Mathematical Extension in Event-B through the Rodin Theory Component

Michael Butler, Issam Maamria

DRAFT: 8 June 2010

1 Introduction

This paper outlines how some of the concepts proposed in [1] to deal with mathematical extension in Event-B will be realised in Rodin from a user perspective. We show the theory component, currently defined by the rule-based prover plug-in, will be extended to allow for new basic predicate definitions, new operator definitions and new inductive data type definitions. The theory component will also support the definition of rewrite rules and inference rules to be used by the rule-based prover.

Mostly the document reflects the outcome of the WP9 meeting in Southampton on 6 May 2010. Various people contributed to the discussion, including Stefan Hallerstede, Alexei Iliasov, Michael Leuschel, Matthias Schmalz and Laurent Voisin.

The first version of mathematical extension will not support the declaration of user defined binders nor will it support rules that require substitution for variables. These will be added later. These proposals do not include support for dependant types nor predicate sub-typing.

2 Theory component

A theory component has a name, a list of global type parameters (global to the theory), and an arbitrary number of definitions and rules:

theory *name*
type parameters $T_1, \ldots, T_n$

{ ⟨*Basic Predicate Definition*⟩
| ⟨ *Operator Definition*⟩
| ⟨ *Data Type Definition*⟩
| ⟨ *Rewrite Rule*⟩
| ⟨ *Inference Rule*⟩ }

We look at each form of definition and rule in turn in the following sections. In the following it is important to recall that the mathematical language has two syntactic categories, *expressions* and *predicates*.

3 Defining new basic predicates

A basic predicate is a property on one or more expressions. For example, the predicate x *divides* y holds when x is an integer divisor of y. This predicate is defined in the following way:

predicate *divides*
  infix
  args *x,y*
  condition $x \in \mathbb{N} \ \wedge \ y \in \mathbb{N}$
  definition $\exists a \cdot y = a \times x$

This declares a new basic predicate *divides*. It is declared as infix with two argments x and y. This declaration makes the predicate E *divides* F syntactically valid for integer expressions E and F. The condition specifies a well-definedness condition – in this case that x and y must be naturals. The argument types are inferred from the condition – in this case it is inferred that x and y are of type $\mathbb{Z}$. The final clause provides the definition of x *divides* y. That is, we have

$$x \textit{ divides } y \quad \Longleftrightarrow \quad \exists a \cdot y = a \times x$$

A new basic predicate can be infix or prefix. For example, if *divides* had been declared as prefix, then $\mathit{divides}(E, F)$ would become syntactically valid. An infix predicate must have exactly two arguments.

Though in this case the arguents are typed with the predefined type $\mathbb{Z}$, in general arguments may be typed using some of the type parameters defined for the theory which makes the predicate polymorphic on those type parameters.

The general structure of a basic predicate definition is as follows:

predicate *Identifier*
  (**prefix** | **infix**)
  args $x_1, \ldots, x_n$
  condition $P(x_1, \ldots, x_n)$
  definition $Q(x_1, \ldots, x_n)$

4 Defining new operators

While a basic predicate forms a predicate from a number of expressions, an operator forms an expression from a number of expressions. We consider an example involving the representation of sequences as functions whose domains are contiguous ranges of naturals starting at 1, i.e., functions from $(1..n) \rightarrow T$. The *seq* operator takes a set s and yields all sequences whose members are in s:

operator *seq*
prefix
args s
condition $s \subseteq T$
definition $\{\ f, n\ \cdot\ f \in (1..n) \rightarrow s\ \mid\ f\ \}$

Here *seq* is declared to be a prefix operator with a single argument represented by s. The well-definedness condition declares s to be a subset of a global type parameter T. Since T is a type parameter, this means that *seq* is polymorphic on type T. The final clause defines the expression $seq(T)$ in terms of the existing expression language. The definition means we have that:

$$seq(s)\ =\ \{\ f, n\ \cdot\ f \in (1..n) \rightarrow s\ \mid\ f\ \}$$

Here is an example of another prefix operator *size* that yields the size of a sequence:

operator *size*
prefix
args m
condition $m \in seq(T)$
definition $card(m)$

Proof obligations are generated to verify the well-definedness of definitions. For example, the definition of *size* leads to a proof obligation requiring that $card(m)$ is well-defined whenever $m \in seq(T)$. This is provable from the condition that m is a sequence since any element of $seq(T)$ has a finite domain and $card(m)$ is well-defined when m is finite (in Section 6 this is expressed as in inference rule).

Operators may be infix in which case they may be declared to be associative and commutative. For example, the concatenation operator on sequences, declared as follows, is associative:

operator $\frown$
infix assoc
args m, n
condition $m \in seq(T)\ \wedge\ n \in seq(T)$
definition $m\ \cup\ \{\ i, x\ \cdot\ i \mapsto x \in n \mid size(m) + i \mapsto x\ \}$

The general form of an operator definition is as follows:

operator *Identifier*
(**prefix** | **infix**) [**assoc**] [**commut**]
args $x_1, \ldots, x_n$
condition $P(x_1, \ldots, x_n)$
definition $E(x_1, \ldots, x_n)$

The core mathematical language will be extended with conditional expressions for use in operator definitions. For example, the *max* operator, that yields the maximum of two integers, is defined using a conditional expression as follows:

operator *max*
infix assoc commut // *declare max to be associative and commutative*
type parameters T
args x, y
condition $x \in \mathbb{Z} \ \wedge \ y \in \mathbb{Z}$
definition if $x \geq y$ then x else y

Declaring an operator to be associative and commutative gives rise to proof obligations to verify these properties. Since the Rodin provers automatically make use of commutativity and associativity properties of operators, to avoid circular proofs, the proof obligations must be specified in terms of the operator definition rather than the operator itself, e.g., the above declaration of *max* will give rise to the following commutativity proof obligation:

$$\text{if } x \geq y \text{ then } x \text{ else } y \quad = \quad \text{if } y \geq x \text{ then } y \text{ else } x$$

5 Rewrite Rules

A rewrite rule is used in automatic or interactive proof to rewrite an expression or predicate in order to faciliate proof. A rewrite involves a left hand pattern and one or more right hands. Each right hand may be guarded by some condition. For example, the following rewrite rule defines two ways of rewriting the expression $card(i..j)$ depending on a condition on i and j:

rewrite *CardIntegerRange*
auto manual complete
vars i, j
condition $i \in \mathbb{Z} \ \wedge \ j \in \mathbb{Z}$
lhs $card(i..j)$
rhs

$i \leq j$	$j - i + 1$
$i > j$	0

This declaration allows the rewrite to be deployed by the rule-based prover plug-in. The above declaration means that the rewrite rule can be used in automatic and interactive proof modes. The 'complete' declaration means that the disjunction of the guards must be true. The variables of the rule (i and j) serve as meta variables that can be matched with any expression of the appropriate type. The condition clause is used to type the variables of rule. For details of the proof obligations associated with rewrites and details of how the rule-based prover applies rewrite rules see [2].

The general form of a rewrite rule for expressions is as follows:

rewrite *Name*
[**auto**] [**manual**] [**complete**]
vars $x_1, \ldots, x_n$
condition $P(x_1, \ldots, x_n)$

lhs $E(x_1, \ldots, x_n)$
rhs

$Q_1(x_1, \ldots, x_n)$	$E_1(x_1, \ldots, x_n)$
$\vdots$	$\vdots$
$Q_m(x_1, \ldots, x_n)$	$E_m(x_1, \ldots, x_n)$

The general form for predicates is as follows:

rewrite *Name*
[**auto**] [**manual**] [**complete**]
vars $x_1, \ldots, x_n$
condition $P(x_1, \ldots, x_n)$
lhs $R(x_1, \ldots, x_n)$
rhs

$Q_1(x_1, \ldots, x_n)$	$R_1(x_1, \ldots, x_n)$
$\vdots$	$\vdots$
$Q_m(x_1, \ldots, x_n)$	$R_m(x_1, \ldots, x_n)$

Currently the predicates may not contain predicate variables though this will be addressed in the future.

6 Inference Rules

An inference rule has a list of hypothesis and a consequent. It is parameterised by one or more variables. For example, the following inference rule has two hypotheses and a consequent that may be inferred from the hypotheses:

rule *FiniteSeq*
vars s, m
given
$s \subseteq T$
$m \in seq(s)$
infer
$finite(m)$

The rule declaration gives rise to a soundness proof obligation.

Here is another inference rule showing that sequence concatenation is closed for elements of $seq(s)$:

rule *Concat1*
vars s, m, n
given $s \subseteq T$
$m \in seq(s)$
$n \in seq(s)$
infer $m \frown n \in seq(s)$

The general form of an inference rule is as follows:

rule *Name*
vars $x_1, \ldots, x_n$
given
$P_1(x_1, \ldots, x_n),\ \ldots,\ P_m(x_1, \ldots, x_n)$
infer
$Q(x_1, \ldots, x_n)$

7 Defining new datatypes

A new datatype declaration defines a new type constructor together with constructor and destructor functions for elements of the new type. For example the usual inductive list type constructor is defined as follows:

datatype *List*
type args T
constructors
nil
$cons(\ head : T,\ tail : List(T)\ )$

This defines

- A new type constructor *List*: $List(T)$ becomes a type for any type T.
- A set operator *List*: $List(s)$ is a set expression – the set of lists whose members are in set s
- Two constructors *nil* and *cons*
- Two destructors *head* and *tail*
- An induction principle on *List*

Proof by induction will be supported though a special reasoner that will generate an induction scheme for any particular hypothesis or goal of a proof.

The general form of an inductive data definition is as follows:

datatype *Ident*
type args $T_1 \ldots T_n$
constructors
$c_1(\ d_1^1 : E_1^1,\ \cdots,\ d_1^j : E_1^j\ )$
$\vdots$
$c_m(\ d_m^1 : E_m^1,\ \cdots,\ d_m^k : E_m^k\ )$

Here E_j^i is a type expression that may include occurrences of the type being defined $Ident(T_1 \ldots T_n)$. If E_j^i does include occurrences of $Ident(T_1 \ldots T_n)$, then E_j^i must be *finitary*, i.e., E_j^i is $Ident(T_1 \ldots T_n)$ or is formed from a cartesian product or an existing inductive data type.

8 Pattern matching with datatypes

When defining basic predicates and operators on inductive types, the usual pattern matching may be used. For example the *size* function on inductive lists is defined as follows:

operator *size*
prefix
args a
condition $a \in List(T)$
definition

match a	
nil	0
$cons(x, b)$	$1 + size(b)$

Since a is of type $List(T)$ the argument a may be matched against each of the constructors for $List$.

Here is an example of an operator definition that removes duplicates in a list. It is dfined using a conditional expression:

operator *remdup*
prefix
args a
condition $a \in List(T)$
definition

match a	
nil	nil
$cons(x, b)$	if $member(x, b)$ then $remdup(b)$ else $cons(x, remdup(b))$

General definition: to be done.

9 Fixed Point definitions

The proposals in [1] suggests a functional predicate definition form where an operator is defined as the solution of some predicate. Rather than supporting this more general form initially, we propose a special case, namely a fixed point definition (based on a suggestion by Laurent). For example, transitive closure of a relation may be defined as follows:

operator *tcl*
prefix
type parameters T
args r
condition $r \in T \leftrightarrow T$
fixpoint y **where**
$r \ \cup \ r; y$
order $\{ \ a \mapsto b \mid a \in T \leftrightarrow T \ \wedge \ a \subseteq b \ \}$

This defines $tcl(r)$ to be the least (under the $\subseteq$ ordering) relation satisfying

$$y \quad = \quad r \ \cup \ r; y$$

The fixpoint definition will lead to the following rewrites and inference rules being automatically declared:

rewrite $tcl(r) \quad \rightsquigarrow \quad r \ \cup \ r; tcl(r)$

rewrite $r \ \cup \ r; tcl(r) \quad \rightsquigarrow \quad tcl(r)$

rule
given $r \ \cup \ r; z \quad \subseteq \quad z$
infer $tcl(r) \ \subseteq \ z$

The general definitions is as follows:

operator *Ident*
prefix
type parameters $T_1, \ldots, T_n$
args $x_1, \ldots, x_n$
condition $P(x_1, \ldots, x_n)$
fixpoint y **where**
$E(y, x_1, \ldots, x_n)$
order O

The definition will lead to proof obligations as follows:

monotonicity: $y \mapsto y' \in O \quad \Rightarrow \quad E(y, x_1, \ldots, x_n) \mapsto E(y', x_1, \ldots, x_n) \in O$
partial order: O is a partial order

10 Records and Data Types

Data type definitions support the definition of variant records: the constructors define the separate forms that a record might take while the destructors provide a means of accessing the fields of a record. The essential difference between this and the current records plug-in is that data types are polymorphic, e.g., we may have lists of integers and lists of names co-existing in the same Event-B model, whereas the records as defined by the records plug-in are not polymorphic. Also the datatypes will come with an induction principle in the case of recursive data types. Later we will explore whether a datatype declaration can be constructed automatically by the records plug-in from the existing records syntax (in a way that is syntactically backwards compatible with the existing records plug-in syntax).

References

[1] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Michael Leuschel, Matthias Schmalz, and Laurent Voisin. *Proposals for Mathematical Extensions for Event-B.* http://deploy-eprints.ecs.soton.ac.uk/216/, 2009.

[2] Issam Maamria, Michael Butler, Andrew Edmunds, and Abdolbaghi Rezazadeh. *On an Extensible Rule-based Prover for Event-B*, November 2009. http://eprints.ecs.soton.ac.uk/18273/.