Pandora's Box

Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper and Erik J. Luit

Eindhoven University of Technology, the Netherlands
r.middelkoop@gmail.com,{c.huizing,r.kuiper,e.j.luit}@tue.nl

1 Introduction

Irrespective of the many different implementation paradigms, it is important that client level specifications allow to balance freedom from implementation bias and properly restricting the possible implementations to the ones that the client desires. Algebraic specification of the black-box behavior of a system provides just this, if a careful choice of what comprises desired output is made. We propose a new notion, canonicity, to achieve this.

We take as client specification an algebraic one, in terms of operators from the client's problem domain. Such a specification generally has multiple algebras as semantics. Rather than designating a specific one, e.g., the initial one, as implementation, we stay at the level of abstraction of the client specification: input and output are in terms of combinations of operators from the algebra. We use that each algebra determines which combinations of specified operators are equal to others as a first criterion that input/output combinations have to satisfy to qualify as an implementation. Then we argue that for a client only certain combinations of, in a sense, basic operators are acceptable to occur as output. We then investigate a notion of canonicity to make this precise.

We thus provide a novel syntax and semantics for client specifications. The semantics matches the client's view of the implementation as a black box.

The paper is structured as follows. We concentrate on looking at algebraic specifications from the perspective of the client and the specifier. In Sect. 2.1, we give a brief overview of first-order logic, on which algebraic specifications are based. In Sect. 2.2, we discuss and formalize algebraic specifications. We introduce a syntax and semantics of algebraic specifications that regards the implementation as a black box, and that is independent of the implementation language. After this, we briefly consider the step towards OO implementations, in Sect. 3. Sect. 4 contains some thoughts about the consequences of the approach and about future work.

We first discuss the meta-level notation that is used.

Functions. $f : A \hookrightarrow B$ introduces a partial function from A to B. *Function* is the set of partial functions. Functions are treated as single-valued relations. Given a property $P(a, b)$, we write '$f(a) = b$ if and only if (iff) $P(a, b)$' to define f as the smallest single-valued relation such that $\forall a \in A, b \in B \bullet (a\ f\ b$ iff $P(a, b)$ holds). $f : A \to B$ introduces a total function from A to B. $Domain(f)$ and $Range(f)$ denote the domain and range of (partial or total) function f.
$f[a \mapsto b]$ is the function like f, but with a mapped to b. If f is a partial function, then this can be used whether or not $a \in Domain(f)$. $f[a \mapsto b, \ldots, c \mapsto d]$ is shorthand for $f[a \mapsto b] \ldots [c \mapsto d]$.

Sequences and sets. We use n as the typical element of the set of natural numbers $\mathbb{N}$ (which includes 0). A sequence $A_1, \ldots, A_n$ can be the empty sequence, whereas sequence $A_0, \ldots, A_n$ has at least one element. A *record* is a tuple indexed by names. When a record *Record* with tuple in $A_1 \times \ldots \times A_n$ is indexed by names $f_1, \ldots, f_n$, we write $Record : f_1 \in A_1 \times \ldots \times f_n \in A_n$.

Record.f_i denotes the value of the field with name f_i of *Record*. Given a sequence Σ, $\Sigma[i]$, $\Sigma[i,j]$ and $\Sigma[i..]$ denote element, consecutive subsequence and postfix. $\langle\rangle$ denotes the empty sequence. $\Sigma_0 \triangleright \Sigma_1$ denotes the concatenation of sequences Σ_0 and Σ_1. We write $\overline{x} \in Seq(x)$ to denote that $\overline{x}$ is a sequence of elements from X. Given a set X, we write $xSet \in Set(X)$ to denote that $xSet$ is a set of elements from X. $|A|$ denotes the length of sequence or set A.

Equality. 'A is B' denotes that A and B are syntactically the same. $A = B$ (strong equality) denotes that A and B are both defined and have the same interpretation, i.e., that both evaluate to a value, and that these values are syntactically the same.

2 Specifier's Perspective

In this section, we look at algebraic specifications from the perspective of the client and the specifier.

(1) In Sect. 2.1, we give a brief overview of first-order logic, on which algebraic specifications are based.
(2) In Sect. 2.2, we discuss and formalize algebraic specifications. We introduce a novel syntax and semantics of algebraic specifications that views the implementation as a black box, and that is independent of the implementation language.

2.1 The Formalism: First-Order Logic

Many-sorted first-order logic is at the basis of many program specification and verification techniques. In this section, we give a brief introduction to many-sorted partial first-order logic with equality to fix the notation and terminology that we use. More thorough treatments can be found in, e.g., (CMR98; ST99). Note that operations in the problem domain are often partial (consider e.g. the division operation i/j, which is undefined when $j = 0$). The use of partial logic allows to specify such operations more directly.

2.1.1 Syntax

Here, we formalize the syntax of multi-sorted first order logic.

Sorts, variables and operators. In this paragraph, we introduce the primitive elements of (the syntax of) a first-order logic.

Definition 2.1. *Sort* is the set of *sorts*

Definition 2.2. *Var* is the set of *variables*. Each variable has associated with it a sort S. Var_S denotes the set of variables of sort S. We write v_S to denote that $v_S \in Var_S$.

The function *VarSort* (Def. 2.3) yields the sort of a variable.

Definition 2.3. $VarSort : Var \to Sort$
$VarSort(v_S) = S$.

In Def. 2.4 we define a notion of an operator.

Definition 2.4. *Op* is the set of *operators*. Every operator has associated with it a tuple of *domain sorts* $\langle S_1, \ldots, S_n\rangle$ and a *range sort* S_0, where $S_0, \ldots, S_n \in Sort$. We write $o : S_1 \times \ldots \times S_n \hookrightarrow S_0$ to denote an operator with domain sorts $\langle S_1, \ldots, S_n\rangle$ and range sort S_0.

The functions *DomainSorts* (Def. 2.5) and *RangeSort* (Def. 2.6) yield the domain sorts and the range sort of an operator.

Definition 2.5. $DomainSorts : Op \to Seq(Sort)$
$DomainSorts(o : S_1 \times \ldots \times S_n \hookrightarrow S_0) = \langle S_1, \ldots, S_n \rangle$.

Definition 2.6. $RangeSort : Op \to Sort$
$RangeSort(o : S_1 \times \ldots \times S_n \hookrightarrow S_0) = S_0$.

Definition 2.7. A *constant* is an operator o such that $DomainSorts(o) = \langle\rangle$. *Constant* is the set of all constants.

Signatures. A signature fixes a set of symbols that are used to construct terms. Every signature contains three sets of special-purpose operators (that come with a predefined meaning, see Sect. 2.1.2).

We first introduce the three sets of predefined operators (Def. 2.8-2.11). Then we formally define signatures (Def. 2.12).

Definition 2.8. *BoolOp* is the set that consists of the two constants $true : \hookrightarrow Bool$ and $false : \hookrightarrow Bool$.

Definition 2.9. *LogConOp* is the set that consists of the usual logical connectives (e.g. $\wedge : Bool \times Bool \hookrightarrow Bool$). *LogConOp* also includes, for every $S \in Sorts$, the operator $=_S : S \times S \hookrightarrow Bool$.

Definition 2.10. *QuantOp* is the set that consists of, for every $S \in Sorts$, the operators $\forall_S : S \times Bool \hookrightarrow Bool$ and $\exists_S : S \times Bool \hookrightarrow Bool$.

Definition 2.11. *PredefinedOp* is the set $BoolOp \cup LogConOp \cup QuantOp$.

A signature *sig* (Def. 2.12) is a record that consists of three parts: a set of sorts $sig.sorts$, a set of operators $sig.ops$, and a subset $sig.tOps$ of $sig.ops$, the subset of total operators. $sig.tOps$ contains every predefined operator which has all domain sorts in $sig.sorts$. Note that this includes the set *BoolOp*. Furthermore, for every operator o in $sig.ops$, every domain and range sort of o is a sort from $sig.sorts$. Note that therefore $Bool \in sig.sorts$, as $BoolOp \in sig.ops$.

Definition 2.12. A *signature* is a record $sig : sorts \in Set(Sort) \times ops \in Set(Op) \times tOps \in Set(Op)$ such that
$sig.tOps \subseteq sig.Ops$, and
$\{o \in PredefinedOp \mid DomainSorts(o) \subseteq sig.sorts\} \subseteq sig.tOps$, and
for every $o \in sig.ops$,
$DomainSorts(o) \subseteq sig.sorts$, and
$RangeSort(o) \in sig.sorts$.
Sig is the set of all signatures.

$SigUnion(sig_0, sig_1)$ (Def. 2.13) returns the signature that unites sig_0 and sig_1 in the obvious way.

Definition 2.13. $SigUnion : Sig \times Sig \to Sig$
$SigUnion(sig_0, sig_1) = sig_2$ iff
$sig_2.sorts = sig_0.sorts \cup sig_1.sorts$, and
$sig_2.ops = sig_0.ops \cup sig_1.ops$, and
$sig_2.tOps = sig_0.tOps \cup sig_1.tOps$.

Terms. Terms are constructed from variables and operators.

Definition 2.14. A *term* t (and its sort) is inductively defined as one of the following:
(1) A variable v of a sort S. In this case, the sort of t is S.
(2) A tuple $\langle o, \langle t_1, \ldots, t_n \rangle\rangle$ of (i) an operator $o : S_1 \times \ldots \times S_n \hookrightarrow S_0$, and (ii) a tuple of terms $\langle t_1, \ldots, t_n \rangle$ such that
for every $i \in [1, n]$, t_i has sort S_i, and
if $o \in \mathit{QuantOp}$, then t_1 is a variable.
In this case, the sort of t is S_0 (the range sort of the operator).
Term is the set of all terms.

We usually write a term $\langle o, \langle t_1, \ldots, t_n \rangle\rangle$ as $o(t_1, \ldots, t_n)$. Additionally, when $o \in \mathit{QuantOp}$, we know that n is 2 and t_1 is a variable v_S, and we usually write the term as $o\ v : S \bullet t_2$. Furthermore, we usually write $o()$ as o if it is clear from the context that o is a used as an term and not as an operator. E.g., we often write *true* instead of *true*() and 1 instead of 1(). Finally, we often write certain well-known operators using infix notation. E.g., we may write $4 + 2$ instead of $+(4, 2)$.

An application (Def. 2.15) is a term that is not a variable, and in which no predefined operators occur (the latter is for technical reasons and does not essentially limit the expressive power).

Definition 2.15. An *application* is a term $o_0(t_1, \ldots, t_n)$ in which no $o_1 \in \mathit{PredefinedOp}$ occurs. *Application* is the set of all applications.

We conclude this section with several definitions that are used in the rest of the chapter, most of which are well-known.

Definition 2.16. Variable v_0 is *free* in term t iff (1) t is v_0, or (2) t is $o\ v_1 : S \bullet t_1$ and v_0 is not v_1 and v_0 is free in t_1, or (3) t is $o(t_1, \ldots, t_n)$ and $o \notin \mathit{QuantOp}$ and there is a $i \in [1, n]$ such that v_0 is free in t_i. $\mathit{Free}(t)$ denotes the set of free variables in t.

Definition 2.17. Term t is *closed* if it contains no free variables. *ClosedAppl* is the set of all closed applications.

Definition 2.18. A *predicate* is a term of sort *Bool*. *Predicate* is the set of all predicates.

Definition 2.19. A *sentence* s is a closed predicate. *Sentence* is the set of all sentences.

Terms (Def. 2.20) essentially maps a signature *sig* to the set of all terms that can be built from operators and sorts from *sig*.

Definition 2.20. $\mathit{Terms} : \mathit{Sig} \to \mathit{Set}(\mathit{Term})$
$t \in \mathit{Terms}(\mathit{sig})$ *iff*
or *there are* $S \in \mathit{sig.sorts}$, $v \in \mathit{Var}_S$ *such that* t *is* v,
or *there are* $o \in \mathit{sig.ops}$, $t_1, \ldots, t_n \in \mathit{Terms}(\mathit{sig})$ *such that* t *is* $o(t_1, \ldots, t_n)$.

Definition 2.21. $o_0 \in \mathit{Op}$ *occurs in* $t \in \mathit{Term}$ iff
there are $o_1 \in \mathit{Op}$, $t_1, \ldots, t_n \in \mathit{Term}$ such that
t is $o_1(t_1, \ldots, t_n)$, and
either o_0 is o_1, or there is an $i \in [1, n]$ such that o_0 occurs in t_i.

$\mathit{ClosedAppls}(\mathit{sig})$ (Def. 2.22) is the set of all closed applications (Def. 2.17) in $\mathit{Terms}(\mathit{sig})$.

Definition 2.22. $\mathit{ClosedAppls} : \mathit{Sig} \to \mathit{Set}(\mathit{ClosedAppl})$
$\mathit{ca} \in \mathit{ClosedAppls}(\mathit{sig})$ *iff* $\mathit{ca} \in \mathit{Terms}(\mathit{sig})$ *and* ca *contains no free variables.*

2.1.2 Semantics

In this section we formalize the well-known notions of an algebra and a valuation to define the semantic meaning of a term. More specifically, we define how to evaluate a term to a value (Def. 2.31). We then use this evaluation to formalize the notion of a model, which relates algebras and sentences.

Algebras. Defs. 2.23 to 2.26 define an algebra and its parts. A carrier function associates certain sorts with non-empty sets of values. An interpretation function associates certain operators with functions. Note that an interpretation function does not associate pre-defined operations with functions. An algebra has an interpretation function that associates operators with functions of which the domain and range values are from the carrier sets of the domain and range sorts of the operator. Also note that the interpretation of an $o \in \mathit{PredefinedOp}$ does not depend on the algebra, but is always the same and is as expected. For example, $\mathit{interpretation}(\wedge : \mathit{Bool} \times \mathit{Bool} \to \mathit{Bool})$ is the function $\mathfrak{and} : \{\mathcal{T}, \mathcal{F}\} \times \{\mathcal{T}, \mathcal{F}\} \to \{\mathcal{T}, \mathcal{F}\}$ such that $\mathfrak{and}(a, b) = \mathcal{T}$ iff $a = \mathcal{T}$ and $b = \mathcal{T}$.

Definition 2.23. $\mathbb{V}$ is the set of *values*.

Definition 2.24. A *carrier function* is a function $\mathit{carrier} : \mathit{Sort} \hookrightarrow \mathit{Set}(\mathbb{V})$ such that
$\mathit{carrier}(\mathit{Bool}) = \{\mathcal{T}, \mathcal{F}\}$, and
for every $S \in \mathit{Domain}(\mathit{carrier})$, $|\,\mathit{carrier}(S)\,| > 0$.
Carrier is the set of all carrier functions.

Definition 2.25. An *interpretation function* is a function $\mathit{interpretation} : \mathit{Op} \hookrightarrow \mathit{Function}$ such that
for every $o \in \mathit{PredefinedOp}$, $\mathit{interpretation}(o)$ is as usual (see e.g. the interpretation of $\wedge$ above).
Interpretation is the set of all interpretation functions.

Definition 2.26. An *algebra* is a record $\mathcal{A} : \mathit{carrier} \in \mathit{Carrier} \times \mathit{interpretation} \in \mathit{Interpretation}$ such that for every $o : S_1 \times \ldots \times S_n \hookrightarrow S_0 \in \mathit{Domain}(\mathcal{A}.\mathit{interpretation})$,
$\mathit{Domain}(\mathcal{A}.\mathit{interpretation}(o)) = \langle \mathcal{A}.\mathit{carrier}(S_1), \ldots, \mathcal{A}.\mathit{carrier}(S_n)\rangle$, and
$\mathit{Range}(\mathcal{A}.\mathit{interpretation}(o)) = \mathcal{A}.\mathit{carrier}(S_0)$.
Alg is the set of all algebras.

For convenience, Def. 2.27 defines a notion of inclusion on algebras in the obvious way.

Definition 2.27. $\subseteq$: $\mathit{Alg} \times \mathit{Alg} \to \mathit{Bool}$
$\mathcal{A}_0 \subseteq \mathcal{A}_1$ iff
$\mathcal{A}_1.\mathit{carrier} \subseteq \mathcal{A}_0.\mathit{carrier}$, and
$\mathcal{A}_1.\mathit{interpretation} \subseteq \mathcal{A}_0.\mathit{interpretation}$.

Valuations. Consider Def. 2.28. A valuation (Def. 2.28), sometimes called a variable assignment, maps variables to values.

Definition 2.28. A *valuation* is a partial function $va : \mathit{Var} \hookrightarrow \mathbb{V}$. *Valuation* is the set of all valuations.

Definition 2.29. *emptyva* is the valuation with $\mathit{Domain}(\mathit{emptyva}) = \{\}$.

Definition 2.30. $va \in \mathit{Valuation}$ is *well-sorted for* $\mathcal{A} \in \mathit{Alg}$ iff for every $v_S \in \mathit{Var}$, if $va(v_S) = \nu$, then $\nu \in \mathcal{A}.\mathit{carrier}(S)$. $\mathit{WellSortedVas}\mathcal{A}$ is the set of all valuations that are well-sorted for $\mathcal{A}$.

Term evaluation. The semantics of terms in total first order logic is well-known (see e.g. (GH93)). Partial logic in addition has to deal with undefined terms, where for a given algebra $\mathcal{A}$, the arguments of an application of operator o evaluate to values outside the domain of $\mathcal{A}.interpretation(o)$, or where a variable is not in the domain of the valuation. To deal with undefined terms, we follow the approach from (CMR98), where logical connectives and quantifiers are total operators that are false when applied to terms that do not evaluate to a value (sometimes called 'negative logic').

Function $Sem(t, va, \mathcal{A})$ (Def. 2.31) evaluates term t in algebra $\mathcal{A}$ under valuation va. The definition of $Sem(t, va, \mathcal{A})$ is straightforward.

Definition 2.31 (semantics of terms)**.** $Sem : Term \times Valuation \times Alg \hookrightarrow \mathbb{V}$
$Sem(t_0, va, \mathcal{A}) = \nu$ *iff*
- *or* $t_0 \in Var$ *and* $\nu = va(t_0)$,
- *or* t_0 *is true and* $\nu = \mathcal{T}$,
- *or* t_0 *is false and* $\nu = \mathcal{F}$,
- *or* *there are* $o \in Op$, $t_1, \ldots, t_n \in Term$ *such that*
  t_0 *is* $o(t_1, \ldots, t_n)$, *and*
  $\nu = \mathfrak{f}(Sem(t_1, va, \mathcal{A}), \ldots, Sem(t_n, va, \mathcal{A}))$ *with* $\mathfrak{f}$ *is* $\mathcal{A}.interpretation(o)$,
- *or* *there are* $v_S \in Var, t_1 \in Term$ *such that*
  - *or* t_0 *is* $\exists v : S \bullet t_1$, *and*
    if there is a $\nu \in \mathcal{A}.carrier(S)$ *such that* $Sem(t_1, va[v_S \mapsto \nu], \mathcal{A}) = \mathcal{T}$,
    then $\nu = \mathcal{T}$, *else* $\nu = \mathcal{F}$,
  - *or* t_0 *is* $\forall v : S \bullet t_1$, *and*
    if for every $\nu \in \mathcal{A}.carrier(S)$, $Sem(t_1, va[v_S \mapsto \nu], \mathcal{A}) = \mathcal{T}$,
    then $\nu = \mathcal{T}$, *else* $\nu = \mathcal{F}$,
- *or* *there are* $t_1, t_2 \in Term$ *such that*
  t_0 *is* $t_1 = t_2$, *and*
  if $Sem(t_1, va, \mathcal{A}) = Sem(t_2, va, \mathcal{A})$, *then* $\nu = \mathcal{T}$, *else* $\nu = \mathcal{F}$.

Models. Def. 2.32 states when an algebra is a model for a sentence. Note that the value of a sentence s does not depend on the valuation as a sentence does not contain free variables. So, if there is a $va \in Valuation$ such that $Sem(s, va, \mathcal{A}) = \mathcal{T}$, then for every $va \in Valuation$, $Sem(s, va, \mathcal{A}) = \mathcal{T}$.

Definition 2.32. $\mathcal{A} \in Alg$ is a *model* for $s \in Sentence$ iff for every $va \in Valuation$, we have $Sem(s, va, \mathcal{A}) = \mathcal{T}$.

Definition 2.33. $\mathcal{A} \in Alg$ is a *model* for $Sen \in Set(Sentence)$ iff for every $s \in Sen$, $\mathcal{A}$ is a model for s.

If Sen is a set of sentences, then $Models(Sen)$ (Def. 2.34) is the set of all models of Sen.

Definition 2.34. $Models : Set(Sentence) \to Set(Alg)$
$\mathcal{A} \in Models(Sen)$ *iff* $\mathcal{A}$ *is a model for* Sen.

2.2 Algebraic Client Specification

In this section, we discuss client specifications that are based on algebraic specifications. Our aim is to come up with a flexible, foundational, notion of specification. To this end, we introduce a novel syntax (Sect. 2.2.1) that incorporates a notion of canonicity, and a novel semantics (Sect. 2.2.2) that closely matches the client's view of the implementation as a black box.

These specifications are particularly suited to situations where the client is interested only in the input/output behavior of the implementation, i.e., situations where the client has a set of possible questions that the implementation should compute the answers to. In such situations, the client wants the implementation to rewrite an input into an equivalent, most basic form. Note that the client can then use this output as the basis for another input.

In our formalism, a client specification consists of an algebraic specification and a canonicity function. An algebraic specification consists of 1) a signature that describes the sorts and operators of the client's problem domain, and 2) a set of sentences, called *axioms* in this context, that formalize the properties that the client desires of the operators in the signature. The canonicity function determines the required format of the output, i.e., which expressions are considered by the client as proper answers, a most basic form.

The meaning of a client specification consisting of a signature *sig*, a set of axioms *ax* and a canonicity function *isCanonical* is the following.

- Every model $\mathcal{A}$ of *ax* provides a notion of equality that is acceptable to the client, formalized by *ax* which expresses the desired properties of the operators in the signature. In particular, two closed applications ca_0 and ca_1 are equal in a model $\mathcal{A}$ iff $Sem(ca_0 = ca_1, emptyva, \mathcal{A}) = \mathcal{T}$.
- Therefore, every model $\mathcal{A}$ of *ax* induces a division of $ClosedAppls(sig)$, which are the closed applications that can be formed using only the operators in the signature, into equivalence classes.
- The canonicity function *isCanonical* determines for each such set of equivalence classes the set of class representatives (with a small caveat: our formalism does not forbid multiple representatives per equivalence class; the representatives of a given class are determined by the specifier through the canonicity function).
- An implementation satisfies the specification iff for one of these sets of equivalence classes, given any input *in* of any equivalence class *EqClass*, the implementation outputs a representative of *EqClass*.

For example, assume that *CS* is the specification of a simple calculator with signature *sig*. Then *sig* defines the sort $\mathbb{N}$, constants such as `1,2,...` and operators such as `+` and `*`. Any closed application built from the operators of the problem domain, i.e., every element of $ClosedAppls(sig)$, represents a question from the problem domain and can be used as an input to the implementation. For example, `+(3,5)` and `*(4,3)` are such closed applications.

For the input `+(3,5)` the client is not satisfied with output `+(2,6)` or `+.+(3,5)` or or, for that matter, `+(3,5)` itself, which are all in the equivalence class of `+(3,5)`. Only the output `8` is acceptable. This is formalized by the notion of canonicity. More generally, the client expects the implementation to transform an input *in* into an output *out* that is a *canonical form* of *in*. That is,

- like *in*, *out* should be expressed using the operators of the client's problem domain, i.e., *out* should be an element of $ClosedAppls(sig)$.
- *out* should be canonical, i.e., it should be of a certain basic form. A possible notion of canonicity for our calculator example is that a closed application is canonical iff it is a constant. E.g., `8` is canonical, but `+(3,5)` is not[1].

[1]Having all elements of the domain as constants in the language is then a consequence, which may or may

- *in* and *out* should be equal. Every model $\mathcal{A}$ of the axioms of the specification provides a notion of equality that is acceptable to the client (as explained above). Which of these notions of equivalence is used by the implementation is up to the implementor. So, in our example, the specifier should ensure that the axioms for `+` are such that for every model $\mathcal{A}$ of the axioms, $Sem(+(\mathtt{3},\mathtt{5}) = \mathtt{8}, emptyva, \mathcal{A}) = \mathcal{T}$ (and that `+(3,5)` is not equal to any other constant).

We formalize the above in more detail in Sections 2.2.1 and 2.2.2.

2.2.1 Syntax of Specifications

In this section, we formalize the notions of algebraic specification, canonicity function and client specification.

An algebraic specification (Def. 2.35) consists of a signature and a set of sentences, called axioms. These axioms only contain operators from the signature.

Definition 2.35. An *algebraic specification* AS is a record $sig \in Sig \times ax \in Set(Sentence)$ such that
$AS.ax \subseteq Terms(AS.sig)$
AlgSpec is the set of all algebraic specifications.

Recall that every model of the axioms of the specification induces a notion of equality that is acceptable to the client. The intention is that given such a model, the canonicity function (Def. 2.36) of a specification determines whether a given closed application is a representative of an equivalence class and thus suitable as output of the implementation. In other words, the canonicity function determines whether a given closed application is 'most basic'. For this to be the case, there should be no equivalent 'more basic' closed application. Which closed applications are equal is determined by the supplied algebra (two closed applications ca_0 and ca_1 are equal in an algebra $\mathcal{A}$ iff $Sem(ca_0 = ca_1, emptyva, \mathcal{A}) = \mathcal{T}$).

Definition 2.36. A *canonicity function* is a function $canonFunc : ClosedAppl \times Alg \rightarrow Bool$.
CanonFunc is the set of all canonicity functions.

A client specification CS (Def. 2.38) consists of an algebraic specification as and a canonicity function *isCanonical* such that for every model of the axioms, every meaningful closed application has a canonical form. The latter is ensured, by $IsEachEqClassRepresented(CS.isCanonical, CS.as) = \mathcal{T}$ (Def. 2.37). More generally, $IsEachEqClassRepresented(canonFunc, AS) = \mathcal{T}$ iff for every model $\mathcal{A}$ of $AS.sig$, for every closed application ca_0 that 1) is defined by $AS.sig$, and 2) can be evaluated to a value using $\mathcal{A}$, there is canonical representation ca_1 of ca_0. In other words, iff for any given notion of equivalence that follows from $AS.ax$, *canonFunc* is such that every equivalence class has at least one representative.

Note that these definitions reflect the intended separation of concerns. The concern of the axioms is to capture the desired properties of the operators and thus define the acceptable notions of equality. Given any such acceptable notion of equality, the concern of the canonicity function is to determine the representatives of the equivalence class of a given closed application.

not be acceptable (having an infinite number of constants is not always acceptable). Hence a more appropriate notion for canonicity is to define an operator for decimal representation, δ, and denote it as juxtaposition. Thus, 241 is not a constant, but an abbreviation for $\delta(1, \delta(2, 4))$.

Definition 2.37. *IsEachEqClassRepresented* : $CanonFunc \times AlgSpec \to Bool$
$IsEachEqClassRepresented(canonFunc, AS) = \mathcal{T}$ iff
for every $ca_0 \in ClosedAppls(AS.sig)$, $\mathcal{A} \in Models(AS.ax)$, $\nu \in \mathbb{V}$,
if $Sem(ca_0, emptyva, \mathcal{A}) = \nu$,
then there is a $ca_1 \in ClosedAppls(AS.sig)$ such that
$canonFunc(ca_1, \mathcal{A}) = \mathcal{T}$, and
$Sem(ca_1, emptyva, \mathcal{A}) = \nu$.

Definition 2.38. A *client specification CS* is a record $as \in AlgSpec \times isCanonical \in CanonFunc$ such that
$IsEachEqClassRepresented(CS.isCanonical, CS.ax) = \mathcal{T}$.
ClientSpec is the set of all client specifications.

Before we present the semantics of algebraic specifications in Sect. 2.2.2, we show three examples of the syntax (Exmpls. 2.1 to 2.3).

Example 2.1.

(*Rationals, specification*)
signature and axioms
Below, we use some self-explanatory shorthand. For technical reasons, we write $new_{Rat}(n, d)$ instead of the more usual n/d. Also note that among the omitted operators and axioms are those that establish that every integer is a rational, which exclude trivial models of the specification.

sorts Rat, Int
operators
$new_{Rat} : Int \times Int \hookrightarrow Rat$
$add : Rat \times Rat \to Rat$
$equals : Rat \times Rat \to Bool$
axioms
$\forall n_0, n_1, d_0, d_1 \in Int \bullet$
$add(new_{Rat}(n_0, d_0), new_{Rat}(n_1, d_0)) = new_{Rat}(n_0 + n_1, d_0)$
$\wedge$ $equals(new_{Rat}(n_0, d_0), new_{Rat}(n_1, d_1)) = true \Leftrightarrow n_0 * d_1 = n_1 * d_0$

operators and axioms for *Int*, the sort that models unbounded integers, are omitted.

A *Rat* r is canonical iff there are $n, d \in Int$ such that 1) r is $new_{Rat}(n, d)$, and 2) n and d are canonical, and 3) the greatest common divider of n and d is 1. A formal definition is straightforward and is therefore omitted.

Example 2.2.

(*Multiple notions of equality*)

This example shows a common pattern for the canonicity function. It also shows how the canonicity function can determine representatives in the case where several notions of equality are induced by the axioms.

Below is the definition of the signature and axioms of the algebraic specification (using some self-explanatory syntactic sugar).

sorts X
operators
$zero : \to X$
$succ : X \to X$
$add : X \times X \to X$

axioms
$\forall x, x_0, x_1 \in X \bullet$
$\quad add(x, zero()) = x$
$\wedge \quad add(x_0, succ(x_1)) = succ(add(x_0, x_1))$
$\wedge \quad succ(succ(x)) \neq succ(x)$
Now consider the following two algebras $\mathcal{A}_0$ and $\mathcal{A}_1$:

1. in $\mathcal{A}_0$, the carrier for X is $\mathbb{N}$, $zero()$ is interpreted as $0 \in \mathbb{N}$, *succ* as the successor function and *add* as the $+$ (i.e., the addition function on natural numbers).

2. in $\mathcal{A}_1$, the carrier for X is *Bool*, $zero()$ is interpreted as $\mathcal{F}$, *succ* as the logical not and *add* as the exclusive or.

Note that both $\mathcal{A}_0$ and $\mathcal{A}_1$ are a model of the axioms.

Next, we define the canonicity function *isCanonical* of the specification. Roughly, a closed application *ca* is canonical iff only *zero* and *succ* occur in *ca* (i.e., *ca* is one of $zero()$, $succ(zero())$, $succ(succ(zero()))$, ...), and there is no equivalent closed application with fewer occurrences of *succ*. This follows a common pattern for the canonicity function, where a closed application *ca* is canonical iff every operator that occurs in *ca* comes from a set of *generators*, and there is no equivalent closed application that consists of fewer applications of these generators.

The definition of the canonicity function uses a helper function *sCount* that returns the number of occurrences of operator *succ* in a closed application in which only *succ* and *zero* occur. It is defined as follows:
$sCount : ClosedAppl \hookrightarrow \mathbb{N}$
$sCount(ca_0) = n$ iff
$\quad ca_0$ is $zero()$ and $n = 0$,
or $\quad$ there is a $ca_1 \in ClosedAppl$ such that ca_0 is $succ(ca_1)$ and $n = 1 + sCount(ca_1)$.
$isCanonical(ca_0, \mathcal{A}) = \mathcal{T}$ iff
only *zero* and *succ* occur in ca_0, and
for every $ca_1 \in ClosedAppl$,
if $\quad$ only *zero* and *succ* occur in ca_1, and
$\quad Sem(ca_0 = ca_1, emptyva, \mathcal{A}) = \mathcal{T}$,
then $\quad sCount(ca_0) \leq sCount(ca_1)$.
Assume that $ca = succ(succ(zero()))$. Note that $isCanonical(ca, \mathcal{A}_0) = \mathcal{T}$, but that $isCanonical(ca, \mathcal{A}_1) = \mathcal{F}$ as $Sem(ca = zero(), emptyva, \mathcal{A}_1) = \mathcal{T}$.

Example 2.3.

(*the Stack of Int example, specification*) Here we present the classic example of an algebraic specification, that of a Stack, in the setting of our specification technique.

Note that the last 2 axioms essentially define equality on stacks.
sorts *Stack*, *Int*
operators
$new_{Stack} : \rightarrow Stack$
$push : Stack \times Int \rightarrow Stack$
$pop : Stack \hookrightarrow Stack$
$top : Stack \hookrightarrow Int$
axioms

$$\begin{array}{ll} \forall s, s_0, s_1 \in \mathit{Stack}, i, i_0, i_1 \in \mathit{Int} \bullet & \\ \quad \mathit{pop}(\mathit{push}(s,i)) = s & \\ \wedge \quad \mathit{top}(\mathit{push}(s,i)) = i & \\ \wedge \quad \mathit{new}_{\mathit{Stack}}() \neq \mathit{push}(s,i) & \\ \wedge \quad \mathit{push}(s1,i1) = \mathit{push}(s2,i2) \Leftrightarrow s1 = s2 \wedge i1 = i2 & \end{array}$$

operators and axioms for Int, the sort that models unbounded integers, are omitted.

The canonicity function follows the common pattern shown in Exmpl. 2.2. A closed application *ca* of sort `Stack` is canonical iff *ca* only consists of the generators $\mathit{new}_{\mathit{Stack}}$, *pop* and integer constants, and there is no equivalent closed application that consists of fewer applications of these generators. We omit the formal definition.

Aside. For brevity, we omit a division of its operators into two sets: *interface operators* and *auxiliary operators*. The intuition of such a division is that interface operators are the verbs of the problem description. Auxiliary operators only serve to axiomatize the interface operators. For example, the example from Hoare's classic paper on data abstraction revolves around sets which are axiomatized using, among others, a *size* operator. But the paper states that only the *insert*, *remove* and *has* operators occur in the abstract program. That is, only these operators are interface operators. The other operators, like *size*, are auxiliary (only used to axiomatize the *SmallIntSet*).

Aside. By interpreting the axioms of an algebraic specification as left-to-right rewrite rules, it is possible to specify the canonicity function indirectly (where a closed application is canonical if none of the rules applies to it). For example, Maude (CDE+02; BJM97) is a program that, given an signature and a set of rewrite rules, allows the user to input a closed application and outputs a closed application (assuming that the rewrite rules are Church-Rosser, terminating and sort-decreasing).

2.2.2 Semantics of Specifications

In this section, we present an intuitive semantics of specifications that is independent of the choice of a programming language.

A client specification intends to capture a set of implementations that are acceptable to the client. A core assumption of the client specification technique is that the client only cares about the input/output behavior of the implementation. Roughly, an implementation is acceptable to the client if, given a meaningful input, the implementation outputs a canonical equivalent. For example, the implementation may be an executable that takes one command line parameter, which is a closed application (typed in by the user). Execution returns a closed application (the answer), and displays it as a string on the screen. Note that the client can use the output as basis for another input.

An answer function (Def. 2.39, illustrated in Fig. 1) is the obvious semantics that matches this black box view of the implementation.

Definition 2.39. An *answer function* is a function $\mathit{answer} : \mathit{ClosedAppl} \hookrightarrow \mathit{ClosedAppl}$.
Answer is the set of all answer functions.

The semantics of an algebraic specification is given in Def. 2.40. Note that given an algebraic specification *CS.as*, not every answer function provides the 'right answer' for any given input ca_0. The intuition is that the semantics of *CS* determines the set of answer functions that do provide the 'right answer' for any given input. Roughly, the semantics of a specification is the

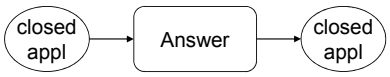

Figure 1: Answer function: black box model of implementation

set of those answer functions for which there exists a model of the axioms such that for every input ca_0 that evaluates to a value, there is a canonical output ca_1 that evaluates to the same value. Note that it suffices that there is *a* model, as every model induces a notion of equivalence that is acceptable to the client.

Definition 2.40. $SemAS : ClientSpec \to Set(Answer)$
$answer \in SemAS(CS)$ iff
there is an $\mathcal{A} \in Models(CS.as.ax)$ such that
for every $ca_0 \in ClosedAppls(CS.as.sig)$, $\nu \in \mathbb{V}$
if $Sem(ca_0, emptyva, \mathcal{A}) = \nu$,
then there is an $ca_1 \in ClosedAppls(CS.as.sig)$ such that
$answer(ca_0) = ca_1$, and
$Sem(ca_1, emptyva, \mathcal{A}) = \nu$, and
$CS.isCanonical(ca_1, \mathcal{A}) = \mathcal{T}$.

This semantics is intuitive, as it directly describes the set of (semantics of) black box implementations that are acceptable to the client. Another advantage is that it is independent of the choice of a programming language. Abstracting the semantics of a program in a concrete programming language to an answer function can be treated as a separate concern.

3 Opening the black box

In OO the world is not viewed in terms of functional expressions but in terms of objects. Therefore, an OO semantics does not readily provide closed applications as i/o. As a consequence, the black box must be opened. Input, a closed application, is translated to a **statement sequence** that can be executed by the OO implementation. The implementation computes for such a statement sequence the result, objects with values, an **evaluation context**. This evaluation context then again has to be translated back to a canonical closed application to display the result to the client. The specifier/implementor contract is therefore extended with translations of *ca*s to (**Translate**) and from (**Display**) the implementation - see Figure 2. By performing the translations in a fixed way, the contract still applies to the black box: the business logic.

More on this can be found in (Mid11).

4 Conclusions and future work

.

We provided a detailed new angle on describing the behavior of a black box - and, given that, made a stab at how to open it. Both the observation that the notion of canonicity may involve semantics and the introduction of the Translate and Display steps raise issues about whether and how completeness of a proof system is attainable.

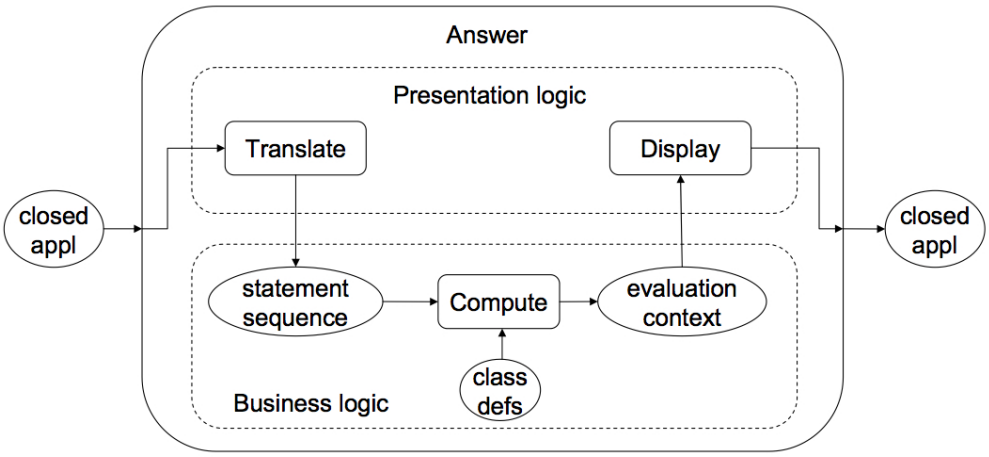

Figure 2: Opening the black box

References

[BJM97] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 67–92. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0030589.

[CDE+02] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narcise Martí-Oliet, José Meseguer, and José F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187 – 243, 2002.

[CMR98] Maura Cerioli, Till Mossakowski, and Horst Reichel. From total equational to partial first order logic. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specifications*, 1998.

[GH93] John V. Guttag and James J. Horning. *Larch: languages and tools for formal specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[Mid11] Ronald Middelkoop. *Capturing and exploiting abstract views of states in OO verification*. PhD thesis, Eindhoven University of Technology, 2011.

[ST99] Donald Sannella and Andrzej Tarlecki. Algebraic preliminaries. In Egidio Astesiano, Hans-Jörg Kreowski, and Bernd Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, chapter 2. Springer, 1999.