Data types for mCRL2

Aad Mathijssen

January 19, 2019

We provide a syntax for the standard data types of the mCRL2 language. This syntax is intended to be a practical mix between standard mathematical notation and the syntax of specification languages and programming languages in general.

1 Basic formalism

The underlying theory of the mCRL2 data types is abstract data types. Abstract data types consist of:

- sorts and operations on these sorts;
- equations on terms made up from operations and variables, where the terms are of the same sort.

Sorts are declared using the keyword **sort**, operations using the keyword **map** and equations using the keyword **eqn**. Variables that are used in the equations need to be declared using the keyword **var**. To distinguish constructor operations from normal operations the former may be declared using the keyword **cons**. However, the use of these constructors is strongly discouraged, because in practise they are implicitly defined via the concrete data types. With the keyword **sort**, also abbreviations for sort expressions may be introduced.

1.1 Expressions

There are three kind of expressions in mCRL2, namely expressions over sorts, over data and over processes. Sort expressions are made up from existing sort names and from representations of predefined data types and type constructors. Data expressions are terms constructed from operations and variables. These data expressions are in the standard functional notation, but in the next sections also mixfix notation is introduced. To increase readability of the operation declarations in the following sections, the sorts are left implicit and underscores indicate the placement of the parameters.

Where clauses may be used as an abbreviation mechanism in data expressions. A where clause is of the form e **whr** $a_0 = e_0, \ldots a_n = a_n$ **end**, with $n \in \mathbb{N}$. Here, e is a data expression and, for all i, $0 \leq i \leq n$, a_i is an identifier and e_i is a data expression. Expression e is called the body and each equation $a_i = e_i$ is called a *definition*. Each identifier a_i is used as an abbreviation for e_i in e, even if a_i is already defined in the context. Also, an identifier a_i may not occur in any of the expressions e_j, $0 \leq j \leq n$. As a consequence, the order in which expressions occur is irrelevant.

2 Predefined data types

A number of predefined data types are provided. For each data type a sort is provided together with a number of predefined operations.

2.1 Booleans

The boolean type is represented by the sort *Bool*. For this sort, we have the following operations.

Operator	*Rich*	`Plain`
true	*true*	`true`
false	*false*	`false`
negation	$\neg\_$	`!_`
conjunction	$\_ \wedge \_$	`_ && _`
disjunction	$\_ \vee \_$	`_ || _`
implication	$\_ \Rightarrow \_$	`_ => _`
universal quantification	$\forall \_:\_.\_$	`forall _:_._`
existential quantification	$\exists \_:\_.\_$	`exists _:_._`

In the quantifiers the first two parameters form a variable declaration, consisting of a name and a sort; the third parameter is the body.

We also have operations that express the equality and inequality of two terms of the same sort, and an operation that expresses the conditional. These operations are available for any predefined sort, and are automatically generated for user defined sorts.

Operator	*Rich*	`Plain`
equality	$\_ \approx \_$	`_ == _`
inequality	$\_ \not\approx \_$	`_ != _`
conditional	$\mathit{if}(\_,\_,\_)$	`if(_,_,_)`
less than	$\_ < \_$	`_ < _`
less than or equal	$\_ \leq \_$	`_ <= _`
greater than or equal	$\_ \geq \_$	`_ >= _`
greater than	$\_ > \_$	`_ > _`

2.2 Numbers

Positive numbers, natural numbers, integers and real numbers (rational number approximations) are represented by the sorts *Pos*, *Nat*, *Int* and *Real*. For these sorts, we have the following operations, where $A, B \in \{ Pos, Nat, Int, Real \}$:

Operator	*Rich*	`Plain`
positive numbers	$1, 2, 3, \ldots$	`1,2,3,...`
natural numbers	$0, 1, 2, \ldots$	`0,1,2,...`
integers	$\ldots, -2, -1, 0, 1, 2, \ldots$	`...,-2,-1,0,1,2,...`
rational numbers	$\ldots$	`...`
	$-1/1, -1/2, -1/3, \ldots$	`-1/1,-1/2,-1/3,...`
	$0/1, 0/2, 0/3, \ldots$	`0/1,0/2,0/3,...`
	$1/1, 1/2, 1/3, \ldots$	`1/1,1/2,1/3,...`
	$\ldots$	`...`
conversion	*A2B*(_)	`A2B(_)`
maximum	*max*(_, _)	`max(_,_)`
minimum	*min*(_, _)	`min(_,_)`
absolute value	*abs*(_)	`abs(_)`
negation	$-$_	`-_`
successor	*succ*(_)	`succ(_)`
predecessor	*pred*(_)	`pred(_)`
addition	_ + _	`_ + _`
subtraction	_ − _	`_ - _`
multiplication	_ * _	`_ * _`
division	_/_	`_ / _`
integer div	_ **div** _	`_ div _`
integer mod	_ **mod** _	`_ mod _`
exponentiation	*exp*(_, _)	`exp(_,_)`
floor	*floor*(_)	`floor(_)`
ceiling	*ceil*(_)	`ceil(_)`
round	*round*(_)	`round(_)`

Explicit type conversions can be done using the conversion operation *A2B*. The other operations perform implicit type conversions, when needed.

3 Type constructors

Type constructors are predefined operations on sorts with which we can construct sorts. For each sort that is constructed this way a number of operations are provided.

3.1 Lists

Lists, where all elements are of sort A, are declared by the sort expression $List(A)$. The following operations are provided for this sort.

Operator	*Rich*	`Plain`
construction	$[\_,\ldots,\_]$	`[_,...,_]`
element test	$\_ \in \_$	`_ in _`
length	$\#\_$	`#_`
cons	$\_ \triangleright \_$	`_ |> _`
snoc	$\_ \triangleleft \_$	`_ <| _`
concatenation	$\_ +\!\!+ \_$	`_ ++ _`
element at position	$\_ \,.\, \_$	`_ . _`
the first element of a list	$head(\_)$	`head(_)`
list without its first element	$tail(\_)$	`tail(_)`
the last element of a list	$rhead(\_)$	`rhead(_)`
list without its last element	$rtail(\_)$	`rtail(_)`

The empty list is represented by an empty list construction, i.e. $[\,]$. Also note that the lists $[a, b]$, $a \triangleright [b]$ and $[a] \triangleleft b$ are all equivalent.

3.2 Sets and bags

Possibly infinite sets and bags where all elements are of sort A are declared by the sort expressions $Set(A)$ and $Bag(A)$, respectively. The following operations are provided for these sorts.

Operator	*Rich*	`Plain`
set enumeration	$\{\_,\ldots,\_\}$	`{ _,...,_ }`
bag enumeration	$\{\_:\_,\ldots,\_:\_\}$	`{ _:_,...,_:_}`
comprehension	$\{\_:\_ \mid \_\}$	`{ _:_ | _ }`
element test	$\_ \in \_$	`_ in _`
bag multiplicity	$count(\_,\_)$	`count(_,_)`
subset/subbag	$\_ \subseteq \_$	`_ <= _`
proper subset/subbag	$\_ \subset \_$	`_ < _`
union	$\_ \cup \_$	`_ + _`
difference	$\_ - \_$	`_ - _`
intersection	$\_ \cap \_$	`_ * _`
set complement	$\bar{\_}$	`!_`
convert set to bag	$Set2Bag(\_)$	`Set2Bag(_)`
convert bag to set	$Bag2Set(\_)$	`Bag2Set(_)`

The empty set of bag is represented by an empty enumeration, i.e. $\{\ \}$. Note that a set enumeration declares a set, not a bag. So e.g. $\{a, b, c\}$ declares the same set as $\{a, b, c, c, a, c\}$. In a bag enumeration the number of times an element occurs has to be declared explicitly.

So e.g. $\{ a : 2, b : 1 \}$ declares a bag consisting of two a's and one b. Also $\{ a : 1, b : 1, a : 1 \}$ declares the same bag. A set comprehension $\{ x : A \mid P(x) \}$ declares the set consisting of all elements x of sort A for which predicate $P(x)$ holds, i.e. $P(x)$ is an expression of sort *Bool*. A bag comprehension $\{ x : A \mid f(x) \}$ declares the bag in which each element x occurs $f(x)$ times, i.e. $f(x)$ is an expression of sort *Nat*. Note that functions P and f have to be total.

3.3 Function types

A function type of total functions from $X_0 \times \ldots \times X_n$ to Y is declared by the sort expression $X_0 \times \ldots \times X_n \to Y$. The following operations are provided for these sorts.

Operator	*Rich*	`Plain`
function application	$\_(\_, \ldots, \_)$	`_(_,...,_)`
lambda abstraction	$\lambda \_ : X_0, \ldots, \_ : X_n . \_$	`lambda _:X0,...,_:Xn._`

Function types may be nested. To make this unambiguous, function arrow $\to$ is right associative, which may be overridden by using parentheses. Also it is not allowed to have sort expressions with $\to$ as its head at the left hand side of a function type. If this is needed, parentheses need to be used.

3.4 Structured types

Structured types consist of *sum* types and *product* types. A structured type is declared by the following sort expression, where $n \in \mathbb{N}^+$ and $k_i \in \mathbb{N}$ with $1 \leq i \leq n$:

$$\begin{aligned} \textbf{struct}\ & c_1(pr_{1,1} : A_{1,1},\ \ldots, pr_{1,k_1} : A_{1,k_1})?is\_c_1 \\ \mid\ & c_2(pr_{2,1} : A_{2,1},\ \ldots, pr_{2,k_2} : A_{2,k_2})?is\_c_2 \\ & \vdots \\ \mid\ & c_n(pr_{n,1} : A_{n,1},\ \ldots, pr_{n,k_n} : A_{n,k_n})?is\_c_n \end{aligned}$$

From this declaration it can be seen that at least 1 summation has to be specified and a summation may consist of 0 products. Each summation i is labelled by a *constructor* c_i and optionally by a *recogniser* is_c_i. Recogniser is_c_i is used to determine if a term of the above sort is constructed with c_i. If a recogniser label is left out, the corresponding ? is also left out. Each product (i, j) is optionally labelled by a *projection* $pr_{i,j}$. With this projection, the j'th element of summation i can be obtained. If a projection label is left out, the corresponding : is left out. If a summation i does not have any products, it is written as $c_i?is\_c_i$ instead of $c_i()?is\_c_i$.

All labels have to be chosen such that no ambiguity can arise. Each sort $A_{i,j}$ has to be a valid sort expression, in which forward references to sort labels may occur. This means that it is allowed to specify systems of equations of structured types.

The following operations are generated for the above sort. Projection and recogniser operations are only generated if the user specified them.

Operator	*Rich*	`Plain`
constructor of summation i	$c_i(\_, \ldots, \_)$	`ci(_,...,_)`
recogniser for constructor i	$is\_c_i(\_)$	`is_ci(_)`
projection (i, j), if declared	$pr_{i,j}(\_)$	`prij(_)`

We give a few examples of sort declarations involving structured sorts. For finite $n \in \mathbb{N}$, an enumerated type can be declared by

sort $Enum = \textbf{struct}\ enum_0?is\_enum_0 \mid \ldots \mid enum_n?is\_enum_n$

This generates constructor operations $enum_i : Enum$, together with recogniser operations $is\_enum_i : Enum \to Bool$, with $0 \leq i \leq n$.

Pairs of elements of sort A and B can be declared as follows:

sort $ABPair = \textbf{struct}\ pair(fst : A, snd : B)$

For this declaration, constructor operation $pair : A \times B \to ABPair$ and projection operations $fst : ABPair \to A$ and $snd : ABPair \to B$ are generated.

Binary trees where all leaves and nodes are labelled with elements of sort A can be declared as follows: Example:

sort $BATree = \textbf{struct}\ leaf(A) \mid node(BATree, A, BATree)$

Or, with projection and recogniser labels:

sort $BATree = \textbf{struct}\ leaf(lval : A)?is\_leaf$
$\quad \mid node(left : BATree, nval : A, right : BATree)?is\_node$

The quantification of an associative operation $f : A \times A \to A$ over all labels in a $BATree$, can be defined in the same way for both declarations of the tree:

map $qf : BATree \to A$
var $t, u : BATree$
$a : A$
eqn $qf(leaf(a)) = a$
$qf(node(t, a, u)) = f(qf(t), f(a, qf(u)))$

The last definition of sort $BATree$ also allows the definition of operation qf without pattern matching:

var $t : BATree$
eqn $qf(t) = if(is\_leaf(t), lval(t), f(qf(left(t)), f(nval(t), qf(right(t)))))$

3.4.1 Pattern matching versus projections and recognisers

When defining *data equations*, it is often the case that pattern matching on the constructors of a structured type is more elegant than using projection and recogniser operations. The last example is a typical instance of this.

When defining *process equations*, pattern matching can not be applied directly on the left hand side of the equation. However, we can apply it indirectly through the use of the summation and conditional operators in the right hand side. As an example, take the following process declaration without pattern matching:

proc $P(t : BATree) = is\_leaf(t) \to get(lval(t)).\delta +$
$is\_node(t) \to get(nval(t)).(P(left(t)) + P(right(t)))$

This declaration is equivalent to the following declaration, which uses pattern matching:

$$\textbf{proc} \quad P(t : BATree) = \begin{array}{ll} \sum_{a:A} & (t \approx leaf(a)) \rightarrow get(a).\delta + \\ \sum_{a:A,u,v:BATree} & (t \approx node(u,a,v)) \rightarrow get(a).(P(u) + P(v)) \end{array}$$

As can be seen from the above example, it is arguable which way is the most elegant to specify process equations.

4 Parsing issues

The following issues are important not only for parsing by computer, but also for parsing by humans.

4.1 Relation with processes

Data terms occur in relation to processes in the following ways:

- action parameters
- arguments of a process reference
- left argument of conditional process terms ($b \rightarrow p \diamond q$)
- right argument of a timed process term ($p@t$)

These last two are ambiguous or hard to read for where clauses, quantifications and infix operations. For this reason, these operations need to be parenthesized.

4.2 Type inference

Data expressions involving numbers, sets, bags or lists may be ambiguous. E.g. namely 1 can be parsed as a term of sort *Pos*, *Nat*, *Int* or *Real*, and { } can be parsed as a term of an arbitrary set or bag sort. For overloaded operations, we have similar problems.

These problems are solved by a type inference system: if the type of an expression can be determined unambiguously, the system is able to infer the type of this expression.

4.3 Priorities and associativity

The prefix operators have the highest priority, followed by the infix operators, followed by the lambda operator together with universal and existential quantification, followed by the where clause. Table 1 lists the infix operators by decreasing priority. The symbols are shown in plain text format and may represent multiple rich text symbols. Operators on the same line have the same priority and associativity. Note that the list operations $\triangleright$, $\triangleleft$ and $+\!\!+$ are split into three priority levels such that expressions with one of these operations as their head symbol are allowed if and only if they match the following pattern, where $b, \ldots, c$, $d, \ldots, e$ and $s, \ldots, t$ are expressions with a priority level greater than $+\!\!+$:

$$b \triangleright \ldots \triangleright c \triangleright s +\!\!+ \ldots +\!\!+ t \triangleleft d \triangleleft \ldots \triangleleft e$$

operators	associativity
`*`, `.`	left
`/`, `div`, `mod`	left
`+`, `-`	left
`|>`	right
`<|`	left
`++`	left
`<`, `>`, `<=`, `>=`, `in`	none
`==`, `!=`	right
`&&`, `||`	right
`=>`	right

Table 1: Precedence of infix operators

4.4 Design decisions

In the development of the language, a number of design decisions were taken. The most important ones are listed here:

- Layout may not have any effect on the semantics of the language.
- Declarations have to be terminated by a semicolon. If this was not required the language would be ambiguous. E.g. consider the following operation declarations:

  $$\begin{aligned} X &= f(g) \\ (k) &= Y \end{aligned}$$

  From the layout it is clear that X is equal to $f(g)$ and Y to (k). However, since layout may not have any effect on the semantics of the language, it is also possible that X is equal to f and Y to $(g)(k)$.

5 Comparison of data languages

The data part of both the μCRL and the mCRL2 languages have a lot in common with functional programming languages. For the features that involve functional programming languages the following comparison is made.

Aspect \ Language	μCRL	mCRL2	Haskell/Clean	MetaOCaml
Purely functional	yes	yes	yes	no
Expressiveness	first-order	higher-order	higher-order	higher-order
Strict	no	no	no	yes
Evaluation	somewhat lazy	somewhat lazy	lazy	eager
Control of eval. order	no	yes	yes	yes
Partial evaluation	yes	yes	no	yes
Polymorphism	no	no	yes	yes
Modules	no	no	yes	yes
Object orientation	no	no	no	yes
Concrete data types	no	yes	yes	yes

Table 2: Comparison of data languages

Note that originally the evaluation in μCRL and mCRL2 was eager, but the addition of just-in-time strategies has moved towards laziness. Since both languages have a non-strict semantics and evaluation is not fully lazy, the evaluation only *approximates* the semantics. However, this problem rarely pops up in practise and if it pops up, it can usually be circumvented by controlling the evaluation order. This can be achieved by the use of *conditional rewrite rules*.