Data Types for GenSpect

Aad Mathijssen

23th September 2004

Basic formalism

Data types in GenSpect are 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

Declaration in GenSpect:

Declaration	Keyword
sorts	**sort**
operations	**op**
variables	**var**
equations	**eqn**

Predefined data types: booleans

Booleans are 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 _:_._`

Predefined data types: booleans (2)

For any sort (predefined or user defined), we have the following operations:

Operator	*Rich*	`Plain`
equality	$\_ = \_$	`_ == _`
inequality	$\_ \neq \_$	`_ <> _`
conditional	$\mathit{if}(\_, \_, \_)$	`if(_,_,_)`

For sort $Bool$, we have:

- equality is bi-implication
- inequality is exclusive-or

Predefined data types: numbers

Positive numbers, natural numbers and integers are represented the sorts Pos, Nat and Int.

For these sorts, we have the following operations:

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,...`
negation	$-\_$	`-_`
addition	$\_ + \_$	`_ + _`
subtraction	$\_ - \_$	`_ - _`
multiplication	$\_ * \_$	`_ * _`
integer div	$\_ \mathbf{div} \_$	`_ div _`
integer mod	$\_ \mathbf{mod} \_$	`_ mod _`

Predefined data types: numbers (2)

And the following operations, where A and B are numeric sorts:

Operator	*Rich*	`Plain`
exponentiation	$exp(\_,\_)$	`exp(_,_)`
increment	$inc(\_)$	`inc(_)`
decrement	$dec(\_)$	`dec(_)`
absolute value	$abs(\_)$	`abs(_)`
maximum	$max(\_,\_)$	`max(_,_)`
minimum	$min(\_,\_)$	`min(_,_)`
less than	$\_ < \_$	`_ < _`
greater than	$\_ > \_$	`_ > _`
less than or equal	$\_ \leq \_$	`_ <= _`
greater than or equal	$\_ \geq \_$	`_ >= _`
conversions	$A2B(\_)$	`A2B(_)`

Type constructors: lists

Singly linked lists consisting of elements of sort A only:

sort $L = List(A)$

The following operations are provided for this sort:

Operator	*Rich*	`Plain`
construction	$[\_, \ldots, \_]$	`[_,...,_]`
length	$\#\_$	`#_`
cons	$\_ \triangleright \_$	`_ |> _`
snoc	$\_ \triangleleft \_$	`_ <| _`
concatenation	$\_ \mathbin{+\!\!+} \_$	`_ ++ _`
element at position	$\_ \hat{} \_$	`_ ^ _`

Lists are constructed from $[\,]$ and $\triangleright$.
$[a, \ldots, z]$ is an abbreviation of $a \triangleright \ldots \triangleright z \triangleright [\,]$.

Type constructors: lists (2)

We also have the following operations:

Operator	*Rich*	`Plain`
empty predicate	$isempty(\_)$	`isempty(_)`
the first element of a list	$lhead(\_)$	`lhead(_)`
list without its first element	$ltail(\_)$	`ltail(_)`
the last element of a list	$rhead(\_)$	`rhead(_)`
list without its last element	$rtail(\_)$	`rtail(_)`

Operations *isempty*, *ltail* and *ltail* have constant time complexity
The other operations have linear time complexity.

The introduced syntax can be used for both cons and snoc lists.
Should we leave the choice to the user?

Type constructors: sets and bags

Sets and bags consisting of elements of sort A only:

sort $S = Set(A)$
$B = Bag(A)$

The following operations are provided:

Operator	*Rich*	`Plain`
set enumeration	$\{\_, \ldots, \_\}$	`{ _,...,_ }`
bag enumeration	$\{\_ : \_, \ldots, \_ : \_\}$	`{ _:_,...,_:_}`
comprehension	$\{\_ : \_ \mid \_\}$	`{ _:_	_ }`

A comprehension is of the form $\{\, x{:}A \mid f(x)\,\}$, where:

- f is a total function of type $A \rightarrow Bool$ for sets
- f is a total function of type $A \rightarrow Nat$ for bags

Type constructors: sets and bags (2)

We also have the following operations:

Operator	*Rich*	`Plain`
size (cardinality)	$\#\_$	`#_`
bag multiplicity / set element test	$\_ \cdot \_$	`_ . _`
element test	$\_ \in \_$	`_ in _`
subset/subbag	$\_ \subseteq \_$	`_ <= _`
proper subset/subbag	$\_ \subset \_$	`_ < _`
union	$\_ \cup \_$	`_ + _`
intersection	$\_ \cap \_$	`_ * _`
difference	$\_ \setminus \_$	`_ \ _`
set complement	$\_'$	`_~`

Note that the empty set or bag is written as an empty enumeration: $\{\ \}$.

Type constructors: function types

A function type of total functions from X to Y:

sort $F = X \rightarrow Y$

The following operations are provided for this sort:

Operator	*Rich*	`Plain`
function application	$\_.\_$	`_ . _`
lambda abstraction	$\lambda\_:\_.\_$	`lambda _:_._`

Type constructors: structured types

General form of structured types, where $n \in \mathbb{N}^+$ and $k_i \in \mathbb{N}$, $1 \leq i \leq n$:

$$\begin{array}{lrl} \textbf{sort} & A = & c_1 : (pr_{1,1} : A_{1,1}) \times \ldots \times (pr_{1,k_1} : A_{1,k_1}) \\ & | & c_2 : (pr_{2,1} : A_{2,1}) \times \ldots \times (pr_{2,k_2} : A_{2,k_2}) \\ & & \vdots \\ & | & c_n : (pr_{n,1} : A_{n,1}) \times \ldots \times (pr_{n,k_n} : A_{n,k_n}) \end{array}$$

Remarks:

- At least 1 *summation*, possibly 0 *products*.
- Each summation i is labelled by a *constructor* c_i.
- Each product (i, j) is labelled by a *projection* $pr_{i,j}$.
- All labels have to be distinct.
- Each sort $A_{i,j}$ has to be either declared or equal to A.
- Projection labels and parentheses are optional.

Type constructors: structured types (2)

The following operations are provided for sort A:

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

A projection operation is only provided when its projection label is declared.

Type constructors: structured type examples

For finite $n \in \mathbb{N}$, an enumerated type can be declared as follows:

sort $\quad Enum = enum_0 \mid \ldots \mid enum_n$

Provided operations, for all i, $0 \leq i \leq n$:

- constructor operation $enum_i :\rightarrow Enum$
- membership operations $is\_enum_i : Enum \rightarrow Bool$

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

sort $\quad ABPair = pair : (fst : A) \times (snd : B)$

Provided operations:

- constructor and membership operation for label *pair*
- projection operations $fst : ABPair \rightarrow A$ and $snd : ABPair \rightarrow B$

Type constructors: structured type examples (2)

Binary trees where all leaves and nodes are labelled with elements of sort A:

sort $\quad T = \mathit{leaf} : (\mathit{lval} : A) \mid \mathit{node} : (\mathit{left} : T) \times (\mathit{nval} : A) \times (\mathit{right} : T)$

Quantification of an associative operation $f : A \times A \to A$ over all labels in a T tree:

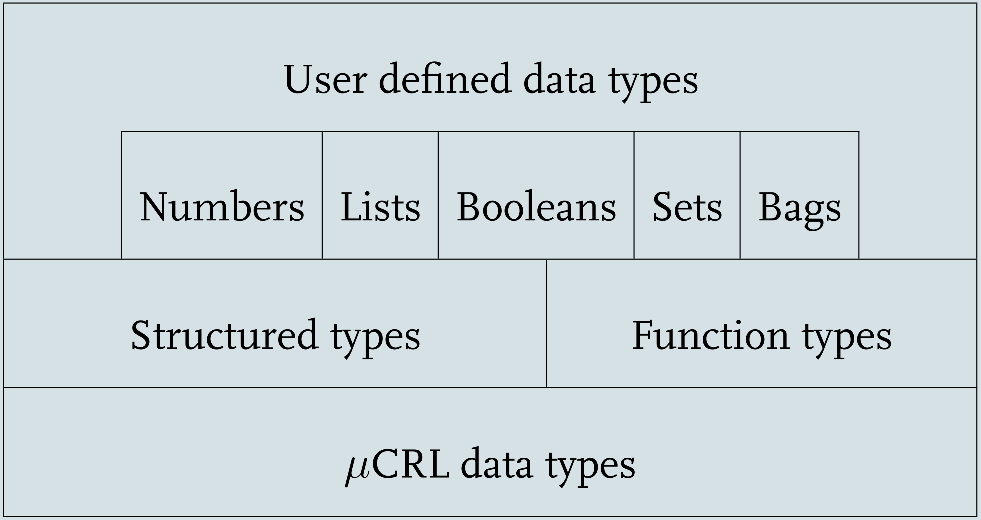
$$
\begin{array}{ll}
\textbf{op} & \mathit{quantf} : T \to A \\
\textbf{var} & t, u : T \\
 & a : A \\
\textbf{eqn} & \mathit{quantf}(\mathit{leaf}(a)) = a \\
 & \mathit{quantf}(\mathit{node}(t, a, u)) = f(\mathit{quantf}(t), f(a, \mathit{quantf}(u)))
\end{array}
$$

Without pattern matching:

$$
\begin{array}{ll}
\textbf{var} & t : T \\
\textbf{eqn} & \mathit{quantf}(t) = \mathit{ite}(\mathit{isleaf}(t), \mathit{lval}(t), \\
 & \qquad\qquad f(\mathit{quantf}(\mathit{left}(t)), f(\mathit{nval}(t), \mathit{quantf}(\mathit{right}(t)))))
\end{array}
$$

Hierarchy

Hierarchy in the context of μCRL:



Extensions

GenSpect representations were provided for the most important data types.

When needed, the following predefined data types can be added:

- characters
- strings

Analogously, the following type constructors can be added:

- $Nat_{mod(n)}$, for finite $n \in \mathbb{N}$
- (infinite) tables, arrays
- stacks, queues
- $\ldots$

Process declarations

Processes are defined by means of process equations.
Variables occurring at the right hand side must occur at the left hand side.
There are three candidate representations.

1) Current μCRL representation:

$$\textbf{proc} \quad P(t:T) = \begin{array}{l} isleaf(t) \;\; \rightarrow get(lval(t)).\delta + \\ isnode(t) \rightarrow get(nval(t)).(P(left(t)) + P(right(t))) \end{array}$$

2) Current μCRL representation extended with pattern matching:

$$\begin{array}{lll} \textbf{proc} & P(leaf(a:A)) & = get(a).\delta \\ & P(node(t:T, a:A, u:T)) & = get(a).(snd(d).P(t,d,e) + \\ & & \quad snd(e).P(u,d,e)) \end{array}$$

Process declarations (2)

3) Representation 2) where variable declarations are separated from the process declarations and a type declaration for the process is added:

$$
\begin{array}{lll}
\textbf{proc} & P : T & \\
\textbf{var} & a : A & \\
 & t, u : T & \\
\textbf{pdef} & P(leaf(a)) & = get(a).\delta \\
 & P(node(t, a, u)) & = get(a).(snd(d).P(t, d, e) + snd(e).P(u, d, e))
\end{array}
$$

Advantages 2) and 3) over 1): avoid membership and projection operations.
Advantages 3) over 2) and 1): closest to the definition of data operations.

Process declarations (3)

A syntactical improvement: shorten process references using an *assignment*.

Example:
We may write $Q(x := c)$ instead of $Q(a, b, c, d)$, if only the second argument has to be changed.

Problems:

- the pattern matching variants may complicate the left hand side of the assignment, e.g. $P(node(t, a, u) := t)$.
- what to do with references to different process equations?

Parsing issues: relation with processes

Occurrences of data terms in relation to processes:

- action parameters
- process declarations (representations 2 and 3)
- arguments of a process reference
- left argument of conditional process terms ($b \rightarrow p$)
- right argument of a timed process term ($p@t$)

Last two are ambiguous / hard to read for quantifications and infix operations. These operations need to be parenthesized.

Parsing issues: type inference

Type inference is needed, because:

- operations may be overloaded
- numbers are ambiguous (1 can be of sort Pos, Nat or Int)

Parsing issues: priorities

Precedence of operators:
postfix $>$ prefix $>$ $\forall\_:\_.\_$, $\exists\_:\_.\_$ $>$ binary $>$ $\lambda\_:\_.\_$

Precedence of binary operators:

<table>
<tr><th>Pr.</th><th>operator</th></tr>
<tr><td>1</td><td>^, .</td></tr>
<tr><td>2</td><td>*, div, mod, \</td></tr>
<tr><td>3</td><td>+, -</td></tr>
<tr><td>4</td><td><, >, <=, >=, <|, |>, ++, in</td></tr>
<tr><td>5</td><td>==, <></td></tr>
<tr><td>6</td><td>&&, ||</td></tr>
<tr><td>7</td><td>=></td></tr>
</table>

Parsing issues: associativity

The following binary operators are associative:

<table>
<tr><th>operator</th><th>associativity</th></tr>
<tr><td>.</td><td>left</td></tr>
<tr><td>*</td><td>left/right</td></tr>
<tr><td>+</td><td>left/right</td></tr>
<tr><td>|></td><td>right</td></tr>
<tr><td><|</td><td>left</td></tr>
<tr><td>++</td><td>left/right</td></tr>
<tr><td>==</td><td>left/right</td></tr>
<tr><td><></td><td>left/right</td></tr>
<tr><td>&&</td><td>left/right</td></tr>
<tr><td>||</td><td>left/right</td></tr>
</table>