Loose Specifications

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at

Specifications

We will introduce various flavors of specifications of ADTs.

- Specification semantics: $sp \to \mathcal{M}(sp)$.
  - Specification sp.
  - Its meaning $\mathcal{M}(sp)$ (an abstract datatype).
- sp is an adequate specification of an ADT $\mathcal{C}$:
  - $\mathcal{C} \subseteq \mathcal{M}(sp)$.
- sp is a strictly adequate specification of an ADT $\mathcal{C}$:
  - $\mathcal{C} = \mathcal{M}(sp)$.
- sp is a (strictly) adequate specification of an algebra A:
  - sp is (strictly) adequate specification of the monomorphic ADT $[A]$.
- sp is polymorphic (monomorphic):
  - sp defines a polymorphic (monomorphic) ADT.

General notions independent of the kind of specification.

Properties of Specifications

- Is the specification inconsistent?
  - Is the specified ADT empty (i.e. does not contain any algebras)?
- Is the specification monomorphic?
  - Are all algebras of the specified ADT isomorphic?
- Are two specifications equivalent?
  - Do they specify the same ADT?
- Does the specification (strictly) adequately describe a given ADT?
  - Assumes that the ADT is mathematically defined by other means.
    - But specification itself is typically the *only* definition of the ADT.
    - Then no mathematical proof of adequacy is possible.
    - Nevertheless, by "executing the specifications" (mechanically evaluating ground terms), we may investigate the properties of the specified ADT to increase our confidence in its adequacy.

All these questions now have a precise meaning.

Loose Specifications

Take logic L.

- Loose specification $sp = (\Sigma, \Phi)$ in L:
  - Signature Σ, set of formulas $\Phi \subseteq L(\Sigma)$.
- Semantics $\mathcal{M}(sp) = Mod_\Sigma(\Phi)$.
  - All Σ-algebras are candidates for the specified ADT.
    - $Mod_\Sigma(\Phi) = Mod_{Alg(\Sigma),\Sigma}(\Phi)$.

A loose specification specifies as the abstract datatype the class of all models of its formula set.

Concrete Syntax

loose spec
  sorts *sort* ...
  opns *operation* ...
  vars *variable*: *sort* ...
  vars *formula* ...
endspec

- Signature $\Sigma = (\{sort, \ldots\}, \{operation, \ldots\})$.
- Set of formulas $\Phi = \{(\forall variable : sort, \ldots\ .\ formula), \ldots\}$.

We will only use the concrete syntax to define specifications.

Example

loose spec
  sorts $el, bool, list$
  opns
    $True : \to bool$
    $False : \to bool$
    $[\,] : \to list$
    $Add : el \times list \to list$
    $\_\,.\,\_ : list \times list \to bool$
  vars $l, m : list, e : el$
  axioms
    $[\,].l = l$
    $Add(e, l).m = Add(e, l.m)$
endspec

Adequate specification of the "classical" list algebra in EL.

Strict Adequacy

Not a *strictly* adequate specification of the "classical" list algebra.

- Carrier set for *bool* may be a singleton.
  - PL: **axiom** $\neg(True = False)$
- Carrier set for *list* may be a singleton.
  - PL: **axiom** $\forall e : el, l : list \,.\, \neg([\,] = Add(e, l))$
- Size of lists may be bound.
  - PL: **axiom** $\forall e_1, e_2 : elem, l_1, l_2 : list \,.$ $Add(e_1, l_1) = Add(e_2, l_2) \Rightarrow e_1 = e_2 \wedge l_1 = l_2$
- Carrier sets may contain extra values ("junk").
  - There may a *bool* value different from *True* and *False*.
    - PL: **axiom** $\forall b : bool \,.\, b = True \vee b = False$
  - There may be *list* values different from those that can be constructed by application of [ ] and *Add*.
    - No axiom can express this in PL, a solution will be later presented.

In PL (not EL or CEL), additional axioms may solve *some* problems.

Example

loose spec
sorts *bool*, *nat*
opns
$True :\to bool$
$False :\to bool$
$0 :\to nat$
$Succ : nat \to nat$
$\_ + \_ : nat \times nat \to nat$
$\_ * \_ : nat \times nat \to nat$
$\_ \leq \_ : nat \times nat \to bool$
vars $m, n : nat, b : bool$
axioms
$\neg(True = False)$
$b = True \vee b = False$
$\neg(0 = Succ(n))$
$Succ(n) = Succ(m) \Rightarrow n = m$
$(0 \leq n) = True$
$(Succ(n) \leq 0) = False$
$(Succ(n) \leq Succ(m)) = (n \leq m)$
$n + 0 = n$
$n + Succ(m) = Succ(n + m)$
$n * 0 = n$
$n * Succ(m) = n + (n * m)$
endspec

Adequate specification of Peano arithmetic in PL (not strictly adequate because *nat* may contain junk).

Proving Strategies for Loose Specifications

Take loose specification $sp = (\Sigma, \Phi)$ in logic L with inference calculus $\vdash$.

- Prove: $\mathcal{M}(sp) \models \varphi$.
  - Every implementation of the specification sp has the property expressed by formula φ.
  - It suffices to prove $\Phi \vdash \varphi$.
    - Formula φ can be derived from the specification axioms Φ.
- Prove: $\mathcal{M}(sp) \subseteq \mathcal{M}(sp')$.
  - Loose specification $sp' = (\Sigma, \Psi)$.
  - Every implementation of the specification sp is also an implementation of the specification sp'.
  - It suffices to prove $\Phi \vdash \Psi$.
    - Every axiom $\psi \in \Psi$ can be derived from the axioms Φ.

Straight-forward reduction of semantic questions to proving.

Expressive Power of Loose Specifications

Not every abstract datatype can be specified by a loose specification.

- Theorem: $\forall A \in Alg(\Sigma) \,.\, (A \models Th_L(\mathcal{M}(sp))) \Rightarrow (A \in \mathcal{M}(sp))$.
  - $Th_L(C) = \{\phi \in L(\Sigma) \mid \forall A \in C : A \models_\Sigma \phi\}$.
    - The theory of a class of algebras w.r.t. a given logic is the set of all formulas of that logic that are satisfied by every algebra of the class.
  - If an algebra also satisfies all the properties (expressible in the given logic) that are satisfied by all algebras of the specified ADT, then it cannot be excluded from the ADT.
- Example:
  - Signature $\mathrm{NAT} = (\{nat\}, \{0 :\to nat, s : nat \to nat\})$.
  - NAT-algebra $N = (\{\mathbb{N}\}, \{0_\mathbb{N}, (\lambda x \,.\, x + 1)\})$.
  - $Th_{EL}(\{N\}) = \{0 = 0, s(0) = s(0), s(s(0)) = s(s(0)), \ldots\}$.
  - Every loose specification in EL of an ADT including N also includes e.g. the algebra $A = (\{0, 1\}, 0, \lambda x \,.\, 1 - x))$

Thus e.g. no strictly adequate specification of Peano arithmetic in EL.

Expressive Power of Loose Specifications

- Theorem: If logic L has a sound and complete calculus and if Φ is recursively enumerable, then $\mathcal{M}(sp)$ is axiomatizable in L.
  - Set S is recursively enumerable, if there is an algorithm that lists all of its elements (running forever, if necessary).
  - A class $\mathcal{C}$ of Σ-algebras is axiomatizable in L, if $Th_L(\mathcal{C})$ is recursively enumerable.
- Consequence: A loose specification cannot specify any ADT whose theory is not-recursively enumerable in the given logic.
  - Gödel's second incompleteness theorem: Peano arithmetic is not axiomatizable in first-order predicate logic.

Thus e.g. no strictly adequate specification of Peano arithmetic in PL.

Expressive Power of Loose Specifications

- Theorem: If the given logic is EL, CEL, or PL and $\mathcal{M}(sp)$ contains an algebra with an infinite carrier set, then $M(sp)$ also contains algebras whose corresponding carrier sets contain "junk".
- Consequence: No ADT with an infinite carrier set can be strictly adequately described by a loose specification in EL, CEL, or PL.
  - Cannot rule out "extra" values in addition to the desired ones.

Thus e.g. no strictly adequate specification of stacks in PL.

Generated Algebras

Take signature $\Sigma = (S, \Omega)$, Σ-algebra A.

- Define set of operations $\Omega_c \subseteq \Omega$ (the constructors).
  - Restricted signature $\Sigma_c = (S, \Omega_c)$.
- A is generated by Ω_c:
  - For each sort $s \in S$ and carrier $a \in A(s)$, there exists a ground term $t \in T_{\Sigma_c,s}$ with $a = A(t)$.
    - Carrier a can be described by a term t that involves only constructors.
  - A is generated if it is generated by Ω.
- $Gen(\Sigma, \Omega_c) := \{A \in Alg(\Sigma) \mid A \text{ is generated by } \Omega_c\}$.
  - The set of all Σ-algebras generated by constructors Ω_c.
  - $Gen(\Sigma) := Gen(\Sigma, \Omega)$.

Generated algebra does not contain "junk" in the carrier sets.

Example

Take signature
$\mathrm{NAT} = (\{nat\}, \Omega = \{0 :\to nat, Succ : nat \to nat, + : nat \times nat \to nat\})$.

- Classical NAT-algebra $A = (\mathbb{N}, 0_{\mathbb{N}}, +_{\mathbb{N}})$.
- Constructors $\Omega_c := \{0 :\to nat, Succ : nat \to nat\}$.
- A is generated by Ω_c:
  - For every $n \in \mathbb{N}, n = A(\underbrace{s(s(s(\ldots(s}_{n \text{ times}}(0))))))$.
- A is also generated by Ω.
  - Any superset of a set of generators is also a set of generators.

Usually one looks for the minimal set of generators.

Algebras Generated in Some Sorts

Take signature $\Sigma = (S, \Omega)$, Σ-algebra A.

- Define set of sorts $S_c \subseteq S$ and set of operations $\Omega_c \subseteq \Omega$ (the constructors) with target sorts in S_c.
  - Restricted signature $\Sigma_c = (S, \Omega_c)$.
- A is generated by Ω_c in S_c:
  - For each sort $s \in S_c$ and carrier $a \in A(s)$, there exists
    - a set X of variables in Σ with $X_s = \emptyset$ for every s in S_c,
    - an assignment $\alpha : X \to A$,
    - and a term $t \in T_{\Sigma_c(X),s}$

    with $a = A(\alpha)(t)$.
    - Value a can be described by a term t that involves only constructors in the generated sorts and variables in the non-generated sorts.
  - A is generated in Sc if it is generated in S_c by Ω.

Algebra does not contain "junk" in the carrier sets of the generated sorts.

Example

- Signature $\mathrm{LIST} = (S, \Omega)$:
  - $S = \{el, list\}$.
  - $\Omega = \{[\,] :\to list, Add : el \times list \to list, \_ \cdot \_ : list \times list \to list$.
- LIST-algebra A:
  - $A(el)$ ... a set of "elements".
  - $A(list)$ ... the set of finite lists of elements.
  - $A([\,])$ ... the empty list.
  - $A(Add)$ adds an element at the front of the list.
  - $A(\cdot)$ concatenates two lists.
- A is generated by $\Omega_c = \{[\,], Add\}$ in $S_c = \{list\}$:
  - Take arbitrary $l = [e_1, e_2, \ldots, e_n] \in A(list)$.
  - Define $X_{el} := \{x_1, x_2, \ldots, x_n\}$.
  - Define $\alpha_{el} := [x_1 \mapsto e_1, x_2 \mapsto e_2, \ldots, x_n \mapsto e_n]$.
  - Then $l = A(\alpha)(Add(x_1, Add(x_2, \ldots, Add(x_n, [\,]))))$.

Proofs by Induction

In generated sorts, the principle of structural induction can be applied.

- Take the LIST-algebra A of the previous example.
  - Notation: c_A for $A(c)$.
  - Knowledge: (1) $\forall l \in list_A : [\,]_A \cdot_A l = l$.
    (2) $\forall e \in el_A, l, k \in list_A :$
    $Add_A(e, l) \cdot_A r = Add_A(e, l \cdot_A r)$.
- Prove: $\forall l \in list_A : l \cdot_A [\,]_A = l$.
- Induction base $l = [\,]_A$:
  - $l \cdot_A [\,]_A = [\,]_A \cdot_A [\,]_A \stackrel{(1)}{=} [\,]_A = l$.
- Induction step $l = Add(e, r)$ (for some $e \in el_A, r \in list_A$).
  - Induction Hypothesis (H): $r \cdot_A [\,]_A = r$.
  - $l \cdot_A [\,]_A = Add(e, r) \cdot_A [\,]_A$
    $\stackrel{(2)}{=} Add(e, r \cdot_A [\,]_A)$
    $\stackrel{(H)}{=} Add(e, r) = l$.

Loose Specifications with Constructors

Take logic L.

- Loose specification with constructors $sp = (\Sigma, \Phi, S_c, \Omega_c)$ in L:
  - Signature $\Sigma = (S, \Omega)$, set of formulas $\Phi \subseteq L(\Sigma)$, generated sorts $S_c \subseteq S$, constructors $\Omega_c \subseteq \Omega$ with target sorts in S_c.
- Semantics $\mathcal{M}(sp) = Mod_{\mathcal{U},\Sigma}(\Phi)$ where
  $\mathcal{U} = \{A \in Alg(\Sigma) \mid A \text{ is generated in } S_c \text{ by } \Omega_c\}$.
  - Only generated Σ-algebras are candidates for the specified ADT.

A loose specification with constructors specifies as the ADT the class of all models of its formula set that are generated by the constructors.

Concrete Syntax

loose spec
  sorts [**generated**] *sort* . . .
  opns [**constr**] *operation* . . .
  vars *variable*: *sort* . . .
  vars *formula* . . .
endspec

- Signature $\Sigma = (\{sort, \ldots\}, \{operation, \ldots\})$.
- Set of formulas $\Phi = \{(\forall variable : sort, \ldots\ .\ formula), \ldots\}$.
- Generated sorts $S_c = \{\textbf{generated}\ sort, \ldots\}$.
- Constructors $\Omega_c = \{\textbf{constr}\ operation, \ldots\}$.

We will only use the concrete syntax to define specifications.

Example

loose spec
  sorts *el*
    generated *bool*
    generated *list*
  opns
    constr $True : \to bool$
    constr $False : \to bool$
    constr $[\,] : \to list$
    constr $Add : el \times list \to list$
    $\_ . \_ : list \times list \to bool$
  vars $l, m : list, e, e_1, e_2 : el$
  axioms
    $\neg(True = False)$
    $\neg([\,] = Add(e, l))$
    $Add(e_1, l_1) = Add(e_2, l_2) \Rightarrow e_1 = e_2$
    $[\,].l = l$
    $Add(e, l).m = Add(e, l.m)$
endspec

Strictly adequate specification of the "classical" list algebra in *PL*.

Example

loose spec
  sorts
    generated *bool*
    generated *nat*
  opns
    constr $True : \to bool$
    constr $False : \to bool$
    constr $0 : \to nat$
    constr $Succ : nat \to nat$
    $\_ + \_ : nat \times nat \to nat$
    $\_ * \_ : nat \times nat \to nat$
    $\_ \leq \_ : nat \times nat \to bool$
  vars $m, n : nat$
  axioms
    $\neg(True = False)$
    $\neg(0 = Succ(n))$
    $Succ(n) = Succ(m) \Rightarrow n = m$
    $(0 \leq n) = True$
    $(Succ(n) \leq 0) = False$
    $(Succ(n) \leq Succ(m)) = (n \leq m)$
    $n + 0 = n$
    $n + Succ(m) = Succ(n + m)$
    $n * 0 = n$
    $n * Succ(m) = n + (n * m)$
endspec

Strictly adequate specification of Peano arithmetic in *PL*.

Specified ADT is Strictly Adequate

Proof requires two parts.

- Peano arithmetic satisfies the specified axioms.
  - Can be easily checked.
- Specified ADT is monomorphic: $\forall B, C \in \mathcal{M}(sp) : B \simeq C$.
  - There is an isomorphism $h : B \to C$.
    - A bijective homomorphism.
  - Definition of unique term representation for every carrier.
    - Simplifies the remainder of the proof.
  - Definition of bijective mapping h:
    - By pattern matching on term representation.
  - Proof that h is a homomorphism:
    - By using properties expressed with the help of the term representation.

Term representation essential for this kind of proofs.

Carriers have Unique Term Representations

Take aribitrary $A \in \mathcal{M}(sp)$.

- $bool_A = \{True_A, False_A\}$ and $True_A \neq False_A$.
  - A is generated by $\{True, False\}$ in $bool$.
  - **axiom** $\neg(True = False)$.
- $nat_A = \{Succ^k(0)_A : k \in \mathbb{N}\}$ and $\forall k \neq l : Succ^k(0)_A \neq Succ^l(0)_A$.
  - A is generated by $\{0, Succ\}$ in nat.
  - Proof by induction on k: $\forall l \neq k : Succ^k(0)_A \neq Succ^l(0)_A$.
    - $k = 0, l \neq 0$: $0_A \neq Succ^l(0)_A$ (by **axiom** $\neg(0 = Succ(n))$).
    - $k \neq 0, l \neq k$: assume $Succ^k(0)_A = Succ^l(0)_A$, show $k = l$.

      Know $l \neq 0$ (by **axiom** $\neg(0 = Succ(n))$).
      Thus $k = k' + 1, l = l' + 1$, it suffices to show $k' = l'$.
      By assumption, $Succ(Succ^{k'}(0))_A = Succ(Succ^{l'}(0))_A$.
      Thus $Succ^{k'}(0)_A = Succ^{l'}(0)_A$.
      By induction hypothesis, $k' = l'$.

Carriers are uniquely described by constructor applications.

Definition of Bijective Mapping

Take arbitrary $B, C \in \mathcal{M}(sp)$.

- h is defined by pattern matching on constructor terms:
  - $h_{bool}(True_B) := True_C$.
  - $h_{bool}(False_B) := False_C$.
  - $h_{nat}(Succ^k(0)_B) = Succ^k(0)_C$, for all $k \geq 0$.
- h is consistently defined:
  - $True_B$ and $False_B$ denote different values.
  - $Succ^k(0)_B$ denote different values for different k.
- h is bijective:
  - $True_C$ and $False_C$ denote different values.
  - $Succ^k(0)_C$ denote different values for different k.

One-to-one correspondence between the carrier sets of B and C.

Homomorphism Proof

- Clear for constructors $True, False, 0, Succ$:
  - Definition of h already expresses homomorphism condition.
- Goal: $\forall m, n \in nat_B \ . \ h(op_B(m, n)) = op_C(h(m), h(n))$.

  $op \ldots +, *, \leq$.

  - $\forall k, l \geq 0 \ . \ h(op_B(Succ^k(0)_B, Succ^l(0)_B)) = op_C(h(Succ^k(0)_B), h(Succ^l(0)_B))$.
    - B and C are generated by $\{0, Succ\}$ in nat.
  - $\forall k, l \geq 0 \ . \ h(op_B(Succ^k(0)_B, Succ^l(0)_B)) = op_C(Succ^k(0)_C, Succ^l(0)_C)$.
    - By definition of h.
  - $\forall k, l \geq 0 \ . \ h(op(Succ^k(0), Succ^l(0))_B) = op(Succ^k(0), Succ^l(0))_C$.
    - By definition of term semantics.

Proof goal is expressed with the help of constructor terms.

Homomorphism Proof

The core of the homomorphism proof.

- Goal: $h((Succ^k(0) + Succ^l(0))_B) = (Succ^k(0) + Succ^l(0))_C$.
  - First simplify left and right hand side of the equation.
- Lemma: $\forall A \in \mathcal{M}(sp) : (Succ^k(0) + Succ^l(0))_A = Succ^{k+l}(0)_A$.
  - Induction base $l = 0$: by **axiom** $n + 0 = n$.
  - Induction step $l = l' + 1$:

$$\begin{aligned}&(Succ^k(0) + Succ^{l'+1}(0))_A\\&= Succ(Succ^k(0) + Succ^{l'}(0))_A\\&= Succ(Succ^{k+l'}(0))_A\\&= Succ^{k+l'+1}(0)_A.\end{aligned}$$

- Simplified goal: $h(Succ^{k+l}(0)_B) = Succ^{k+l}(0)_C$.
  - By definition of h.

Similar for the homomorphism proofs of the other operations.

Freely Generated Algebras

Take signature $\Sigma = (S, \Omega)$, Σ-algebra A.

- Define set of operations $\Omega_c \subseteq \Omega$ (the constructors).
  - Restricted signature $\Sigma_c = (S, \Omega_c)$.
- A is freely generated by Ω_c:
  - For each sort $s \in S$ and carrier $a \in A(s)$, there exists exactly one ground term $t \in T_{\Sigma_c,s}$ with $a = A(t)$.
    - Carrier a can be described by a unique term t that involves only constructors.
  - A is freely generated if it is generated by Ω.
- A is freely generated by Ω_c in S_c:
  - Analogous definition as for generated by ... in ... .

Algebras have unique constructor term representations for the carrier sets of the freely generated sorts.

Example

- The "classical" BOOL-algebra $(\{true, false\}, \ldots)$:
  - Freely generated by $\{True, False\}$.
  - Not freely generated by $\{True, False, \neg\}$.
- The "one-element" BOOL-algebra $(\{\#\}, \ldots)$.
  - Freely generated by $\{True\}$ and by $\{False\}$.
  - Not freely generated by $\{True, False\}$.
- The "classical" NAT-algebra $(\mathbb{N}, \ldots)$:
  - Freely generated by $\{0, Succ\}$.
  - Not freely generated by $\{0, Succ, +\}$.
- The "classical" INT-algebra $(\mathbb{Z}, \ldots)$:
  - $\mathrm{INT} = (int, \{0 :\to int, \mathrm{Succ} : int \to int, \mathrm{Pred} : int \to int\})$.
  - Not freely generated by any subset of operations.

A set of free constructors cannot be extended.

Inductive Function Definitions

Freely generated algebras allow inductive function definitions.

- Signature $\mathrm{LIST} = (S, \Omega)$:
  - $S = \{el, list\}$.
  - $\Omega = \{[\,] :\to list, Add : el \times list \to list, \_ . \_ : list \times list \to list\}$.
- Classical LIST-algebra A as in the previous example.
  - A is generated by $\Omega_c = \{[\,], Add\}$ in $S_c = \{list\}$:
- Inductive definition of function $g : A(list) \to \mathbb{N}$.
  - $g([\,]_A) = 0$.
  - $g(Add(x, t)_A) = g(t_A) + 1$ for all $x \in X, t \in T_{\Sigma_c(X), list}$.

Inductive definition by "pattern matching" on constructor terms (independent of the nature of the carrier set).

Loose Specifications with Free Constructors

Take logic L.

- Loose specification with free constructors $sp = (\Sigma, \Phi, S_c, \Omega_c)$ in L:
  - Signature $\Sigma = (S, \Omega)$, set of formulas $\Phi \subseteq L(\Sigma)$, freely generated sorts $S_c \subseteq S$, constructors $\Omega_c \subseteq \Omega$ with target sorts in S_c.
- Semantics $\mathcal{M}(sp) = Mod_{\mathcal{U},\Sigma}(\Phi)$ where
  $\mathcal{U} = \{A \in Alg(\Sigma) \mid A \text{ is freely generated in } S_c \text{ by } \Omega_c\}$.
  - Only freely generated Σ-algebras are candidates for the specified ADT.

A loose specification with free constructors specifies the class of all models of its formula set that are freely generated by the constructors.

Concrete Syntax

loose spec
  sorts [**freely generated**] *sort* ...
  opns [**constr**] *operation* ...
  vars *variable*: *sort* ...
  vars *formula* ...
endspec

- Signature $\Sigma = (\{sort, \ldots\}, \{operation, \ldots\})$.
- Set of formulas $\Phi = \{(\forall variable : sort, \ldots \ . \ formula), \ldots\}$.
- Generated sorts $S_c = \{$**freely generated** $sort, \ldots\}$.
- Constructors $\Omega_c = \{$**constr** $operation, \ldots\}$.

Also mixing of generated sorts with freely generated sorts possible.

Example

loose spec
  sorts *el*
    freely generated *bool*
    freely generated *list*
  opns
    constr $True :\to bool$
    constr $False :\to bool$
    constr $[\,] :\to list$
    constr $Add : el \times list \to list$
    $\_ . \_ : list \times list \to bool$
  vars $l, m : list, e, e_1, e_2 : el$
  axioms
    $[\,].l = l$
    $Add(e, l).m = Add(e, l.m)$
endspec

Strictly adequate specification of the "classical" list algebra in *EL*; the non-constructor operation is inductively defined.

Example

loose spec
  sorts
    freely generated *bool*
    freely generated *nat*
  opns
    constr *True* $:\to bool$
    constr *False* $:\to bool$
    constr $0 :\to nat$
    constr *Succ* $: nat \to nat$
    $\_ + \_ : nat \times nat \to nat$
    $\_ * \_ : nat \times nat \to nat$
    $\_ \leq \_ : nat \times nat \to bool$
  vars $m, n : nat$
  axioms
    $(0 \leq n) = True$
    $(Succ(n) \leq 0) = False$
    $(Succ(n) \leq Succ(m)) = (n \leq m)$
    $n + 0 = n$
    $n + Succ(m) = Succ(n + m)$
    $n * 0 = n$
    $n * Succ(m) = n + (n * m)$
endspec

Strictly adequate specification of the "classical" list algebra in *EL*; the non-constructor operations are inductively defined.

Summary

A couple of core messages...

- A loose specification describes a class of models as an ADT.
  - To check whether a given algebra implements the specification (i.e., whether it is an element of the specified ADT):
    - Check whether the algebra satisfies the specification axioms.
  - Carrier sets may collapse to singletons (or be too "small").
    - In *PL*, additional axioms can prevent this.
    - Non-equalities of operation results (injectiveness of operations).
  - Carrier sets may contain junk.
    - In *PL*, an additional axiom can prevent this for a finite carrier set.
    - Axiom enumerates constants that denote all carriers of the sort.

Without constructors, loose specifications are generally clumsy because many "boring" axioms are needed.

Summary (Contd)

- Loose specifications with constructors.
  - Every carrier is denoted by some constructor term.
  - Thus junk is removed from (also infinite) carrier sets.
  - Induction proofs on term representation of carriers become possible.
  - Problem: not all carrier sets have term representations.
    - ADT "real" (carrier set is not countable).
- Loose specifications with free constructors.
  - Every carrier is denoted by exactly one constructor term.
  - Thus the collapse of carrier sets is prevented.
  - Inductive function definitions by pattern matching on term representations of carriers become possible.
  - Problem: not all carrier sets have unique term representations.
    - ADT "set" (no unique representation at all).
    - ADT "integer" (unique representation is unconvient).

With constructors, loose specifications become easy to use.

Summary (Contd)

So what is the role of loose specifications...

- Loose specifications are good for specifying requirements.
  - May specify zero, one, many datatypes (polymorphic ADTs).
  - Thus allow arbitrarily many implementations.
    - A loose specification may not have any model (implementation) at all!
  - Specification axioms can (should) be abstract.
    - Later verification that concrete implementation satisfies the axioms.
- Loose specifications are not good for specifying designs.
  - Not descriptions of concrete algorithms/implementations.
- Loose specifications are generally not executable.
  - No engines to execute loose specifications for rapid prototyping.

Loose specifications are for *reasoning*, not for *executing*; they are the basis of program specification languages such as Larch/C++.