation.

Theorem 9.10 *The institution $\mathcal{SET}$ has amalgamation.*

Proof L. et F be a pushout diagram with $F(i) = \Sigma_i = (G_i, O_i)$ for $i \in \{0, 1, 2\}$, $F(f) = \sigma_f = (\sigma_G^f, \sigma_O^f)$ for $f \in \mathrm{V}$, and let $\Sigma_{cl} = (G_{cl}, O_{cl})$ be the pushout of F with co-limit morphisms $\iota_i; \Sigma_i \to \Sigma_{cl}$ for $i \in \{0, 1, 2\}$. Given Σ_i-structures $A_i = (A_G^i, A_O^i)$ such that $A_1|_{\sigma_f} = A_0 =$

$A_2|_{\sigma_g}$, where $f : 0 \to 1$ and $g : 0 \to 2$. Then we define the amalgamated sum $A = (A_G, A_O)$ of A_1 and A_2 with respect to A_0 by

$$A_G(g) = A^i_G(g_i) \text{ if } (\iota_G)_i(g_i) = g \text{ for some } i \in \{0, 1, 2\} \text{ and } g_i \in G_i$$

and

$$A_O = \{(o, v_i) \mid \exists i \in \{0, 1, 2\}\ (o_i, v_i) \in A^i_O,\ (\iota_O)_i(o_i) = o\},$$

For A_G to be well-defined consider i, j with $(\iota_G)_i(g_i) = g = (\iota_G)_j(g_j)$. Then we have to show that $A^i_G(g_i) = A^j_G(g_j)$. Assume that there exists $f : i \to j$ in V then we have $\sigma^f_O(g_i) = g_j$ and $A_j|_{\sigma_f} = A_i$, which by the definition of σ-reduct implies $\sigma^f_G; A^j_G = A^i_G$, and therefore $A^j_G(g_j) = A^j_G(\sigma^f_G(g_i)) = A^i_G(g_i)$.

A similar argument shows that if $(o, v_i) \in A_O$ and $(o, v_j) \in A_O$ then $v_i = v_j$. □

Formulas

The Σ-formulas are first-order formulas over expressions denoting sets and elements in sets. Expressions can be tested for equality and membership. An important category of expressions, called schema-expressions, denote sets of elements of schema-type.

Expressions Expressions describe either given-sets or values in sets constructed from given-sets; the values can, of course, be themselves sets. These expressions are built from names in G and O, formation of tuples, selection of an element in a tuple, formation of an element of schema-type, selection of a component of an element of schema-type and function application.

$$E ::= id \mid (E, \dots, E) \mid E.i \mid {<}x_1 := E, \dots, x_n := E{>} \mid E.x \mid E(E)$$

An identifier *id* in O denotes a value of the set $\bar{A}_G(\tau(id))$ and a given-set name g in G denotes a given-set $A_G(g)$, which is an element of the set $\bar{A}_G(\mathcal{P}(g)) = 2^{A_G(g)}$. The function application $E_1(E_2)$ is well-defined if E_1 is of type $\mathcal{P}(T_1 \times T_2)$ and E_2 is of type T_1. The result is of type T_2. If E_1 represents the graph of a total function then $E_1(E_2)$ yields the result of that function applied to E_2. However, if E_1 is the graph of a partial function or not a functional at all then an arbitrary value from the $\bar{A}_G(T_2)$ is chosen as the result for the situations where E_2 is not in the domain of that function or if several results are associated with E_2 in E_1.

The remaining expressions denote sets. These are set-extension and comprehension, powerset construction, cartesian product and schemas.

$$E ::= \dots \mid \{E, \dots, E\} \mid \{S \bullet E\} \mid \mathcal{P}(E) \mid E \times \dots \times E \mid S$$

Let V be a recursive enumerable set of variable names such that V and F are disjoint. Given a signature $\Sigma = (G, O, \tau)$ and a set of variables $X \subseteq V$ together with a function $\tau_X : X \to \mathcal{T}(G)$ then an *environment* ϵ is a pair $(\Sigma, (X, \tau_X))$. An expression E is well-formed with respect to ϵ if

- $E = id$ and $id \in X \cup O \cup G$. The type of E w.r.t. ϵ is

$$\tau^{\epsilon}(E) = \begin{cases} \tau_X(id) & \text{if } id \text{ is in } X, \\ \tau(id) & \text{if } id \text{ is in } O \text{ and} \\ \mathcal{P}(id) & \text{if } id \text{ is in } G \setminus O. \end{cases}$$

- $E = (E_1, \ldots, E_n)$ and each E_i is well-formed for all $1 \leq i \leq n$. Then $\tau^{\epsilon}(E) = \tau^{\epsilon}(E_1) \times \ldots \times \tau^{\epsilon}(E_n)$.
- $E = E'.i$, $\tau^{\epsilon}(E') = T_1 \times \ldots \times T_n$ and $1 \leq i \leq n$. The type of E is T_i.
- $E = {<}x_1 := E_1, \ldots, x_n := E_n{>}$, $x_i \in V$, $x_i \neq x_j$ and each E_i is well-formed. The type of E is ${<}x_1 : \tau^{\epsilon}(E_1), \ldots, x_n : \tau^{\epsilon}(E_n){>}$.
- $E = E'.x$, $\tau^{\epsilon}(E') = {<}x_1 : T_1, \ldots, x_n : T_n{>}$ and $x = x_i$ for some $1 \leq i \leq n$. The type of E is T_i.
- $E_1(E_2)$, $\tau^{\epsilon}(E_1) = \mathcal{P}(T_1 \times T_2)$ and $\tau^{\epsilon}(E_2) = T_1$. The type of E is T_2.
- $E = \{E_1, \ldots, E_n\}$, each E_i is well-formed and all E_i have the same type T for $1 \leq i \leq n$. The type of E is $\mathcal{P}(T)$.
- $E = \{S \bullet E'\}$, S is well-formed and has type $\mathcal{P}({<}x_1 : T_1, \ldots, x_n : T_n.{>}$ and E' is well-formed with respect to $\epsilon[{<}x_1 : T_1, \ldots, x_n : T_n{>}]$, where $\epsilon[{<}x_1 : T_1, \ldots, x_n : T_n{>}] = (\Sigma, (X', \tau'_X))$ is given by $X' = X \cup \{x_1, \ldots, x_n\}$ and

$$\tau'_X(id) = \left\{ \begin{array}{ll} T_i & \text{if } id = x_i \text{ for some } 1 \leq i \leq n \\ \tau_X(id) & \text{else} \end{array} \right)$$

  The type of E is $\mathcal{P}(\tau^{\Sigma'}(E'))$.
- $E = \mathcal{P}(E')$ and E' is well-formed. The type of E is $\mathcal{P}(\tau^{\epsilon}(E'))$.
- $E = E_1 \times \ldots \times E_n$ and each E_i is well-formed. The type of E is $\mathcal{P}(\tau^{\epsilon}(E_i) \times \ldots \times \tau^{\epsilon}(E_n))$.
- $E = S$ and S is well-formed schema-expression with respect to Σ, where well-formedness of schema-expressions are defined on page 172. The type of E is the type of S with respect to ϵ.

Let E be an expression well-formed with respect to $\epsilon = (\Sigma, (X, \tau_X))$ and let $A = (A_G, A_O)$ be a Σ-structure. The semantics of an expression E with respect to the **environment ?** $\beta = (A, A_X)$, where $A_X = \{(x_1, v_1) \ldots (x_n, v_n)\}$, $X = \{x_1, \ldots, x_n\}$ and $v_i \in \bar{A}_G(\tau_X(x_i))$ for all $1 \leq i \leq n$, is defined as follows:

- $[\![id]\!]^{\beta} = v$ if $(id, v) \in A_X$ and $id \in X$ or $(id, v) \in A_O$ and $id \in O$ or $[\![id]\!]^{\beta} = A_G(id)$ if id is in $G \setminus O$.
- $[\![(E_1, \ldots, E_n)]\!]^{\beta} = ([\![E_1]\!]^{\beta}, \ldots, [\![E_n]\!]^{\beta})$.
- $[\![E.i]\!]^{\beta} = v_i$ if $[\![E]\!]^{\beta} = (v_1, \ldots, v_n)$.

- $[\![<x_1 := E_1, \ldots, x_n := E_n>]\!]^\beta = \{(x_1, [\![E_1]\!]^\beta), \ldots, (x_n, [\![E_n]\!]^\beta)\}$.
- $[\![E.x]\!]^\beta = v_i$ if $[\![E]\!]^\beta = \{(x_1, v_1), \ldots, (x_n, v_n)\}$ and $x = x_i$.
- $[\![E_1(E_2)]\!]^\beta = v$ if v is unique with $([\![E_2]\!]^\beta, v)$ in $[\![E_1]\!]^\beta$. If another v' with $([\![E_2]\!]^\beta, v')$ in $[\![E_1]\!]^\beta$ exists or if none exists then v is an arbitrary element of $\bar{A}_G(T_2)$, where $\tau^\epsilon(E_1) = \mathcal{P}(T_1 \times T_2)$.
- $[\![\{E_1, \ldots, E_n\}]\!]^\beta = \{[\![E_1]\!]^\beta, \ldots, [\![E_n]\!]^\beta\}$.
- $[\![\{S \bullet E\}]\!]^\beta = \{[\![E]\!]^{\beta[v]} \mid v \in [\![S]\!]^\beta\}$.
For a value $v = \{(x_1, v_1), \ldots, (x_n, v_n)\}$ from $[\![S]\!]^\beta$, $\beta[v]$ is the environment $(A, A_X[v])$ such that
$$A_X[v] = A_X \setminus \{(id, v') \mid (id, v') \in A_X,\ \exists 1 \leq i \leq n : id = x_i\} \cup v$$
- $[\![\mathcal{P}(E)]\!]^\beta = 2^{[\![E]\!]^\beta}$.
- $[\![E_1 \times \ldots \times E_n]\!]^\beta = [\![E_1]\!]^\beta \times \ldots \times [\![E_n]\!]^\beta$.

Schema-expressions A schema denotes a set of elements of schema-type which have the form $\{(x_1, v_1), \ldots, (x_n, v_n)\}$ and are called *bindings*. A simple schema of the form $x_1 : E_1, \ldots, x_n : E_n$ defines the identifiers of a schema and a set of possible values for each identifier.

We require that the identifiers of a schema are not taken from the same set F as the given-set names in G and the identifier names of O. Thus we assume a recursive enumerable set V with $V \cap F = \{\}$ from which the identifiers of a schema are taken. The reason is that the identifier names of a schema can be bound by a quantifier and that we have to prevent that an identifier is renamed by a signature morphism to the name of a bound identifier. This would lead to problems with the satisfaction condition for $\mathcal{SET}$.

Given a schema S we can a define a new schema $S|P$ having as elements all the elements of S satisfying the predicate P. We can form the negation, disjunction, conjunction and implication of schema-expressions, which correspond to the complement, union and intersection of the sets denoted by the arguments. For the disjunction, conjunction and implication of schema-expressions the type of the arguments have to be compatible, that is, if two components have the same name, they have to have the same type. The type of the result has as components the union of the components of the arguments, with all duplicates removed. Adjustments of the type of schemas can be made by using hiding and renaming, where hiding hides some components of a schema-type and renaming renames some identifiers. An existentially quantified schema $\exists S_1.S_2$ denotes the set of all bindings of the identifiers of S_2 without the ones in S_1 such that there exists a binding in S_1 such that the union of the bindings is an element of S_2. An universally quantified schema $\forall S_1.S_2$ is an abbreviation for

$\neg\exists S_1.\neg S_2$.

$$\begin{array}{lll} S & ::= & x_1 : E, \ldots, x_n : E \mid (S|P) \mid \neg S \mid S \vee S \mid S \wedge S \mid S \Rightarrow S \\ & & \mid \forall S.S \mid \exists S.S \mid S \setminus [x_1, \ldots, x_n] \mid S[x_1/y_1, \ldots, x_n/y_n] \\ & & \mid S \ \mathit{Decor} \mid E \end{array}$$

A schema-expression S is well-formed with respect to an envrionment $\epsilon = (\Sigma, (X, \tau_X))$ with $\Sigma = (G, O, \tau)$, if

- $S = x_1 : E_1, \ldots, x_n : E_n$, $x_i \in V$ and E_i is well-formed and has type $\mathcal{P}(T_i)$ for each $1 \leq i \leq n$. The type of S is $\mathcal{P}(<x_1 : T_1, \ldots, x_n : T_n>)$.
- $S = S'|P$ and P is well-formed with respect to $\epsilon' = \epsilon[T]$, where T is the type of S' with respect to ϵ. The type of S is T.
- $S = \neg S'$ and S' is well-formed. The type of S is $\tau^\epsilon(S')$.
- $S = S_1 \ op \ S_2$, S_1 and S_2 have compatible types, and S_1 and S_2 are well-formed for each $op \in \{\vee, \wedge, \Rightarrow\}$. Two schema-types $<x_1 : T_1, \ldots, x_n : T_n>$ and $<x'_1 : T'_1, \ldots, x'_m : T'_m>$ are compatible if for all i, j such that $x_i = x'_j$ we have $T_i = T'_j$. The type of S has as components the union of the components of the type of S_1 and S_2 with the duplicates removed.
- $S = \exists S_1.S_2$, S_1 and S_2 are well-formed with respect to ϵ and their types are compatible. The type of S is the type of S_2 with all the identifiers removed which occur in S_1.
- $S = S' \setminus [x_1, \ldots, x_n]$ and S is well-formed. Note that it is not required that the x_i have to be identifiers of the type of S'. The type of S is the type of S' without the identifier x_i if x_i occurs in the type of S for all $1 \leq i \leq n$.
- $S = S'[x_1/y_1, \ldots, x_n/y_n]$ and S is well-formed. Note that it is not required that the x_i have to be identifiers of the type of S'. The type of S is the type of S' where x_i is replaced by y_i if x_i is an identifier of S'. Note that the function from the identifiers of the type of S' to the identifiers of the type of S defined by this replacement has to be injective.
- $S = S' \ \mathit{Decor}$ and S' is well-formed. *Decor* is a finite sequence of elements from $\{', !, ?\}$. The type of S is $\mathcal{P}(<\bar{x}_1 : T_1, \ldots, \bar{x}_n : T_n>)$ if S' is of type $\mathcal{P}(<x_1 : T_1, \ldots, x_n : T_n>)$ $\bar{x}_i$ is the decorated form of x_i, for example, if *Decor* is ! then $\bar{x}_i$ is $x_i!$.
- $S = E$ and E is well-formed with type $\mathcal{P}(<x_1 : T_1, \ldots, x_n : T_n>)$ The type of S is $\mathcal{P}(<x_1 : T_1, \ldots, x_n : T_n>)$

If a schema-expression S is well-defined with respect to ϵ, its semantics $[\![S]\!]^\beta$ with respect to a structure $A = (A_G, A_O)$ and an **environment** $\beta = (A, A_X)$ is defined as follows:

- $[\![x_1 : E_1, \ldots, x_n : E_n]\!]^\beta = \{\{(x_1, v_1), \ldots, (x_n, v_n)\} \mid v_i \in [\![E_i]\!]^\beta, \ 1 \leq i \leq n\}$.
- $[\![S|P]\!]^\beta = \{v \in [\![S]\!]^\beta \mid \beta[v] \models^{\mathcal{SET}} P\}$. The satisfaction relation $\models^{\mathcal{SET}}$ is defined on page 174.

- $[\![\neg S]\!]^\beta = \{v \mid v \notin [\![S]\!]^\beta\}$.
- $[\![S_1 \mathit{ op } S_2]\!]^\beta = \{v \in \bar{A}_G(T) \mid v = v_1 \cup v_2,\ v_1 \in [\![S_1]\!]^\beta \mathit{ op }\ v_2 \in [\![S_2]\!]^\beta\}$ for $op \in \{\vee, \wedge, \Rightarrow\}$, where T is the type of S_1 *op* S_2. Note that $v \in \bar{A}_G(T)$ guarantees that if $(x, a) \in v_1$, $(x, a') \in v_2$ and $v = v_1 \cup v_2$ then $a = a'$.
- $[\![\exists S_1.S_2]\!]^\beta = \{v \in \bar{A}_G(\tau^\epsilon(\exists S_1.S_2)) \mid \exists v_1 \in [\![S_1]\!]^\beta\ v_1 \cup v \in [\![S_2]\!]^\beta\}$. Note that the elements of $[\![S_2]\!]^\beta$ contain at most one pair (x_i, v_i) for each identifier x_i and therefore if $v_1 \cup v$ is in $[\![S_2]\!]^\beta$ then for all $(x_i, v_i) \in v$ and $(x'_j, v'_j) \in v_1$ with $x_i = x'_j$ we have $v_i = v'_j$.
- $[\![S \setminus [y_1, \ldots, y_n]]\!]^\beta = \{v|_{\{x_1,\ldots,x_m\}} \mid v \in [\![S]\!]^\beta\}$, where $\{x_1, \ldots, x_m\}$ is the set of identifiers of the type of S without the identifiers $y_1, \ldots, y_n$ and $v|_X$ denotes the binding v restricted to the identifiers in the set X.
- $[\![S[y_1/y'_1, \ldots, y_n/y'_n]]\!]^\beta = \{\bar{f}(v) \mid v \in [\![S]\!]^\beta\}$, where f is the function from the identifiers of type S to the identifiers of type S' defined by $[y_1/y'_1, \ldots, y_n/y'_n]$ as follows:

  $$f(id) = \begin{cases} y'_i & \text{if } y_i = id \text{ for some } 1 \leq i \leq n \\ id & \text{else} \end{cases}$$

  and $\bar{f}$ is the extension of f to bindings.
- $[\![S' \mathit{ Decor}]\!]^\beta = \{\{(\bar{x}_1, v_1), \ldots, (\bar{x}_n, v_n)\} \mid \{(x_1, v_1), \ldots, (x_n, v_n)\} \in [\![S']\!]^\beta\}$. $\bar{x}_i$ is the identifier x_i decorated with *Decor*. For example, if *Decor* is $'$ then $\bar{x}_i$ is $x_i{}'$.

Formulas The formulas in $\mathsf{Sen}_{\mathcal{SET}}(\Sigma)$ are the usual first-order formulas built on the membership predicate and the equality between expressions.

$$\begin{aligned} P \ ::=\ & \text{true} \mid \text{false} \mid E \in E \mid E = E \mid \neg P \mid P \vee P \mid P \wedge P \\ & \mid P \Rightarrow P \mid \forall S.P \mid \exists S.P \end{aligned}$$

A formula P is well-formed in an environment $\epsilon = (\Sigma, (X, \tau_X))$ if

- $P = E_1 \in E_2$, $\tau^\epsilon(E_2) = \mathcal{P}(\tau^\epsilon(E_1))$ and E_1 and E_2 are well-formed.
- $P = (E_1 = E_2)$, $\tau^\epsilon(E_1) = \tau^\epsilon(E_2)$ and E_1 and E_2 are well-formed.
- $P = \neg P'$ and P' is well-formed.
- $P = P_1 \mathit{ op } P_2$ and P_1 and P_2 are well-formed for $op \in \{\vee, \wedge, \Rightarrow\}$.
- $P = \forall S.P'$, S is well-formed and has type T and P' is well-formed with respect to $\epsilon[T]$.
- $P = \exists S.P'$, S is well-formed and has type T and P' is well-formed with respect to $\Sigma[T]$.

Given a signature-morphism $\sigma : \Sigma \to \Sigma'$ and a formula P well-formed with respect to $\epsilon = (\Sigma, (X, \tau_X))$ then the formula $\bar{\sigma}(P)$ is well-formed with respect to $(\Sigma', (X, \tau'_X))$, where $\tau'_X = \tau_X; T(\sigma_G)$, and $\bar{\sigma}(P)$ is given by:

- $\bar{\sigma}(id) = id$ if $id \in X$, $\bar{\sigma}(id) = \sigma_O(id)$ if $id \in O$ and $\bar{\sigma}(id) = \sigma_G(id)$ if $id \in G \setminus O$.

- $\bar{\sigma}((E_1, \dots, E_n)) = (\bar{\sigma}(E_1), \dots, \bar{\sigma}(E_n))$.
- $\bar{\sigma}(E.i) = \bar{\sigma}(E).i$.
- $\bar{\sigma}(<x_1 := E_1, \dots, x_n := E_n>) = <x_1 := \bar{\sigma}(E_1), \dots, x_n := \bar{\sigma}(E_n)>$.
- $\bar{\sigma}(E.x) = \bar{\sigma}(E).x$.
- $\bar{\sigma}(E_1(E_2)) = \bar{\sigma}(E_1)(\bar{\sigma}(E_2))$.
- $\bar{\sigma}(\{E_1, \dots, E_n\}) = \{\bar{\sigma}(E_1), \dots, \bar{\sigma}(E_n)\}$.
- $\bar{\sigma}(\{S \bullet E\}) = \{\bar{\sigma}(S) \bullet \bar{\sigma}(E)\}$.
- $\bar{\sigma}(\mathcal{P}(E)) = \mathcal{P}(\bar{\sigma}(E))$.
- $\bar{\sigma}(E_1 \times \ldots \times E_n) = \bar{\sigma}(E_1) \times \ldots \times \bar{\sigma}(E_n)$.
- $\bar{\sigma}(x_1 : E_1, \dots, x_n : E) = x_1 : \bar{\sigma}(E_1), \dots, x_n : \bar{\sigma}(E_n)$.
- $\bar{\sigma}(S|P) = \bar{\sigma}(S)|\bar{\sigma}(P)$.
- $\bar{\sigma}(\neg S) = \neg\bar{\sigma}(S)$.
- $\bar{\sigma}(S_1 \ op\ S_n) = \bar{\sigma}(S_1)\ op\ \bar{\sigma}(S_n)$ for $op \in \{\vee, \wedge, \Rightarrow\}$.
- $\bar{\sigma}(\exists S_1.S_2) = \exists\bar{\sigma}(S_1).\bar{\sigma}(S_2)$ and $\bar{\sigma}(\forall S_1.S_2) = \forall\bar{\sigma}(S_1).\bar{\sigma}(S_2)$.
- $\bar{\sigma}(S \setminus [x_1, \dots, x_n]) = \bar{\sigma}(S) \setminus [x_1, \dots, x_n]$.
- $\bar{\sigma}(S[x_1/y_1, \dots, x_n/y_n]) = \bar{\sigma}(S)[x_1/y_1, \dots, x_n/y_n]$.
- $\bar{\sigma}(E_1 \in E_2) = \bar{\sigma}(E_1) \in \bar{\sigma}(E_2)$.
- $\bar{\sigma}(E_1 = E_2) = \bar{\sigma}(E_1) = \bar{\sigma}(E_n)$.
- $\bar{\sigma}(\text{true}) = \text{true}$ and $\bar{\sigma}(\text{false}) = \text{false}$.
- $\bar{\sigma}(\neg P) = \neg\bar{\sigma}(P)$.
- $\bar{\sigma}(P_1\ op\ P_2) = \bar{\sigma}(P_1)\ op\ \bar{\sigma}(P_2)$ for $op \in \{\vee, \wedge, \Rightarrow\}$.
- $\bar{\sigma}(\forall S.P) = \forall\bar{\sigma}(S).\bar{\sigma}(P)$ and $\bar{\sigma}(\exists S.P) = \exists\bar{\sigma}(S).\bar{\sigma}(P)$.

Definition 9.11 ($\mathsf{Sen}_{\mathcal{SET}}$) *The functor* $\mathsf{Sen}_{\mathcal{SET}}$ *from* $\text{Sign}_{\mathcal{SET}}$ *to* Set *maps each signature* Σ *to its set of* Σ*-formulas and each signature morphism* σ *from* Σ *to* Σ' *to the translation of* Σ*-formulas to* Σ'*-formulas given by* $\bar{\sigma}$.

Validity of a well-formed formula P in $\beta = (A, A_X)$, $\beta \models^{\mathcal{SET}} P$, is defined by:

- $\beta \models^{\mathcal{SET}}$ true.
- $\beta \models^{\mathcal{SET}} E_1 \in E_2$ iff $[\![E_1]\!]^\beta \in [\![E_2]\!]^\beta$.
- $\beta \models^{\mathcal{SET}} E_1 = E_2$ iff $[\![E_1]\!]^\beta = [\![E_2]\!]^\beta$.
- $\beta \models^{\mathcal{SET}} \neg P$ iff not $\beta \models^{\mathcal{SET}} P$.

- $\beta \models^{\mathcal{SET}} P_1 \ op\ P_2$ iff $\beta \models^{\mathcal{SET}} P_1 \ op\ \beta \models^{\mathcal{SET}} P_2$ for $op \in \{\vee, \wedge, \Rightarrow\}$.
- $\beta \models^{\mathcal{SET}} \forall S.P$ iff $\beta[v] \models^{\mathcal{SET}} P$ for all $v \in [\![S]\!]^{\beta}$.
- $\beta \models^{\mathcal{SET}} \exists S.P$ iff $\beta[v] \models^{\mathcal{SET}} P$ for some $v \in [\![S]\!]^{\beta}$.

Definition 9.12 (Satisfaction) *Given an signature Σ, a formula P which is well-formed with respect to $(\Sigma, (\{\}, \tau_X))$ and a Σ-structure A then $A \models_{\Sigma}^{\mathcal{SET}} P$ if $(A, \{\}) \models^{\mathcal{SET}} P$.*

Theorem 9.13 (The Institution $\mathcal{SET}$) *The category* $\textsc{Sign}_{\mathcal{SET}}$*, the functor* $\mathsf{Str}_{\mathcal{SET}}$*, the functor* $\mathsf{Sen}_{\mathcal{SET}}$ *and the family of satisfaction relations given by* $\models_{\Sigma}^{\mathcal{SET}}$ *define the institution* $\mathcal{SET} = \langle \textsc{Sign}_{\mathcal{SET}}, \mathsf{Str}_{\mathcal{SET}}, \mathsf{Sen}_{\mathcal{SET}}, \models^{\mathcal{SET}} \rangle$.

For $\mathcal{SET}$ to be an institution we have to show that the satisfaction condition holds. For the proof of the satisfaction condition we need that the value of a translated expression $\bar{\sigma}(E)$ in $\beta = (A', A'_X)$ is the same as the value of E in the σ-reduct of A' and A'_X.

Lemma 9.14 *Let A' be a Σ'-structure, σ a signature morphism from Σ to Σ' and E a well-formed expression with respect to Σ. Then*

$$[\![\bar{\sigma}(E)]\!]^{\beta'} = [\![E]\!]^{\beta'|_\sigma}$$

where $\beta'|_\sigma = (A', A'_X)|_\sigma$ is defined as $(A'|_\sigma, A'_X)$.

Proof L. et $\Sigma = (G, O, \tau)$ and $\Sigma' = (G', O', \tau')$, $\sigma = (\sigma_G, \sigma_O)$ be a signature morphism from Σ to Σ' and $A' = (A'_G, A'_O)$ a Σ'-structure. We give the proof only for the less obvious cases.

Let $E = id$ where $id \in X$ then $[\![\bar{\sigma}(id)]\!]^{\beta'} = [\![id]\!]^{\beta'} = v$ where $(id, v) \in A'_X$ and by the definition of $\beta'|_\sigma$ we have $[\![id]\!]^{\beta'|_\sigma} = v$.

Let $E = id$ where $id \in O$ then $[\![\bar{\sigma}(id)]\!]^{\beta'} = [\![\sigma_O(id)]\!]^{\beta'} = v$ where $(\sigma_O(id), v) \in A'_O$. Applying the definition of σ-reduct we get $(id, v) \in A'|_\sigma$ and thus $[\![id]\!]^{\beta'|_{\sigma'}} = v$.

If id is in $G \setminus O$ then by the definition of signature morphisms we have $\sigma(id) \in G' \setminus O'$ and

$$[\![\bar{\sigma}(id)]\!]^{\beta'} = A'_G(\sigma_G(id)) = (A'|_\sigma)(id) = [\![\bar{\sigma}(id)]\!]^{\beta'|_\sigma}.$$

Let $E = \{S \bullet E\}$ then

$$\begin{aligned}
[\![\bar{\sigma}(\{S \bullet E\})]\!]^{\beta'} &= [\![\{\bar{\sigma}(S) \bullet \bar{\sigma}(E)\}]\!]^{\beta'} \\
&= \{[\![\bar{\sigma}(E)]\!]^{\beta'[v]} \mid v \in [\![\bar{\sigma}(S)]\!]^{\beta'}\} \\
&= \{[\![E]\!]^{\beta'[v]|_\sigma} \mid v \in [\![S]\!]^{\beta'|_\sigma}\} \\
&= [\![\{S \bullet E\}]\!]^{\beta'|_\sigma}.
\end{aligned}$$

For the last equation to hold we have to show that $\beta'[v]|_{\sigma'} = (\beta'|_\sigma)[v]$.

$$\begin{aligned}(A', A'_X)[v]|_\sigma &= (A', A'_X[v])|_\sigma \\ &= (A'|_\sigma, A'_X[v]) \\ &= (A'|_\sigma, A'_X)[v] \\ &= ((A', A'_X)|_\sigma)[v].\end{aligned}$$

Let $S = x_1 : E_1, \ldots, x_n : E_n$ then

$$\begin{aligned}[\![\bar\sigma(S)]\!]^{\beta'} &= [\![x_1 : \bar\sigma(E_1), \ldots, x_n : \bar\sigma(E_n)]\!]^{\beta'} \\ &= \{\{(x_1, v_1), \ldots, (x_n, v_n)\} \mid v_i \in [\![\bar\sigma(E_i)]\!]^{\beta'},\ 1 \leq i \leq n\} \\ &= \{\{(x_1, v_1), \ldots, (x_n, v_n)\} \mid v_i \in [\![E_i]\!]^{\beta'|_\sigma},\ 1 \leq i \leq n\} \\ &= [\![x_1 : E_1, \ldots, x_n : E_n]\!]^{\beta'|_\sigma}.\end{aligned}$$

□

THEOREM 9.15 (SATISFACTION CONDITION) *For all signature morphisms σ from Σ to Σ' in* $\mathrm{SIGN}_{\mathcal{SET}}$*, all structures A in* $\mathsf{Str}_{\mathcal{SET}}(\Sigma')$ *and formulas φ in* $\mathsf{Sen}_{\mathcal{SET}}(\Sigma)$ *we have*

$$A'|_\sigma \models^{\mathcal{SET}} \varphi \textit{ iff } A' \models^{\mathcal{SET}} \bar\sigma(\varphi).$$

PROOF A. ssume that we are given a signature morphism $\sigma : \Sigma \to \Sigma'$ and a Σ'-structure A'. Let φ be the formula $E \in E'$ from $\mathsf{Sen}_{\mathcal{SET}}(\Sigma)$. We have $A \models^{\mathcal{SET}} \bar\sigma(E \in E')$ iff $A \models^{\mathcal{SET}} \bar\sigma(E) \in \bar\sigma(E')$ iff $[\![\bar\sigma(E)]\!]^\beta \in [\![\bar\sigma(E')]\!]^\beta$, where $\beta = (A, \{\})$. Now we have $[\![\bar\sigma(E)]\!]^\beta = [\![E]\!]^{(A'|_\sigma, \{\})}$ and similar $[\![\bar\sigma(E')]\!]^\beta = [\![E']\!]^{(A'|_\sigma, \{\})}$. Thus we get $[\![\bar\sigma(E)]\!]^\beta \in [\![\bar\sigma(E')]\!]^\beta$ iff $[\![\bar\sigma(E)]\!]^{(A'|_\sigma, \{\})} \in [\![\bar\sigma(E')]\!]^{(A'|_\sigma, \{\})}$ iff $A'|_\sigma \models^{\mathcal{SET}} E \in E'$.

The argumentation is similar for the other formulas. □

9.2 Abstract Machines in Z

In Z, schemas are used for modeling the state space, the initial states and the operations of abstract machines.

The state space of the counter is given by the following schema:

Counter

$c : N$

and the initial state:

Init

Counter

$c =$ zero

Then the increment and decrement operation are given as

$$\begin{array}{|l}
\hline \texttt{Inc} \\
\Delta\texttt{Counter} \\
\hline
c' = \texttt{succ}(c) \\
\hline
\end{array}$$

and

$$\begin{array}{|l}
\hline \texttt{Dec} \\
\Delta\texttt{Counter} \\
\hline
\neg(c = \texttt{zero}) \\
\texttt{succ}(c') = c \\
\hline
\end{array}$$

where $\Delta\texttt{Counter}$ is the default notion for the schema that has as components the components of the state before and after the operation: $\Delta\texttt{Counter} = \texttt{Counter} \wedge \texttt{Counter}'$.

Abstracting from the Z-syntax we can reformulate the above example in $\texttt{SL}_{\mathcal{SET}}$ as $I_\Phi \Sigma_C$ where $\Sigma_C = (\{N\}, \{\texttt{zero}, \texttt{succ}, \texttt{Counter}, \texttt{Init}, \texttt{Inc}, \texttt{Dec}\}, \tau)$ with

- $\tau(\texttt{zero}) = N$
- $\tau(\texttt{succ}) = \mathcal{P}(N \times N)$
- $\tau(\texttt{Counter}) = \tau(\texttt{Init}) = \mathcal{P}(<c : N>)$
- $\tau(\texttt{Inc}) = \tau(\texttt{Dec}) = \mathcal{P}(<c : N, c' : N>)$

The set of formulas Φ is the union of Φ_C and Φ_N, where Φ_N restricts the interpretation of N to be isomorphic to the set of natural numbers with 0 and the successor function, and Φ_C is the set:

$$\begin{array}{ll}
\{ & \texttt{Counter} = c : N, \\
 & \texttt{Init} = (\texttt{Counter} \mid c = \texttt{zero}), \\
 & \texttt{Inc} = (\texttt{Counter} \wedge \texttt{Counter}' \mid c' = \texttt{succ}(c)), \\
 & \texttt{Dec} = (\texttt{Counter} \wedge \texttt{Counter}' \mid \texttt{succ}(c') = c \wedge \neg(c = \texttt{zero})) \quad \}
\end{array}$$

A possible model of $I_\Phi \Sigma_C$ is the Σ_C-structure $A = (A_G, A_O)$, where $A_G(N) = \mathbb{N}$ and

$$A_O = \begin{array}{lll}
\{ & (\texttt{zero}, 0), (\texttt{succ}, \{(n_1, n_2) \mid n_2 = n_1 + 1,\ n_1 \in \mathbb{N},\ n_2 \in \mathbb{N}\}), & \\
 & (\texttt{Counter}, \{(c, v) \mid v \in \mathbb{N}\}), & \\
 & (\texttt{Init}, \{(c, 0)\}), & \\
 & (\texttt{Inc}, \{\{(c, v), (c', v')\} \mid v' = v + 1\}), & \\
 & (\texttt{Dec}, \{\{(c, v), (c', v')\} \mid v' + 1 = v,\ v \neq 0\}) & \}
\end{array}$$

Let $\Sigma = (G, O, \tau)$ be a signature in $\mathcal{SET}$. A schema-type

$$T = <x_1 : T_1, \ldots , x_n : T_n>$$

defines a signature $\Sigma' = (G, O \cup \{x_1, \ldots, x_n\}, \tau)$,where $\tau'(x_i) = T_i$ and $\tau'(id) = \tau(id)$ for $id \in O$.[1]

Given a Σ-structure $A = (A_G, A_O)$ then an element $\{(x_1, v_1), \ldots, (x_n, v_n)\}$ of schema-type T defines a Σ'-structure $A' = (A_G, A_O \cup \{(x_1, v_1), \ldots, (x_n, v_n)\})$.

Definition 9.16

Given a signature $\Sigma = (G, O, \tau)$, a schema-expression S of type $\mathcal{P}(<x_1 : T_1, \ldots, x_n : T_n>)$ and a Σ-structure $A = (A_G, A_O)$. Define an abstract datatype (Σ_S, M_S^A) by

- *$\Sigma_S = (G, O \cup \{x_1, \ldots, x_n\}, \tau_S)$, where $\tau_S(x_i) = T_i$ for $1 \leq i \leq n$ and $\tau_S(id) = \tau(id)$ for $id \in O$ and*
- *$M_S^A = \{(A_G, A_O \cup v_S) \mid v_S \in [\![S]\!]^{((A_G,A_O),\{\})}\}$.*

This definition can be extended to abstract datatypes $\textsc{Sp} = (\Sigma, M)$ in $\textsc{Adt}_{\mathcal{SET}}$ by taking the union of all M_S^A for $A \in M$:

$$\textsc{Sp}_S = (\Sigma_S, \bigcup_{A \in M} M_S^A).$$

Since a specification of an abstract machine in Z consists of a schema for the state space, the initial states, and the operations, a specification of an abstract machine in Z can be also viewed as a collection of abstract datatypes for the state space, the initial states, and the operations.

In the example of the counter, let $\Sigma = (G, O, \tau)$, where $G = \{N\}$, $O = \{\mathtt{zero}, \mathtt{succ}\}$, $\tau(\mathtt{zero}) = N$ and $\tau(\mathtt{succ}) = \mathcal{P}(N \times N)$. Then the schema for the increment operation

$$(\mathtt{Counter} \wedge \mathtt{Counter}' \mid c' = \mathtt{succ}(c)),$$

which is the same as

$$(c : N,\ c' : N \mid c' = \mathtt{succ}(c)),$$

defines the abstract datatype $\textsc{Sp}_{\mathtt{Inc}} = (\Sigma_{\mathtt{Inc}}, M_{\mathtt{Inc}})$, where $\Sigma_{\mathtt{Inc}}$ is the signature

$$(\{N\}, \{\mathtt{zero}, \mathtt{succ}, c, c'\}, \tau_{\mathtt{Inc}})$$

with $\tau_{\mathtt{Inc}}(c) = \tau_{\mathtt{Inc}}(c') = N$, $\tau_{\mathtt{Inc}}(\mathtt{zero}) = N$ and $\tau_{\mathtt{Inc}}(\mathtt{succ}) = \mathcal{P}(N \times N)$, and

$$M_{\mathtt{Inc}} = \{(A_G, A_O \cup \{(c, v), (c', v')\}) \mid v' = [\![\mathtt{succ}(c)]\!]^{((A_G,A_O),\{\})},\ (A_G, A_O) \in M\}.$$

Note that we get similar abstract datatypes when using $\mathtt{RSL}_{\mathcal{SET}}$ to define the counter and translating them to abstract datatypes in $\textsc{Adt}_{\mathcal{SET}}$ (cf. Section 7.1).

Also the operations on schemas are similar to the operations found in $\mathtt{SL}_{\mathcal{SET}}$. Let $\textsc{Sp}$ be an $\mathtt{SL}_{\mathcal{SET}}$ expression with semantics (Σ, M). If

[1] Note that Σ' is not a signature as defined in Definition 9.1 because $\{x_1, \ldots, x_n\}$ is not a subset of F since, for technical reasons, we had to require that the set of variable names and the set of identifier names are disjoint. However, we can assume that O' is the set $O \cup \{\bar{x}_1, \ldots, \bar{x}_n\}$ where the $\bar{x}_i$ are suitable renamings of x_i to symbols in F not occurring in O.

- $S = x_1 : E_1, \dots, x_n : E_n$ then $[\![\mathrm{SP}]\!]_S = \big[\!\big[I_{\{x_i \in E_i | 1 \leq i \leq n\}} T_\sigma \mathrm{SP}\big]\!\big]$, where σ is the inclusion of Σ into Σ_S.
- $S = S'|P$ then $[\![\mathrm{SP}]\!]_S = \big[\!\big[I_{\{P\}} \mathrm{SP}_{S'}\big]\!\big]$.
- $S = S_1 \wedge S_2$ then $[\![\mathrm{SP}]\!]_S = [\![T_{\sigma_1} \mathrm{SP}_{S_1} + T_{\sigma_2} \mathrm{SP}_{S_2}]\!]$. The signature morphisms σ_1 and σ_2 are the inclusions of the signatures Σ_{S_1} and Σ_{S_2} into $\Sigma_{S_1 \wedge S_2}$. This is needed because, in contrast to the union of abstract datatypes, the types of S_1 and S_2 are only required to be compatible in the union of S_1 and S_2.
- $S = S_1 \vee S_2$ then $[\![\mathrm{SP}]\!]_S = [\![T_{\sigma_1} \mathrm{SP}_{S_1} \vee T_{\sigma_2} \mathrm{SP}_{S_2}]\!]$. The signature morphisms σ_1 and σ_2 are the same as defined for the union of schemas.
- $S = S' \setminus [x_1, .., x_n]$ then $[\![\mathrm{SP}]\!]_S = [\![D_\sigma \mathrm{SP}_{S'}]\!]$, where σ is the inclusion of Σ_S into $\Sigma_{S'}$.
- $S = S'[x_1/y_1, .., x_n/y_n]$ then $[\![\mathrm{SP}]\!]_S = [\![T_\sigma \mathrm{SP}_{S'}]\!]$, where σ_G is the identity and $\sigma_O(x) = y_i$, if $x = x_i$ for some i and $\sigma_O(x) = x$ if $x \neq x_i$ for all i.

However, we do not have corresponding $\mathtt{SL}_{\mathcal{SET}}$ operations for $\neg S$, $S \Rightarrow S$, $\forall S.S$ and $\exists S.S$. While their semantics could be easily defined, for example, $[\![\neg(\Sigma, M)]\!] = (\Sigma, \{m \in \mathsf{Str}_{\mathcal{I}}(\Sigma) \mid m \notin M\})$, one would need additional proof rules for the entailment of specification, as in the case of disjunction (cf. Chapter 8).

There are some important differences between modeling the operations as schemas and as abstract datatypes. Defining the operations as schemas depends on the base institution to provide schemas. This fixes the institution to $\mathcal{SET}$. In contrast, in the approach presented in this thesis, we have separated the logic for defining the operations of an abstract machine from the base institution. Thus our approach applies to any institution whose category of signatures is (finitely) cocomplete and whose structure functor preserves (finite) colimits. On the other side, since schemas are part of the institution $\mathcal{SET}$, it is possible to reason within $\mathcal{SET}$ itself about the operations of an abstract machine. In our approach only the operations in $\mathtt{RSL}_{\mathcal{SET}}$ can be used to refer to other operations.

Another difference is that expressions in $\mathtt{RSL}_{\mathcal{SET}}$ carry information about the type of the relation. The type of an operation in the Z style specification of abstract machines is kept informal. Conventions, like the use of Δ`Counter` and descriptive text that surrounds a schema definition, provides this information.

9.3 Z-style specifications using LSL

The theory of relations, presented in the previous chapters, is based on a newly constructed institution $\mathcal{R}_{\mathcal{I}}$ with respect to an, arbitrary exact institution $\mathcal{I}$ with a (finitely) cocomplete category of signatures. Then the language $\mathtt{RSL}_{\mathcal{I}}$ is used to define relations. However, the problem with this approach is that $\mathtt{RSL}_{\mathcal{I}}$ is an abstract language, which is intended mainly as a vehicle for more concrete languages to define their semantics in and for theoretical studies. Another problem is that no tools exist for $\mathtt{RSL}_{\mathcal{I}}$.

To address this problem we shall adopt a method used by the Z community for the specification of sequential systems [57]. The specification language Z itself almost has no language support for defining sequential system. It is a matter of convention and the explanations used in the text surrounding a Z schema, whether this schema is to be interpreted as a definition of a relation or the specification of a state space.

In the following, we shall use a similar approach based on the Larch Shared Language. The advantage is that we can use tools designed for the Larch Shared Language, like the Larch Shared Language checker and the Larch Prover to check our specifications and to prove properties. However, since some information, for example, whether a specification is meant as specifying a state space or defining a relation, is left implicit, the use of these tools is limited.

Note that we could have used any suitable specification language, that is, a specification language with loose semantics, like Pluss [8] or CASL [45].

That this approach is possible, is the result of the strong relationship between $\textsc{Adt}_{\mathcal{R}_{\mathcal{I}}}$ and $\textsc{Adt}_{\mathcal{I}}$ established in Section 5.6.

DEFINITION 9.17 *An abstract datatype* $\textsc{Sp} = (\Sigma, M)$ *from* $\textsc{Adt}_{\mathcal{I}}$ *is a* relation of type Θ_R *if there exists a relation* $R = (\Theta_R, M_R)$ *of type* $\Theta_R = (\mathrm{D}_R, \Gamma_R)$ *in* $\textsc{Adt}_{\mathcal{R}_{\mathcal{I}}}$ *and an* $\textsc{Adt}_{\mathcal{I}}$*-morphism* σ *from* $\coprod \Gamma_R; \mathsf{Sig}^{\mathcal{A}}$ *to* Σ *such that* $M|_{\sigma} = \overline{M}_R$*.*

As an example we shall give the birthday book specification of Spivey [56], which we have discussed in Section 6.2.

The state of a birthday book is given by the predicate `known`, a boolean function, and the (total) function `birthday`, mapping the name of a person to his birthday. The value of `birthday` for a name n is arbitrary if `known`(n) is not true.

```
BirthdayBook : trait
   introduces
      known: Name → Bool
      birthday : Name → Date
```

The initial birthday book contains no entries.

```
InitBirthday : trait
   includes
      BirthdayBook
   asserts
      ∀ n:Name   ¬known(n)
```

The trait `DeltaBirthdayBook` introduces two copies of `known` and `birthday`, an unprimed copy for the state components before the operation and a primed copy for the state components after the execution of the operation. Any operation that wants to change the state of the birthday book includes `DeltaBirthday`.

```
DeltaBirthdayBook : trait
   includes
      BirthdayBook,
      BirthdayBook(known' for known,
                   birthday' for birthday)
```

DeltaBirthday is the colimit of the type <BirthdayBook × BirthdayBook>$_{\texttt{Env}}$.

In contrast, the trait ThetaBirthdayBook is included in the trait of all operations that do not want to change the state of the birthday book.

```
ThetaBirthdayBook : trait
   includes
      BirthdayBook,
      BirthdayBook(known' for known,
                   birthday' for birthday)
   asserts
      forall n:Name
         known'(n) ⇔ known(n);
         birthday'(n) = birthday(n)
```

The operation adding a new birthday to a birthday book is specified by the trait AddBirthday. The input parameters are the name of the person to add, $\texttt{name}_{\texttt{in}}$, and the birthday of that person, $\texttt{date}_{\texttt{in}}$. The operation is defined only when that person does not already have an entry in the birthday book.

```
AddBirthday : trait
   includes
      DeltaBirthdayBook
   introduces
      name_in:  Name
      date_in:  Date
   asserts
      forall n:Name
         ¬known(name_in);
         known'(n) ⇔ n = name_in  ∨  known(n);
         birthday'(n) = (if n = name_in
                         then date_in
                         else birthday(n))
```

AddBirthday is a relation of type

$$<[\texttt{name}_{\texttt{in}} : \texttt{Name}] \times [\texttt{date}_{\texttt{in}}] \times \texttt{BirthdayBook} \times \texttt{BirthdayBook}>.$$

The FindBirthday operation returns for a given person with name $\texttt{name}_{\texttt{in}}$ its birthday $\texttt{date}_{\texttt{out}}$. The inclusion of ThetaBirthdayBook guarantees that the state of the birthday book is not changed.

```
FindBirthday : trait
   includes
      ThetaBirthdayBook
   introduces
      name$_{in}$:  Name
      date$_{out}$: Date
   asserts
      equations
         known(name$_{in}$);
         date$_{out}$ = birthday(name$_{in}$)
```

For a given date $\texttt{today}_{\texttt{in}}$, the Remind operation returns the set of persons whose birthday is $\texttt{today}_{\texttt{in}}$.

```
Remind : trait
   includes
      ThetaBirthdayBook,
      Set(Name,NameSet)
   introduces
      today$_{in}$: Date
      cards$_{out}$: NameSet
   asserts
      forall n:Name
         n ∈ cards$_{out}$ ⇔
            birthday(n) = today$_{in}$ ∧ known(n)
```

10 Conclusion

In this thesis we have defined a framework for the specification of dynamic behavior of software systems. This framework is motivated by the state as algebra approach and the model-oriented language Z. From the state as algebra approach we have used the idea of modeling the environment and the state components as structures of an institution. However, in contrast to the state as algebra approach, states in our framework are modeled by structures from any suitable institution not only those having algebras as their structures. From Z we have used the idea that environment, state spaces, and relations between state spaces are specified using the same logic, and how more complex relations can be constructed from simpler ones by means of the schema calculus. However, we differ from Z in that our framework can be instantiated by different institutions while the approach of Z can only work because of the particular logical system used by Z.

We have defined a new institution $\mathcal{R}_{\mathcal{I}}$, which is based on an exact institution $\mathcal{I}$ with a cocomplete category of signatures. Abstract datatypes in $\mathcal{R}_{\mathcal{I}}$ are relations between abstract datatypes from $\mathcal{I}$; this means that a relation is a pair (Θ, M) consisting of a diagram $\Theta = (\mathrm{D}, \Gamma : \mathrm{D} \to \textsc{Adt}_{\mathcal{I}})$ and a set M of families of structures $m_d \in \mathsf{Mod}^{\mathcal{A}}(\Gamma(d))$ for $d \in \mathrm{D}$ such that $m_{d'}|_{\Gamma(f)} = m_d$ for each morphism $f : d \to d'$ in D.

Most of the known institutions have a cocomplete category of signatures, and their structure functor preserves colimits. In addition, we have defined two new institutions, the institution $\mathcal{LSL}$, which is a variant of equational logic with constraints, used to model the logical system of the Larch Shared Language, and the institution $\mathcal{SET}$, which models the logical system of the specification language Z and which is based on set theory. Further, it is possible, by defining a new institution, to instantiate our framework with models of the state more appropriate to the problem domain. For example, one could define an institution whose structures model states having components that denote arrays and pointer structures.

We have shown that the category of signatures of $\mathcal{R}_{\mathcal{I}}$ is cocomplete and that the structure functor of $\mathcal{R}_{\mathcal{I}}$ preserves colimits. This implies, for example, that relations on complex states can be uniquely constructed by relations on simpler states. As an example consider two state spaces given by abstract datatypes $\textsc{Sp}_1$ and $\textsc{Sp}_2$ such that both are extensions of the abstract datatype `Env`. Further, let R_1 be a relation of type $<\textsc{Sp}_1 \times \textsc{Sp}_1>_{\texttt{Env}}$ and R_2 be a relation of type $<\textsc{Sp}_2 \times \textsc{Sp}_2>$. Now we can form a state space $\textsc{Sp}$ which consists of the components of $\textsc{Sp}_1$ and $\textsc{Sp}_2$ sharing the components of `Env`. Thus, $\textsc{Sp}$ is the pushout of $\textsc{Sp}_1$ and $\textsc{Sp}_2$ with respect to `Env`. Then we can define the relation R of type $<\textsc{Sp} \times \textsc{Sp}>_{\texttt{Env}}$ acting as R_1 on the state components from $\textsc{Sp}_1$ and as R_2 on the state components from $\textsc{Sp}_2$ as the pushout of R_1 and R_2 with respect to the identity on `Env`. That this is always possible is a consequence of the fact that $\textsc{Sign}_{\mathcal{R}_{\mathcal{I}}}$ is cocomplete; then the fact that $\mathsf{Str}_{\mathcal{R}_{\mathcal{I}}}$ preserves colimits implies that the pairs (A, A') in R can be uniquely constructed from pairs (A_1, A'_1) in R_1 and pairs (A_2, A'_2) in R_2 as the amalgamated sums $A = A_1 +_{A_0} A_2$ and $A' = A'_1 +_{A'_0} A'_2$. Here, A_0 is the environment that A_1 and A'_1 have in common and A'_0 is the environment that A_2 and A'_2 have in common. Note that A_0 and A'_0 have to be the same environment to make the construction work.

The advantage of modeling relations as abstract datatypes in $\mathcal{R}_\mathcal{I}$ is that we could reuse the institution independent theory of abstract datatypes. For example we could use the specification language $\mathtt{SL}_\mathcal{I}$, which is based on the operations impose, translate, derive, and union, as the foundation of the specification language $\mathtt{RSL}_\mathcal{I}$ for the specification of relations. $\mathtt{RSL}_\mathcal{I}$ is not intended to be a specification language used in practice, but as more practical specification languages, like CASL [45], can be defined in terms of the operations of $\mathtt{SL}_\mathcal{I}$, more practical specification languages for the specification of relations could be defined using the operations of $\mathtt{RSL}_\mathcal{I}$.

In this thesis we have added disjunction to the operations on abstract datatypes. The disjunction of two abstract datatypes over the same signature forms the union of their model classes. Disjunction was not treated before mainly because the disjunction of two abstract datatypes given by sets of formulas Φ_1 and Φ_2 cannot be represented by one set of formulas in every institution. This is only possible for restricted classes of sets of formulas and for certain logics, like finite sets of formulas and first-order logic. This is in contrast to the union of two abstract datatypes for which $I_{\Phi_1}\Sigma + I_{\Phi_2}\Sigma$ is the same as $I_{\Phi_1 \cup \Phi_2}\Sigma$ in any institution. While the usefulness of disjunction for the definition of classical abstract datatypes is questionable, this is not the case for the construction of relations because disjunction allows to combine relations defined on different domains to a relation defined on the union of these domains.

There are two ways of proving properties of relations and entailment of relations. The first way takes advantage of modeling relations as abstract datatypes because this allows us to apply the proof calculus for proving properties from abstract datatypes to prove properties from relations and to prove entailment between relations.

A second way is given by the translation $\mathtt{tr}$ of $\mathtt{RSL}_\mathcal{I}$-expressions to $\mathtt{SL}_\mathcal{I}$-expressions, which is defined in Section 7.1. One property of this translation is that it preserves entailment, that is, $R \models^{\mathcal{R}_\mathcal{I}} \varphi$ if and only if $\mathtt{tr}(R) \models^\mathcal{I} \varphi$ and, similarly, $R \models^{\mathcal{R}_\mathcal{I}} R'$ if and only $\mathtt{tr}(R) \models^\mathcal{I} \mathtt{tr}(R')$. This makes it possible to reuse theorem provers for the institution $\mathcal{I}$ for the institution $\mathcal{R}_\mathcal{I}$.

Both strategies can be combined by first applying the rules of the proof calculus to reduce a goal $R \models^{\mathcal{R}_\mathcal{I}} R'$ into subgoals $R_1 \models^{\mathcal{R}_\mathcal{I}} \varphi_1$ to $R_n \models^{\mathcal{R}_\mathcal{I}} \varphi_n$ and then proving $\mathtt{tr}(R_i) \models^{\mathcal{R}_\mathcal{I}} \varphi_i$ by using a theorem prover for $\mathcal{I}$.

We have already discussed that $\mathtt{RSL}_\mathcal{I}$ can be used as the basis for a more practical specification language for the specification of the dynamic behavior of software systems. However, there is a second way to use the results of this thesis to specify software systems. This way has the advantage that it can be used with any specification language whose semantics are abstract datatypes.

The idea is that each relation $R = (\Theta, M_R)$ in $\textsc{Adt}_{\mathcal{R}_\mathcal{I}}$ corresponds to an abstract datatype

(Σ_{cl}, M'_R) in $\textsc{Adt}_{\mathcal{I}}$ such that M_R and M'_R are isomorphic. This correspondence can be used to interpret an abstract datatype (Σ, M) with respect to $\mathcal{I}$ as a relation (Θ, M_R) provided that we know the type of the relation. We need to know the type of the relation because from Σ it is impossible to get the type of the relation back as there may be several relations having the same signature Σ_{cl} as the colimit of $\Gamma_R; \mathsf{Sig}^{\mathcal{A}}$. If we keep the type of a relation as an informal annotation with the specification of abstract datatypes in $\mathcal{I}$, then we can use any specification language having abstract datatypes in $\mathcal{I}$ as the semantics of its specifications for the specification of the dynamic behavior of software systems. As an example of this style we have given in Section 9.3 a specification of the birthday book example of the Z-reference manual using the Larch Shared Language. Another example of this style can be found in Baumeister [6].

Note that this approach is similar to the way the dynamic behavior of software systems are specified in Z. In Z states and operations on states are both presented as schemata, which correspond to abstract datatypes (cf. Section 9.2); only naming conventions and the text surrounding a Z-schema determine whether a schema represents an operation or a state space.

10.1 Future Work

Since the results of this thesis are parameterized by any suitable institutions, we can apply our framework to all the other state as algebra approaches with the exception of Gurevich's Abstract State Machines. We cannot apply our results directly to Gurevich's Abstract State machines because the state of an Abstract State Machine is modeled as an unsorted algebra, and for the institutions of unsorted equational and first-order logic the structure functor does not preserve coproducts (cf. Diaconescu et al [17]), and thus it does not preserve arbitrary colimits, which is required by the construction of $\mathcal{R}_{\mathcal{I}}$. However, the structure functor preserves colimits of all diagrams D that have a unique bottom element, that is, D has an element $\perp$ and for all objects $d \neq \perp$ in D there exists a morphism $f : \perp \to d$ in D. This is the type of diagrams we are mostly interested in because they are used in the type of relations of the form $<\textsc{Sp}_1 \times \cdots \times \textsc{Sp}_n>_{\mathtt{Env}}$. Thus we could, in the construction of $\mathcal{R}_{\mathcal{I}}$, restrict ourself to types $(\mathrm{D}, \Gamma : \mathrm{D} \to \textsc{Adt}_{\mathcal{I}})$ where D has this bottom element. This allows a lot of the results in this thesis to be applied to Gurevich's Abstract State Machines. However, modularity results, that is, that the category of signatures constructed in that way is cocomplete and that the structure functor preserves colimits, do not carry over. The problem here is that the colimit of a diagram of types which have a unique bottom element need not yield a signature (D_{cl}, Γ_{cl}) where D_{cl} has a unique bottom element.

One application of our results could be, for a particular instantiation of our framework, to extend $\mathtt{RSL}_{\mathcal{I}}$ by additional operations, like the local function update of Gurevich or the elementary modifiers of the implicit state approach. Then the inference system has to be extended by special rules coping with these new operations. For example, if R is given by an update instruction $f(t) := t'$ in the style of Abstract State Machines, then one would get additional inference rules $R \vdash f'(t) = t'$, $R \vdash x \neq t \Rightarrow f'(x) = f(x)$, and $R \vdash g'(x) = g(x)$

for all function symbols in the state signature which are not f. However, the translation of $\mathtt{RSL}_{\mathcal{I}}$-expressions to $\mathtt{SL}_{\mathcal{I}}$-expression may not be possible anymore.

A closer look at the institution independent proof-calculus for entailment between abstract datatypes is needed, in particular, for entailments where the right side contains a derive operation. As we have seen in Section 7.4, this appears quite often when proving refinement, and proving them can get quite tedious. One possibility would be to encode $\mathtt{RSL}_{\mathcal{I}}$ and its proof-calculus into a higher-order logic framework, for instance Isabelle [47], and make it parametric by a suitable encoding of the logical-system used to model the states.

What we could only touch in this thesis is a theory of abstract machines based on our approach (cf. Chapter 6). One problem is how to combine abstract machines, for example, use abstract machines as arguments to the operations of other abstract machines. In the current approach it is possible to pass the *state* of an abstract machine as an argument, but information hiding is lost because the operation has access to all the state components. Similarly, one would want to use abstract machines as state components. The modularity results allow us to use the state of another abstract machine as a sub-state of another, but again loosing encapsulation.

A Larch Traits

In the following we present the traits FiniteMap and Set, which are based on the traits SetBasics and DerivedOrders from the library of the Larch Shared Language. A description of this library can be found in the book about Larch by Guttag and Horning [34].

A.1 FiniteMap

```
FiniteMap (M, D, R): trait
  % An M is a map from D's to R's.
  introduces
    {}: → M
    update: M, D, R → M
    apply: M, D → R
    defined: M, D → Bool
  asserts
    M generated by {}, update
    M partitioned by apply, defined
    ∀ m: M, d, d1, d2: D, r: R
      apply(update(m, d2, r), d1) ==
        if d1 = d2 then r else apply(m, d1);
      ¬defined({}, d);
      defined(update(m, d2, r), d1) ==
        d1 = d2 ∨ defined(m, d1)
  implies
    Array1 (update for assign, apply for [],
            M for A, D for I, R for E)
    converts apply, defined
      exempting ∀ d: D apply({}, d)
```

A.2 Set

```
Set (E, C): trait
  % Common set operators
  includes
    SetBasics,
    Integer,
    DerivedOrders (C, ⊆ for ≤, ⊇ for ≥,
                   ⊂ for <, ⊃ for >)
  introduces
    ∉ : E, C → Bool
    delete: E, C → C
    {}: E → C
     ∪ ,  ∩ , -: C, C → C
    size: C → Int
  asserts
    ∀ e, e1, e2: E, s, s1, s2: C
      e ∉ s == ¬(e ∈ s);
      { e } == insert(e, {});
      e1 ∈ delete(e2, s) == e1 ≠ e2 ∧ e1 ∈ s;
      e ∈ (s1 ∪ s2) == e ∈ s1 ∨ e ∈ s2;
      e ∈ (s1 ∩ s2) == e ∈ s1 ∧ e ∈ s2;
      e ∈ (s1 - s2)  == e ∈ s1 ∧ e ∉ s2;
      size({}) == 0;
      size(insert(e, s)) ==
        if e ∉ s then size(s) + 1 else size(s);
      s1 ⊆ s2 == s1 - s2 = {}
  implies
    AbelianMonoid (∪ for ∘, {} for unit, C for T),
    AC (∩, C),
    JoinOp (∪, {} for empty),
    MemberOp ({} for empty),
    PartialOrder (C, ⊆ for ≤, ⊇ for ≥,
                  ⊂ for <, ⊃ for >)
    C generated by {}, {}, ∪
    ∀ e: E, s, s1, s2: C
      s1 ⊆ s2 ⇒ (e ∈ s1 ⇒ e ∈ s2);
      size(s) ≥ 0
    converts
      ∈, ∉, {}, delete, size, ∪, ∩, -:C,C→C,
      ⊆, ⊇, ⊂, ⊃
```

A.3 SetBasics

```
SetBasics (E, C): trait
  % Essential finite-set operators
  introduces
    {}: → C
    insert: E, C → C
    ∈ : E, C → Bool
  asserts
    C generated by {}, insert
    C partitioned by ∈
    ∀ s: C, e, e1, e2: E
```

```
      ¬(e ∈ {});
      e1 ∈ insert(e2, s) == e1 = e2 ∨ e1 ∈ s
  implies
    InsertGenerated ({} for empty)
    ∀ e, e1, e2: E, s: C
      insert(e, s) ≠ {};
      insert(e, insert(e, s)) == insert(e, s);
      insert(e1, insert(e2, s)) ==
        insert(e2, insert(e1, s))
    converts ∈
```

A.4 DerivedOrder

```
DerivedOrders (T): trait
  % Define any three of the comparison operators,
  % given the fourth
  introduces
    ≤, ≥, <, >: T, T → Bool
  asserts ∀ x, y: T
    x < y == x ≤ y ∧ ¬(x = y);
    x ≥ y == y ≤ x;
    x > y == y < x
  implies
    ∀ x, y: T
      x ≤ y == x < y ∨ x = y
    converts ≥, <, >
    converts ≤, <, >
    converts ≤, ≥, >
    converts ≤, ≥, <
```

Bibliography

[1] Jean-Raymond Abrial. *The B-Book. Assigning Programs to Meanings.* Cambridge University Press, first edition, 1996.

[2] Jiri Adamek, Horst Herrlich, and Georg Strecker. *Abstract and concrete categories.* Wiley, 1990.

[3] Egidio Astesiano and Martin Wirsing. An introduction to ASL. Technical Report MIP–8609, Universität Passau, May 1986.

[4] Egidio Astesiano and Elena Zucca. D-oids: a model for dynamic data–types. *Mathematical Structures in Computer Science*, 5(2):257–282, 1995.

[5] Michael Barr and Charles Wells. *Category Theory for Computing Science.* Prentice Hall, New York, 1990.

[6] Hubert Baumeister. Using algebraic specification languages for model-oriented specifications. Technical Report MPI-I-96-2-003, Max-Planck-Institut für Informatik, Saarbrücken, Germany, February 1996.

[7] J. A. Bergstra, J. Heering, and P. Klint. Module algebra. *Journal of the ACM*, 37(2):335–372, April 1990.

[8] Michel Bidoit. *Pluss, un langage pour le developement de specifications algebriques modulaires.* PhD thesis, Université de Paris-sud, 1989.

[9] Michel Bidoit and Rolf Hennicker. Behavioural theories and the proof of behavioural properties. *Theoretical Computer Science*, 165(1):3–55, September 30 1996.

[10] Tomasz Borzyszkowski. Correctness of the logical system for structured specifications. In Francesco Parisi Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12 International Workshop, WADT '97, June 1997, Selected Papers*, pages 107–121, Tarquinia, Italy, June 1998. Springer.

[11] Ruth Breu. A normal form for structured algebraic specifications. Technical report, MIP, 1989.

[12] Stephen Brien and John Nicholls. Z base standard version 1.0. Oxford University Computing Laboratory Programming Research Group, November 30 1992.

[13] R. M. Burstall and J. A. Goguen. The semantics of Clear, a specification language, February 1980.

[14] María Victoria Cengarle. *Formal Specifications with Higher-Order Parameterization.* PhD thesis, Fakultät für Mathematik, Ludwig-Maximilians-Universität, München, 1994.

[15] Ingo Claßen, Hartmut Ehrig, and Dietmar Wolz. *Algebraic specification techniques and tools for software development: the ACT approach.* Number 1 in AMAST series in computing. World Scientific, London, 1993.

[16] P. Dauchy and M.-C. Gaudel. Algebraic specifications with implicit state, February 1994.

[17] Răzvan Diaconescu, Joseph Goguen, and Petros Stefaneas. Logical support for modularisation. In Gerard Huet and Gordon Plotikin, editors, *Proceedings of a Workshop on Logical Frameworks*, 1991. Revision of 24 March 1993.

[18] H. Ehrig, E. G. Wagner, and J. W. Thatcher. Algebraic constraints for specifications and canonical from results. In *Proceeding of the 10th International Colloquium on Automata, Languages and Programming*, number 154 in LNCS, pages 188–202. Springer, 1983.

[19] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and initial Semantics.* Number 6 in EATCS Monographs on Theoretical Computer Science. Springer, 1985.

[20] Hartmut Ehrig and Fernando Orejas. Dynamic abstract data types, an informal proposal. *Bulletin of the EATCS*, 53:162–169, June 1994.

[21] Jorge Farrés-Casals. *Verification in ASL and Related Specification Languages.* PhD thesis, Department of Computer Science, The University of Edinburgh, July 1992. Published as technical reports ECS-LFCS-92-220 and CST-92-92.

[22] Loe M. Feijs and H. B. Jonkers. *Formal Specification and Design*, volume 35 of *Cambridge tracts in theoretical computer science.* Cambridge Univ. Press, Cambridge, 1992.

[23] Harald Ganzinger. Programs as transformations of algebraic theories (extended abstract). *Informatik Fachberichte*, 50:22–41, 1981.

[24] J. Goguen and J. Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. In *Proceedings, ICALP*, number 140 in LNCS, pages 265–281. Springer, 1982.

[25] J. A. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, January 1992.

[26] Joseph Goguen and Grant Malcolm. A hidden agenda. Report CS97–538, University of California at San Diego, April 1997.

[27] Joseph A. Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In Hartmut Ehrig and Fernando Orejas, editors, *Recent Trends in Data Type Specification, 9th Workshop on Specification of Abstract Data Types Joint with the 4th COMPASS Workshop, Selected Papers, Caldes de Malavella, Spain 1992*, number 785 in LNCS, 1993.

[28] Joseph A. Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. Technical report, SRI International, October 1993.

[29] Martin Große-Rhode. *Specification of Transition Categories, An Approach to Dynamic Abstract Data Types.* PhD thesis, Fachbereich 13 — Informatik, Technische Universität, Berlin, 1995.

[30] Martin Große-Rhode. Transition specifications for dynamic abstract data types. *Applied Categorical Structures*, 5:265–308, 1997.

[31] Alexandre Grothendieck. Catégories fibrées et descente. In *Revêtements étales et groupe fondamental, Séminaire de Géométrie Algébraique du Bois-Marie 1960/61, Exposé VI.* Institut des Hautes Études Scientifiques, 1963. Reprinted in Lecture Notes in Mathematics, Volume 224, Springer, 1971, pp. 145–194.

[32] Yuri Gurevich. Evolving algebras: An attempt to discover semantics. *Bulletin of the EATCS*, 43:264–284, February 1991.

[33] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In Egon Börger, editor, *Specification and Valication Methods*, pages 7–36. Clarendon Press, Oxford, 1995.

[34] John V. Guttag and J. Horning. *LARCH: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer, New York, 1993.

[35] David Harel. Dynamic logic. In Dov M. Gabbay and Franz Guenthner, editors, *Handbook of philosophical logic : vol. 2: extensions of classical logic*, volume 165 of *Synthese library*, pages 497–604. Kluwer, Dordrecht, 1984.

[36] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined, resume. In Bernard Robinet and Reinhard Wilhelm, editors, *ESOP '86: European Symposium on Programming, Saarbrücken*, number 213 in LNCS, pages 187–196. Springer, March 17–19 1986.

[37] Eric C. R. Hehner. *A Practical Theory of Programming.* Texts and Monographs in Computer Science. Springer, 1993.

[38] Rolf Hennicker, Martin Wirsing, and Michel Bidoit. Proof systems for structured specifications with observability operators. *Theoretical Computer Science*, 173(2):393–443, February 28 1996.

[39] C. A. R. Hoare. Proving correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[40] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice Hall international series in computer science. Prentice Hall, New York, second edition, 1990.

[41] Thomas Lehmann and Jacques Loeckx. The specification language of OBSCURE. In Don Sannella and Andrzej Tarlecki, editors, *Proceedings of the 5th Workshop on Recent Trends in Data Type specification*, number 332 in LNCS, pages 131–153, Gullane, Scotland, September 1988. Springer.

[42] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of Abstract Data Types*. Wiley-Teubner, 1996.

[43] Saunders Mac Lane. *Categories for the working mathematician*. Graduated Texts in Mathematics. Springer, fourth edition, 1988.

[44] José Meseguer. General logics. technical report SRI-CSL-89-5, SRI International, March 1989.

[45] Peter D. Mosses. CoFI: The common framework initiative for algebraic specification and development. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Proceedings of the Seventh Joint Conference on Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, number 1214 in LNCS, Lille, France, April 1997. Springer.

[46] Tobias Nipkow. Non-deterministic data types: Models and implementation. *Acta Informatica*, 22:629–661, 1986.

[47] Lawrence C. Paulson. *Isabelle: a generic theorem prover; with contributions by Tobias Nipkow*. Number 828 in LNCS. Springer, Berlin, 1994.

[48] H. Reichel. Partial algebras — a sound basis for structural induction. In P. Burmeister, B. Ganter, C. Herrman, K. Keimel, W. Poguntke, and R. Wille, editors, *Universal Algebra and its links with logic, algebra, combinatorics and computer science. Proceedings of the '25. Arbeitstagung über Allgemeine Algebra'*, R&E: research and exposition in mathematics, pages 230–240. Heldermann Verlag, Berlin, Darmstadt, 1983.

[49] H. Reichel. Software specification by behavioural canons. TU Magdeburg, 1986.

[50] Donald Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76(2/3):165–210, February/March 1988.

[51] Donald Sannella and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: Implementation revisited. *Acta Informatica*, 25:233–281, 1988.

[52] Donald Sannella and Andrzej Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269, 1997.

[53] Donald Sannella and Martin Wirsing. A kernel language for algebraic specification and implementation. In M. Karpinski, editor, *Colloquium on Foundations of Computation Theory*, number 158 in LNCS, pages 413–427, Berlin, 1983. Springer.

[54] Donald Sannella and Martin Wirsing. Specification languages. In E. Astesiano, H.-J. Kreowski, and B Krieg-Brückner, editors, *Algebraic Foundations of Systems Specifications*, chapter 7. Chapman and Hall, September 15th 1995. draft.

[55] Oliver Schoett. *Data Abstraction and the Correctness of Modular Programming*. PhD thesis, University of Edinburgh, 1986.

[56] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge tracts in theoretical computer science*. Cambridge Univ. Press, Cambridge, GB, repr. 1992 edition, 1988.

[57] J. Michael Spivey. *The Z Notation: A Reference Manual*. International series in computer science. Prentice Hall, New York, 2nd edition, 1992.

[58] A. Tarlecki, R. Burstall, and J. Goguen. Some fundamental algebraic tools for the semantics of computation, Part 3: Indexed categories. *Theoretical Computer Science*, 91:239–264, 1991.

[59] Roel Wieringa. Equational specification of dynamic objects. Technical Report 91-1, Faculty of Mathematics and Computer Science, Vrije Universiteit Amsterdam, 1991.

[60] Roel J. Wieringa. A formalization of objects using equational dynamic logic. Technical Report 91-2, Faculty of Mathematics and Computer Science, Vrije Universiteit Amsterdam, 1991.

[61] Martin Wirsing. Structured specifications: Syntax, semantics and proof calculus. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and algebra of specification: proceedings of the NATO Advanced Study Institute on Logic and Algebra of Specification, Marktoberdorf, Germany, July 23-August 4, 1991*, number 94 in NATO ASI series, pages 411–442. Springer, 1992.