Data-Oriented System Development

Prof. Martin Wirsing

30.10.2002

Algebraic Specification with CASL

Goals

- Get to know algebraic specifications
- Write first specifications with CASL
- Use reachability and free extensions for CASL

Simple Algebraic Specifications

Definition:

Let $\Sigma = (S, F, P)$ be a signature and E a set of (closed) Σ-formulas. Then $SP = \langle \Sigma, E \rangle$ is an axiomatic specification.

SP is called algebraic specification, if P is empty. (i. e. if only the equality symbol occurs as a predicate.)

Example (Associativity):

spec ASSOC

sorts	Elem
ops	$\_ \circ \_$: Elem $\times$ Elem $\rightarrow$ Elem
vars	x, y, z : Elem
axioms	$(x \circ y) \circ z = x \circ (y \circ z)$

end

Example (Simple Database):

spec Database =

sorts	Database; String; Nat
ops	initial: Database;
	look-up: Database $\times$ String $\rightarrow$? Nat;
	update: Database $\times$ String $\times$ Nat $\rightarrow$ Database
vars	s: Database; v,w: String; n: Nat
axioms	[initial]% $\neg$def look-up(initial,w);
	[look-up1]% $v = w \Rightarrow$ look-up(update(s,v,n),w) = n;
	[look-up2]% $v \neq w \Rightarrow$ look-up(update(s,v,n),w) = look-up(s,w)

end

Example (BOOL):

spec BOOL1 =

sorts	Bool
ops	true, false: Bool
axioms	$\neg$(true = false);
	$\forall x$: Bool.x = true $\vee$ x = false

end

For extending specifications by additional sorts, operations and axioms we use the keyword **then**:

spec BOOL =
BOOL1 **then**
ops not _ : Bool $\rightarrow$ Bool;
_ and _, _ or _ : Bool $\times$ Bool $\rightarrow$ Bool
vars x: Bool
axioms not(true) = false;
not(false) = true;
true and x = x;
false and x = false;
true or x = true;
false or x = x
end

Specifications with Constructors

Definition (Constructors):

Given $\Sigma = (S, F, P)$ and $s \in S$.

- An expression g of the form

  generated { **sorts** s; **ops** C}

  introduces the operations C as constructors for s.
  It is required syntactically that every $c \in C$ has the domain s ("generating constraint"), i.e.

$$C \subseteq \bigcup_{w \in S^*} F_{\langle w,s \rangle}$$

- A Σ-structure A satisfies g ($A \models g$), if for every element $a \in A_s$, there is an assignment $\nu : X \to A$ with $\nu(x) = a$ (and $x \in X_s$) and a term $t \in T((S, C), (X_{s'})_{s' \neq s})$. s.t.

$$A, \nu \models x = t$$

  C is then called a constructor set for s.

Example:

The declaration of a constructor set replaces the axiom that constrains the domain to true, false.

spec BOOL_g =
  generated { **sorts** Bool; **ops** true, false: Bool }
  axioms ¬ (true = false)
end

Abbreviation for "**generated** { **sorts** Bool; **ops** true, false: Bool }":
  generated type Bool ::= true | false

Note: In the literature an algebraic specification is called flat algebraic specification if:

1. every sort is constructed via **generated { sort** s,**ops** $F_{w,s}$ } and
2. the axioms consist of equations and quantified equations.

Constructor sets can be defined as data type declarations in BNF-expressions. A datatype declaration for the sort s is written as

$$s ::= A_1 \mid \ \ldots \mid \ A_n$$

Example (Natural numbers):

spec NAT_g =
  generated type Nat ::= zero | succ(Nat)
  vars x, y: Nat
  axioms $\neg$(zero = succ(x));
  succ(x) = succ(y) $\Rightarrow$ x = y
end

Example (Database):

spec Database_g=
  generated type Database ::= initial | update(Database;String;Nat)
  ops look-up: Database $\times$ String $\rightarrow$ Nat
  vars s: Database; v,w: String; n: Nat
  axioms [initial]% $\neg$def look-up(initial,w);
  [look-up1]% $v = w \Rightarrow$ look-up(update(s,v,n),w) = n;
  [look-up2]% $v \neq w \Rightarrow$ look-up(update(s,v,n),w) = look-up(s,w)
end

Specification of Initial and Free Structures

Definition (Free sub structures):

- Let $\Sigma = (S, F, P)$ and $s \in S$. An expression g of the form

$$\textbf{free}\{\ \textbf{sorts}\ s; \textbf{ops}\ C\ \}$$

  introduces sort s and the operations C as free constructors for s.
  Syntactically we require: for every $c \in C$,

$$C \subseteq \bigcup_{w \in S^*} F_{\langle w,s \rangle}$$

- Let S_0 be the set of all sorts of C which are different from s. A Σ-structure A fullfils g ($A \models g$), if the substructure $A|_{(S,C)}$ is a free extension of $A|_{(S_0,\emptyset)}$.
- Abreviation:

$$\textbf{free type}\ s ::= c_1 \mid \ldots \mid c_n \qquad (\text{for } C = \{c_1, \ldots, c_n\})$$

Example (Natural numbers):

spec NAT_f=
  free type Nat ::= zero | succ(Nat)
end

spec NAT_f1 =
  BOOL1 and NAT_f **then**
  ops eq: Nat × Nat → Bool
  vars x,y: Nat
  axioms eq(zero,zero) = true;
  eq(zero,succ(x)) = false;
  eq(succ(x),succ(y)) = eq(x,y);
  eq(x,y) = eq(y,x)
end

Example (WORD):

spec WORD=
  sorts Alph
  free types Word ::= make(Alph) | add(Word;Alph)
end

Example (Lists):

The structure for finite lists is a free extension of the element type Elem.

spec LIST0=

sorts	Elem
free types	List ::= nil | cons(Elem; List)
ops	first: List →? Elem;
	rest: List →? List
vars	x:Elem; l:List
axioms	¬def first(nil);
	¬def rest(nil);
	first(cons(x,l)) = x;
	rest(cons(x,l)) = l

end

Remark: first and rest are the selector operations for the arguments of cons.
Write shortly:

free types List ::= nil | cons(first: Elem; rest: List)

Example (Database):

spec DB_f=

sorts Data; Key

free types Database ::= initial | update(Database;Key;Data)

axioms [initial]% ¬def look-up(initial,w);

[look-up1]% $v = w \Rightarrow$ look-up(update(s,v,n),w) = n;

[look-up2]% $v \neq w \Rightarrow$ look-up(update(s,v,n),w) = look-up(s,w)

end

Specification of Basic Computational Structures

Example (Stack):

spec STACK =

sorts	Elem
free types	Stack ::= empty | push(Elem; Stack)
ops	top: Stack $\rightarrow$? Elem;
	pop: Stack $\rightarrow$? Stack
vars	d:Elem; s:Stack
axioms	$\neg$def top(empty);
	$\neg$def pop(empty);
	top(push(d,s)) = d;
	pop(push(d,s)) = s

end

Example (Loose finite Sets):

spec LSETNAT =
    BOOL1 **and** NAT_f1 **then**
    generated type Set ::= empty | add(Nat; Set)
    ops iselem: Nat × Set → Bool
    vars b: Bool; x,y: Nat; s: Set
    axioms iselem(x, empty) = false;
    iselem(x, add(y,s)) = or(eq(x, y), iselem(x, s))
end

Example (Binary Trees):

spec TREE =

sorts	Elem
free type	Tree ::= empty|node(Tree;Elem;Tree)
ops	left,right:Tree→Tree;
	label:Tree→? Elem
vars	t_1, t_2 :Tree; d :Elem
axioms	left(empty) =empty;
	¬def label(empty);
	right(empty) =empty;
	left(node(t_1, d, t_2)) = t_1;
	label(node(t_1, d, t_2)) = d;
	right(node(t_1, d, t_2)) = t_2

end

Structured Specifications

Let $Spec$ be the set of all specifications, $Sign$ the set of all signatures.

For every SP $\in Spec$ is

- Sig(SP) $\in Sign$ the signature of SP and
- Mod(SP) $\subseteq$ Struct(Sig(SP)) w.r.t. Mod(SP) $\in$ Alg(Sig(SP)) the class of models of SP

Definition (Sum of two Specifications):

$$\begin{array}{rcl} \mathrm{Sig}(\mathrm{SP}_1 \ \textbf{and} \ \mathrm{SP}_2) & = & \mathrm{Sig}(\mathrm{SP}_1) \cup \mathrm{Sig}(\mathrm{SP}_2) \\ \mathrm{Mod}(\mathrm{SP}_1 \ \textbf{and} \ \mathrm{SP}_2) & = & \{\mathrm{A} \in \mathrm{Alg}(\mathrm{sig}(\mathrm{SP}_1 \ \textbf{and} \ \mathrm{SP}_2)) \mid \\ & & \mathrm{A} \mid_{Sig(SP_1)} \in \mathrm{Mod}(\mathrm{SP}_1) \text{ and} \\ & & \mathrm{A} \mid_{Sig(SP_2)} \in \mathrm{Mod}(\mathrm{SP}_2) \ \} \end{array}$$

Example (Specification NAT and BOOL):

$$\text{Mod(NAT } \textbf{and} \text{ BOOL)} = \{A \in \text{Alg(Sig(NAT } \textbf{and} \text{ BOOL))} \mid A|_{Sig(NAT)} \in \text{Mod(NAT)}, A|_{Sig(BOOL)} \in \text{Mod(BOOL)} \}$$

Note: Shared sort and function symbols get identified. If shared parts of $SP1$ and $SP2$ are inconsistent, then is $SP1$ **and** $SP2$ inconsistent.

Example (Inconsistency):

$SP_1 = \langle(\{s\}, \{a, b : s, f : s \to s\}, \neg(a = b)\rangle$

$SP_2 = \langle(\{s\}, \{a, b, c : s\}, (a = b)\rangle$

$\text{Sig}(SP_1 \textbf{ and } SP_2) = (\{s\}, \{a, b, c : s, f : s \to s\})$

$\text{Mod}(SP_1 \textbf{ and } SP_2) = \emptyset$

The extension of signature SP by new sorts, function symbols and axioms using then can be reduced to the sum.

Definition (Extension):

SP **then sorts** S; **ops** F; **axioms** E **end** $=_{def}$
SP **and** $\langle (sorts(Sig(SP)) \cup S, ops(Sig(SP)) \cup F), E \rangle$

Definition (Hiding):

$$\begin{array}{lcl} \text{Sig}(SP \textbf{ hide } (S_1,F_1)) & = & \Sigma^- \\ \text{Mod}(SP \textbf{ hide } (S_1,F_1)) & = & \{B|_{\Sigma^-} \mid B \in \text{Mod}(\Sigma)\} \end{array}$$

Let S_1, F_1 be lists of sort and function symbols, let $\Sigma = (S, F)$ be a signature.
$\Sigma^- =_{def} \Sigma - (S_1, F_1)$ denotes the signature where

- all sort and function symbols of S_1 and F_1 and
- all function symbols that have an element of S_1 in their functionality

are discarded.

Example (Sorting Lists of Natural Numbers):

spec INSSORT =
(LISTNAT **then**
  ops insert: Nat $\times$ List $\rightarrow$ List;
  sort: List $\rightarrow$ List
  vars x,y: Nat; l: List
  axioms insert(x, empty) = cons(x, empty);
  insert(x, cons(y, l)) = cons(x, cons(y, l));
  when x leq y **else** cons(y, insert(x, l));
  sort(empty) = empty;
  sort(cons(x, l)) = insert(x, sort(l))
) **hide** insert
end

Definition (Signature Morphism):

- Let $\Sigma = (S, F)$ and $\Sigma' = (S', F')$ be signatures. A signature morphism is a renaming of sorts and function symbols in such a way that the funcionality of the renamed function symbols respects the renaming of the sorts. Formally a mapping
  $\sigma = (\sigma_{sort}, \sigma_{op})$ with $\sigma_{sort} : S \to S' \qquad \sigma_{op} : F \to F'$
  is called signature morphism ($\sigma : \Sigma \to \Sigma'$), if for all $f \in F_{\langle\langle s_1,\ldots,s_n\rangle,s\rangle}$
  $\sigma_{op}(f) : \sigma_{sort}(s_1), \ldots, \sigma_{sort}(s_n) \to \sigma_{sort}(s)$
- Let $\sigma : \Sigma \to \Sigma'$ be an injective signature morphism, $A \in$ Alg(Σ). The σ-translation $\sigma(A)$ of A is the algebra with
  $\sigma(A)_{\sigma_{sort}(s)} =_{def} A_s$ and $\sigma_{op}(f)^{\sigma(A)} =_{def} f^A$
- Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism, B $\in$Alg(Σ'). The σ-reduct B$|_\sigma$ is the Σ-algebra with
  $(B|_\sigma)_s =_{def} B_{\sigma_{sort}(s)}$ and $f^{B|_\sigma} =_{def} (\sigma_{op}(f))^B$

A theory morphism $\alpha : SP1 \rightarrow SP2$ preserves the theory of $SP1$; i.e. $SP2$ satisfies all axioms, generating and free extension constraints of $SP1$ (modulo renaming).

Definition (Theory Morphism, View):

- A theory morphism $\alpha : SP_1 \rightarrow SP_2$ is a signature morphism $\alpha : Sig(SP_1) \rightarrow Sig(SP_2)$, such that for every model $M \in Mod(SP_2)$ holds:

$$(*) \qquad M|_\alpha \in Mod(SP_1)$$

- Let α_0 be a finite mapping between characters. If the signature morphism $\alpha : Sig(SP_1) \rightarrow Sig(SP_2)$ is well defined and a theory morphism, then it defines a View:

$$\textbf{view}\ SM : SP_1\ \textbf{to}\ SP_2 = \alpha_0$$

Parameterised Specifications

A parameterised specification (or generic specification) is of the form

spec SN[SP] =
    Body
end

where

- SN is the name of the parameterised specification
- SP is the name of the formal parameter specification
- Body is the body of the specification

SN is well defined, if the body extends the parameter, i. e. if
SP **then** Body
is a well defined specification.

Example (Parameterised Lists):

spec ELEM =
  sorts Elem
end

spec LIST[ELEM]
  free type List[Elem] ::= nil | cons(first: ?Elem; rest: ?List[Elem])
  ops _ ++ _: List[Elem] $\times$ List[Elem] $\rightarrow$ List[Elem]
  vars e: Elem; l, l': List[Elem]
  axioms nil ++ $l = l$;
  cons(e, l) ++ l' = cons(e, l ++ l')
  ops reverse(nil) = nil;
  reverse(cons(e, l)) = reverse(l) ++ cons(e, nil)
end

Definition (Parameter Passing):

Let **spec** SN[SP] = Body **end** be a generic specification with formal parameter SP, let AP be an actual parameter and SM:SP **to** AP = σ_0 a theory morphism from SP to AP, then

SN[**view** SM] or SN[AP **fit** σ_0]

denotes the instantiation of SN with AP.

Abbr.: SN[AP]

Semantically: (SP **then** Body) **with** σ_0 **and** AP

Example (Instantiation):

The instantiation of LIST[ELEM] with natural numbers is defined via renaming:
LIST[NAT **fit** [Elem $\mapsto$ Nat]]

Summary

- A CASL specification describes abstractly a class of structures with a signature.
- CASL supports "loose" and "free" datatypes with **generated** and **free**.
- Specifications can be structured.
- Renaming is done using signature morphisms.
- Specifications can be parameterised.