# CASL

## Common Algebraic Specification Language

# Guided Tour

Peter D. Mosses (BRICS, Aarhus, Denmark)

External Relations Coordinator of

CoFI: The Common Framework Initiative

for algebraic specification and development

# Plan

- Background
- CASL
  - basic specifications
  - structured specifications
  - architectural specifications
  - libraries of specifications
- Foreground

# Background

- Lack of a common framework meant
  - industrial acceptance: *discouraged*
  - dissemination: *hindered*
  - tool applicability: *limited*
- Why not agree on a common framework?
  - concepts
  - language
  - methodologies

# Aims

- *a common framework*—a collaborative effort
- for *algebraic* specification & development
- *attractive*
  - short term: to researchers; long term: for use in industry
- providing a *common specification language*
  - syntax: uniform, user-friendly; semantics: straightforward
- subsuming *many* previous frameworks
- good documentation & *tool support*
- *free*—but protected!

# Scope

- specifying functional requirements
- formal development & verification of software
- relation of specifications to:
  - informal requirements
  - implemented code
- prototyping, theorem-proving
- libraries, reuse, evolution
- providing tool interoperability

# CASL:
## Common Algebraic Specification Language

- critical selection of existing constructs
- expressive, simple, pragmatic
- specifying requirements and design
  for conventional software packages
- restrictions to sub-languages
- extensions to higher-order, state-based, concurrent, …

# CASL Design Process

- design started: September 1995
- design proposed: May 1997
  - summary, abstract syntax, semantics
  - tentatively approved by IFIP WG1.3
- design finalized: October 1998
  - complete language summary available
  - formal semantics, rationale are being updated

# Parts of CASL

- Basic specifications
  - declarations, definitions, axioms
- Structured specifications
  - translations, reductions, unions, extensions, freeness
  - named specifications, generic specifications, views
- Architectural specifications
  - implementation units, composition
- Specification libraries
  - distributed

# Aspects of CASL design

- Design issues
  - methodology, tools, aesthetics
- Semantic concepts
  - formal (institutions, environments)
  - informal (expansions, scopes)
- Language constructs
  - abstract syntax (structure, annotations)
  - concrete syntax (input, display)

# Presentation

| Specification | Pragmatic issues | Semantic concepts | Language constructs |
|---|---|---|---|
| Basic | ⇒ | ⇒ | ⇒ |
| Structured | ⇒ | ⇒ | ⇒ |
| Architectural | ⇒ | ⇒ | ⇒ |
| Libraries | ⇒ | ⇒ | ⇒ |

# Basic specifications

- Design issues
- Semantic concepts
- Language constructs
  - concrete syntax (display format)
  - trivial illustrative examples

# Design issues

- Partial and total functions
- Logic and predicates
- Classes of models
- Overloading
- Subsorts
- Datatype constructors/selectors

# Partial and total functions

- error propagation implicit (strictness)
  - any argument undefined $\Rightarrow$ result undefined
- totality an important special case
  - all arguments defined $\Rightarrow$ result defined
- domain of definition may be implicit
  - not always a natural sort/subsort
- equations may require definedness or not
  - existential equations require it, strong do not

# Logic and predicates

- classical 2-valued first-order logic
  - standard interpretation of connectives
- predicates
  - better than (total) Boolean functions!
  - any argument undefined $\Rightarrow$ unsatisfied
- quantification, connectives
  - expressiveness
  - restrictions in sublanguages

# Classes of models

- all models
  - loose semantics
- generated models
  - only expressible values considered
  - induction proofs sound
- initial (and freely-generated) models
  - minimal satisfaction, definedness
  - may not exist (disjunction, negation, ...)

# Overloading

- declaration
  - same symbol, various profiles (arg/result sorts)
    - » e.g. '+' on integers, reals, strings, …
- use
  - intended profile determined by context
  - disambiguation when needed by specifying profile (or result sort) of applications

# Subsorts

- embeddings
  - not necessarily inclusions (e.g. Int values represented differently from Real values)
- commute with overloaded functions
  - values independent of profiles
  - 2+2=4 (natural numbers, integers, …)
- no ‘regularity’ conditions on declarations
  - for simplicity, modularity

# Datatype constructors/selectors

- frequently needed
  - correspond to (unions of) record types, enumeration types
- abbreviate tedious declarations, axioms
  - for constructors and selectors
- may be loose, generated, or free

# Semantic concepts

- Signatures
- Models
- Sentences
- Satisfaction
- Sort generation constraints
- Institution
- Semantic functions

# Signatures

$\Sigma = (S, TF, PF, P, \leq)$

- sorts $S$
- operations: total $TF$, partial $PF$ (disjoint)
- predicates $P$
- subsort embeddings $\leq$
- unrestricted overloading

# Models

$M \in \mathbf{Mod}(\Sigma)$, $\Sigma = (S, TF, PF, P, \leq)$

- carrier sets are non-empty
- embedding, overloading are compatible
  - embeddings commute with overloaded ops
- operations are partial functions
  - necessarily total when so declared
- predicates are relations on carriers

# Sentences

$\Phi \in \mathbf{Sen}(\Sigma), \Sigma = (S, TF, PF, P, \leq)$

- closed, many-sorted first-order formulae
- atomic formulae:
  - equations: strong, existential
  - definedness, membership assertions
  - fully-qualified predicate applications
- terms:
  - fully-qualified operation applications, variables, sorted terms, casts

# Satisfaction

$M \models \Phi$, $M \in \mathbf{Mod}(\Sigma)$, $\Phi \in \mathbf{Sen}(\Sigma)$

- the value of a term may be undefined
- when value of argument undefined:
  - predicate never holds
  - operation result always undefined
- existential equation: defined *and* equal
- strong equation: both undefined *or* equal

# Sort generation constraints

$\Sigma = (S, TF, PF, P, \leq)$

- sentences $\Phi \in \mathbf{Sen}(\Sigma)$:
  $(S', F') \subseteq (S,F)$, where $F = TF \cup PF$
- satisfaction $M \models \Phi$:
  - when carriers of sorts in $S'$ generated by functions in $F'$ (possibly from sorts not in $S'$)

# Institution

$M \in \mathbf{Mod}(\Sigma)$, $\Phi \in \mathbf{Sen}(\Sigma)$, $\Sigma = (S, TF, PF, P, \leq)$

- morphisms $\sigma: \Sigma \to \Sigma'$
  - preserve overloading, embeddings, totality
  - determine reducts $\mathbf{Mod}(\sigma) : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$
- homomorphisms $h: M_1 \to M_2$ $\quad (M_1, M_2 \in \mathbf{Mod}(\Sigma))$
  - preserve operation values, holding predicates
- translation preserves satisfaction
  - $M' \models \sigma(\Phi)$ iff $\mathbf{Mod}(\sigma)(M') \models \Phi$

# Semantic functions

- expansion of atomic formulae
  - gives a set of well-sorted fully-qualified atoms
  - well-formed when unique (up to embeddings)
- reduction from subsorts to embeddings
  - $\Sigma = (S, TF, PF, P, \leq) \ \mapsto$
    $\Sigma^{\#} = (S, TF, PF, P) \cup \{\mathrm{emb}_{s,s'} \mid \ s \leq s' \} \cup \ldots$
- basic specification semantics:
  - signature $\Sigma$, class of models $M \subseteq \mathbf{Mod}(\Sigma)$

# Language constructs

- Signature declarations
  - sorts, operations, predicates, datatypes
  - sort generation constraints
- Axioms
  - variable declarations
  - quantifiers, connectives, atoms, terms
- Scope rules: declaration before use

# Sorts

- declarations

  **sorts** *Elem, List*

  **sorts** *Nat* < *Int*

- subsort definitions

  **sort** *Pos* = { *n:Nat* • *n* > *0* }

# Operations

- declarations

  **ops** $0 : Nat$ ;
  $suc : Nat \to Nat$ ;
  $pre : Nat \to ? Nat$ ;
  $__+__ : Nat \times Nat \to Nat$, **assoc, comm, unit** $0$

- definitions

  **ops** $1 : Nat = suc(0)$ ;
  $dbl(n{:}Nat) : Nat = n+n$

# Predicates

- declarations

  **preds** $odd : Nat$ ;

  $__ < __ : Nat \times Nat$

- definitions

  **preds** $even(n{:}Nat) \Leftrightarrow \neg\, odd(n)$ ;

  $__ \leq __ (m,n{:}Nat) \Leftrightarrow m < n \vee m = n$

# Datatypes

- non-linear visibility within types

**type** *Set* ::= *Ø* | {|__|}*(Elem)* | __∪__*(Set; Set)*

**free type** *Bit* ::= *0* | *1*

**free type** *Pair* ::= *pair(left,right:Elem)*

**free type** *List* ::= *nil* | *cons(hd:?Elem; tl:?List)*

**free types** *List* ::= *nil* | **sort** *NeList;*
*NeList* ::= *cons(hd:Elem; tl:List)*

# Sort generation constraints

- types

  **generated type** *Bag ::= empty* | *add(Elem; Bag)*

  **generated types** *Nat ::= 0* | **sort** *Pos;*

  *Pos ::= suc(pre:Nat)*

- declarations

  **generated**

  { **sorts** *Pos < Nat ;*

  **ops** *0 : Nat ;  suc : Nat → Pos* }

# Axioms

- lists of axioms

  **vars** $m,n : Nat;\ p : Pos$

  **axioms** $n \prec m \Rightarrow \ldots\ ;\ \ suc(n)=p \Rightarrow \ldots\ ;\ \ldots$

  **vars** $x, y, z : Elem$

  - $x \preceq x$
  - $x \preceq y \wedge y \preceq z \Rightarrow x \preceq z$

- quantifiers

  $\forall n{:}Nat \bullet \exists\ m{:}Nat \bullet n \prec m$

  $\forall p{:}Pos \bullet \exists!\ n{:}Nat \bullet suc(n)=p$

# Axioms…

- logical connectives
  - $even(n) \Leftrightarrow \neg\, odd(n)$
  - $m \leq n \Leftrightarrow m < n \vee m = n$
  - $m < n \Rightarrow \neg\, n = 0$
  - $even(m+n)$ *if* $odd(m) \wedge odd(n)$

# Atomic formulae

- definedness
  - $def\ pre(suc(n)) \wedge \neg\ def\ pre(0)$
- equations: existential, strong
  - $def\ pre(n) \Rightarrow suc(pre(n)) \stackrel{e}{=} pre(suc(n))$
  - $\neg\ ok(x, e) \Rightarrow find(x, cons(e,l)) = find(x, l)$
- subsort membership
  - $n \in Pos \Rightarrow def\ pre(n)$
- predicate application

# Terms

- constants and variables
  - $nil$, $\emptyset$, $empty_set$, $n$, $n'$, $CURRENT_STATE$
- applications (functional, mixfix notation)
  - $cons(e, l)$, $\{|e|\} \cup s$, $(op\ pre{:}Nat \rightarrow ?Nat)(n)$
- sorted terms (identity, embedding)
  - $dbl(suc(n){:}Nat)$
- casts (projection)
  - $pre(n\ as\ Pos)$

# Basic specification example

**free types** $Nat ::= 0 \mid$ **sort** $Pos;$

$Pos ::= suc(pre{:}Nat)$

**op** $pre : Nat \rightarrow ?Nat$

**axioms** $\neg\, def\ pre(0);$

$\forall n{:}Nat \bullet \quad pre(suc(n)) = n$

**pred** $even__ : Nat$

**var** $n{:}Nat$

- $even\ 0$
- $even\ suc(n) \Leftrightarrow \neg\ even\ n$

# Break!

# Structured specifications

- Design issues
- Semantic concepts
- Language constructs

# Design issues

- No model structure
- Names of symbols
- Generic extension

# No model structure

- structure of specification
  ≠ specification of structure!
- models of structured specifications
  - same kind as for basic specifications
- separate architectural specifications

# Names of symbols

- general principle:
  same name, same thing!
  - the “name” of an operation or predicate includes its profile, so overloading does not require identification (except with subsorts)
- applies both to unions and extensions
  - but not to named specifications in libraries
  - clashes can be eliminated by renaming

# Names of symbols…

- auxiliary symbols are to be hidden
  - hidden symbols are irrelevant
  - revealed symbols are significant
- tools may warn about duplicate declarations
  - no problem when same origin

# Generic extension

- specification definitions
  - reuse by reference to name
- generic parameters
  - parameters intended to vary
  - reuse requires instantiation
  - not intended for defining arbitrary functions on specifications (macros)

# Semantic concepts

- signature, class of models
  - as for basic specifications
- institution-independent structure
  - many-sorted, subsorted, …
  - reduct homomorphisms
- local environment
  - visible symbol declarations

# Semantic functions

- compositional
- denotations of complete specifications:
  - signatures and model-classes
- denotations of extending parts:
  - (partial) functions from signatures to extended signatures, and from corr. model classes to model classes

# Language constructs

- translation, reduction, union, extension, free extension, local specifications
- specification definitions
- generic parameters, instantiation, views, compound identifiers

# Translation & reduction

- translation

  NAT **with** *Nat* $\mapsto$ *Natural*, *suc* $\mapsto$ *succ*__

  NAT **with** **op** __+__ $\mapsto$ *plus*,
  **pred** __<__ $\mapsto$ *lt*

  TRUTH-VALUES **with** **sort** *Bool*, **ops** …

- reduction

  NAT **hide** *Pos*, *suc*

  NAT **reveal** *Nat*, *0*, __+__ , __<__ $\mapsto$ *lt*

# Unions & extensions

- union (disjoint, conjoint)

  NAT **and** STRING

  MONOID **and** COMMUTATIVE

- extension (symbols and/or properties)

  NAT **then**

  **sort** $Nat < Int$ ; **ops** $__+__ : Int \times Int \to Int$; …

  BAG **then** **axiom** $\forall b{:}Bag \bullet b \cup b = b$

  – conservativeness: use annotations

# Free specifications

- initiality

  **free**

  { **sort** *Nat* ; **ops** *0* : *Nat* ; *suc* : *Nat* → *Nat* }

- free extension

  **sort** *Elem* **then**

  **free**

  { **type** *Set* ::= $\emptyset$ | {|__|}(*Elem*) | __∪__(*Set*; *Set*)

  **op** __∪__ : *Set* × *Set* → *Set*,

  **assoc, comm, idem, unit** $\emptyset$ }

# Local specifications

- implicit hiding of auxiliary symbols

**sort** *Nat*
**op** *0 : Nat*
**then local**
**op** *suc : Nat → Nat*
**within**
**op** *__+__ : Nat × Nat → Nat*
**vars** *m, n : Nat*
- *m+0 = m*
- *m+suc(n) = suc(m+n)*

# Named specifications

- only closed specifications can be named
- intended for inclusion in libraries

```
spec NAT =
        free { … }

spec INT =
        NAT then
        free { … }
```

# Generic parameters

- parameters are closed subspecifications

  **spec** ELEM = **sort** *Elem*

  **spec** LIST [ELEM] =
  **free type** *List* ::= *nil* | *cons(Elem; List)*

- instantiation provides fitting arguments

  LIST [NAT **fit** *Elem* ↦ *Nat*]

# Generic parameters…

- sharing between parameter symbols preserved by fitting

  **spec** PAIR [**sort** *Elem1*] [**sort** *Elem2*] =
  **free type** *Pair ::= pair(Elem1; Elem2)*

- sharing between body and arguments restricted to explicit imports (⇒ pushout)

  **spec** LIST [ELEM] **given** NAT =
  **free type** *List ::= nil* | *cons(Elem; List)*
  **op** *length : List* → *Nat*

# Compound identifiers

- for avoiding unintentional name clashes

**spec** ELEM = **sort** *Elem*

**spec** LIST [ELEM] =
**free type** *List[Elem]* ::=
*nil* | *cons(Elem; List[Elem])*

declared sorts now *List[Nat]*, *List[Char]* in:

LIST [NAT **fit** *Elem* ↦ *Nat*] **and**
LIST [CHAR **fit** *Elem* ↦ *Char*]

# Views

- named specification morphisms

  **view** PO2NAT **:** PARTIAL_ORDER **to** NAT **=**
  *Elem* $\mapsto$ *Nat*, __≤__ $\mapsto$ __≤__

  ORD_LIST[**view** PO2NAT]

- implicit identity maps
- implicit unique extension to morphism

  ORD_LIST[NAT]

- generic views

# Structured specification example

**spec** LIST_WITH_ORDER [PARTIAL_ORDER] =
**free type** *List[Elem] ::= nil | cons(hd:?Elem; tl:?List[Elem])*
**then local op** *insert : Elem × List[Elem] → List[Elem];*
**vars** *x,y : Elem; l : List[Elem]*
**axioms**
*insert(x,nil) = cons(x,nil);*
*x≤y ⇒ insert(x,cons(y,l)) = cons(x,insert(y,l));*
*¬(x≤y) ⇒ insert(x,cons(y,l)) = cons(y,insert(x,l))*
**within op** *order[__≤__] : List[Elem] → List[Elem]*
**vars** *x : Elem; l : List[Elem]*
**axioms**
*order[__≤__](nil) = nil;*
*order[__≤__](cons(x,l)) = insert(x,order[__≤__](l))*
**end**

# Architectural specifications

- Design issues
- Semantic concepts
- Language constructs

# Design issues

- Reusability
- Interfaces
- Decomposition of tasks – composition of results

# Reusability

- structure of specifications ⇒ reusable specifications
- specification of structure ⇒ reusable design/implementation units
  - e.g. structure implementation of LIST [NAT] as:
    - » implementation $N$ of NAT and
    - » a function $F$ extending any such $N$ to an implementation of LIST [NAT] and
    - » compose by applying $F$ to $N$

# Interfaces

- explicit assumptions that parts of the implementation make about others
- explicit guarantees about results under the given assumptions

# Decomposition/composition

- architectural specifications specify decomposition of implementation development task into sub-tasks
- also how to compose results of sub-tasks into result of main task

# Semantic concepts

- model of architectural specification:
  - collection of (named) unit constructors
  - unit (constructor) resulting from composition
- unit constructors:
  - constants
  - functions from units to units
    - » persistent, extending arguments
    - » arguments must be compatible

# Language constructs

- architectural specification definitions
- unit declarations, unit definitions
- unit specifications
- unit expressions

# Architectural specifications

- definitions

**arch spec** IMP_NAT_LIST =
  **units** $N$ : NAT ;
    $F$ : NAT $\rightarrow$ LIST[NAT]
  **result** $F[N]$

# Unit declarations & definitions

- declarations

  $N$ : NAT

  $L$ : LIST[NAT]

  $L$ : LIST[NAT] **given** $N$

  $F$ : NAT $\rightarrow$ LIST[NAT]

- definitions

  $L = F[N]$

  $L = F[N$ **fit** $\ldots]$ **hide** $\ldots$

# Unit specifications

- definition

  **unit spec** GEN_LIST = NAT → LIST[NAT]

- use

  $F$ : GEN_LIST

- architectural specifications may also be used as unit specifications

# Unit expressions

- application (to compatible arguments)

  *F[N]*, *F[N* **fit**…*]*

- abstraction

  λ*N:NAT* • …*N*…

- translation, reduction

  *U* **with**…, *U* **hide**…, …

- amalgamation (of compatible units)

  *N* **and** *C*

# Architectural specification example

**spec** LIST[ELEM] =

**free type** *List[Elem]* ::=

*nil* | *cons(Elem; List[Elem])*

**arch spec** CN_LIST =

**units** *C* : CHAR ;

*N* : NAT ;

*F* : ELEM → LIST[ELEM]

**result** *F[C* **fit** *Elem* ↦ *Char]* **and**

*F[N* **fit** *Elem* ↦ *Nat]*

# Libraries of specifications

- Design issues
- Semantic concepts
- Language constructs

# Design issues

- visibility of symbols between specifications
  - none
- visibility of specification names
  - linear
- distributed libraries on Internet
  - indirect links
  - explicit downloading
- version control

# Semantic concepts

- global environment
  - mapping specification names to their denotations
- directory of registered libraries
  - mapping library names to their registered URLs
  - giving access to previous versions of libraries
- state of the universe (Internet, local files)

# Language constructs

- Local libraries
- Distributed libraries

# Local libraries

- definition before use

**library** CoFI_Beans

**spec** Nat = …

**view** …

**arch spec** … = …Nat…

**unit spec** …

# Distributed libraries

- downloading

  **from** CoFI_Beans **get** List, Nat $\mapsto$ Natural

- version

  **library** CoFI_Beans **version** …

  **from** CoFI_Plant **version** … **get** …

  – default is current (registered) version

# Foreground

- Language design tasks
- Semantics tasks
- Tools tasks
- Methodology tasks
- Reactive system tasks
- Overall coordination:
  - Peter D. Mosses (Aarhus) Sep 95 – Aug 98
  - Don Sannella (Edinburgh) Sep 98 – …

# Language design tasks

- coordination:
  - Bernd Krieg-Brückner (Bremen)
- finalization of CASL design
- documentation of design
- sub-languages
  - total, many-sorted, equational, …
- extensions
  - higher-order, state-based, …

# Semantics tasks

- coordination:
  - Andrzej Tarlecki (Warsaw)
- update of formal semantics to v0.99
- choice of proof system for CASL

# Methodology tasks

- coordination:
  - Michel Bidoit (Cachan)
- user's guide
- case studies
- relation of specifications to requirements and code
- development processes

# Tools tasks

- coordination:
  - Hélène Kirchner (Nancy)
- parsers
  - 4 prototypes implemented
- interoperability of existing tools
- formatting (LaTeX package, HTML filter)
- static analysis, proof tools
  - prototype implemented

# Reactive system tasks

- coordination:
  - Egidio Astesiano (Genova)
- extensions of CASL for reactive system specification
- combination of CASL with concurrency

# External relations tasks

- coordination:
  - Peter D. Mosses (Aarhus)
- relation to existing languages
  - ASF+SDF, CAFEOBJ (so far superficially)
  - help needed from experts!
- liaison with IFIP WG1.3
- web presentation of CoFI
- tutorial

# Join now!

- all task groups welcome new participants
  - contact the coordinators
- all cofi-* mailing lists open to subscription
  - moderated
  - administered by Majordomo program
  - cofi-list@brics.dk:
    low-volume, announcements only
- all documents available on the web/ftp

# References

- CoFI Home Page
  - http://www.brics.dk/Projects/CoFI/
- CASL Summary
  - …/Documents/CASL/Summary/
- CoFI Rationale
  - *EATCS Bulletin* 59, June1996
  - Proc. TAPSOFT'97, *LNCS* 1214, 1997