Some basic notions concerning the mCRL2 data library

Jeroen Keiren

20th March 2023

1 mCRL2

The language mCRL2 consists of data and processes. The data part contains an equational specification. One can define sorts, functions working upon these sorts, and describe the meaning of these functions by means of equational axioms. The process part contains processes described in the style of CCS, CSP or ACP, with the particular process syntax taken from ACP. It basically consists of a set of uninterpreted actions that may be parametrized with data and time.

2 The syntax of mCRL2

See [**?**, Appendix B].

3 Data specification

A data specification consists of a number of *sort*s, a number of *constructor*s for each sort, a number of *map*pings, and a set of *equation*s. A data specification is an equational specification, in which sorts denote types. The semantics of a sort is a set. The elements of the semantics of a sort are described by its constructors, whereas the mappings are functions defined on the semantics of sorts. The equations (axioms) describe equational properties of functions and elements of the semantics. Note that every element of the semantics of a sort can be constructed from its constructors, this is also known as "no junk". It may however be the case that an element can be described by several constructors, hence this construction does not satisfy the "no confusion" property. The only exception to this are the booleans; *true* and *false* are distinct elements.

3.1 Syntax

We define the syntax that is used to describe data in mCRL2.

Definition 3.1 (Sort expressions) We assume a set of basic sorts S_{Basic}. Sort expressions S are defined as follows, where $\mathbb{B} \in S_{Basic}$, and $\to$ is right-associative:

$$S ::= S_{Basic} \mid S_{Container} \mid S \times \cdots \times S \to S \mid S_{Struct}$$

with $S_{Container}$ being defined as:

$$S_{Container} ::= List(S) \mid Set(S) \mid Bag(S)$$

The syntax of structured sorts S_{Struct} is defined as follows (with p a string):

$$S_{Struct} ::= p(proj*)?p$$

in which $proj$ has the following syntax:

$$proj ::= S \mid p{:}S$$

Structured sorts, with $n \in \mathbb{N}^+$, $k_i \in \mathbb{N}$ with $1 \leq i \leq n$, in general have the following form:

$$\begin{aligned} \textbf{struct} \quad & c_1(pr_{1,1} : S_{1,1}, \ldots, pr_{1,k_1} : S_{1,k_1})?isc_1 \\ \mid \quad & c_2(pr_{2,1} : S_{2,1}, \ldots, pr_{2,k_2} : S_{2,k_2})?isc_2 \\ \mid \quad & \vdots \\ \mid \quad & c_n(pr_{n,1} : S_{n,1}, \ldots, pr_{n,k_n} : S_{n,k_n})?isc_n; \end{aligned}$$

We refer to c_i as the constructors of the structured sort. $S_{i,j}$ are the sorts of the arguments of the constructors. $pr_{i,j}$ are names for optional projection functions, retrieving the corresponding argument for a constructor. isc_i are the names of optional recognizer functions, returning a boolean value.

We call the set $S_{Container}$ the container sorts and $S \backslash \{S_{Basic} \cup S_{Container} \cup S_{Struct}\}$ the set of function sorts. In $S_0 \times \cdots \times S_n \to S$ we refer to $S_0, \ldots, S_n$ as the domain, and to S as the codomain of the sort.

The language also supports sort aliases like $S_0 = S_1$. In this case only one of the two is treated as a sort, and the other is a reference to it.

Example 3.2 (Sort aliases) Consider a specification which has $LNat = List(Nat)$. Now data expressions x of sort $LNat$ and x of sort $List(Nat)$ are equivalent.

Definition 3.3 (Variables) We assume a set V of variable names with their associated sorts.

We write V_s to refer to variables of sort s.

Definition 3.4 (Operations) The set of operations Ω consists of a set of constructors Ω_C and a set of mappings Ω_M, i.e.

$$\Omega = \Omega_C \cup \Omega_M$$

All elements in Ω can be described, with n a function symbol, and S a sort, as follows

$$\Omega ::= n{:}S$$

Ω_C only contains expressions of the following form.

$$\Omega_C ::= n{:}S_B \mid n{:}S \times \cdots \times S \to S_B$$

This means that there are only constructors for basic sorts.

In the remainder we write $\Omega_{C,s}$ to denote the constructors of sort s. That is those n in Ω_C with codomain s.

Definition 3.5 (Data expressions) We inductively define data expressions e, with sort expressions S and variables x as follows:

$$\begin{array}{rcl} e & ::= & x \mid n \mid e(e,\ldots,e) \mid \lambda x{:}S,\ldots,x{:}S.e \mid \forall x{:}S,\ldots,x{:}S.e \mid \exists x{:}S,\ldots,x{:}S.e \\ & \mid & e \ \mathbf{whr}\ x = e,\ldots,x = e \ \mathbf{end} \mid \{x{:}S \mid e\} \end{array}$$

Here $e(e,\ldots,e)$ denotes application of data expressions, $\lambda x{:}S,\ldots,x{:}:S.e$ denotes abstraction. $\forall x{:}S,\ldots,x{:}S.e$ and $\exists x{:}S,\ldots,x{:}S.e$ describe universal and existential quantification. $\{x{:}S \mid e\}$ denotes set or bag comprehension. Note that in the remiander we will write Λ to denote any binding operator when describing rules that apply to all binding operators, i.e. λ, $\exists$, $\forall$, $\{\}$.

Convention 3.6 (System defined operators) We write system defined operators as infix operators, hence we write $b_1 \wedge b_2$ for $and(b_1, b_2)$, etc. For system defined operators we also have a notion of operator precedence. See Section 4 for a complete overview of system defined sorts.

Definition 3.7 (Signature) A signature Σ is a structure (S_{Basic}, Ω), where S_{Basic} is a set of sorts and Ω is a set of operations. S_{Basic} contains at least $\mathbb{B}, \mathbb{N}^+, \mathbb{N}, \mathbb{Z}, \mathbb{R}$.

We now define the validity of data expressions with a number of syntax-directed derivation rules.

Definition 3.8 (Valid data expressions) We assume a context Γ, which is a set of typing statements of variables and operations used in the typing derivations. Note that we write $\Gamma, x{:}s$ as shorthand for $\Gamma \cup \{x{:}s\}$ and $\exists^1_s$ to denote that there is *exactly one* such s.

$$\frac{x{:}s \in \Gamma}{\Gamma \vdash x{:}s}\,(Var) \qquad \frac{n{:}s \in \Gamma}{\Gamma \vdash n{:}s}\,(Op)$$

$$\frac{\Gamma, x_0{:}s_0,\ldots,x_n{:}s_n \vdash e{:}s}{\Gamma \vdash (\lambda x_0{:}s_0,\ldots,x_n{:}s_n.e){:}s_0 \times \cdots \times s_n \to s}\,(Abs)$$

$$\frac{\exists^1_{s0,\ldots,s_n}(\Gamma \vdash t{:}s_0 \times \cdots \times s_n \to s \quad \Gamma \vdash t_0{:}s_0 \quad \ldots \quad \Gamma \vdash t_n{:}s_n)}{\Gamma \vdash t(t_0,\ldots,t_n){:}s}\,(Appl)$$

$$\frac{\exists^1_{s_0,\ldots,s_n}(\Gamma \vdash x_1{:}s_1 \quad \Gamma \vdash e_1{:}s_1 \quad \ldots \quad \Gamma \vdash x_n{:}s_n \quad \Gamma \vdash e_n{:}s_n \quad \Gamma, x_0{:}s_0,\ldots,x_n{:}s_n \vdash e{:}s)}{\Gamma \vdash (e \ \mathbf{whr}\ x_0 = e_0,\ldots,x_n = e_n \ \mathbf{end}){:}s}\,(Where)$$

$$\frac{\Gamma, x_0{:}s_0,\ldots,x_n{:}s_n \vdash e{:}\mathbb{B}}{\Gamma \vdash (\forall x_0{:}s_0,\ldots,x_n{:}s_n.e){:}\mathbb{B}}\,(Forall) \qquad \frac{\Gamma, x_0{:}s_0,\ldots,x_n{:}s_n \vdash e{:}\mathbb{B}}{\Gamma \vdash (\exists x_0{:}s_0,\ldots,x_n{:}s_n.e){:}\mathbb{B}}\,(Exists)$$

$$\frac{\Gamma, x{:}s \vdash e{:}\mathbb{B}}{\Gamma \vdash \{x{:}s \mid e\}{:}Set(s)}\,(SetComp) \qquad \frac{\Gamma, x{:}s \vdash e{:}\mathbb{N}}{\Gamma \vdash \{x{:}s \mid e\}{:}Bag(s)}\,(BagComp)$$

The Σ-algebra is characterized by means of equational logic. We describe the syntax of formulae in the equational logic.

Definition 3.9 (Equations) Consider data expressions e, the syntactic set of equations adheres to

$$E ::= e = e \mid e \to e = e$$

We also introduce two rules for validity of equations.

$$\frac{\exists^1_s(\Gamma \vdash d{:}s \quad \Gamma \vdash e{:}s)}{\Gamma \vdash d = e}\,(Eq) \qquad \frac{\Gamma \vdash c : \mathbb{B} \quad \exists^1_s(\Gamma \vdash d{:}s \quad \Gamma \vdash e{:}s)}{\Gamma \vdash c \to d = e}\,(CondEq)$$

Definition 3.10 (Data specification) A data specification consists of sorts, constructors, mappings and equations, i.e.

$$D = (S, \Omega, E)$$

3.2 Semantics

Definition 3.11 (Σ-Algebra) A Σ-algebra A assigns meaning to signature $\Sigma = (S_{Basic}, \Omega)$ by assigning a carrier set $A(s)$ to each sort s. $A(s)$ is the set containing the elements of sort s. Furthermore it assigns a total function $A(n{:}s)$, yielding an element of sort $A(s)$, to each operation $n{:}s \in \Omega$. We write $A(n)$ for $A(n{:}s)$ when s is clear from the context.

Note that all elements from carrier set $A(s)$ can be obtained from the constructors $n{:}s \in \Omega_C$.

Example 3.12 Suppose we have a signature $\Sigma = (S_{Basic}, \Omega)$, with

$$\begin{aligned} S_{Basic} &= \{Nat\} \\ \Omega_C &= \{zero{:}Nat, succ{:}Nat \to Nat\} \\ \Omega_M &= \{add{:}Nat \times Nat \to Nat\} \end{aligned}$$

We give semantics to this by Σ-algebra A, where:

$$\begin{aligned} A(Nat) &= \mathbb{N} \\ A(Nat \to Nat) &= \mathbb{N} \to \mathbb{N} \\ A(Nat \times Nat) \to Nat &= \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ A(zero) &= 0 \\ A(succ) &= \lambda n{:}\mathbb{N}.(n+1) \\ A(add) &= \lambda m, n{:}\mathbb{N} \times \mathbb{N}.(m+n) \end{aligned}$$

All elements of a carrier set $A(s)$ of sort s can be obtained by inductively applying the constructors $n : s \in \Omega_C$ of s. This process is referred to as *enumeration*. Note that this requires all elements of other sorts $s' \in S$ that occur as subexpression in s.

Example 3.13 As an example of obtaining all elements of a sort from its constructors, let us again look at the natural numbers Nat. We take a data expression representing a natural number n, which according to the constructors of Nat could be either 0 or $succ(n')$—the successor of another natural number. We repeat this process for n', and continue this way until all variables have been eliminated. A tree representation of this is given in Figure 1. Note that this process does not necessarily terminate.

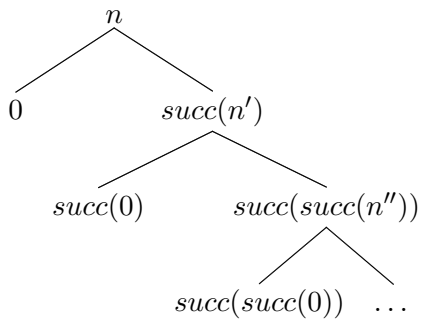

Figure 1: Enumeration of natural numbers using 0 and *succ*

3.2.1 Variables and data expressions

In order to define the semantics of data expressions, we introduce the notion of assignment. Using V_s to denote the variables of sort s from V, an assignment of a set of variables V for a Σ-algebra A ($\alpha : V \to A$) is a family $\alpha = (\alpha_s)_{s \in S}$ of total functions $\alpha_s : V_s \to A(s)$.

A meaning is given to a data expression e over a signature Σ extended with variables V—in short $e \in T_{\Sigma(V)}$—using a Σ-algebra A and an assignment α. It is called the value of e for α and is denoted by $A(\alpha)(e)$. In the sequel we also write $e \in T_{\Sigma(V),s}$ to denote the data expressions over signature Σ and variables V of sort s. With $\Lambda\!\!\!\Lambda$ we denote abstraction in the semantic domain.

Definition 3.14 We define the notion of value inductively on the structure of e:

- $A(\alpha)(x) = \alpha_s(x)$, where $x \in V_s$ and $s \in S$;
- $A(\alpha)(n) = A(n)$;
- $A(\alpha)(e(u_0, \ldots, u_n)) = A(\alpha)(e)(A(\alpha)(u_0), \ldots, A(\alpha)(u_n))$ where $e \in T_{\Sigma(V),s_0 \times \cdots \times s_n \to s}$ and $u_i \in T_{\Sigma(V),s_i}$
- $A(\alpha)(\Lambda x_0{:}s_0, \ldots, x_n{:}s_n.e) = \Lambda\!\!\!\Lambda d_0 \in A(s_0), \ldots d_n \in A(s_n).A(\alpha[x_i := d_i]_{0 \le i \le n})(e)$
- $A(\alpha)(e$ **whr** $x_0 = e_0, \ldots, x_n = e_n$ **end** $= A(\alpha[x_i := d_i]_{0 \le i \le n})(e)$ **where** $d_0 \in A(s_0) = A(\alpha)(e_0), \ldots d_n \in A(s_n) = A(\alpha)(e_n)$ **end**

We introduce the syntactic notion of substitution. For a signature $\Sigma = (S, \Omega)$ and associated sets of variables V and W, a substitution is a family $\sigma = (\sigma_s)_{s \in S}$ of functions $\sigma_s : V_s \to T_{\Sigma(W),s}$. We denote this by $\sigma : V \to T_{\Sigma(W)}$.

Definition 3.15 (Substitution) We define the application of a substitution σ to a data expression inductively as follows:

- $\sigma(x) = \sigma_s(x)$, where $x \in V_s$ and $s \in S$
- $\sigma(n) = n$, where $n : S \in \Omega$
- $\sigma(e(u_0, \ldots, u_n)) = \sigma(e)(\sigma(u_0), \ldots, \sigma(u_n))$

- $\sigma(\Lambda x_0, \ldots, x_n.e) = \Lambda y_0, \ldots, y_n.\sigma[x_i := y_i]_{0 \leq i \leq n}(e)$, if y_i does not occur free in e and $\sigma(y_i) = y_i$ (for all $0 \leq i \leq n$)
- $\sigma(e \textbf{ whr } x_0 = e_0, \ldots, x_n = e_n \textbf{ end}) = \sigma[x_i := y_i]_{0 \leq i \leq n}(e) \textbf{ whr } y_0 = \sigma(e_0), \ldots, y_n = \sigma(e_n)$, if y_i does not occur free in e and $\sigma(y_i) = y_i$ (for all $0 \leq i \leq n$)

Remark 3.16 There is a close relation between the syntactic notion of substitution and the semantic notion of assignment. Assignments may simulate substitutions. For all signatures Σ, associated sets of variables V, W, substitutions $\sigma : V \to T_{\Sigma(W)}$, Σ-algebras A, assignments $\beta : W \to A$, and data expressions $e \in T_{\Sigma(V)}$, we have:

$$A(\beta)(\sigma(e)) = A(\alpha)(e)$$

where assignment $\alpha : V \to A$ is defined by $\alpha(x) = A(\beta)(\sigma(x))$, for all $x \in V$.

3.2.2 Equational logic

The semantics of a formula in equational logic is expressed by a satisfaction relation.

Definition 3.17 (Satisfaction relation) For a Σ-algebra A, a condition $c \in T_{\Sigma(V),\mathbb{B}}$, a sort $s \in S$ and data expressions $d, e \in T_{\Sigma(V),s}$, the satisfaction relation $\models_{EL}$ is defined by:

$A \models_{EL} c \to d = e$ iff $A(\alpha)(d) = A(\alpha)(e)$ and $(A(\alpha)(c) = true)$, for all assignments $\alpha : V \to A$

If $A \models_{EL} c \to d = e$, we say that $d = e$ is valid in A. If c is omitted, we consider it to be $true$.

Definition 3.18 (Model) For a set of equations $E \subseteq EL(\Sigma)$, a Σ-algebra A is called a model of E if $A \models_{EL} eq$, for all $eq \in E$.

We denote the class of all models of E as $Mod_{EL}(E)$.

Definition 3.19 (Logical consequence) An equation $eq \in EL(\Sigma)$ is called a logical consequence of a set of equations E, iff eq is valid in all models of E, denoted $E \models_{EL} eq$. That is

$$E \models_{EL} eq \text{ iff } A \models_{EL} eq, \text{ for all } A \in Mod_{EL}(E)$$

3.2.3 Data specification

In mCRL2 the set of basic sorts S_{Basic} is obtained by those sorts preceded with the **sort** keyword. Function sorts may be used implicitly in the specification. The constructor functions Ω_C are those functions preceded be the **cons** keyword. The mappings Ω_M are preceded by **map**. Finally, the equations $EL(\Sigma)$ are preceded by **eqn**.

3.3 Some notions on sorts

3.3.1 Finiteness

For some applications it is interesting to know whether sort is finite.

Let function $DependentSorts : \Omega_C \to 2^S$, that obtains the sorts on which a constructor depends, be defined inductively as:

$$DependentSorts(n{:}s) = \begin{cases} \emptyset & \text{if } s \in S_{Basic} \\ \bigcup_{0 \le i \le n}(\{s_i\} \cup Sorts(s_i)) & \text{if } s = s_0 \times \cdots \times s_n \to s' \end{cases}$$

Let the function $Sorts : S \to 2^S$, that obtains the sorts on which its argument depends, be defined recursively as:

$$Sorts(s) = \begin{cases} \bigcup_{n \in \Omega_{C,s}} DependentSorts(n) & \text{if } s \in S_{Basic} \\ Sorts(s') & \text{if } s \in S_{Container} \\ \bigcup_{0 \le i \le n} Sorts(s_i) \cup \{s'\} & \text{if } s = s_0 \times \cdots \times s_n \to s' \\ \bigcup_{0 \le i \le n} \bigcup_{0 \le j \le m_i} Sorts(s_{i,j}) & \text{if } s = \textbf{struct } c_i(pr_{i,1}{:}s_{i,1}, \ldots, pr_{i,m_i}{:}s_{i,m_i})?isc_i \\ & \quad \text{for } 0 \le i \le n, 1 \le j \le m_i \end{cases}$$

We inductively define predicate $Finite : S \to \mathbb{B}$ as follows:

$$Finite(s) = \begin{cases} \Omega_{C,s} \neq \emptyset \wedge s \notin Sorts(s) & \\ \quad \wedge (\forall n \in \Omega_{C,s} : & \\ \quad (\forall s' \in DependentSorts(n) : Finite(s'))) & \text{if } s \in S_{Basic} \\ Finite(s') & \text{if } s = Set(s') \\ false & \text{if } s \in S_{Container} \text{ and } s \neq Set(s') \text{ for all } s' \\ (\forall i : Finite(s_i)) \wedge Finite(s') & \text{if } s = s_0 \times \cdots \times s_n \to s' \\ s \notin Sorts(s) \wedge (\forall s' \in Sorts(s) : Finite(s'))) & \text{if } s = \textbf{struct } c_i(pr_{i,1}{:}s_{i,1}, \ldots, pr_{i,m_i}{:}s_{i,m_i})?isc_i \\ & \quad \text{for } 0 \le i \le n, 1 \le j \le m_i \end{cases}$$

3.3.2 Equivalence of sorts

For this topic see the note titled "An algorithm to find a representant for sorts in the context of sort aliases and recursive sorts".

3.4 Some notions on data expressions

Definition 3.20 (Free variables) We inductively define the set of free variables $FV(e)$ of a data expression e as follows:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(n) &= \emptyset \\ FV(e(e_0, \ldots, e_n)) &= FV(e) \cup \bigcup_{0 \le i \le n} FV(e_i) \\ FV(\lambda x_0{:}s_0, \ldots, x_n{:}s_n.e) &= FV(e) \setminus \{x_i \mid 0 \le i \le n\} \end{aligned}$$

Definition 3.21 (Closed) We say that a data expression e is closed iff $FV(e) = \emptyset$.

4 Predefined sort specifications

mCRL2 provides a number of predefined sorts. For an overview of these sorts, with the full specifications that are generated see [**?**, Appendix A].

5 Type checking

In mCRL2 types are only made explicit in function declarations and abstractions. For places where type information is not explicitly included these can be inferred by a type-checking algorithm if the specification is typable. This algorithm attempts to infer the types of expressions in the program from their contexts. Type checking is outside the scope of this document.

5.1 Type conversion rules

The toolset supports a number of convenience functions `A2B` in order to cast numeric types. In this $A \neq B$, and $A, B \in \{Pos, Nat, Int, Real\}$. Of these, the type checker only uses upcasts, e.g. `Pos2Nat`.

6 Rewriting

The rewriters used in mCRL2 [**?**] take the equations as defined in the data specification and interpret these from left to right as rewrite rules. A rewrite rule means that an expression matching the left hand side of this rule may be rewritten to the right hand side of the rule. Additionally, a rewrite rule may be equipped with a condition, such that an expression is only rewritten if the condition holds. Such rewrite rules are referred to as conditional rewrite rules.

Rewriting of a data expression is done by repeatedly applying rewrite rules until a normal form is reached, that is, until no rewrite rules can be applied to the expression. Note that the mCRL2 toolset supports multiple rewrite strategies, which do not have the same termination conditions. However, whenever the rewrite system derived from the data specification is terminating, the rewriters will also terminate.

Rewriting is a purely syntactic manipulation. Given a set of rewrite rules a normal form for a data expression is returned. If two expressions have the same normal form, they are considered equal.

In rewriting we can parametrize a rewriter with a set of substitutions which are performed during rewriting. This is mostly for performance reasons.

We consider a rewrite system R.

Definition 6.1 (Rewrite step) There is a rewrite step $t \to_R u$ if there is a position π (of a subterm of t), a rewrite rule $l \to r$ in R and a substitution σ, such that $t|\pi = \sigma(l)$ and $u = t[\pi](\sigma(r))$, with $t|\pi$ the subterm of t at position π, and $t[\pi]u$ is t with its subterm at position π replaced by u.

Definition 6.2 (Normal form) At term t is in normal form when there is no rewrite step $t \to_R u$.

Definition 6.3 (Rewrite sequence) A rewrite sequence $t \twoheadrightarrow_R u$ is a sequence of rewrite steps, such that $t \twoheadrightarrow_R v$ and $v \rightarrow_R u$.

Definition 6.4 (Normal forms of a term) The normal forms of a term t are those terms that occur as the last term in a rewrite sequence starting with t.

7 Proving

Proving is, like rewriting, a syntactic operation. However, where in rewriting the equations are only interpreted left to right, the provers use the equations to their full extent. The provers determine whether or not an expression of sort Bool is a tautology or a contradiction. An in depth discussion of a prover for mCRL2 is presented by Luc Engelen [**?**].

8 Equality checking

In practice a lot of equality checking is performed, for example for evaluating conditions in rewriters. In its basic form, the question is merely whether two terms are equal. In addition, like in rewriting, substitutions may be provided to check whether to terms are equal under a certain substitution. This is again for performance reasons. Checking for equality can be performed by a rewriter as well as by a prover.

The requirements for an equality checker Eq are as follows:

$$\begin{aligned} Eq(\mathit{true}, \mathit{false}) &\equiv \mathit{false} \\ Eq(e, e') &\implies e = e' \end{aligned}$$

Thus, *true* and *false* are different, and if two terms are said to be equal, they are indeed equal.

9 Enumeration

Let $\mathit{range}(e)$ be the set of all possible values that data expression e can attain. A sort S is *enumerable* iff a function enum_S exists that maps an arbitrary data expression e of sort S to a finite set of closed data expressions $\{e_1, \ldots, e_k\}$, such that $\mathit{range}(e) = \bigcup_{i=1}^{k} \mathit{range}(e_i)$. By repeatedly applying enum_S to non-constant data expressions x_i, a tree expansion of the data expression is obtained. The leafs of this tree form a finite representation of the data expression. For data expressions of finite sort, this tree is always finite as well.

10 Mapping an data specification to the signature

In this section we map the current implementation of a data specification to the signature as described in this document.

10.1 Details of the implementation up to revision 5967

We now map parts of the current (procedural style) implementation in the mCRL2 toolset to the signature as described above. This is the implementation directly built upon the ATerm library as it was available up to revision 5967 of the mCRL2 toolset.

10.1.1 Sort expressions

Sort expressions are the *sort expressions* S. Internally referred to as `SortExpr`. Note that Sort identifiers, Arrow sorts, Sort references, Structured sorts and Container sorts are all sort expressions.

Sort identifiers are the basic sorts S_B. Internally referred to as `SortId`.

Arrow sorts are the function sorts $S \backslash \{S_{Basic} \cup S_{Container}\}$. Internally referred to as `SortArrow`.

Sort references are the aliases $S_0 = S_1$. Internally referred to as `SortRef`.

Container sorts are the $List(S)$, $Set(S)$ and $Bag(S)$ sorts. Internally these are referred to as `SortCons(Constructor type, SortExpr)`, in which `SortExpr` denotes the element sort, and `Constructor type` is one of `SortList, SortSet, SortBag` to denote List, Set and Bag sorts respectively.

Structured sorts are the $Struct(S)$ sorts. Internally these are referred to as `SortStruct(StructCons+)`, in which `StructCons+` is a non-empty list of constructors. A constructor has the form `StructCons(Name, StructProj*, Recogniser?)`. In this, `Name` represents the name of the constructor (a string), `Recogniser?` represents the name of the recogniser function for this constructor, which may be *nil* to denote that there is no recogniser. `StructProj*` is a (possibly empty) list of expressions that denotes the arguments of the constructor, with their optional projection functions. An expression for an argument has the following form: `StructProj(Name?, SortExpr)`, in this `Name?` denotes the name of the argument, which is either a string or *nil* denoting that the argument has no name. `SortExpr` denotes the sort of the argument.

10.1.2 Data expressions

Data expressions are the data expressions. Internally referred to as `DataExpr`.

Data variable is a variable of a certain sort. Internally referred to as `DataVarId`.

Data operation is a function with its sort, an element of Ω. Internally referred to as `OpId`.

Data application is an expression applied to a (number of) argument(s). Internally referred to as `DataAppl`. Note that the sorts must match!

Binder is an expression that denotes abstraction, i.e. either a lambda abstraction, or a universal or existential quantification. This is denoted internally as `Binder(BindingOperator, DataVarId+, DataExpr)`, in which `BindingOperator` is one of `Forall`, `Exists`, `Lambda`, `SetBagComp`, `SetComp`, `BagComp`, `DataVarId+` is a non-empty list of variables over which is abstracted in `DataExpr`.

10.2 Some notes on the data implementation up to revision 5967

The data implementation in the mCRL2 toolset had two—relatively closely related—tasks. First of all it completed the equational specification for standard data types (Bool, Pos, Nat, Int, Real). Furthermore it translated complex constructs (List, Set, Bag, structured sort) that are present in the mCRL2 format *after typechecking* (before data implementation) away

to a simpler construct, and generated the corresponding equational specification, to facilitate efficient rewriting.

10.2.1 Transformations for standard data types

In the data implementation phase, the following transformations were made:

- Rules for Bool, Pos, Nat, Int and Real were added
- Standard functions $\approx, \not\approx, \mathit{if}$ were added
- Numerical pattern matching was implemented, this made sure that explicit casts (e.g. Pos2Nat) were translated away. This way `Pos2Nat(p:Pos)` may be rewritten to `p:Nat`.

Observe that the rewrite rules that were introduced for the standard data types are closely tailored to the setting in which rewriters are used. With this we mean that the equational specification is constructed in such a way that efficient rewriting is possible. Therefore generation of equational specifications for system defined sorts is closely linked to the rewriter that is used.

10.2.2 Data implementation of complex constructs

Complex constructs were implemented to a more simple form in order to facilitate a simpler rewriting mechanism. Data implementation included the implementation of:

- Sort references
- Structured sorts
- List, Set and Bag sorts
- Binders
- Numeric constants

10.2.3 The need for a data implementation

As we have seen in the previous sections, it can be argued that the data implementation is an implementation detail, tailored to efficient rewriting, and should be treated as such.

As a result of this, the data library gets an interface conforming to the current internal format *before data implementation*, i.e. in the data library we should support high-level concepts such as structured sorts, lists, sets, bags, binders and numeric constants. This might typically also lead to an interface which could more easily incorporate support for provers.

I would therefore propose such a high-level interface for the data library, but in order to support facilities like rewriting and theorem proving, it should be able to make a systematic translation between the data format and a format supported by e.g. a rewriter of a prover, this way treating low-level details really as implementation issue for the used backend.

This results in a format in which the most fundamental transformation work, namely that concerning system defined sorts, is always done immediately after type checking, i.e. after typechecking numbers, lists, sets and bags are translated to the format as described in

Appendix A of [**?**]. However, as we also want to work on a level that remains close to the level of theory, concepts such as binding and existential and universal quantification should remain available at an interface level in the implementation. For more details on the specific format that is used in the new data library see Section 11.7.

11 Designing a new data library

In the basic design for the new implementation of a data library we want to stick as close to the theory as possible. Furthermore we want to integrate enumeration, rewriting and in the end proving with this library.

11.1 Goals of the library

The goals of the data library is providing an interface that matches closely with the mCRL2 data types. In the current implementation, translations are made to a small core of the language (the so called data implementation). Furthermore translations are made towards a rewriter-specific format. Translations like these have to be prevented in the library we are designing, as these imply performance penalties. In an ideal situation no such translations remain.

Currently all system defined constructors/mappings/equations are added to the data specification. They are also included in the format which is saved on disk. This causes incompatibilities between files generated by different version of the toolset. Therefore a stable storage format is desired.

An important concern in the implementation is minimizing runtime memory usage. Performance is of lesser importance then memory use.

11.2 Sort expressions

First of all we provide a design for the library part concerned with sort expressions. Note that utility functions may be added when the need arises.

11.2.1 Classes

We provide an overview of the classes with a short description, we also describe their public methods:

sort_expression	
description	This denotes any sort expression that can be constructed from the theory, hence any sort occurring in the signature of a specification.
subclass of	

superclass of	• basic_sort • structured_sort • container_sort • function_sort
public methods	**is_basic_sort** Returns *true* iff this expression is a basic sort. **is_structured_sort** Returns *true* iff this expression is a structured sort. **is_container_sort** Returns *true* iff this expression is a container sort. **is_function_sort** Returns *true* iff this expression is a function sort.

basic_sort

description	This represents a basic sort, corresponding directly to the theory.
subclass of	sort_expression
superclass of	
public methods	**name** Returns the name of the basic sort.

function_sort

description	This denotes a function sort $S_0 \times ... \times S_n \to S$, where $S_0, \ldots, S_n$ is the domain, and S is the codomain.
subclass of	sort_expression
superclass of	
public methods	**domain** Returns the domain of the function sort. The domain of $S_0 \times \cdots \times S_n \to S$ is $S_0, \ldots, S_n$. **codomain** Returns the codomain of the function sort. The codomain of $S_0 \times \cdots \times S_n \to S$ is S.

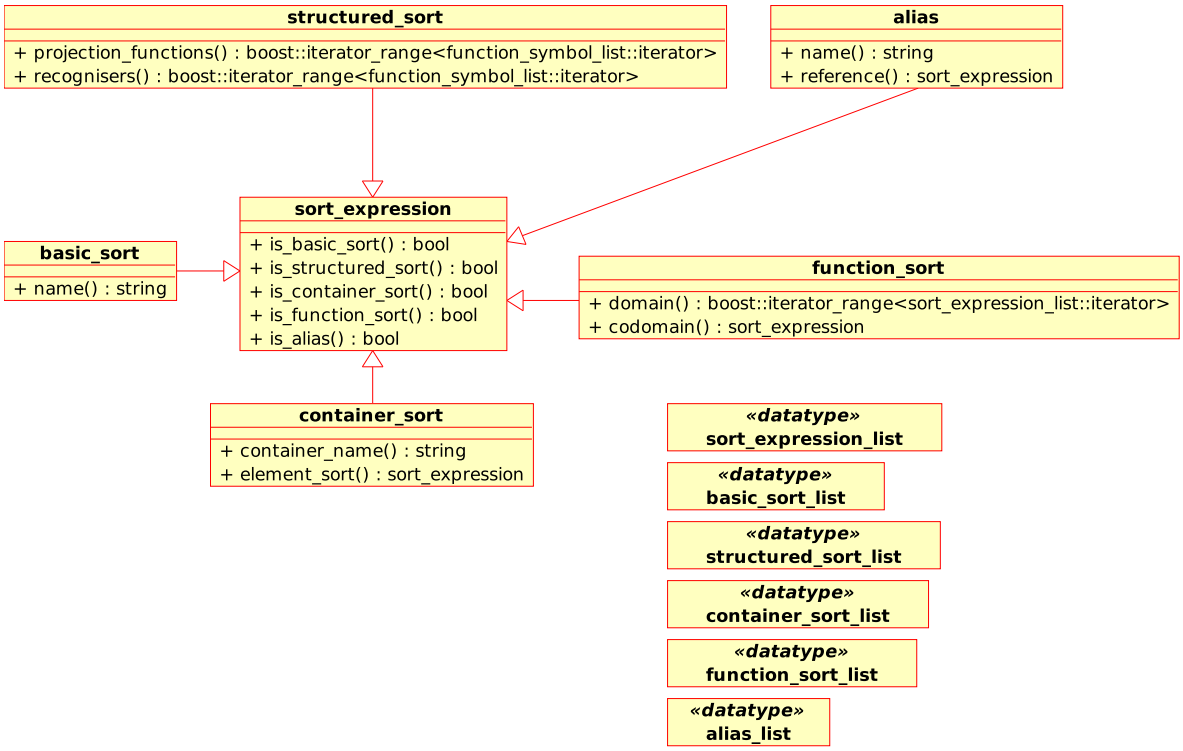

Figure 2: Class model for the sort expressions (in namespace `data`)

structured_sort	
description	This denotes a structured sort $$\begin{aligned} \textbf{struct}\quad & c_1(pr_{1,1} : S_{1,1}, \ldots, pr_{1,k_1} : S_{1,k_1})?isc_1 \\ \mid\quad & c_2(pr_{2,1} : S_{2,1}, \ldots, pr_{2,k_2} : S_{2,k_2})?isc_2 \\ \mid\quad & \vdots \\ \mid\quad & c_n(pr_{n,1} : S_{n,1}, \ldots, pr_{n,k_n} : S_{n,k_n})?isc_n; \end{aligned}$$ where $c_1, \ldots, c_n$ are called the constructors. In the sequel, where we write $Struct_{example}$, this entire structured sort should be substituted. $pr_{i,j}$ are called the projection functions. Note that the sort of projection function function $pr_{i,j}$ is $Struct_{example} \to S_{i,j}$. isc_i are called recogniser functions. The sort of recogniser function isc_i is $Struct_{example} \to Bool$
subclass of	sort_expression
superclass of	

public methods	
	projection_functions Returns the projection functions of the structured sort.
	recognizers Returns the recogniser functions of the structured sort.
	structured_sort_constructors Returns the constructors of the structured sort, including the projection and recogniser functions.

container_sort

description	This represents container sorts. These are sorts with a name and an element sort. An example of this is $List(Nat)$, representing the List of natural numbers. In this, List is the name of the container, and Nat is the element sort.
subclass of	sort_expression
superclass of	
public methods	
	container_name Returns the name of the container. For example the container name of $List(Nat)$ is $List$.
	element_sort Returns the sort of the elements of the container. For example the element sort of $List(Nat)$ is Nat.

11.2.2 Lists of sorts

For all classes concerned with sorts, list types are proved of the form *class_list*. This amounts to the following types:

sort_expression_list List of sort expressions

basic_sort_list List of basic sorts

structured_sort_list List of structured sorts

container_sort_list List of container sorts

function_sort_list List of function sorts

alias_list List of aliases

All these list types provide iterator interfaces.

11.3 Data expressions

11.3.1 Classes

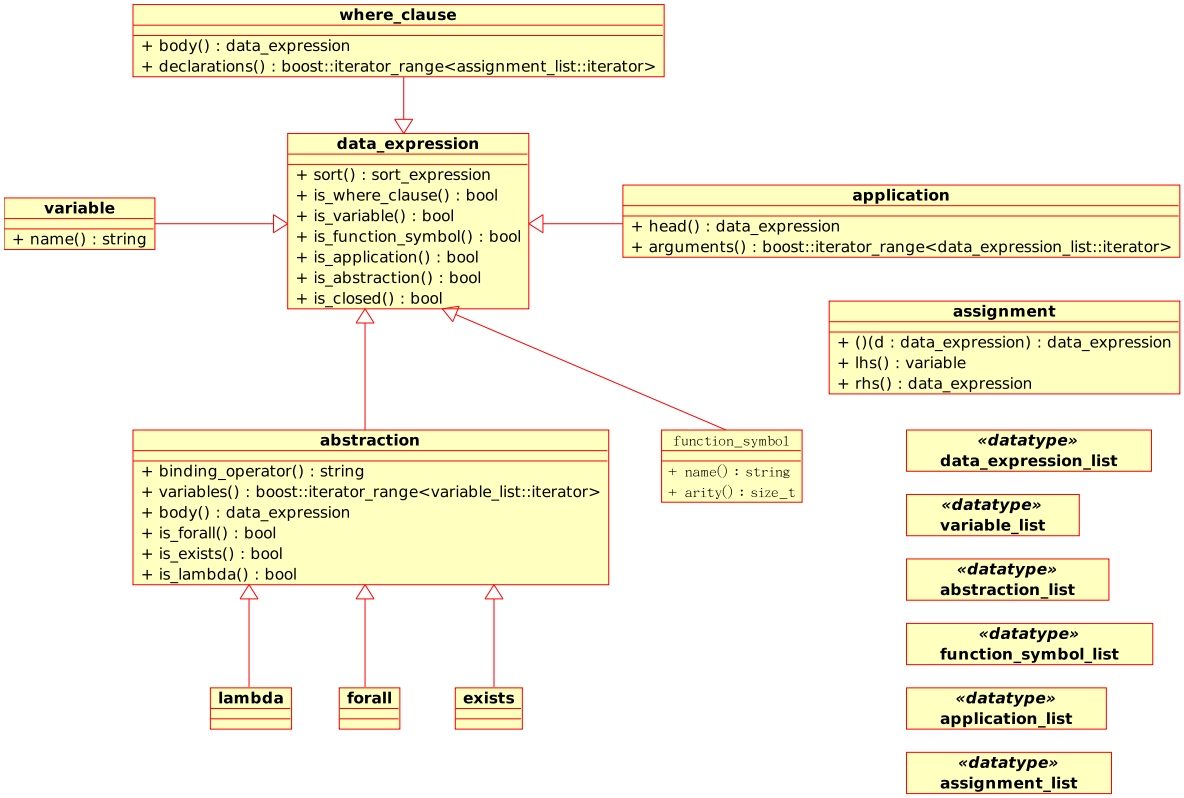

Figure 3: Class model for the data expressions (in namespace `data`)

data_expression

description	This denotes a data expression. The data expression structure corresponds directly to the definition in the theory.
subclass of	
superclass of	• variable • function_symbol • application • abstraction • where_clause
public methods	**sort** Returns the sort of the data expression. **is_variable** Returns *true* iff the data expression is a variable. **is_function_symbol** Returns *true* iff the data expression is a function symbol. **is_abstraction** Returns *true* iff the data expression is an abstraction. **is_application** Returns *true* iff the data expression is an application. **is_where_clause** Returns *true* iff the data expression is a where clause. **is_closed** Returns *true* iff the data expression is closed.

variable

description	This denotes a variable.
subclass of	data_expression
superclass of	
public methods	**name** Returns the name of the variable.

function_symbol

description	This denotes function symbols.
subclass of	data_expression

superclass of	
public methods	**name** Returns the name of the function symbol. **arity** Returns the arity of this function.

abstraction

description	This denotes lambda abstraction. Body is the data expression which is abstracted from, variables contains the abstraction variables.
subclass of	data_expression
superclass of	• lambda • forall • exists
public methods	**binding_operator** Returns the binding operator of the abstraction. **variables** Returns the binding variables of the abstraction. Note that this is never empty. **body** Returns the body of the abstraction, i.e. the expression that is abstracted. **is_lambda** Returns *true* iff the abstraction is a lambda abstraction. **is_forall** Returns *true* iff the abstraction is a universal quantification. **is_exists** Returns *true* iff the abstraction is an existential quantification.

application

description	This denotes application of a data expression (head) to a number of other data expressions (arguments).
subclass of	data_expression
superclass of	

public methods	
	head Returns the head of the application. For example the head of $t(t_0, \ldots, t_n)$ is t.
	arguments Returns the arguments of the application. For example the arguments of $t(t_0, \ldots, t_n)$ are $t_0, \ldots, t_n$.

lambda

description	This denotes lambda abstraction. This is an abstraction with binding operator lambda.
subclass of	abstraction
superclass of	
public methods	

forall

description	This denotes universal quantification. This is an abstraction with binding operator forall.
subclass of	abstraction
superclass of	
public methods	

exists

description	This denotes existential quantification. This is an abstraction with binding operator exists.
subclass of	abstraction
superclass of	
public methods	

where_clause

description	This denotes a where clause, e.g. e **whr** $x_0 = e_0, \ldots x_n = e_n$. Note that this behaves as a beta redex.
subclass of	data_expression
superclass of	

public methods	
	body Returns the body of the where clause.
	declarations Returns the local declarations of the where clause.

assignment

description	This describes an assignment of a value (data expression) to a data variable.
subclass of	
superclass of	
public methods	
	()(d) Returns d in which the assignment has been applied.
	lhs Returns the variable which is assigned a value.
	rhs Returns the data expression which is assigned to the variable.

11.3.2 Lists of data expressions

For all classes concerned with data expressions, list types are proved of the form *class_list*. This amounts to the following types:

data_expression_list List of data expressions

variable_list List of variables

abstraction_list List of abstractions

function_symbol_list List of function symbols

application_list List of applications

assignment_list List of assignments

All these list types provide iterator interfaces.

11.4 Data specification

A data specification represents a signature Σ with an associated set of equations E.

11.4.1 Classes

data_specification

description	This represents a data specification, directly from the theory. Note that sort aliases are included in the specification explicitly.[1]
subclass of	
superclass of	

[1]Note that additional methods may be added as the need arises.

public methods

sorts Returns all sorts in the specification, except the function sorts.

constructors Returns all constructors declared in the specification.

mappings Returns all mappings declared in the specification.

equations Returns all equations declared in the specification.

aliases(s) Returns all aliases of sort s.

dependent_sorts(constructor) Returns all sorts on which a constructor depends. See also Section 3.3.1.

dependent_sorts(sort) Returns all sorts on which a sort depends. See also Section 3.3.1.

constructors(sort) Returns all constructors of *sort*.

mappings(sort) Returns all mappings which have *sort* as a result.

equations(d) Returns all equations with d as the head of one of its sides.

is_certainly_finite(s) Returns true iff sort s is definately finite.

default_expression Returns a valid data expression according to this data specification of the given sort s. If no valid expression can be found, an exception is thrown. It returns a minimal term. When selecting function symbols, constructor symbols have a preference over mappings. For each sort, the same term is returned.

add_sort Adds a sort to the specification.

add_alias Adds an alias, so a name for a (possibly new) sort to the specification.

add_constructor Adds a constructor to the specification.

add_mapping Adds a mapping to the specification.

add_equation Adds an equation to the specification.

add_sorts(sl) Adds all sorts in sl to the specification.

add_constructors(fl) Adds all constructors in fl to the specification.

add_mappings(fl) Adds all mappings in fl to the specification.

add_equations(el) Adds all equations in el to the specification.

alias	
description	This denotes an alternative name for a sort, i.e. $S_0 = S_1$.
subclass of	sort_expression
superclass of	
public methods	**name** Returns the name of the alias. For example the name of the alias $S = T$ is S. **reference** Returns the sort the name refers to. For example the reference of the alias $S = T$ is T.

data_equation	
description	This represents a conditional equation, in which variables may be used.
subclass of	
superclass of	
public methods	**variables** Return the variables from the variable declaration section of the data equation. **condition** Returns the condition of the equation. **lhs** Returns the left hand side of the equation. **rhs** Returns the right hand side of the equation.

11.4.2 Lists

In addition to the lists defined so far, we also have the following list:

data_equation_list A list of data equations.

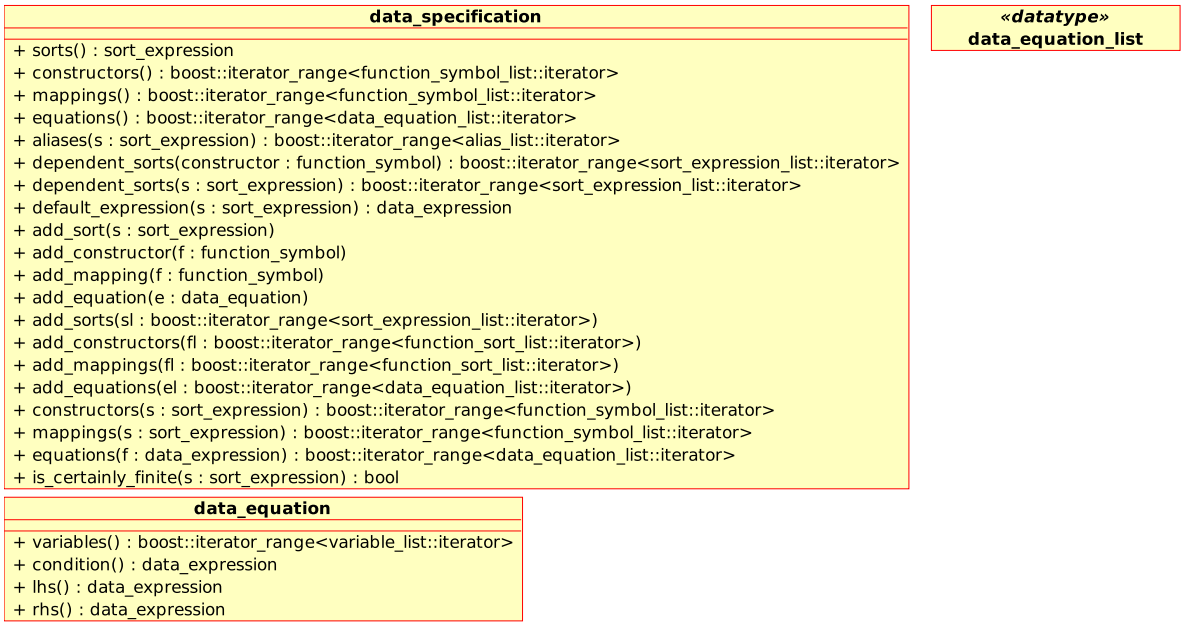

Figure 4: Class model for the data specification (in namespace `data`)

11.5 Utilities

This section describes a number of utilities for simplifying data expressions, and deciding equality. Note that this section merely describes minimal interfaces the actual components should adhere to.

11.5.1 Classes

rewriter	
description	A rewriter uses the equations from a data specification to obtain a normal form for a data expression. Upon construction time of the rewriter object a data specification is provided.
subclass of	
superclass of	
public methods	**()(d)** returns the normal form of d **()(d, sigma)** returns the normal form of d. On the fly the substitutions in *sigma* are performed. This is a variant of the rewriter which allows for more efficient rewriting in some cases.

enumerator_data_expression	
description	Contains a data expression which should be enumerated, and the variables that should be expanded
subclass of	24
superclass of	
public methods	**data_expression** Gives the data expression it contains. **variables** Gives the variables to be expanded.

public methods	
	enumerate(d) Gives the set of all possible values that d can attain.
	enumerate_one_level(ed) Gives the set resulting from expanding all variables of ed in the data expression of ed exactly once.

equality_checker

description	Determines whether two expressions are equal. If equality cannot be decided, false is returned.
subclass of	
superclass of	
public methods	
	()(t,t') Returns *yes* if t and t' are equal, *no* if they are not equal, *unknown* otherwise.
	()(t,t', sigma) Returns *yes* if t and t' are equal under substitution *sigma*, *no* if they are not equal under *sigma*, *unknown* otherwise.

prover

description	A prover determines whether a boolean expression is either a tautology or a contradiction. Note that it may be the case that a formula is neither a tautology, nor a contradiction.
subclass of	
superclass of	
public methods	
	is_tautology(d) Returns *yes* if data expression d is a tautology, *no* if d is not a tautology, *unknown* otherwise.
	is_contradiction(d) Returns *yes* if data expression d is a contradiction, $false$ if d is not a contradiction, *unknown* otherwise.

fresh_variable_generator

description	Generates variables with names that do not appear in the given context.
subclass of	
superclass of	

public methods	
	add_to_context Adds a term to the context.
	hint Returns the current hint that is used in the generated variable name.
	()(v) Returns a fresh variable of the same sort as v using the name of v as hint.
	()() Returns a fresh variable of sort *sort* using *hint*.
	set_context Sets the context to given term.
	set_hint Sets the hint.
	set_sort Sets the sort of the variables to be generated.
	sort Returns the current sort of variables that are generated.

data_eliminator

description	Facilitates elimination of unused or unnecessary parts of a data specification.
subclass of	
superclass of	
public methods	
	keep_sort Keep the sort, and all sorts it depends on.
	keep_data_from_data_expression Keep the sorts and mappings used in data expression.
	() Remove all sorts and data expressions that should not be kept from the data specification.

11.5.2 Lists

For the classes in utilities we also have lists, with their corresponding iterators:

enumerator_data_expression_list A list of enumerator data expressions.

11.5.3 Type definitions

In order to denote that an answer is unknown in the equality checker and the prover we introduce a tri-valued type **answer** with the obvious meaning.

11.5.4 Utility functions

There are also a number of utility functions in the data library that are plain functions. These will be described in this section.

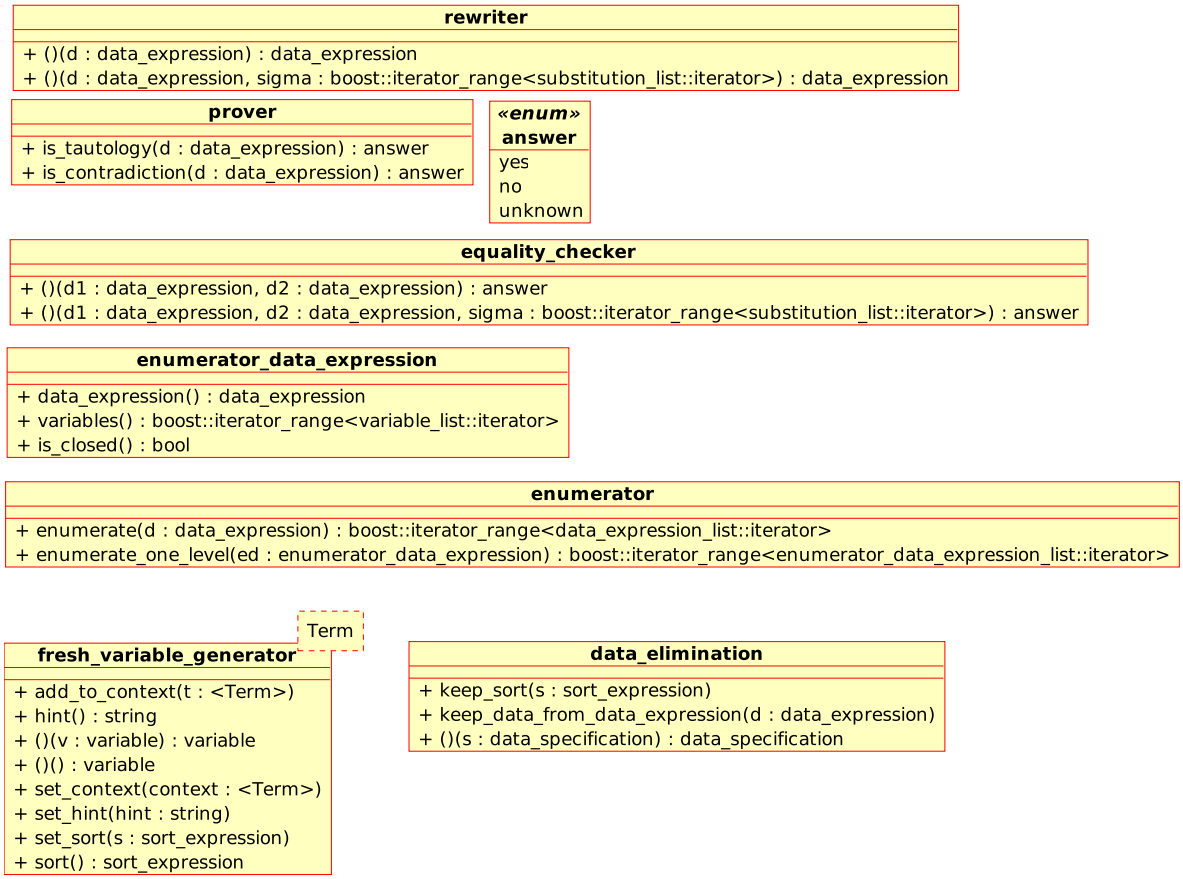

Figure 5: Class model for the utilities (in namespace `data`)

find First of all there are a number of find functions which follow the find functions in the details section of the current data library. We will only give their names here.

find_data_expression

find_data_variable

find_data_application

find_function_symbol

find_abstraction

find_all_data_expressions

find_all_data_variables

find_all_data_applications

find_all_function_symbols

find_all_abstractions

find_sort_expression

find_basic_sort

find_structured_sort

find_container_sort

find_alias

find_all_sort_expressions

find_all_basic_sorts

find_all_structured_sorts

find_all_container_sorts

find_all_aliases

replace Additionally there are a number of functions to facilitate replacements. Again this follows the same pattern as in the current implementation, and we provide reference to the names only.

replace_data_expressions

replace_data_variables

replace_basic_sorts

replace_sort_expressions

11.6 System defined functions and equations

For each system defined sort, i.e. *Bool*, *Pos*, *Nat*, *Int* and *Real*, utility functions will be created to add the declarations of these sorts, with their constructors, mappings and equations to a data specification. For other system defined mappings and equations similar functions will be constructed. Note that for everything that is added in this way, it is recorded that it is system defined. Furthermore utility functions will be added to facilitate easy manipulation of these system defined sorts.

11.7 The internal format used in the new data library design

As noted before, the format used in the new data library is an intermediate format between the internal format after typechecking and the internal format after data implementation as used up to revision 5976. In essence it is the format after typechecking, except for some minor adjustments. Namely, numbers, lists, sets and bags are stored in their concrete, or implemented, form, i.e. they are stored according to the specification given in Appendix A of [**?**]. This is the format assumed in the data library. Furthermore there are facilities that automatically import the necessary declarations for standard data types. This functionality is provided by the function `make_complete`.

The rewriters still use their original (undocumented) formats, however, as we now do have binders available explicitly on the data library level (along with where clauses), these need to be translated away to a rewrite rules. This translation is carried out internally in the interface to the rewriters that is provided by the data library.