Introduction to CASL, the Common Algebraic Specification Language

Franz Lichtenberger
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Franz.Lichtenberger@risc.uni-linz.ac.at
Contents

- Historical remarks
- General remarks about CASL
- Underlying Concepts
- CASL by examples
  (Basic Specifications only)
Historical Remarks

- First papers on Algebraic Specification of Abstract Data Types, around 1975: Liskov/Zilles, Guttag/Horning, ADJ-Group (Goguen, Thatcher, Wagner, Wright)
- Several AlgSpec languages developed in the next 20 years
- The specification language defined by CoFI: CASL – The Common Algebraic Specification Language
Algebraic Specification

- Observation/Claim/Thesis:
  - Data types are algebras (set(s)+operations+axioms)
  - Abstract data types (ADTs) are classes of (usually heterogeneous) algebras
  - (Software) Systems can be specified by ADTs.
- Thus:
  - ADTs / Software systems can/should be specified algebraically
- CASL: a language for specifying requirements and designs, not a programming language
CASL

- Core of a family of languages:
  - restrictable (e.g. for executability)
  - extendable (higher order, state based, reactive, modal, ...)
- CASL specifications denote classes of models
- CASL has a complete formal definition
- Abstract and concrete syntax are defined formally
- CASL has a complete formal semantics
- The foundations of CASL are rock-solid!  
  (Claim by CoFI)
CASL Specifications

- Basic specifications (this talk)
- Structured specifications
  - Large and complex specifications are easily built out of simpler ones by means of (a small number of) specification building operations
- Architectural specifications
  - impose structure on implementations
- Libraries
  - are named collections of named specifications
  - The CASL Basic Libraries contain the standard datatypes
Underlying concepts

- CASL is based on standard concepts of algebraic specification
- Basic specifications declare symbols, and give axioms and constraints
- The semantics of a basic specification is (a signature and) a class of models:
  - A signature $\Sigma$ corresponding to the symbols introduced by the specification
  - A class of $\Sigma$-models, corresponding to those interpretations of the signature that satisfy the axioms and constraints of the specification
- When a model $M$ satisfies a specification $SP$ we write $M \models SP$
Specifications

- CASL specifications may declare
  - sorts
  - subsorts
  - operations
  - predicates
- A spec is called many-sorted if it has no subsort specifications (otherwise subsorted)
- A spec is called algebraic if it has no predicate declarations
Sorts

- A *sort* is a symbol which is interpreted as a set, called a *carrier set*.
- The Elements of carrier sets are abstract representations of data: numbers, characters, lists, trees, etc.
- A *sort* is approx. a *type* in a programming language.
- CASL allows *compound sort* symbols, i.e. List[Int]
Subsorts

- Subsort declarations are interpreted as *embeddings*
- Set inclusion would be sufficient for, e.g. $\text{Nat} < \text{Int}$
- Embedding is necessary for, e.g. $\text{Char} < \text{String}$
  (Char and String are disjoint)
- An embedding is a 1-1 function
Operations

- Operations may be declared as *total* or *partial*
- An operation symbol consists of its *name* together with its *profile*
- *Profile:* number and sort of arguments, and *result sort*
- An operations is interpreted as a *total* or a *partial function* from the Cartesian product of the carrier sets of the arguments to the carrier set of the result sort
- The result of applying an operation is *undefined* if any of the arguments is *undefined*
- *Constant:* operation with no arguments, interpreted as an element of the carrier set of the result sort
Predicates

- A predicate symbol consists of a name and its profile
- Profile: number and sorts of the arguments, but no result sort
- Predicates are different from boolean-valued operations!!!
- Predicates are used to form atomic formulas, rather than terms
- A predicate symbol is interpreted as a relation on (i.e., a subset of) the Cartesian product of the carrier sets of the argument sorts
- Predicates are never undefined, they just do not hold if any of the arguments is undefined (two-valued logic)
- For boolean-valued operations: three-valued logic (true, false, undefined)
Overloading

- Operation and predicate symbols may be *overloaded*, i.e.
- can be declared with different profiles in the same specification
- Examples:
  - ‘empty’ for empty list and empty set
  - < : predicate on unrelated sorts such as Char and Int
- Overloading has to be *compatible* with embeddings between subsorts, i.e.
- for sorts Nat < Int, operation +, predicate <: interpretations are required to be such that it makes no difference whether the embedding from Nat to Int is applied to the arguments (and the result) or not
Atomic Formulas

- predicate applications
- equations (strong or existential)
  - existential: both sides are defined and equal
  - strong: hold as well, if both sides are undefined
- definedness assertions
- subsort membership assertions
Axioms

- Axioms are *formulas of first-order logic*
- *Logical connectors* have usual interpretation
- Quantification: universal, existential, unique-existential
- Interpretation of quantification: completely standard!
- Variables in formulas range over the carrier sets of specified sorts
- An axiom either holds or does not hold in a particular model: there is no “maybe” or undefinedness about holding (regardless of whether the values of terms occurring in the axioms are defined)
Constraints

- Sort generation constraints eliminate “junk” from specific carrier sets, i.e. restrict the class of models
- Default case: all sets allowed as carriers
- *Generated*: no junk
  
  all elements can be obtained by consecutively applying the operations of the sort in question
- *Free*: no junk, no confusion
  
  generated, and no equations hold except those implied by the axioms
CASL by examples

- Simple specifications can be written essentially as in many other algebraic specification languages
- CASL provides useful abbreviations and annotations
- Tools: the Heterogeneous Tools Set (HETS) is the main analysis tool for CASL;
  - it provides a parser, static analysis and translation to an intermediate/exchange format (so called A-terms)
- Useful only together with other tools like theorem provers (Isabelle/HOL, etc.), SW-development environments, ...
- Not tried out yet!
Loose specifications

```plaintext
spec Strict_Partial_Order =
%%% Let's start with a simple example !
  sort Elem
  pred __<__ : Elem * Elem
%%% pred abbreviates predicate
  forall x, y, z:Elem
   . not (x < x) %(strict)%
   . (x < y) => not (y < x) %(asymmetric)%
   . (x < y) \& (y < z) => (x < z) %(transitive)%
%%%{ Note that there may exist x, y such that
    neither x < y nor y < x. }%
end
```
Specification extension

spec Total_Order =
  Strict_Partial_Order
then
  forall x, y:Elem
  . (x < y) ∨ (y < x) ∨ x = y %(total)%
end
Abbreviations

spec Total_Order_With_MinMax =
   Total_Order
then
   ops min(x, y :Elem): Elem = x when x < y else y;
   max(x, y :Elem): Elem = y when min (x, y) = x else x
end
abbreviates
   forall x,y:Elem . min(x,y) = x when x<y else y
which abbreviates
   (x<y => min(x,y)=x) \land (not(x<y) => min(x,y)=y)
Pretty-printing

spec Partial_Order =
  Strict_Partial_Order
then
  pred __<=__(x, y :Elem) <=> (x < y) \lor x = y
end

“less or equal” can be pretty-printed using

%display __<=__ %LATEX __\le__

Not tried out yet!
Redundancy with %implies annotation

\text{spec Partial\_Order\_1 =}
\hspace{1em} \text{Partial\_Order}
\text{then %implies}
\hspace{1em} \text{forall x, y, z:Elem}
\hspace{2em} \text{. (x <= y) \land (y <= z) => (x <= z) % (transitive) %}
\text{end}

\text{Can be used to generate the proof obligation}

\text{Partial\_Order \models (x <= y) \land (y <= z) => (x <= z)}
Attributes

\[
\text{spec Monoid =}
\begin{align*}
\text{sort Monoid} \\
\text{ops 1 : Monoid;}
\end{align*}
\]

\[
\_\_*\_ : \text{Monoid * Monoid -> Monoid, assoc, unit 1}
\]

\textbf{assoc} abbreviates, as expected, the following axiom:

\[
\forall x, y, z : \text{Monoid} . \ (x * y) * z = x^* (y * z)
\]
Generic specifications via parameters

spec Generic_Monoid [sort Elem] =
  sort Monoid
  ops inj : Elem -> Monoid;
    1 : Monoid;
    __*__ : Monoid * Monoid -> Monoid, assoc, unit 1
  forall x, y:Elem
    . inj (x) = inj (y) => x = y
end
Datatype declarations by constructors

spec Container [sort Elem] =
  type Container ::= empty | insert(Elem; Container)
  pred __is_in__ : Elem * Container
  forall e, e':Elem; C:Container
    . not (e is_in empty)
    . (e is_in insert (e', C)) <=> e = e' V (e is_in C)
end

Abbreviation for:

sort Container
ops empty: Container;
    insert: Elem * Container -> Container
Generated Specifications

spec Generated_CONTAINER [sort Elem] =
    generated type Container ::= empty | insert(Elem; Container)
    pred __is_in__ : Elem * Container
    forall e, e':Elem; C:Container
    . not (e is_in empty)
    . (e is_in insert (e', C)) <=> e = e' \lor (e is_in C)
end

- Generated types allow induction over the constructors!
Free specifications

\[
\begin{align*}
\text{spec Natural} &= \quad \text{free} \\
\text{type Nat} &::= 0 \mid \text{suc(Nat)} \\
\end{align*}
\]

Equivalent to

\[
\begin{align*}
\text{generated type Nat} &::= 0 \mid \text{suc(Nat)} \\
\forall x,y: \text{Nat} . \ \text{suc}(x)=\text{suc}(y) \Rightarrow x=y \\
\forall x:\text{Nat} . \ \neg(0 = \text{suc}(x))
\end{align*}
\]
Enumerated types

- Free datatype declarations are particularly convenient for defining enumerated types

```plaintext
spec Color =
    free type RGB ::= Red | Green | Blue
    free type CMYK ::= Cyan | Magenta | Yellow | Black
end
```

With `generic` instead of `free` one would have to add

```plaintext
not(Red=Green) \land not(Red=Blue) \land ...
```
Freeness constraints

\[
\text{spec Integer = free \{type Int ::= 0 | suc(Int) | pre(Int)}
\]
\[
\text{  forall x:Int}
\]
\[
\text{    . suc (pre (x)) = x}
\]
\[
\text{    . pre (suc (x)) = x\}}
\]
end
Predicates and freeness

- Predicates hold minimally in models of free specifications

```
spec Natural_Order =
    Natural
then
free {pred __<__ : Nat * Nat
    forall x, y:Nat
    . 0 < suc (x)
    . (x < y) => (suc (x) < suc (y))}
end
```
Inductive definitions of operations/predicates

```
spec Natural_Arithmetic =
  Natural_Order
then
  ops 1: Nat = suc (0);
   ___+___ : Nat * Nat -> Nat;
   ___*___ : Nat * Nat -> Nat
forall x, y:Nat
  . x + suc (y) = suc (x + y)
  . x * 0 = 0
  . x * suc (y) = (x * y) + x
end
```