Relating Abstract Datatypes and Z-Schemata[⋆]

Hubert Baumeister

University of Munich, Institute of Computer Science,
Oettingenstr. 67, D-80358 Munich, Germany
baumeist@informatik.uni-muenchen.de

Abstract. In this paper we investigate formally the relationship between the notion of abstract datatypes in an arbitrary institution, found in algebraic specification languages like Clear, ASL, and CASL; and the notion of schemata from the model-oriented specification language Z. To this end the institution $\mathcal{S}$ of the logic underlying Z is defined, and a translation of Z-schemata to abstract datatypes over $\mathcal{S}$ is given. The notion of a schema is internal to the logic of Z, and thus specification techniques of Z relying on the notion of a schema can only be applied in the context of Z. By translating Z-schemata to abstract datatypes, these specification techniques can be transformed to specification techniques using abstract datatypes. Since the notion of abstract datatypes is institution independent, this results in a separation of these specification techniques from the specification language Z and allows them to be applied in the context of other, e.g. algebraic, specification languages.

1 Introduction

As already noted by Spivey [12], schema-types, as used in the model-oriented specification language Z, are closely related to many-sorted signatures, and schemata are related to the notion of abstract datatypes found in algebraic specification languages.

Z is a model-oriented specification language based on set-theory. In the model-oriented approach to the specification of software systems, specifications are explicit system models constructed out of either abstract or concrete primitives. This is in contrast to the approach used with algebraic or property-oriented specification languages like CASL [10], which identifies the interface of a software module, consisting of sorts and functions, and states the properties of the interface components using first-order formulas.

Specifications written in Z are structured using schemata and operations on schemata. A schema denotes a set of bindings of the form $\{(x_1, v_1), \ldots, (x_n, v_n)\}$. Operations on schemata include restriction of the elements of a schema to those satisfying a formula; logical operations like negation, conjunction, disjunction, and quantification; and renaming and hiding of the components of a schema. Schemata, and thus the structuring mechanism of Z, are elements of the logic

[⋆] This research has been partially supported by ESPRIT working group 29432 (CoFI WG).

used by Z. This, on one hand, has the advantage of using Z again to reason about the structure of a specification, but, on the other hand, has the disadvantage that development methods and theoretical results referring to the structure of specifications cannot be easily transfered to other specification languages based on different logics.

In contrast, the structuring primitives of property-oriented specification languages can be formulated independent from the logic underlying the particular specification language. This is done by using the notion of an institution introduced by Goguen and Burstall [6] to formalize the informal notion of a logical system. The building blocks of specifications are abstract datatypes, which consist of an interface and a class of possible implementations of that interface. Operations on abstract datatypes are the restriction of the implementations to those satisfying a set of formulas; the union of abstract datatypes; and hiding, adding and renaming of interface components. What exactly constitutes the components of an interface and how they are interpreted in implementations depends on the institution underlying the specification language. For example, in the institution of equational logic the components of an interface are sorts and operations. The implementations interpret the sorts as sets and the operations as functions on these sets.

The goal of this paper is to formalize the relationship between schemata and abstract datatypes, and to show a correspondence between the operations on abstract datatypes and operations on schemata. This relationship can be used to transfer results and methods from Z to property-oriented specification languages and vice versa. For example, the Z-style for the specification of sequential systems can be transfered to property-oriented specification languages [2]. Further, the correspondence between operations on abstract datatypes and operations on schema suggests new operations on abstract datatypes like negation and disjunction.

However, we cannot compare schemata with abstract datatypes in an arbitrary institution; instead, we have to define first an institution $\mathcal{S}$ which formalizes the notion of the set-theory used in Z, and then compare schemata with abstract datatypes in this institution. The definition of the institution $\mathcal{S}$ has the further advantage that it can be used to define a variant of the specification language CASL, CASL-$\mathcal{S}$, based on set-theory instead of order-sorted partial first-order logic. This is possible because the semantics of most of CASL is largely independent from a particular institution (cf. Mossakowski [9]).

2 Institutions and Abstract Datatypes

The notion of institutions attempts to formalize the informal notion of a logical system, and was developed by Goguen and Burstall [6] as a means to define the semantics of the specification language Clear [4] independent from a particular logic.

Definition 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 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* $\sigma : \Sigma \to \Sigma'$ *the reduct 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*

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

We may write $M \models^{\mathcal{I}} \varphi$ for a class of Σ-structures M and a Σ-formula φ instead of $\forall m \in M\colon\ m \models^{\mathcal{I}} \varphi$, and similar for $m \models^{\mathcal{I}} \Phi$ and $M \models^{\mathcal{I}} \Phi$ for a set of Σ-formulas Φ and a Σ-structure m.

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 $\mathsf{Str}_{\mathcal{I}}(\Sigma)$.

The basic operations on abstract datatypes are I_Φ (impose), D_σ (derive), T_σ (translate), and $+$ (union) (cf. Sannella and Wirsing [11]):

Impose allows to impose additional requirements on an abstract datatype. The semantics of an expression $I_\Phi(\Sigma, M)$ is the abstract datatype (Σ, M') where M' consists of all Σ-structures m in M satisfying all formulas in Φ, i.e.

$$I_\Phi(\Sigma, M) = (\Sigma, \{m \in M \mid m \models^{\mathcal{I}} \Phi\}).$$

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 $T_\sigma(\Sigma, M)$ denotes an abstract datatype (Σ', M') where M' contains all Σ'-structures m which are extensions of some Σ-structure m in M, i.e.

$$T_\sigma(\Sigma, M) = (\Sigma', \{m' \in \mathsf{Str}_{\mathcal{I}}(\Sigma') \mid m'|_\sigma \in M\}).$$

The *derive* operation allows to hide parts of a signature. $D_\sigma(\Sigma', M')$ denotes the abstract datatype having as signature the domain of σ and as models the translations of the models of $\textsc{Sp}$ by $\_|_\sigma$, i.e.

$$D_\sigma(\Sigma', M') = (\Sigma, \{m'|_\sigma \mid m' \in M'\}).$$

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. To form the union of two specifications of different signatures Σ_1 and Σ_2, one has to provide a signature Σ and signature morphisms $\sigma_1 : \Sigma_1 \to \Sigma$ and $\sigma_2 : \Sigma_2 \to \Sigma$, and write $T_{\sigma_1}\mathrm{S{\small P}}_1 + T_{\sigma_2}\mathrm{S{\small P}}_2$. The semantics of $(\Sigma, M_1) + (\Sigma, M_2)$ is the abstract datatype (Σ, M') where M' is the intersection M_1 and M_2, i.e.

$$(\Sigma, M_1) + (\Sigma, M_2) = (\Sigma, M_1 \cap M_2).$$

3 The Institution $\mathcal{S}$

In this section we introduce the components of the institution $\mathcal{S}$ formalizing a reasonable large subset of the logic underlying the specification language Z from the Z standard [13]. What is missing, for example, are generic definitions. Other constructs, like the free type construct, can be easily added and their semantics defined by transformation as it is done in the Z standard. We also don't treat the types and operations defined in the prelude and the mathematical toolkit, e.g. natural numbers; these can be easily added if needed.

Note that this is not an attempt to give a semantics to the Z specification language. The relationship between $\mathcal{S}$ and Z is similar to the relationship between the institution of equational logic and the semantics of a specification language based on this institution.

3.1 Signatures

A signature Σ in $\mathrm{S{\small IGN}}_{\mathcal{S}}$ consists of a set of names for *given-types* G and a set of *global-variables* O. Each global-variable id in O is associated with a type $\tau(id)$ built from the names of given-types and the constructors cartesian product, power-set, and schema-type. Note that $\mathcal{S}$ has no type constructor 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 arguments the graph of a function and also return the graph of a function.

Definition 2 (Signatures). *Let F and V be two disjoint, recursive enumerable sets of names. A signature Σ in $\mathrm{S{\small IGN}}_{\mathcal{S}}$ is a tuple (G, O, τ) where G and O are finite disjoint subsets of F. The function τ assigns each name in O a type in $\mathcal{T}(G)$, where $\mathcal{T}(G)$ is inductively defined by:*

- *$G \subseteq \mathcal{T}(G)$*
- *(product-type) $T_1 \times \cdots \times T_n \in \mathcal{T}(G)$ for $T_i \in \mathcal{T}(G)$, $1 \le i \le n$*
- *(power-set-type) $\mathcal{P}(T) \in \mathcal{T}(G)$ 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 $x_i \in V$ and $x_i \neq x_j$ for $1 \le i, j \le n$.*

Note that the elements of $\mathcal{T}(G)$ are names of types and not sets; therefore the type constructors $\mathcal{P}(\_)$ and $\times$ should not be confused with the familiar operations on sets.

The function $\mathcal{T}$, mapping a set of given-type names G to $\mathcal{T}(G)$, is extended to a functor from SET to SET by extending the function $f : G \to G'$ to a function $\mathcal{T}(f) : T(G) \to T(G')$ as follows:

– $\mathcal{T}(f)(g) = f(g)$ 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-types and the set of global-variables.

Definition 3 (Signature-Morphisms). *A signature morphism σ from a signature (G, O, τ) to a 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 $\tau; \mathcal{T}(\sigma_G) = \sigma_O; \tau'$.*

The category SIGN$_\mathcal{S}$ has as objects signatures $\Sigma = (G, O, \tau)$ and as morphisms signature morphisms $\sigma = (\sigma_G, \sigma_O)$ as defined above.

Example 1. As an example consider the following small Z specification of a bank account which defines a given-type *Integer*, a global-variable $+$, and a schema *ACCOUNT*:

$[Integer]$

| $+ : Integer \times Integer \to Integer$

```
┌─ACCOUNT──────────────────────
│ bal : Integer
└──────────────────────────────
```

The signature of this specification is $\Sigma = (\{Integer\}, \{+, ACCOUNT\}, \tau)$ where τ maps $+$ to the type $\mathcal{P}(Integer \times Integer \times Integer)$ and $ACCOUNT$ to the type $\mathcal{P}(<bal : Integer>)$. Note that the function type of $+$ is translated to the type $\mathcal{P}(Integer \times Integer \times Integer)$ of its graph.

A property necessary for writing modular specifications is the cocompleteness of the category of signatures of an institution.

Theorem 1. *The category* SIGN$_\mathcal{S}$ *is finitely cocomplete.*

The colimit of a functor $F : J \to$ SIGN$_\mathcal{S}$ is given by the colimits of the sets of given-type names and the sets of global-variables. Let $F(i) = (G_i, O_i, \tau_i)$. If the functor F_G from J to SET is defined by $F_G(i) = G_i$ and the functor F_O from J to SET by $F_O(i) = O_i$, then the colimit (G, O, τ) of F is given by the colimit G of F_G and the colimit O of F_O. The function τ is uniquely determined by the colimit property of G. Note that, because we have assumed that the set of given-type names and the set of global-variables are finite, SIGN$_\mathcal{S}$ is only finitely cocomplete

3.2 Structures

Given a signature $\Sigma = (G, O, \tau)$, a Σ-structure A interprets each given-type in G as a set from SET and each global-variable id in O as a value of the set corresponding to the type of id.

Definition 4 (Σ-structures). *For a given signature $\Sigma = (G, O, \tau)$ the category $\mathsf{Str}_{\mathcal{S}}(\Sigma)$ 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 $v_i \in \bar{A}_G(\tau(o_i))$. 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 $<x_1 : T_1, \ldots, x_n : T_n> \in \mathcal{T}(G)$.

Example 2. An example of a structure A over the signature defined in Ex. 1 consists of a function A_G mapping *Integer* to $\mathbb{Z}$ and the set

$$A_O = \{(ACCOUNT, \{\{(bal, n)\} \mid n \in \mathbb{Z}\}), (+, graph(\lambda(x, y).x + y))\}.$$

The notation $graph(f)$ is used to denote the graph of a function $f : T \to T'$.

A morphism h from a Σ-structure A to a Σ-structure B is a family of functions between the interpretations of the given-types which is compatible with the interpretations of the global-variables in O.

Definition 5 (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. $\bar{h}$ is the extension of $h : A_G \Rightarrow B_G$ to $h : \bar{A}_G \Rightarrow \bar{B}_G$ given by:*

- $\bar{h}_T(v) = h_T(v)$ *for* $T \in G$ *and* $v \in \bar{A}_G(T)$
- $\bar{h}_T((v_1, \ldots, v_n)) = (\bar{h}_{T_1}(v_1), \ldots, \bar{h}_{T_n}(v_n))$ *for* $T = T_1 \times \cdots \times T_n \in \mathcal{T}(G)$ *and* $(v_1, \ldots, v_n) \in \bar{A}_G(T)$
- $\bar{h}_T(S) = \{\bar{h}_{T'}(v) \mid v \in S\}$ *for* $T = \mathcal{P}(T') \in \mathcal{T}(G)$ *and* $S \in \bar{A}_G(T)$
- $\bar{h}_T(\{(x_1, v_1), \ldots, (x_n, v_n)\}) = \{(x_1, \bar{h}_{T_1}(v_1)), \ldots, (x_n, \bar{h}_{T_n}(v_n))\}$
  for $T = <x_1 : T_1, \ldots, x_n : T_n> \in \mathcal{T}(G)$ *and* $\{(x_1, v_1), \ldots, (x_n, v_n)\} \in \bar{A}_G(T)$

Definition 6 (σ-reduct). *Given a signature morphism σ from $\Sigma = (G, O, \tau)$ to $\Sigma' = (G', O', \tau')$ in* SIGN$_{\mathcal{S}}$ *and a Σ'-structure $A = (A_G, A_O)$, the σ-reduct of A, written $A|_\sigma$, is the Σ-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 7 ($\mathsf{Str}_{\mathcal{S}}$)**.** *The contravariant functor* $\mathsf{Str}_{\mathcal{S}}$ *from* $\textsc{Sign}_{\mathcal{S}}$ *to* $\textsc{Cat}$ *assigns to each signature* Σ *the category having as objects* Σ*-structures and as morphisms* Σ*-homomorphisms, and to each* $\textsc{Sign}_{\mathcal{S}}$*-morphism* σ *from* Σ *to* Σ' *the functor from the category* $\mathsf{Str}_{\mathcal{S}}(\Sigma')$ *to the category* $\mathsf{Str}_{\mathcal{S}}(\Sigma)$ *mapping a* Σ*-structure* A *and a* Σ*-homomorphism to their* σ*-reduct.*

If an institution has amalgamation, two structures A and B over different signatures Σ_A and Σ_B can be always combined provided that the common components of both signatures are interpreted the same in A and B. This allows to build larger structures from smaller ones in a modular way. An institution has amalgamation if and only if its structure functor preserves pushouts, i.e. maps pushout diagrams in $\textsc{Sign}_{\mathcal{I}}$ to pullback diagrams in the category of categories. The functor $\mathsf{Str}_{\mathcal{S}}$ not only preserves pushouts but also arbitrary finite colimits.

Theorem 2. *The functor* $\mathsf{Str}_{\mathcal{S}}$ *preserves finite colimits.*

3.3 Expressions

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 S, denote sets of elements of schema-type. Schema-expressions will be discussed later in this paper.

$$\begin{aligned} E ::= {} & id \mid (E,\ldots,E) \mid E.i \mid \texttt{<}x_1 := E,\ldots,x_n := E\texttt{>} \mid E.x \mid E(E) \\ & \mid \{E,\ldots,E\} \mid \{S \bullet E\} \mid \mathcal{P}(E) \mid E \times \ldots \times E \mid S \end{aligned}$$

The function application $E_1(E_2)$ is well-formed 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. Otherwise, if E_1 is the graph of a partial function or not functional at all, then the result of the function application where E_2 is not in the domain of that function or where several results are associated with E_2 in E_1 is not specified in the Z standard [13]. This leaves room for different treatments of undefinedness. A possible choice described in the standard and which we will adopt in this paper is to choose an arbitrary value from $\bar{A}_G(T_2)$ in these cases.

Well-formedness of expressions over a signature $\Sigma = (G, O, \tau)$ is defined wrt. an *environment* $\epsilon = (\Sigma, (X, \tau_X))$ which consists of the signature Σ, a set of variables $X \subset V$, and a function $\tau_X : X \to \mathcal{T}(G)$ mapping a variable to its type.

We use the notation $\epsilon[\texttt{<}x_1 : T_1,\ldots,x_n : T_n\texttt{>}]$ to denote the environment $(\Sigma, (X', \tau'_X))$ given by $X' = X \cup \{x_1,\ldots,x_n\}$ and

$$\tau'_X(id) = \begin{cases} T_i & \text{if } id = x_i \text{ for some } 1 \leq i \leq n \\ \tau_X(id) & \text{else} \end{cases}$$

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 wrt. ϵ is

$$\tau^\epsilon(E) = \begin{cases} \tau_X(id) & \text{if } id \text{ is in } X, \\ \tau(id) & \text{if } id \text{ is in } O, \\ \mathcal{P}(id) & \text{if } id \text{ is in } G. \end{cases}$$

- $E = (E_1, \ldots, E_n)$ and each E_i is well-formed for all $1 \leq i \leq n$. Then $\tau^\epsilon(E) = \tau^\epsilon(E_1) \times \ldots \times \tau^\epsilon(E_n)$.
- $E = E_1.i$, $\tau^\epsilon(E_1) = T_1 \times \ldots \times T_n$ 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 ${<}x_1 : \tau^\epsilon(E_1), \ldots, x_n : \tau^\epsilon(E_n){>}$.
- $E = E_1.x$, $\tau^\epsilon(E_1) = {<}x_1 : T_1, \ldots, x_n : T_n{>}$ 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 $\mathcal{P}(T)$.
- $E = \{S \bullet E_1\}$, S is well-formed and has type $\mathcal{P}({<}x_1 : T_1, \ldots, x_n : T_n{>})$, and E_1 is well-formed with respect to $\epsilon[{<}x_1 : T_1, \ldots, x_n : T_n{>}]$. The type of E is $\mathcal{P}(\tau^{\epsilon'}(E_1))$.
- $E = \mathcal{P}(E_1)$ and E_1 is well-formed. The type of E is $\mathcal{P}(\tau^\epsilon(E_1))$.
- $E = E_1 \times \ldots \times E_n$ 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 a well-formed schema-expression with respect to ϵ (well-formedness of schema-expressions is defined later in this paper.) The type of E is the type of S with respect to ϵ.

Let E be an expression well-formed with respect to an environment $\epsilon = (\Sigma, (X, \tau_X))$, and let $A = (A_G, A_O)$ be a Σ-structure. The semantics of an expression E is given with respect to a *variable binding* β compatible with the environment ϵ. Let $X = \{x_1, \ldots, x_n\}$, a variable binding $\beta = (A, A_X)$ *compatible* with ϵ consists of a Σ-structure A and a set $A_X = \{(x_1, v_1) \ldots (x_n, v_n)\}$ with $v_i \in \bar{A}_G(\tau_X(x_i))$ for all $1 \leq i \leq n$.

If $v = \{(x_1, v_1), \ldots, (x_n, v_n)\}$ is an element of type $T = {<}x_1 : T_1, \ldots, x_n : T_n{>}$ then the notation $\beta[v]$ is used to describe the variable binding (A, A'_X) where (x_i, v_i) is in A'_x iff (x_i, v_i) is in v or (x_i, v_i) is in A_X but not in v.

Now the semantics of an expression E wrt. β 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.
- $[\![(E_1, \ldots, E_n)]\!]^\beta = ([\![E_1]\!]^\beta, \ldots, [\![E_n]\!]^\beta)$.
- $[\![E.i]\!]^\beta = v_i$ 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 T_2 is the co-domain of the E_1, that is, $\tau^\epsilon(E_1) = \mathcal{P}(T_1 \times T_2)$.

- $[\![\{E_1, \ldots, E_n\}]\!]^\beta = \{[\![E_1]\!]^\beta, \ldots, [\![E_n]\!]^\beta\}$.
- $[\![\{S \bullet E\}]\!]^\beta = \{[\![E]\!]^{\beta[v]} \mid v \in [\![S]\!]^\beta\}$.
- $[\![\mathcal{P}(E)]\!]^\beta = 2^{[\![E]\!]^\beta}$.
- $[\![E_1 \times \ldots \times E_n]\!]^\beta = [\![E_1]\!]^\beta \times \ldots \times [\![E_n]\!]^\beta$.

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*. Thus the *type of a schema* is $\mathcal{P}(<x_1 : T_1, \ldots, x_n : T_n>)$ if T_i is the type of v_i for $1 \leq i \leq n$.

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. Given a schema S, the schema $S|P$ has as elements all the elements of S satisfying the predicate P. Other operations on schemata include forming the negation, disjunction, conjunction and implication of schemata. Negation, disjunction and conjunction 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 to the type of schemas can be made by using hiding and renaming. Hiding hides some components of a schema-type and renaming renames some components. A particular kind of renaming is decorating the identifiers with finite sequences of elements from $\{', !, ?\}$.

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 and the union of the bindings is an element of S_2. An universally quantified schema $\forall S_1.S_2$ is an abbreviation for $\neg\exists S_1.\neg S_2$.

$$\begin{aligned} S ::= {} & x_1 : E, \ldots, x_n : E \mid (S|P) \mid \neg S \mid S \vee S \mid S \wedge S \mid S \Rightarrow S \\ & \mid \forall S.S \mid \exists S.S \mid S \setminus [x_1, \ldots, x_n] \mid S[x_1/y_1, \ldots, x_n/y_n] \\ & \mid S \ Decor \mid E \end{aligned}$$

Note that the schema operations ΔS and ΞS, used in Z for the specification of sequential systems, are only convenient abbreviations for schema expressions involving the schema operations defined above. For example, ΔS is the same as the conjunction of the schema S with S', and ΞS is the same as the schema $S \wedge S'|(x_1 = x_1' \wedge \ldots \wedge x_n = x_n')$ given that the type of S is $\mathcal{P}(<x_1 : T_1, \ldots, x_n : T_n>)$.

A schema-expression S is well-formed with respect to an environment $\epsilon = (\Sigma, (X, \tau_X))$ 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 $\mathcal{P}(<x_1 : T_1, \ldots, x_n : T_n>)$.
- $S = S_1|P$ and P is well-formed with respect to $\epsilon' = \epsilon[T]$, where the type of S_1 is $\mathcal{P}(T)$. The type of S is $\mathcal{P}(T)$.
- $S = \neg S_1$ and S_1 is well-formed. The type of S is $\tau^\epsilon(S_1)$.

- $S = S_1\ op\ S_2$, S_1 and S_2 have compatible types, S_1 and S_2 are well-formed, and $op \in \{\vee, \wedge, \Rightarrow\}$.
  Two types $\mathcal{P}(<x_1 : T_1, \ldots, x_n : T_n>)$ and $\mathcal{P}(<x'_1 : T'_1, \ldots, x'_m : T'_m>)$ are compatible if for all i and 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.
- $S = S_1 \setminus [x_1, \ldots, x_n]$ and S is well-formed. Note that it is not required that the x_i have to be identifiers of the type of S_1. The type of S is the type of S_1 without the identifier x_i if x_i occurs in the type of S for all $1 \leq i \leq n$.
- $S = S_1[x_1/y_1, \ldots, x_n/y_n]$ and S is well-formed. Note that it is not required that the x_i have to be identifiers of the type of S_1. The type of S is the type of S_1 where x_i is replaced by y_i if x_i is an identifier of S_1. Note that the mapping from the identifiers of the type of S_1 to the identifiers of the type of S defined by this replacement has to be injective.
- $S = S_1$ *Decor* and S_1 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_1 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 $\mathcal{P}(<x_1 : T_1, \ldots, x_n : T_n>)$.

Let v be the set $\{(x_1, v_1), \ldots, (x_n, v_n)\}$ and X be a set of variables, then $v|_X$ denotes the binding v restricted to the identifiers in the set X, i.e. the set $\{(x_i, v_i) \mid x_i \in X \wedge (x_i, v_i) \in v \wedge 1 \leq i \leq n\}$.

If a schema-expression S is well-formed with respect to ϵ, its semantics $[\![S]\!]^\beta$ with respect to a structure $A = (A_G, A_O)$ and a variable binding $\beta = (A, A_X)$ compatible with ϵ is defined as follows:

- $[\![x_1 : E_1, \ldots, x_n : E_n]\!]^\beta = \{\{(x_1, v_1), \ldots, (x_n, v_n)\} \mid v_i \in [\![E_i]\!]^\beta,\ 1 \leq i \leq n\}$.
- $[\![S|P]\!]^\beta = \{v \in [\![S]\!]^\beta \mid \beta[v] \models^{\mathcal{S}} P\}$. The satisfaction relation $\models^{\mathcal{S}}$ is defined in Sect. 3.4.
- $[\![\neg S]\!]^\beta = \{v \in \bar{A}_G(T) \mid v \notin [\![S]\!]^\beta\}$ and S has type T.
- $[\![S \setminus [y_1, \ldots, y_n]]\!]^\beta = \{v|_{\{x_1,\ldots,x_m\}} \mid v \in [\![S]\!]^\beta\}$, where $\{x_1, \ldots, x_m\}$ is the set of identifiers of the type of S without the identifiers $y_1, \ldots, y_n$.
- $[\![S_1\ op\ S_2]\!]^\beta = \{v \in \bar{A}_G(T) \mid v|_{X_1} \in [\![S_1]\!]^\beta\ op\ v|_{X_2} \in [\![S_2]\!]^\beta\}$ where op is in $\{\vee, \wedge, \Rightarrow\}$, $\mathcal{P}(T)$ is the type of $S_1\ op\ S_2$, and X_1 and X_2 are the set of components of schemata S_1 and S_2, respectively.
- $[\![\exists S_1.S_2]\!]^\beta = \{v \in \bar{A}_G(T) \mid \exists v_1 \in [\![S_1]\!]^\beta : (v_1 \cup v)|_{X_2} \in [\![S_2]\!]^\beta\}$ where $\mathcal{P}(()T)$ X_2 is the set of components of schema S_2 and $\mathcal{P}(T)$ is the type of $\exists S_1.S_2$.
- $[\![S[y_1/y'_1, \ldots, y_n/y'_n]]\!]^\beta = \{\bar{f}(v) \mid v \in [\![S]\!]^\beta\}$ where f is the function from the identifiers of type S to the identifiers of type S_1 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 canonical extension of f to bindings.
- $[\![S_1\ Decor]\!]^\beta = \{\{(\bar{x}_1, v_1), \ldots, (\bar{x}_n, v_n)\} \mid \{(x_1, v_1), \ldots, (x_n, v_n)\} \in [\![S_1]\!]^\beta\}$. $\bar{x}_i$ is the identifier x_i decorated with $Decor$. For example, if $Decor$ is $'$ then $\bar{x}_i$ is $x_i{}'$.

3.4 Formulas

The formulas in $\mathsf{Sen}_{\mathcal{S}}(\Sigma)$ 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.
- $P = \neg P_1$ and P_1 is well-formed.
- $P = P_1\ op\ P_2$, P_1 and P_2 are well-formed, and $op \in \{\vee, \wedge, \Rightarrow\}$.
- $P = \forall S.P_1$, S is well-formed and has type $\mathcal{P}(T)$ where T is a schema-type and P_1 is well-formed with respect to $\epsilon[T]$.
- $P = \exists S.P_1$, S is well-formed and has type $\mathcal{P}(T)$ where T is a schema-type and P_1 is well-formed with respect to $\epsilon[T]$.

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 $id \in G$.
- $\bar{\sigma}((E_1, \ldots, E_n)) = (\bar{\sigma}(E_1), \ldots, \bar{\sigma}(E_n))$.
- $\bar{\sigma}(E.i) = \bar{\sigma}(E).i$.
- $\bar{\sigma}(<x_1 := E_1, \ldots, x_n := E_n>) = <x_1 := \bar{\sigma}(E_1), \ldots, x_n := \bar{\sigma}(E_n)>$.
- $\bar{\sigma}(E.x) = \bar{\sigma}(E).x$.
- $\bar{\sigma}(E_1(E_2)) = \bar{\sigma}(E_1)(\bar{\sigma}(E_2))$.
- $\bar{\sigma}(\{E_1, \ldots, E_n\}) = \{\bar{\sigma}(E_1), \ldots, \bar{\sigma}(E_n)\}$.
- $\bar{\sigma}(\{S \bullet E\}) = \{\bar{\sigma}(S) \bullet \bar{\sigma}(E)\}$.
- $\bar{\sigma}(\mathcal{P}(E)) = \mathcal{P}(\bar{\sigma}(E))$.
- $\bar{\sigma}(E_1 \times \ldots \times E_n) = \bar{\sigma}(E_1) \times \ldots \times \bar{\sigma}(E_n)$.
- $\bar{\sigma}(x_1 : E_1, \ldots, x_n : E) = x_1 : \bar{\sigma}(E_1), \ldots, x_n : \bar{\sigma}(E_n)$.
- $\bar{\sigma}(S|P) = \bar{\sigma}(S)|\bar{\sigma}(P)$.
- $\bar{\sigma}(\neg S) = \neg\bar{\sigma}(S)$.
- $\bar{\sigma}(S_1\ op\ S_n) = \bar{\sigma}(S_1)\ op\ \bar{\sigma}(S_n)$ for $op \in \{\vee, \wedge, \Rightarrow\}$.
- $\bar{\sigma}(\exists S_1.S_2) = \exists\bar{\sigma}(S_1).\bar{\sigma}(S_2)$ and $\bar{\sigma}(\forall S_1.S_2) = \forall\bar{\sigma}(S_1).\bar{\sigma}(S_2)$.
- $\bar{\sigma}(S \setminus [x_1, \ldots, x_n]) = \bar{\sigma}(S) \setminus [x_1, \ldots, x_n]$.
- $\bar{\sigma}(S[x_1/y_1, \ldots, x_n/y_n]) = \bar{\sigma}(S)[x_1/y_1, \ldots, x_n/y_n]$.
- $\bar{\sigma}(E_1 \in E_2) = (\bar{\sigma}(E_1) \in \bar{\sigma}(E_2))$.
- $\bar{\sigma}(E_1 = E_2) = (\bar{\sigma}(E_1) = \bar{\sigma}(E_n))$.

- $\bar{\sigma}(\text{true}) = \text{true}$ 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 8 ($\mathsf{Sen}_\mathcal{S}$). *The functor* $\mathsf{Sen}_\mathcal{S}$ *from* $\textsc{Sign}_\mathcal{S}$ *to* $\textsc{Set}$ *maps each signature* Σ *to the set formulas well-formed wrt.* $\epsilon = (\Sigma, (\{\}, \tau_X))$ *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 $\beta = (A, A_X)$, $\beta \models^\mathcal{S} P$, is defined by:

- $\beta \models^\mathcal{S}$ true.
- $\beta \models^\mathcal{S} E_1 \in E_2$ iff $[\![E_1]\!]^\beta \in [\![E_2]\!]^\beta$.
- $\beta \models^\mathcal{S} E_1 = E_2$ iff $[\![E_1]\!]^\beta = [\![E_2]\!]^\beta$.
- $\beta \models^\mathcal{S} \neg P$ iff not $\beta \models^\mathcal{S} P$.
- $\beta \models^\mathcal{S} P_1 \ op\ P_2$ iff $\beta \models^\mathcal{S} P_1 \ op\ \beta \models^\mathcal{S} P_2$ for $op \in \{\vee, \wedge, \Rightarrow\}$.
- $\beta \models^\mathcal{S} \forall S.P$ iff $\beta[v] \models^\mathcal{S} P$ for all $v \in [\![S]\!]^\beta$.
- $\beta \models^\mathcal{S} \exists S.P$ iff $\beta[v] \models^\mathcal{S} P$ for some $v \in [\![S]\!]^\beta$.

Definition 9 (Satisfaction). *Given a signature* Σ*, a formula* P *which is well-formed with respect to* $(\Sigma, (\{\}, \tau_X))$*, and a* Σ*-structure* A*, then* $A \models^\mathcal{S}_\Sigma P$ *if* $(A, \{\}) \models^\mathcal{S} P$.

Theorem 3 (The Institution $\mathcal{S}$). *The category* $\textsc{Sign}_\mathcal{S}$*, the functor* $\mathsf{Str}_\mathcal{S}$*, the functor* $\mathsf{Sen}_\mathcal{S}$ *and the family of satisfaction relations given by* $\models^\mathcal{S}_\Sigma$ *define the institution* $\mathcal{S} = \langle \textsc{Sign}_\mathcal{S}, \mathsf{Str}_\mathcal{S}, \mathsf{Sen}_\mathcal{S}, \models^\mathcal{S} \rangle$.

Example 3. To complete our small example of a bank account we define the schema $\Delta ACCOUNT$ and the operation $UPDATE$ adding n to the balance of the account:

$$\Delta ACCOUNT = ACCOUNT \wedge ACCOUNT'$$

$$\begin{array}{|l}
\multicolumn{1}{l}{UPDATE} \\
\Delta ACCOUNT \\
n : Integer \\
\hline
bal' = bal + n \\
\hline
\end{array}$$

The abstract datatype in $\mathcal{S}$ corresponding to this specification consists of the signature:

$$\Sigma_{BA} = (\{Integer\}, \{+, ACCOUNT, \Delta ACCOUNT, UPDATE\}, \tau)$$

where τ is given by

$$\tau(id) = \begin{cases} \mathcal{P}(Integer \times Integer \times Integer) & if\ id = + \\ \mathcal{P}(<bal : Integer>) & if\ id = ACCOUNT \\ \mathcal{P}(<bal : Integer,\ bal' : Integer>) & if\ id = \Delta ACCOUNT \\ \mathcal{P}(<bal : Integer,\ bal' : Integer,\ n : Integer>) & if\ id = UPDATE \end{cases}$$

The following set of formulas specifies the schemata $ACCOUNT$, $\Delta ACCOUNT$ and the $UPDATE$ operation:

$$\Phi = \begin{array}{l}\{ACCOUNT = (bal : Integer),\\ \ \Delta ACCOUNT = ACCOUNT \land ACCOUNT',\\ \ UPDATE = ((\Delta ACCOUNT \land (n : Integer)) \mid bal' = bal + n)\}.\end{array}$$

A Σ_{BA}-structure $A = (A_G, A_O)$ satisfying Φ is given by function A_G mapping *Integer* to $\mathbb{Z}$ and the set

$$A_O = \begin{array}{l}\{(+, graph(\lambda(x,y).x+y)),\\ \ (ACCOUNT, \{\{(bal, x)\} \mid x \in \mathbb{Z}\}),\\ \ (\Delta ACCOUNT, \{\{(bal, x), (bal', y)\} \mid x, y \in \mathbb{Z}\}),\\ \ (UPDATE, \{\{(bal, x), (bal', y), (n, z)\} \mid x, y, z \in \mathbb{Z} \land y = x + z\})\}.\end{array}$$

Now the interpretation of $UPDATE$ in A, denoted by $A(UPDATE)$, defines an operation that given an integer n transforms a state, which is an element of the interpretation of $ACCOUNT$ in A, to another state:

$$(\{(bal, x)\}, z) \mapsto \{(bal, y)\} \text{ iff } \{(bal, x), (bal', y), (n, z)\} \text{ is in } A(UPDATE).$$

4 Relating Abstract Datatypes to Schemata

Let $\Sigma = (G, O, \tau)$ be a signature in $\mathcal{S}$. 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 $A = (A_G, A_O)$, an element $\{(x_1, v_1), \ldots, (x_n, v_n)\}$ of type T defines a Σ'-structure $A' = (A_G, A_O \cup \{(x_1, v_1), \ldots, (x_n, v_n)\})$.

Definition 10. *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 $A = (A_G, A_O)$. We 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* $1 \le i \le n$ *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* $\textsc{Adt}_{\mathcal{S}}$ *by taking the union of all* M_S^A *for* $A \in M$:

$$\textsc{Sp}_S = (\Sigma_S, \bigcup_{A \in M} M_S^A).$$

[1] Note that Σ' is not a signature as defined in Def. 2 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.

Example 4. Given $\Sigma = (\{\mathit{Integer}\}, \{+\}, \tau)$, then the signatures corresponding to the schemata *ACCOUNT*, *ΔACCOUNT*, and *UPDATE* are:

$$\begin{aligned}
\Sigma_A &= (\{\mathit{Integer}\}, \{+, \mathit{bal}\}, \tau_A),\\
\Sigma_{\Delta A} &= (\{\mathit{Integer}\}, \{+, \mathit{bal}, \mathit{bal}'\}, \tau_{\Delta A}),\\
\Sigma_U &= (\{\mathit{Integer}\}, \{+, \mathit{bal}, \mathit{bal}', n\}, \tau_U).
\end{aligned}$$

The next theorem relates the operations on schemata with the operations on abstract datatypes:

Theorem 4. *Let* $\textsc{Sp} = (\Sigma, M)$ *be an abstract datatype in* $\mathcal{S}$*. If*

- $S = x_1 : E_1, \ldots, x_n : E_n$*, then* $\textsc{Sp}_S = I_{\{x_i \in E_i | 1 \le i \le n\}} T_\sigma \textsc{Sp}$ *where* σ *is the inclusion of* Σ *into* Σ_S*.*
- $S = S_1 | P$*, then* $\textsc{Sp}_S = I_{\{P\}} \textsc{Sp}_{S_1}$*.*
- $S = S_1 \wedge S_2$*, then* $\textsc{Sp}_S = T_{\sigma_1} \textsc{Sp}_{S_1} + T_{\sigma_2} \textsc{Sp}_{S_2}$*. 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 *in the union of* S_1 *and* S_2 *are only required to be compatible.*
- $S = S_1 \setminus [x_1, .., x_n]$*, then* $\textsc{Sp}_S = D_\sigma \textsc{Sp}_{S_1}$ *where* σ *is the inclusion of* Σ_S *into* Σ_{S_1}*.*
- $S = S_1[x_1/y_1, .., x_n/y_n]$*, then* $\textsc{Sp}_S = T_\sigma \textsc{Sp}_{S_1}$ *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* $x \ne x_i$ *for all* i*.*

Example 5. Given $\textsc{Sp} = (\Sigma, M)$ and $\mathit{UPDATE} = (\Delta \mathit{ACCOUNT} \wedge (n : \mathit{Integer}) \mid \mathit{bal}' = \mathit{bal} + n)$ we can write $\textsc{Sp}_U = (\Sigma_U, M_U)$ as:

$$\textsc{Sp}_U = I_{\{\mathit{bal}'=\mathit{bal}+n\}}(T_{\sigma_1} \textsc{Sp}_{\Delta A} + T_{\sigma_2} I_{\{n \in \mathit{Integer}\}} T_{\sigma_3} \textsc{Sp}).$$

Here, σ_1 is the inclusion of $\Sigma_{\Delta A}$ into Σ_U, σ_3 the inclusion of Σ into $\Sigma_{(n:\mathit{Integer})}$, and σ_2 the inclusion of $\Sigma_{(n:\mathit{Integer})}$ into Σ_U. $\Sigma_{(n:\mathit{Integer})} = (\{\mathit{Integer}\}, \{+, n\}, \tau')$ is the signature corresponding to the schema $(n : \mathit{Integer})$.

What about the other schema operations $\neg S$, $S_1 \vee S_2$, $S_1 \Rightarrow S_2$, and $\exists S_1 . S_2$? The existential quantifier is the same as hiding the schema variables of S_1 in the conjunction of S_1 and S_2. Let $x_1, \ldots, x_n$ be the schema variables of S_1, then $\exists S_1 . S_2$ and $(S_1 \wedge S_2) \setminus [x_1, .., x_n]$ have the same semantics. This yields the following theorem:

Theorem 5. *Let* $\textsc{Sp} = (\Sigma, M)$ *be an abstract datatype in* $\mathcal{S}$*, and* $S = \exists S_1 . S_2$ *a well-formed schema expression. Then*

$$\textsc{Sp}_S = D_\sigma(T_{\sigma_1} \textsc{Sp}_{S_1} \wedge T_{\sigma_2} \textsc{Sp}_{S_2})$$

where σ_1 *and* σ_2 *are the inclusions of* Σ_{S_1} *and* Σ_{S_2} *into* $\Sigma_{S_1 \wedge S_2}$*, and* σ *is the inclusion of the signature of the whole expression into* $\Sigma_{S_1 \wedge S_2}$*.*

It is easy to define negation, disjunction and implication on abstract datatypes:

Definition 11. *Let (Σ, M), (Σ, M_1) and (Σ, M_2) be abstract datatypes in an arbitrary institution $\mathcal{I}$, define:*

$$\begin{aligned}\neg(\Sigma, M) &= (\Sigma, \{m \in \mathsf{Str}_{\mathcal{I}}(\Sigma) \mid m \notin M\})\\ (\Sigma, M_1) \vee (\Sigma, M_2) &= (\Sigma, M_1 \cup M_2)\\ (\Sigma, M_1) \Rightarrow (\Sigma, M_2) &= (\Sigma, \{m \in \mathsf{Str}_{\mathcal{I}}(\Sigma) \mid m \in M_1 \Rightarrow m \in M_2\})\end{aligned}$$

What is the relationship of these operations to the corresponding schema operations? Disjunction can be treated similar to conjunction; however, while it seems natural to expect $\textsc{Sp}_{\neg S} = \neg\textsc{Sp}_S$, this does not hold. The reason is that in $\textsc{Sp}_{\neg S}$ the negation of S is interpreted within a given abstract datatype $\textsc{Sp}$ while the negation of $\textsc{Sp}_S$ also permits the negation of $\textsc{Sp}$ itself. If $(A_G, A_O \cup v)$ is a model of $\textsc{Sp}_{\neg S}$, then v is not in $[\![S]\!]^\beta$ and (A_G, A_O) is always a model of $\textsc{Sp}$. On the other hand, if $(A_G, A_O \cup v)$ is a model of $\neg\textsc{Sp}_S$, either v is not in $[\![S]\!]^\beta$ or (A_G, A_O) is not a model of $\textsc{Sp}$. The solution is to add the requirement that (A_G, A_O) is a model of $\textsc{Sp}$ to $\neg\textsc{Sp}_S$. Implication has a similar problem.

Theorem 6. *Let $\textsc{Sp} = (\Sigma, M)$ be an abstract datatype in $\mathcal{S}$. If*

- *$S = S_1 \vee S_2$, then $\textsc{Sp}_S = T_{\sigma_1}\textsc{Sp}_{S_1} \vee T_{\sigma_2}\textsc{Sp}_{S_2}$. The signature morphisms σ_1 and σ_2 are the inclusions of the signatures Σ_{S_1} and Σ_{S_2} into $\Sigma_{S_1 \vee S_2}$.*
- *$S = \neg S_1$, then $\textsc{Sp}_S = \neg\textsc{Sp}_{S_1} + T_{\sigma_{S_1}}\textsc{Sp}$ where σ_S is the inclusion of the Σ into Σ_S.*
- *$S = S_1 \Rightarrow S_2$, then $\textsc{Sp}_S = (T_{\sigma_1}\textsc{Sp}_{S_1} \Rightarrow T_{\sigma_2}\textsc{Sp}_{S_2}) + T_{\sigma_S}\textsc{Sp}$. The signature morphisms σ_1 and σ_2 are the inclusions of the signatures Σ_{S_1} and Σ_{S_2} into $\Sigma_{S_1 \Rightarrow S_2}$.*

5 Conclusion

In this paper we have formalized the relationship between the structuring mechanism in Z and the structuring mechanism of property-oriented specification languages. Z specifications are structured using schemata and operations on schemata, which are based on the particular logic underlying Z. In contrast, property-oriented specifications are structured using abstract datatypes and operations on abstract datatypes, which can be formulated largely independent of the logic used for the specifications.

The advantage of having the structuring mechanism represented as part of the logic is that it is possible to reason within that logic about the structure of specifications. The disadvantage is that it is not easy to transfer results and methods to be used with a different logic and specification language. For example, the specification of sequential systems in Z consists of a schema for the state space and a schema for each operation. In the example of the bank account the schema $ACCOUNT$ defines the state space of the bank account, and the schema $UPDATE$ defines the update operation that changes the state of the account. Using the results of this paper we can use abstract datatypes instead of schemata for the specification of sequential systems and the bank account specification can be written without the use of schemata as a CASL-$\mathcal{S}$ specification as follows:

spec *BASE* =
  sort *Integer*
  op $+ : \mathcal{P}(\mathit{Integer} \times \mathit{Integer} \times \mathit{Integer})$

spec *ACCOUNT* = *BASE* **then**
  op *bal*: *Integer*

spec *ΔACCOUNT* = *ACCOUNT* **and {** *ACCOUNT* **with** $\mathit{bal} \mapsto \mathit{bal}'$ **}**

spec *UPDATE* = *ΔACCOUNT* **then**
  op n : *Integer*
  axioms $\mathit{bal}' = \mathit{bal} + n$

Note that this specification does not make any reference to schemata anymore. Instead of schemata the structuring facilities of CASL-$\mathcal{S}$ are used. Since these structuring facilities are institution independent[2], this allows the use of the Z-style for the specification of sequential systems also with other specification languages. For example, this specification style can be used in the state as algebra approach (e.g. [1, 2, 5, 7]).

In the process of relating schemata and their operations to abstract datatypes we have defined the operations negation, disjunction and implication on abstract datatypes, which were previously not defined. Further work needs to be done to study the relationship of these new operations with the other operations on abstract datatypes, and how to integrate the new operations into proof calculi, like that of Hennicker, Wirsing, and Bidoit [8]. Work in this direction has been done for the case of disjunction in Baumeister [3].

References

1. Hubert Baumeister. Relations as abstract datatypes: An institution to specify relations between algebras. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT 95, Proceedings of the Sixth Joint Conference on Theory and Practice of Software Development*, number 915 in LNCS, pages 756–771, Århus, Denmark, May 1995. Springer.
2. 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.
3. Hubert Baumeister. *Relations between Abstract Datatypes modeled as Abstract Datatypes*. PhD thesis, Universität des Saarlandes, Saarbrücken, May 1999.
4. R. M. Burstall and J. A. Goguen. The semantics of Clear, a specification language, February 1980.
5. Hartmut Ehrig and Fernando Orejas. Dynamic abstract data types, an informal proposal. *Bulletin of the EATCS*, 53:162–169, June 1994.

[2] To be precise, CASL is parameterized by the notion of an institution with symbols (cf. Mossakowski [9]). However, it is easy to show that $\mathcal{S}$ is an institution with symbols.

6. 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.
7. Yuri Gurevich. Evolving algebras: An attempt to discover semantics. *Bulletin of the EATCS*, 43:264–284, February 1991.
8. 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.
9. Till Mossakowski. Specifications in an arbitrary institution with symbols. In Didier Bert and Christine Choppy, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Bonas, France, September 1999; Selected Papers;*, volume 1827 of *LNCS*. Springer, 2000.
10. 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.
11. 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.
12. 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.
13. Z notation, final committee draft, cd 13568.2, August 24 1999. Available at `http://web.comlab.ox.ac.uk/oucl/research/groups/zstandards/index.html`.