Syntactic Theory Functors for Specifications with Partiality

Magne Haveraaen[1], Markus Roggenbach[2], and Håkon Robbestad Gylterud[1]

[1] Dept. of Informatics, University of Bergen, Norway
[2] Dept. of Computer Science, Swansea University, United Kingdom
magne.haveraaen@uib.no, m.roggenbach@swan.ac.uk, hakon.gylterud@uib.no

Abstract

Here we suggest using "syntactic theory functors" (STFs) as a means to organise algebraic specifications of partial data types. Algebraic specifications of partial data types often introduce much syntactic clutter and is error prone in itself. STFs are a syntactic structuring mechanism for specifications, allowing systematic treatment of such specifications to avoid the pitfalls. As a syntactic structuring mechanism the STF can be left in the specification text to indicate how the partial specification will be encoded. The STF can also be expanded to expose the full specification text with all the detail and clutter in place.

1 Motivation

Algebraic specifications are a useful formalism for defining APIs and generic code. Such specification formalisms include the very simple equational and conditional equational logics, the logic of boolean expressions (quantifier free first order predicate logic) familiar to software developers, and standard first order predicate logic which is preferred by pre/post specifications. Thus algebraic specifications subsume pre/post specifications, which are focussed on the individual algorithm and not the API as such.

The weakness of algebraic specifications is in handling partiality or preconditions. While algebraic specifications are good at defining properties like what is the first element of a queue, or that two operations are inverses, like popping versus pushing an element on a stack, or division versus multiplication, they are not good for defining special cases like the first element of an empty queue, popping an empty stack, or dividing by zero. The former can be handled by choosing some arbitrary designated value as the first element of an empty queue, popping an empty stack can be kludged to be an empty stack, but division by zero has no viable answer since zero is an absorbing element for multiplication and thus causes deeper problems.

There are many approaches to partiality in algebraic specifications, see [4] for a discussion. These approaches turn out to introduce "syntactic clutter" in the machinery used for dealing with partiality, making it difficult to convey the important properties of an API, and can be very error prone in getting right. Here is a straight forward stack specification.

```
specification Stack = type S, E;
  function new: -> S;        function push: S, E -> S
  function pop: S -> S;      function top: S -> E
  axiom s:S, e:E pop(push(s,e)) = s;       axiom s:S, e:E top(push(s,e)) = e
```

The error algebra version deals with the problematic cases of the empty stack.

```
specification Stack_error = type S, E
  function new: -> S;        function push: S, E -> S
  function pop: S -> S;      function top: S -> E
  % New error constants
```

```
function error_s : -> S;   function error_e : -> E
% New axioms defining the error situations
axiom pop(new()) = error_s();            axiom top(new()) = error_e()
% Error propagation axioms for each of the operations of the Stack signature
axiom s:S push(s,error_e()) = error_s(); axiom e:E push(error_s(),e) = error_s()
axiom pop(error_s()) = error_s();        axiom top(error_s()) = error_e()
% Protected axioms from the Stack specification
axiom s:S, e:E s!=error_s() && e!=error_e() => pop(push(s,e)) = s
axiom s:S, e:E s!=error_s() && e!=error_e() => top(push(s,e)) = e
```

Here we can deduce that `pop(push(s,error_e()))=error_s()`, but cannot deduce that `pop(push(s,error_e()))=s`. Without this care in introducing the appropriate prerequisits, the latter deduction would imply that `s=error_s()` for all stacks `s`.

We suggest applying "syntactic theory functors" (STFs) as a means to organise such specifications. STFs are a syntactic structuring mechanism for specifications and were introduced at NWPT'18. They allow a systematic manipulation of declarations, adding new and modifying existing axioms of a specification. As syntactic structuring mechanisms they can be kept in the specification text thus indicating which partiality mechanism will be used, or they can be expanded giving the correspondingly "cluttered" specification in the semantically intended way. Careful design of the STFs will ensure the intended specification, avoiding inadvertent mistakes.

2 Syntactic Theory Functors

Syntactic theory functors are an institution [1] independent syntactic structuring mechanism. STFs work on discrete institutions, i.e., any logic with a model theory, e.g., a programming system, consisting of

- Interface declarations, called *signatures* $\mathbf{Sig}_{\mathrm{id}}$ or APIs.
- A specification formalism for(Σ) which defines the set of *formulae* for each signature Σ.
- A collection of *models* mod(Σ) for each signature.
- A *satisfaction relation* $\models_\Sigma \subseteq \mathrm{mod}(\Sigma) \times \mathrm{for}(\Sigma)$ that defines which models satisfy a formulae.

A *theory* $\langle \Sigma, \Phi \rangle$ consists of a signature Σ and a set of formulae $\Phi \subseteq \mathrm{for}(\Sigma)$. A specification structuring mechanism is syntactic if, when applied to a theory, it can by expanded (flattened) to a theory, i.e., it can be seen as a mapping from theories to theories. The STFs are theory mappings with additional requirements.

Definition 1 (Syntactic Theory Functor (STF))**.** *Let* $\mathbf{Th}_{\mathrm{id}}$ *denote the theories of an institution. A mapping* $F : \mathbf{Th}_{\mathrm{id}} \to \mathbf{Th}_{\mathrm{id}}$ *from theories to theories is a* syntactic theory functor *if the application of* F *on theories can be decomposed*

$$F(\Sigma, \Phi) = \langle F_{\mathtt{sig}}(\Sigma), F_{\mathtt{base}}(\Sigma) \cup F_{\mathtt{for},\Sigma}(\Phi) \rangle$$

where

- $F_{\mathtt{sig}} : \mathbf{Sig}_{\mathrm{id}} \to \mathbf{Sig}_{\mathrm{id}}$ *is a mapping of signatures to signatures,*
- $F_{\mathtt{base}} : \mathbf{Sig}_{\mathrm{id}} \to \mathbf{Set}$ *is a mapping from signatures to sets of formulae with* $F_{\mathtt{base}}(\Sigma) \subseteq \mathrm{for}(F_{\mathtt{sig}}(\Sigma))$*, and*

- $F_{\mathtt{for},\Sigma} : \mathrm{for}(\Sigma) \to \mathrm{for}(F_{\mathtt{sig}}(\Sigma))$ *is a function for every signature* Σ.

STFs are additive, i.e., $F(\Sigma, \Phi_1 \cup \Phi_2) = F(\Sigma, \Phi_1) \cup F(\Sigma, \Phi_2)$. STFs compose, i.e., given STFs F and G, then $F;G$ is also an STF. Some STFs are model consistent, i.e., whenever $F(\Sigma, \Phi) = \langle \Sigma', \Phi' \rangle$, then there is a mapping $\mu : \mathrm{mod}(\Sigma') \to \mathrm{mod}(\Sigma)$ s.t. $\mu(M') \models_\Sigma \varphi \iff M' \models F(\Sigma, \{\varphi\})_2$. Model consistent STFs build institutions with non-trivial morphisms between signatures and corresponding morphisms on formulae and models. Such institutions have a very well behaved translation of formulae and models between different signatures. STFs that do not build institutions may have a weaker reuse of formulae, e.g., they may preserve or reflect only some kind of formulae from one signature to another. The STFs for partial specifications will normally not be model consistent, as the purpose is to allow models which are less constrained for the arguments that break preconditions than for those arguments that satisfy the precondition—a flexibility not allowed by the source specification.

3 Approaches to partiality and corresponding STFs

In the paper [4] Peter Mosses presents succinctly a list of approaches to partial specifications: error algebras, ok-predicates, exception algebras, labeled algebras, various forms of order sorted algebras, and classified algebras. We will develop STFs for most of these formalisms, showing that the notational complexity and error proneness of writing these specifications can be handled nicely. Finally we will look into guarded algebras [2].

To illustrate the approach we will sketch an error STF. The error STF is parameterised by a listing earg of the error constants and the axioms that returns the error element. It does the following transformation to a specificiation.

- $\mathrm{error}_{\mathrm{earg,sig}}$ create a new signature where the provided error constants have been added to the signature.
- $\mathrm{error}_{\mathrm{earg,base}}$ add the new axioms and error propagation rules for each of the existing operations.
- $\mathrm{error}_{\mathrm{earg,for}}$ modify each existing axiom by prepending checks for the relevant error elements.

If the given parameter list does not appropriately match the argument theory, the error STF will provide relevant error messages. For the stack example the parameter list might be:

```
earg = type S, E; function error_s : -> S; function error_e : -> E % The error elements
  % Axioms defining the error situations
  function new: -> S;          function pop: S -> S;          function top: S -> E
  axiom pop(new()) = error_s(); axiom top(new()) = error_e()
```

Expanding the error STF on the `Stack` specification yields the `Stack_error` specification above.

4 Tools for reasoning

The translations for handling partiality in algebraic specifications typically involve enlarging the API (signatures) of the specification, adding some axioms based on the declarations, and modifying axioms by introducing conditionals.

Now many of the standard reasoning tools have problems dealing with large sets of conditional formulae. Here we discuss some of these difficulties and point to tools and methods which alleviate these problems.

5 Summary

In this presentation we will demonstrate how STFs can simplify writing partial specifications, making them more readable and comprehensible by not cluttering up the specifications. We show this for a range of approaches to partiality in algebraic specifications.

We also give examples of tool support for reasoning about such partial specifications, and prove/disprove simple claims about the transformed specifications at the STF level.

References

[1] J.A. Goguen and R.M. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.

[2] M. Haveraaen and E.G. Wagner. Guarded algebras: Disguising partiality so you won't know whether it's there. In D. Bert, C. Choppy, P. Mosses, editors, *Selected Papers from 14th Int. Workshop on Algebraic Development Techniques, WADT '99*, volume 1827 of *Lecture Notes in Computer Science*, pages 182–200. Springer, 2000.

[3] P. Mosses. *CASL Reference Manual: The Complete Documentation of the Common Algebraic Specification Language*, volume 2960 of *Lecture Notes in Computer Science*. Springer, 2004.

[4] P. Mosses. The use of sorts in algebraic specifications. In M. Bidoit and C. Choppy, editors, *Selected Papers from 8th Workshop on Specification of Abstract Data Types and 3rd COMPASS Workshop, ADT/COMPASS '91*, volume 655 of *Lectures Notes in Computer Science*, pages 66–92. Springer, 1993.