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. \( f : A \rightarrow B \) 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 \( \{f(a) = b \text{ if and only if } P(a, b)\} \) to define \( f \) as the smallest single-valued relation such that \( \forall a \in A, b \in B \bullet (a \ f b \iff P(a, b) \text{ holds}) \). \( f : A \rightarrow B \) introduces a total function from \( A \) to \( B \). \( \text{Domain}(f) \) and \( \text{Range}(f) \) 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 \( a \in \text{Domain}(f) \).

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 \( \text{Record} \) with tuple in \( A_1 \times \ldots \times A_n \) is indexed by names \( f_1, \ldots, f_n \), we write \( \text{Record} : f_1 \in A_1 \times \ldots \times f_n \in A_n \).
Record.f denotes the value of the field with name f of Record. Given a sequence Σ, Σ[i], Σ[i, j] and Σ[i..] denote element, consecutive subsequence and postfix. ⟨⟩ denotes the empty sequence. Σ₀ ⊕ Σ₁ denotes the concatenation of sequences Σ₀ and Σ₁. We write π ∈ Seq(x) to denote that π is a sequence of elements from X. Given a set X, we write xSet ∈ 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. VarS denotes the set of variables of sort S. We write vS to denote that vS ∈ VarS.

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

Definition 2.3. VarSort : Var → Sort

VarSort(vS) = 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 ⟨S₁, . . . , Sₙ⟩ and a range sort S₀, where S₀, . . . , Sₙ ∈ Sort. We write o : S₁ × . . . × Sₙ → S₀ to denote an operator with domain sorts ⟨S₁, . . . , Sₙ⟩ and range sort S₀.
The functions \textit{DomainSorts} (Def. 2.5) and \textit{RangeSort} (Def. 2.6) yield the domain sorts and the range sort of an operator.

\textbf{Definition 2.5.} \textit{DomainSorts}: \(\text{Def. 2.5} \)
\(\text{DomainSorts}(o : S_1 \times \ldots \times S_n \rightarrow S_0) = (S_1, \ldots, S_n)\).

\textbf{Definition 2.6.} \textit{RangeSort}: \(\text{Def. 2.6} \)
\(\text{RangeSort}(o : S_1 \times \ldots \times S_n \rightarrow S_0) = S_0\).

\textbf{Definition 2.7.} A constant is an operator \(o\) such that \(\text{DomainSorts}(o) = ()\). \textit{Constant} is the set of all constants.

\textit{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).

\textbf{Definition 2.8.} \textit{BoolOp} is the set that consists of the two constants \(true : \rightarrow \text{Bool}\) and \(false : \rightarrow \text{Bool}\).

\textbf{Definition 2.9.} \textit{LogConOp} is the set that consists of the usual logical connectives (e.g. \(\land : \text{Bool} \times \text{Bool} \rightarrow \text{Bool}\)). \textit{LogConOp} also includes, for every \(S \in \text{Sorts}\), the operator \(=^S : S \times S \rightarrow \text{Bool}\).

\textbf{Definition 2.10.} \textit{QuantOp} is the set that consists of, for every \(S \in \text{Sorts}\), the operators \(\forall^S : S \times \text{Bool} \rightarrow \text{Bool}\) and \(\exists^S : S \times \text{Bool} \rightarrow \text{Bool}\).

\textbf{Definition 2.11.} \textit{PredefinedOp} is the set \(\text{BoolOp} \cup \text{LogConOp} \cup \text{QuantOp}\).

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

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

\(\text{SigUnion}(\text{sig}_0, \text{sig}_1) \) (Def. 2.13) returns the signature that unites \(\text{sig}_0\) and \(\text{sig}_1\) in the obvious way.

\textbf{Definition 2.13.} \textit{SigUnion}: \(\text{Sig} \times \text{Sig} \rightarrow \text{Sig}\)
\(\text{SigUnion}(\text{sig}_0, \text{sig}_1) = \text{sig}_2 \) iff
\(\text{sig}_2.\text{sorts} = \text{sig}_0.\text{sorts} \cup \text{sig}_1.\text{sorts}\), and
\(\text{sig}_2.\text{ops} = \text{sig}_0.\text{ops} \cup \text{sig}_1.\text{ops}\), and
\(\text{sig}_2.\text{tOps} = \text{sig}_0.\text{tOps} \cup \text{sig}_1.\text{tOps}\).
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 $\langle o, (t_1, \ldots, t_n) \rangle$ of (i) an operator $o : S_1 \times \cdots \times S_n \to S_0$, 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 $o \in \text{QuantOp}$ 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 $\langle o, (t_1, \ldots, t_n) \rangle$ as $o(t_1, \ldots, t_n)$. Additionally, when $o \in \text{QuantOp}$ 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 $o_1 \in \text{PredefinedOp}$ 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 $o \in \text{QuantOp}$ and there is a $i \in [1, n]$ such that $v_0$ is free in $t_i$. $\text{Free}(t)$ denotes the set of free variables in $t$.

Definition 2.17. Term $t$ is closed if it contains no free variables. $\text{ClosedAppl}$ is the set of all closed applications.

Definition 2.18. A predicate is a term of sort $\text{Bool}$. $\text{Predicate}$ is the set of all predicates.

Definition 2.19. A sentence $s$ is a closed predicate. $\text{Sentence}$ is the set of all sentences.

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

Definition 2.20. $\text{Terms} : \text{Sig} \to \text{Set}(\text{Term})$

$t \in \text{Terms}(\text{sig})$ iff

- there are $S \in \text{sig.sorts}$, $v \in \text{Vars}$ such that $t = v$,
- there are $o \in \text{sig.ops}$, $t_1, \ldots, t_n \in \text{Terms}(\text{sig})$ such that $t = o(t_1, \ldots, t_n)$.

Definition 2.21. $o_0 \in \text{Op}$ occurs in $t \in \text{Term}$ iff

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

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

Definition 2.22. $\text{ClosedAppl} : \text{Sig} \to \text{Set}(\text{ClosedAppl})$

$\text{ca} \in \text{ClosedAppl}(\text{sig})$ iff $\text{ca} \in \text{Terms}(\text{sig})$ and $\text{ca}$ 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. Dffs. 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 $o \in \text{PredefinedOp}$ does not depend on the algebra, but is always the same and is as expected. For example, interpretation($\land : \text{Bool} \times \text{Bool} \to \text{Bool}$) is the function $\text{and} : \{T,F\} \times \{T,F\} \to \{T,F\}$ such that $\text{and}(a, b) = T$ iff $a = T$ and $b = T$.

Definition 2.23. $V$ is the set of values.

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

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

Definition 2.26. An algebra is a record $A : \text{carrier} \in \text{Carrier} \times \text{interpretation} \in \text{Interpretation}$ such that for every $o : S_1 \times \ldots \times S_n \to S_0 \in \text{Domain}(A.\text{interpretation})$, $\text{Domain}(A.\text{interpretation}(o)) = (A.\text{carrier}(S_1), \ldots, A.\text{carrier}(S_n))$, and $\text{Range}(A.\text{interpretation}(o)) = A.\text{carrier}(S_0)$. 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. $\subseteq : \text{Alg} \times \text{Alg} \to \text{Bool}$ $A_0 \subseteq A_1$ iff $A_1.\text{carrier} \subseteq A_0.\text{carrier}$, and $A_1.\text{interpretation} \subseteq A_0.\text{interpretation}$.

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 $\text{va} : \text{Var} \to V$. Valuation is the set of all valuations.

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

Definition 2.30. $\text{va} : \text{Valuation}$ is well-sorted for $A : \text{Alg}$ iff for every $v_S \in \text{Var}$ if $\text{va}(v_S) = \nu$, then $\nu \in A.\text{carrier}(S)$. $\text{WellSortedVasA}$ is the set of all valuations that are well-sorted for A.
Term evaluation. The semantics of terms in total first order logic is well-known (see e.g. \cite{GH93}). Partial logic in addition has to deal with undefined terms, where for a given algebra \( A \), the arguments of an application of operator \( o \) evaluate to values outside the domain of \( A.\text{interpretation}(o) \), or where a variable is not in the domain of the valuation. To deal with undefined terms, we follow the approach from \cite{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 \( \text{Sem}(t, va, A) \) (Def. 2.31) evaluates term \( t \) in algebra \( A \) under valuation \( va \). The definition of \( \text{Sem}(t, va, A) \) is straightforward.

**Definition 2.31** (semantics of terms).  
\[
\text{Sem}(t_0, va, A) = \nu \text{ iff } \\
or \quad t_0 \in \text{Var} \text{ and } \nu = \text{va}(t_0), \\
or \quad t_0 \text{ is true and } \nu = T, \\
or \quad t_0 \text{ is false and } \nu = F, \\
or \quad \text{there are } o \in \text{Op}, t_1, \ldots, t_n \in \text{Term} \text{ such that } \\
\quad \quad t_0 = o(t_1, \ldots, t_n), \text{ and } \\
\quad \quad \nu = \{ \text{Sem}(t_1, va, A), \ldots, \text{Sem}(t_n, va, A) \} \text{ with } \nu \text{ is } A.\text{interpretation}(o), \\
or \quad \text{there are } v_S \in \text{Var}, t_1 \in \text{Term} \text{ such that } \\
\quad \quad t_0 = \exists v: S \bullet t_1, \text{ and } \\
\quad \quad \text{if there is a } \nu \in A.\text{carrier}(S) \text{ such that } \text{Sem}(t_1, va[v_S \rightarrow \nu], A) = T, \\
\quad \quad \quad \text{then } \nu = T, \text{ else } \nu = F, \\
or \quad \quad \quad \text{if for every } \nu \in A.\text{carrier}(S), \text{Sem}(t_1, va[v_S \rightarrow \nu], A) = T, \\
\quad \quad \quad \quad \text{then } \nu = T, \text{ else } \nu = F, \\
or \quad \text{there are } t_1, t_2 \in \text{Term} \text{ such that } \\
\quad \quad t_0 = t_1 = t_2, \text{ and } \\
\quad \quad \text{if } \text{Sem}(t_1, va, A) = \text{Sem}(t_2, va, A), \text{ then } \nu = T, \text{ else } \nu = F.
\]

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 \( va \in \text{Valuation} \) such that \( \text{Sem}(s, va, A) = T \), then for every \( va \in \text{Valuation} \), \( \text{Sem}(s, va, A) = T \).

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

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

If \( Sen \) is a set of sentences, then \( \text{Models}(Sen) \) (Def. 2.34) is the set of all models of \( Sen \).

**Definition 2.34.** \( \text{Models} : \text{Set(Sentence)} \rightarrow \text{Set(Alg)} \) \( A \in \text{Models}(Sen) \) iff \( A \) 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 $\text{sig}$, a set of axioms $\text{ax}$ and a canonicity function $\text{isCanonical}$ is the following.

- Every model $\mathcal{A}$ of $\text{ax}$ provides a notion of equality that is acceptable to the client, formalized by $\text{ax}$ 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 $\text{Sem}(ca_0 = ca_1, \text{emptyva}, \mathcal{A}) = \text{T}$.

- Therefore, every model $\mathcal{A}$ of $\text{ax}$ induces a division of $\text{ClosedAppls}(\text{sig})$, which are the closed applications that can be formed using only the operators in the signature, into equivalence classes.

- The canonicity function $\text{isCanonical}$ 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 $\text{in}$ of any equivalence class $\text{EqClass}$, the implementation outputs a representative of $\text{EqClass}$.

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

For the input $+(3, 5)$ the client is not satisfied with output $+(2, 6)$ or $\times+(3, 5)$ 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 $\text{in}$ into an output $\text{out}$ that is a canonical form of $\text{in}$. That is,

- like $\text{in}$, $\text{out}$ should be expressed using the operators of the client’s problem domain, i.e., $\text{out}$ should be an element of $\text{ClosedAppls}(\text{sig})$.

- $\text{out}$ 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 \( A \) 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 \( A \) of the axioms, \( \text{Sem} + (3, 5) = 8, \text{emptyva}, A = T \) (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

\[
\text{sig} \in \text{Sig} \times \text{ax} \in \text{Set(Sentence)}
\]

such that

\[
AS.\text{ax} \subseteq \text{Terms(AS.\text{sig})}
\]

\( \text{AlgSpec} \) 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 \( A \) iff \( \text{Sem}(ca_0 = ca_1, \text{emptyva}, A) = T \)).

**Definition 2.36.** A canonicity function is a function \( \text{canonFunc} : \text{ClosedAppl} \times \text{Alg} \rightarrow \text{Bool} \). \( \text{CanonFunc} \) is the set of all canonicity functions.

A client specification \( CS \) (Def. 2.38) consists of an algebraic specification \( as \) and a canonicity function \( \text{isCanonical} \) such that for every model of the axioms, every meaningful closed application has a canonical form. The latter is ensured, by \( \text{IsEachEqClassRepresented}(CS.\text{isCanonical}, CS.\text{as}) = T \) (Def. 2.37). More generally, \( \text{IsEachEqClassRepresented}(\text{canonFunc}, AS) = T \) iff for every model \( A \) of \( AS.\text{sig} \), for every closed application \( ca_0 \) that 1) is defined by \( AS.\text{sig} \), and 2) can be evaluated to a value using \( A \), there is canonical representation \( ca_1 \) of \( ca_0 \). In other words, iff for any given notion of equivalence that follows from \( AS.\text{ax} \), \( \text{canonFunc} \) 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, \( \delta \), and denote it as juxtaposition. Thus, 241 is not a constant, but an abbreviation for \( \delta(1, \delta(2, 4)) \).*

---

265
Definition 2.37. \textit{IsEachEqClassRepresented} is a function \( \text{CanonFunc} \times \text{AlgSpec} \rightarrow \text{Bool} \) that checks if every equivalence class is represented. For every \( ca_0 \in \text{ClosedAppls}(AS.\text{sig}) \), \( A \in \text{Models}(AS.\text{ax}) \), and \( \nu \in \mathbb{V} \):

\[
\text{Sem}(ca_0, \text{emptyva}, A) = \nu \quad \text{if there is a } ca_1 \in \text{ClosedAppls}(AS.\text{sig}) \text{ such that }
\]
\[
\text{canonFunc}(ca_1, A) = \text{T}, \quad \text{and } 
\]
\[
\text{Sem}(ca_1, \text{emptyva}, A) = \nu.
\]

Definition 2.38. A client specification \( CS \) is a record \( \text{as} \in \text{AlgSpec} \times \text{isCanonical} \in \text{CanonFunc} \) such that \( \text{IsEachEqClassRepresented}(CS.\text{isCanonical}, CS.\text{ax}) = \text{T} \). \( \text{ClientSpec} \) 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 \( \text{newRat}(n, d) \) 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.

\begin{itemize}
\item \textbf{sorts} \( \text{Rat, Int} \)
\item \textbf{operators}
\begin{itemize}
\item \( \text{newRat} : \text{Int} \times \text{Int} \rightarrow \text{Rat} \)
\item \( \text{add} : \text{Rat} \times \text{Rat} \rightarrow \text{Rat} \)
\item \( \text{equals} : \text{Rat} \times \text{Rat} \rightarrow \text{Bool} \)
\end{itemize}
\item \textbf{axioms}
\begin{itemize}
\item \( \forall n_0, n_1, d_0, d_1 \in \text{Int} \cdot 
\begin{align*}
\text{add}(\text{newRat}(n_0, d_0), \text{newRat}(n_1, d_0)) &= \text{newRat}(n_0 + n_1, d_0) \\
\text{equals}(\text{newRat}(n_0, d_0), \text{newRat}(n_1, d_1)) &= \text{true} \iff n_0 \cdot d_1 = n_1 \cdot d_0
\end{align*}
\end{itemize}
\end{itemize}

operators and axioms for \( \text{Int} \), the sort that models unbounded integers, are omitted.

A \( \text{Rat} \) \( r \) is canonical iff there are \( n, d \in \text{Int} \) such that 1) \( r \) is \( \text{newRat}(n, d) \), 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).

\begin{itemize}
\item \textbf{sorts} \( X \)
\item \textbf{operators}
\begin{itemize}
\item \( \text{zero} : \rightarrow X \)
\item \( \text{succ} : X \rightarrow X \)
\item \( \text{add} : X \times X \rightarrow X \)
\end{itemize}
\end{itemize
}
Pandora’s Box

Middelkoop, Huizing, Kuiper, Luit

axioms

\[ \forall x, x_0, x_1 \in X \bullet \]
\[ \text{add}(x, \text{zero}()) = x \]
\[ \wedge \text{add}(x_0, \text{succ}(x_1)) = \text{succ}(\text{add}(x_0, x_1)) \]
\[ \wedge \text{succ}(\text{succ}(x)) \neq \text{succ}(x) \]

Now consider the following two algebras \( A_0 \) and \( A_1 \):

1. in \( A_0 \), the carrier for \( X \) is \( \mathbb{N} \), \( \text{zero}() \) is interpreted as 0 \( \in \mathbb{N} \), \( \text{succ} \) as the successor function and \( \text{add} \) as the + (i.e., the addition function on natural numbers).

2. in \( A_1 \), the carrier for \( X \) is \( \text{Bool} \), \( \text{zero}() \) is interpreted as \( \text{false} \), \( \text{succ} \) as the logical not and \( \text{add} \) as the exclusive or.

Note that both \( A_0 \) and \( A_1 \) are a model of the axioms.

Next, we define the canonicity function \( \text{isCanonical} \) of the specification. Roughly, a closed application \( ca \) is canonical iff only \( \text{zero} \) and \( \text{succ} \) occur in \( ca \) (i.e., \( ca \) is one of \( \text{zero}() \), \( \text{succ} (\text{zero}()) \), \( \text{succ} (\text{succ} (\text{zero}())) \),...), and there is no equivalent closed application with fewer occurrences of \( \text{succ} \). 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 \( \text{succ} \) in a closed application in which only \( \text{succ} \) and \( \text{zero} \) occur. It is defined as follows:

\[ \text{sCount} : \text{ClosedAppl} \rightarrow \mathbb{N} \]
\[ \text{sCount}(ca_0) = n \text{ iff } \]
\[ \begin{cases} 
\text{ca}_0 \text{ is zero}() \text{ and } n = 0, \\
\text{or there is a } ca_1 \in \text{ClosedAppl} \text{ such that } ca_0 \text{ is succ}(ca_1) \text{ and } n = 1 + \text{sCount}(ca_1). 
\end{cases} \]

\[ \text{isCanonical}(ca_0, A) = T \text{ iff } \]
\[ \begin{cases} 
\text{only zero and succ occur in } ca_0, \text{ and} \\
\text{for every } ca_1 \in \text{ClosedAppl} \text{ if } \]
\[ \begin{cases} 
\text{only zero and succ occur in } ca_1, \text{ and} \\
\text{Sem}(ca_0 = ca_1, emptyva, A) = T, 
\end{cases} \]
\[ \text{then } \text{sCount}(ca_0) \leq \text{sCount}(ca_1). \]

Assume that \( ca = \text{succ} (\text{succ} (\text{zero}())) \). Note that \( \text{isCanonical}(ca, A_0) = T \), but that \( \text{isCanonical}(ca, A_1) = F \) as \( \text{Sem}(ca = \text{zero}(), emptyva, A_1) = T \).

---

Example 2.3.

(\text{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

\[ \text{newStack} : \rightarrow \text{Stack} \]
\[ \text{push} : \text{Stack} \times \text{Int} \rightarrow \text{Stack} \]
\[ \text{pop} : \text{Stack} \rightarrow \text{Stack} \]
\[ \text{top} : \text{Stack} \rightarrow \text{Int} \]

axioms
∀s, s₀, s₁ ∈ Stack, i, i₀, i₁ ∈ Int •

\[\text{pop(push(s, i))} = s\]
\[\text{top(push(s, i))} = i\]
\[\text{newStack()} \neq \text{push(s, i)}\]
\[\text{push(s₁, i₁)} = \text{push(s₂, i₂)} \iff s₁ = s₂ \land i₁ = i₂\]

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 newStack, 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 \(\text{answer} : \text{ClosedAppl} \rightarrow \text{ClosedAppl}\).

**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

268
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. \( \text{Sem}_{\text{AS}} : \text{ClientSpec} \rightarrow \text{Set}(\text{Answer}) \)

\[
\exists A \in \text{Models}(\text{CS.as.ax}) \quad \forall ca_0 \in \text{ClosedAppls}(\text{CS.as.sig}) \quad \nu \in \text{V} \\
\text{if } \text{Sem}(ca_0, \text{emptyva}, A) = \nu, \\
\text{then there is an } ca_1 \in \text{ClosedAppls}(\text{CS.as.sig}) \text{ such that} \\
\text{answer}(ca_0) = ca_1, \text{ and} \\
\text{Sem}(ca_1, \text{emptyva}, A) = \nu, \text{ and} \\
\text{CS.asCanonical}(ca_1, A) = T.
\]

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 \( \text{ca} \) 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


