Proposals for Mathematical Extensions for Event-B

J.-R. Abrial, M. Butler, M Schmalz, S. Hallerstede, L. Voisin

9 January 2009

Mathematical Extensions

1 Introduction

In this document we propose an approach to support user-defined extension of the mathematical language and theory of Event-B.

The proposal consists of considering three kinds of extension:

(1) Extensions of set-theoretic expressions or predicates: example extensions of this kind consist of adding the transitive closure of relations or various ordered relations.

(2) Extensions of the library of theorems for predicates and operators.

(2) Extensions of the Set Theory itself through the definition of algebraic types such as lists or ordered trees using new set constructors.

2 Brief Overview of Mathematical Language Structure

A full definition of the mathematical language of Event-B may be found in [1]. Here we give a very brief overview of the structure of the mathematical language to help motivate the remaining sections.

Event-B distinguishes *predicates* and *expressions* as separate syntactic categories. Predicates are defined in term of the usual basic predicates ($\top$, $\bot$, $A = B$, $x \in S$, $y \leq z$, etc), predicate combinators ($\neg$, $\wedge$, $\vee$, etc) and quantifiers ($\forall$, $\exists$). Expressions are defined in terms of constants (0, $\varnothing$, etc), (logical) variables (x, y, etc) and operators ($+$, $\cup$, etc).

Basic predicates have expressions as arguments. For example in the predicate $E \in S$, both E and S are expressions. Expression operators may have expressions as arguments. For example, the set union operator has two expressions as arguments, i.e., $S \cup T$. Expression operators may also have predicates as arguments. For example, set comprehension is defined in terms of a predicate P, i.e., $\{ x \mid P \}$.

2.1 Typing rules

All expressions have a type which is one of three forms:

- a basic set, that is a predefined set ($\mathbb{Z}$ or BOOL) or a carrier set provided by the user (i.e., an identifier);
- a power set of another type, $\mathbb{P}(\alpha)$;
- a cartesian product of two types, $\alpha \times \beta$

This are the types currently built-in to the Rodin tool. In Section 6 we will see a proposal for how new types could be defined by a user. An expression E has a type $\mathbf{type}(E)$ provided E satisfies typing rules. Each expression operator has a typing rule which we write in the form of an inference rule. For example, the following typing rule

for the set union operator specifies that $S \cup T$ has type $\mathbb{P}(\alpha)$ provided both S and T have type $\mathbb{P}(\alpha)$:

$$\frac{\textbf{type}(S) = \mathbb{P}(\alpha), \quad \textbf{type}(T) = \mathbb{P}(\alpha)}{\textbf{type}(S \cup T) = \mathbb{P}(\alpha)}$$

This rule is polymorphic on the type variable α which means that union is a polymorphic operator.

The following table shows examples of typing rules for the intersection and addition operators:

Expression operators	**Type rules**
$S \cap T$	$\dfrac{\textbf{type}(S) = \mathbb{P}(\alpha), \quad \textbf{type}(T) = \mathbb{P}(\alpha)}{\textbf{type}(S \cap T) = \mathbb{P}(\alpha)}$
$E + F$	$\dfrac{\textbf{type}(E) = \mathbb{Z}, \quad \textbf{type}(F) = \mathbb{Z}}{\textbf{type}(E + F) = \mathbb{Z}}$

The arguments of a basic predicate must satisfy typing rules. The following table shows typing rules for set membership and inequality predicates:

Basic Predicates	**Type rules**
$E = F$	$\textbf{type}(E) = \alpha, \quad \textbf{type}(F) = \alpha$
$E \in S$	$\textbf{type}(E) = \alpha, \quad \textbf{type}(S) = \mathbb{P}(\alpha)$
$E \leq F$	$\textbf{type}(E) = \mathbb{Z}, \quad \textbf{type}(F) = \mathbb{Z}$

Note that the rules for $E = F$ and $E \in S$ are polymorphic on the type variable α which means that equality and set membership are polymorphic predicates.

It should be noted that an expression of type *BOOL* is not a predicate. The type *BOOL* consists of the values *TRUE* and *FALSE*, both of which are expressions. These are different to the basic predicates $\top$ and $\bot$. The $bool$ operator is used to convert a predicate into a boolean expression, i.e., $bool(x > y)$. A boolean expression E is converted to a predicate by writing $E =$ *TRUE*. We have that $bool(\top) =$ *TRUE*.

2.2 Function application

It is instructive to consider the relationship between operators and function application in Event-B. An Event-B function $f \in A \pfun B$ is a special case of a set of pairs so the type of f is $\mathbb{P}(\textbf{type}(A) \times \textbf{type}(B))$. The functionality of f is an additional property defined by a predicate specifying a uniqueness condition:

$$\forall x, y, y' \cdot x \mapsto y \in f \ \land \ x \mapsto y' \in f \ \implies \ y = y'$$

The domain of f, written $dom(f)$, is the set $\{ \ x \mid \exists y \cdot x \mapsto y \in f \ \}$. Application of f to x is written $f(x)$ which is well-defined provided $x \in dom(f)$.

It is important to note that f is not itself an operator, it is simply an expression. The operator involved here is implicit – it is the *function application* operator that takes two arguments, f and x. To make the operator explicit, function application could have been written as $apply(f, x)$, where $apply$ is the operator and f and x are the arguments. However, in the Rodin tool, the shorthand $f(x)$ must be used.

Variables in the mathematical language are typed by set expressions. This means, for example, that a variable may represent a function since a function is a special case of a set (of pairs). Variables may not represent expression operators or predicates in the mathematical language. This means that, while we can quantify over sets (including functions), we cannot quantify over operators or predicates.

2.3 Well-definedness

Our consideration of function application just now has referred to the well-definedness of expressions. Along with typing rules as defined above, all expression operators come with well-definedness predicates. We write $\textbf{WD}(E)$ for the well-definedness predicate of expression E. The following table gives examples of well-definedness conditions for several operators, including the function application operator:

Expression operators	**Well-definedness**
$\textbf{WD}(\ F(E)\ )$	$\textbf{WD}(F), \quad \textbf{WD}(E), \quad F \in \alpha \pfun \beta, \quad E \in dom(F)$
$\textbf{WD}(\ E/F\ )$	$\textbf{WD}(E), \quad \textbf{WD}(F), \quad F \neq 0$
$\textbf{WD}(\ card(E)\ )$	$\textbf{WD}(E), \quad finite(E)$
$\textbf{WD}(\ S \cup T\ )$	$\textbf{WD}(S), \quad \textbf{WD}(T)$

Thus, an expression $F(E)$ is well-defined provided both F and E are well-defined, that F is a partial function and that E is in the domain of F. In the Rodin tool, well-

definedness conditions give rise to proof obligations for expressions that appear in models. The well-definedness conditions are themselves written as predicates in the Event-B mathematical language.

3 Specifying Basic Predicates

In Section 2 we saw how basic predicates come with typing rules and how expression operators come with typing rules and well-definedness rules. In the current version of Rodin these rules are embedded in the implementation rather than being defined in a library of rules. Our aim is to migrate towards having rule libraries which can be easily extended by users in order to add new basic predicates and operators. In this section we outline the form that the rules for introducing new basic predicates should take. We will address expression operators in Section 5. We avoid defining a concrete syntax for the form of the rule libraries and focus on the structure.

A basic predicate has a name and a list of arguments and is introduced by rules for typing those arguments. The general form is shown in the following table:

Basic Predicate	**Type rule**
$pred(x_1, \ldots, x_n)$	$\textbf{type}(x_1) = \alpha_1 \quad \cdots \quad \textbf{type}(x_n) = \alpha_n$

Along with the typing rules, the basic predicate is defined in terms of existing predicates as shown in the following table:

Basic Predicate	**Definition**
$pred(x_1, \ldots, x_n)$	$P(x_1, \ldots, x_n)$

Here $P(x_1, \ldots, x_n)$ stands for any predicate term with $x_1, \ldots, x_n$ as free variables. The defining predicate $P(x_1, \ldots, x_n)$ cannot refer to the newly introduced basic predicate $pred$.

As an example, we introduce two basic predicates for symmetry (sym) and asymmetry ($asym$) of relations. We first define the typing rules:

Basic Predicate	Type rule
$sym(R)$	$\textbf{type}(R) \ = \ \alpha \leftrightarrow \beta$
$asym(R)$	$\textbf{type}(R) \ = \ \alpha \leftrightarrow \beta$

We then specify the definitions of these predicates:

Basic Predicate	Definition
$sym(R)$	$R = R^{-1}$
$asym(R)$	$R \cap R^{-1} = \varnothing$

4 Theorem libraries

Having introduced new predicates, such as those in the previous section, we may wish to specify and prove theorems about these. For example, the following theorem shows that union preserves symmetry of relations:

$\textbf{Theorem}\ [\alpha, \beta]$ *thm1*	$\forall R, S \cdot R \in \alpha \leftrightarrow \beta \ \wedge \ S \in \alpha \leftrightarrow \beta \wedge$ $sym(R) \ \wedge \ sym(S) \quad \Longrightarrow \quad sym(R \cup S)$

Since the the basic predicates are polymorphic on the types α and β, the theorem is also polymorphic on these types. The polymorphic type parameters are indicted explicitly in square brackets on the left-hand side. The theorem also has a label (*thm1*) for reference.

The following proof of Theorem *thm1* shows how the definition of sym, introduced in the previous section, is used:

$$
\begin{array}{ll}
 & sym(R) \ \land \ sym(S) \\
\iff & \quad \text{“Definition of } sym\text{”} \\
 & R = R^{-1} \ \land \ S = S^{-1} \\
\implies & \quad \text{“Leibnitz”} \\
 & R \cup S = R^{-1} \cup S^{-1} \\
\iff & \quad \text{“Theorem: } (R \cup S)^{-1} = R^{-1} \cup S^{-1}\text{”} \\
 & R \cup S = (R \cup S)^{-1} \\
\iff & \quad \text{“Definition of } sym\text{”} \\
 & sym(R \cup S)
\end{array}
$$

This proof also illustrates the use of an existing theorem.

Currently many theorems about basic predicates (and expression operators) are directly implemented in the Rodin provers. The aim is to migrate towards libraries of polymorphic theorems. The libraries should be extendable by users. It should be possible both to prove any newly introduced theorem and to use those theorems in proofs. Before being proved, however, newly introduced theorems need to be checked for type correctness. A type checker will need to use the typing rules for newly introduced basic predicates in order to type check theorems involving those predicates. In our example, the typing rule for $sym(R)$ required R to be a polymorphic relation. In the case of Theorem *thm1*, the antecedent is sufficient to check this.

Theory libraries should be organised according to a taxonomy based on structures, e.g., theories about sets, relations, integers, etc.

5 Defining New Operators

In this section we outline the way in which new operators may be introduced. Along with typing and well-definedness rules, operator definitions should be provided. We identify four different forms by which an operator may be defined. We assume an operator has the form $op(x_1, \ldots, x_n)$ where op is the operator name and $x_1, \ldots, x_n$ are expression arguments. We envisage a syntactic layer where, for example, binary operators could be written in infix form with a special symbol representing an operator (as already supported by Rodin for the pre-defined opertors, e.g., $S \cup T$). We will not address this syntactic layer here. Note we require that operators are never overloaded.

The general form of a typing rule for a new operator is as follows:

Expression operator	Type rule
$op(x_1, \ldots, x_n)$	$\dfrac{\textbf{type}(x_1) = \alpha_1, \quad \ldots, \quad \textbf{type}(x_n) = \alpha_n}{\textbf{type}(\ op(x_1, \ldots, x_n)\ ) = \alpha}$

A rule such as this will be used by a type checker for expressions involving op. We have already seen examples of typing rules for operators in Section 2.1.

The general form of a well-definedness rule for a new operator is as follows:

Expression operator	Well-definedness
$\textbf{WD}(\ op(x_1, \ldots, x_n)\ )$	$\textbf{WD}(x_1), \quad \ldots, \quad \textbf{WD}(x_n), \quad P(x_1, \ldots, x_n)$

Well-definedness of $op(x_1, \ldots, x_n)$ depends on the well-definedness of its arguments along with a possible additional condition $P(x_1, \ldots, x_n)$ which is a predicate in the mathematical language. For example, in Section 2.3 we saw that well-definedness of function application $F(E)$ requires that the first argument F is functional and that the second argumnent E is in the domain of F. The additional condition $P(x_1, \ldots, x_n)$ must itself be well-defined:

$$\textbf{WD}(x_1), \ldots, \textbf{WD}(x_n) \quad \Longrightarrow \quad \textbf{WD}(\ P(x_1, \ldots, x_n)\ )$$

Many operators have no additional condition, i.e, their well-definedness depends only on the well-definedness of their arguments.

As described for basic predicates in Section 4, we may specify theorems about polymorphic operators with the intention that these would be added to an appropriate theory library. Proofs of these theorems would rely on the operator definitions. We now consider the four different forms that definitions may take.

5.1 Direct Definition

A *direct definition* defines $op(x_1, \ldots, x_n)$ directly in terms of an expression E. It has the following form:

Expression operator	Direct definition
$op(x_1, \ldots, x_n)$	$op(x_1, \ldots, x_n) \ \widehat{=}\ E(x_1, \ldots, x_n)$

The defining expression E should not refer to the newly defined operator *op*. In Section 6 we will introduce a recursive form of definition. The type of the defining expression should be the same as the declared type of the operator:

$$\textbf{type}(\ E(x_1, \ldots, x_n)\ ) \quad = \quad \textbf{type}(\ op(x_1, \ldots, x_n)\ )$$

Furthermore, the defining expression should itself be well-defined under the well-definedness conditions of the operator:

$$\textbf{WD}(\ op(x_1, \ldots, x_n)\ ) \quad \Longrightarrow \quad \textbf{WD}(\ E(x_1, \ldots, x_n)\ )$$

For example, let us introduce the symmetric difference operator on sets, *symdiff*. The typing rule is as follows:

Expression operator	**Type rule**
symdiff(S, T)	$\dfrac{\textbf{type}(S) = \mathbb{P}(\alpha), \quad \textbf{type}(T) = \mathbb{P}(\alpha)}{\textbf{type}(\ \textit{symdiff}(S, T)\ ) = \mathbb{P}(\alpha)}$

Well-definedness has no additional condition:

Expression operator	**Well-definedness**
WD(*symdiff*(S, T))	**WD**(S), **WD**(T)

The operator is defined directly in terms of subtraction and union:

Expression operator	**Direct definition**
symdiff(S, T)	*symdiff*$(S, T) \ \widehat{=} \ (S \setminus T) \cup (T \setminus S)$

5.2 Conditional Direct Definition

A *conditional direct definition* defines $op(x_1, \ldots, x_n)$ with several distinct cases and each case is guarded by a predicate. It has the following form:

Expression operator	**Conditional direct definition**
$op(x_1, \ldots, x_n)$	$op(x_1, \ldots, x_n) \;\widehat{=}$ **case** $C_1 : E_1(x_1, \ldots, x_n)$ $\ldots$ **case** $C_m : E_m(x_1, \ldots, x_n)$

Each defining expression E_i should not refer to the newly defined operator op. The type of each defining expression should be the same as the declared type of the operator (each $i \in 1..m$):

$$\textbf{type}(\ E_i(x_1, \ldots, x_n)\ ) \quad = \quad \textbf{type}(\ op(x_1, \ldots, x_n)\ )$$

Each guard and each defining expression should be well-defined as follows (each $i \in 1..m$):

$$\begin{array}{rcl} \textbf{WD}(\ op(x_1, \ldots, x_n)\ ) & \Longrightarrow & \textbf{WD}(\ C_i\ ) \\ \textbf{WD}(\ op(x_1, \ldots, x_n)\ ) \ \wedge\ C_i & \Longrightarrow & \textbf{WD}(\ E_i(x_1, \ldots, x_n)\ ) \end{array}$$

Furthermore, the case guards should be pairwise distinct and should cover the well-definedness condition of the operator:

$$\begin{array}{rcll} C_i \wedge C_j & \Longrightarrow & \bot, & \text{each } i, j \in 1..m,\ i \neq j \\ \textbf{WD}(\ op(x_1, \ldots, x_n)\ ) & \Longrightarrow & C_1 \vee \cdots \vee C_m & \end{array}$$

For example, let us introduce the max operator on two integers. The typing rule is as follows:

Expression operator	**Type rule**
$max(x, y)$	$\dfrac{\textbf{type}(x) = \mathbb{Z}, \quad \textbf{type}(y) = \mathbb{Z}}{\textbf{type}(\ max(x, y)\ ) = \mathbb{Z}}$

Well-definedness has no additional condition:

Expression operator	**Well-definedness**
$\textbf{WD}(\ max(x, y)\ )$	$\textbf{WD}(x), \quad \textbf{WD}(y)$

The operator is defined in terms of two cases:

Expression operator	Conditional direct definition
$max(x, y)$	$max(x, y) \;\widehat{=}$ **case** $x \geq y$: x **case** $x < y$: y

5.3 Extensional Definition

When introducing a set operator, it can be convenient to define it in terms of the membership conditions for the resulting set. Such an *extensional definition* takes the following form:

Expression operator	Extensional definition
$op(x_1, \ldots, x_n)$	$y \in op(x_1, \ldots, x_n) \iff P(y, x_1, \ldots, x_n)$

Here $P(y, x_1, \ldots, x_n)$ is a predicate term with $y, x_1, \ldots, x_n$ as free variables. The extensional form can only be used when the type of the operator is $\mathbb{P}(\alpha)$.

$P(y, x_1, \ldots, x_n)$ cannot refer to the newly defined operator *op* and should be well-typed under the typing rules for operator arguments. The defining predicate should be well-defined under the well-definedness condition for the operator:

$$\textbf{WD}(\ op(x_1, \ldots, x_n)\ ) \implies \textbf{WD}(\ P(y, x_1, \ldots, x_n)\ )$$

For example, set union is defined in terms of disjunction in this way:

Expression operator	Extensional definition
$union(S, T)$	$y \in union(S, T) \iff y \in S \ \vee\ y \in T$

5.4 Functional Predicate Definition

The *functional predicate definition* has a similar shape to the extensional form of definition except that instead of defining a condition for when $y \in op(x_1, \ldots, x_n)$ we define a condition for when $y = op(x_1, \ldots, x_n)$. It takes the following form:

Expression operator	Functional predicate definition
$op(x_1, \ldots, x_n)$	$y = op(x_1, \ldots, x_n) \iff P(y, x_1, \ldots, x_n)$

The defining predicate must be type consistent and well-defined as for the extensional form of definition. While the extensional form was only applicable when the type of op is $\mathbb{P}(\alpha)$, the functional predicate definition is applicable for any type.

We need to ensure that the definition is consistent, i.e., that a value for y exists. We also need to ensure that the value for y is unique, hence the name 'functional predicate'. Consistency is shown by associating the following proof obligation with the functional predicate form:

$$\mathbf{WD}(\ op(x_1, \ldots, x_n)\ ) \implies \exists y \cdot P(y, x_1, \ldots, x_n)$$

Uniqueness is shown by the following proof obligation:

$$\mathbf{WD}(\ op(x_1, \ldots, x_n)\ ) \ \wedge\ P(y, x_1, \ldots, x_n) \ \wedge\ P(y', x_1, \ldots, x_n) \implies y = y'$$

For example, function application is defined using a functional predicate definition:

Expression operator	Functional predicate definition
$apply(f, x)$	$y = apply(f, x) \iff x \mapsto y \in f$

Here, we have made the application operator explicit for clarity. The consistency and uniqueness proof obligations can be discharged directly from the well-definedness condition for function application:

$$f \in \alpha \nrightarrow \beta \ \wedge\ x \in dom(f) \implies \exists y \cdot x \mapsto y \in f$$
$$f \in \alpha \nrightarrow \beta \ \wedge\ x \in dom(f) \ \wedge\ x \mapsto y \in f \ \wedge\ x \mapsto y' \in f \implies y = y'$$

5.5 Axiomatic Definition

In some cases it is pragmatic not to define an operator using one of the other determinate forms. Instead we define some axioms expressing properties that we require an operator or set of operators to satisfy. In this case we provide typing and well-definedness rules as before. The axiomatic definition then takes the following form:

Expression operators	Axiomatic definition
$op_1(x_1^1, \ldots)$ $\vdots$ $op_m(x_1^m, \ldots)$	$\text{label}_1: \quad P_1(\ x_1^1, \ldots, x_1^m, \ldots\ )$ $\vdots$ $\text{label}_k: \quad P_k(\ x_1^1, \ldots, x_1^m, \ldots\ )$

The defining axioms must be type consistent and well-defined under the well-definedness condition for the operator. In the case of an axiomatic definition we do not attempt to prove uniqueness or completeness of the definitions. We just accept the axioms we provide. The guideline is that wherever possible we should use the other forms when introducing an operator. We resort to an axiomatic definition when it is too inconvenient to do otherwise.

As an example of an axiomatic definition, consider the addition and multiplication operators on integers. They are typed as follows:

Expression operator	Type rules
$plus(x, y)$	$\dfrac{\textbf{type}(x) = \mathbb{Z}, \quad \textbf{type}(y) = \mathbb{Z}}{\textbf{type}(\ plus(x, y)\ ) = \mathbb{Z}}$
$mult(x, y)$	$\dfrac{\textbf{type}(x) = \mathbb{Z}, \quad \textbf{type}(y) = \mathbb{Z}}{\textbf{type}(\ mult(x, y)\ ) = \mathbb{Z}}$

Here we present distribution axioms for these operators that are intended to be illustrative rather than in any way comprehensive:

Expression operators	Axiomatic definition
$plus(x, y)$ $mult(x, y)$	ax1: $\quad plus(x, mult(y, z)) \ = \ mult(plus(x, y), plus(x, z))$ ax2: $\quad mult(x, plus(y, z)) \ = \ plus(mult(x, y), mult(x, z))$

6 Extending Set Theory with Algebraic Types

Before proposing an extension mechanism to it (section 6.2), let us review how our current Set Theory has been built (section 6.1).

6.1 A Review of the Basic Set-theoretic Constructs

The Set Theory used in Event-B is based on two constructors: the set comprehension constructor and the pair constructor.

The set comprehension constructor, $\{ \mid \}$, constructs a set from a predicate. The pairing constructor, $\mapsto$, constructs a pair from two expressions. The set of all possible set comprehensions is denoted by means of the power set operator $\mathbb{P}$ and the set of all pairs is denoted by means of the cartesian product operator $\times$. The two operators $\mathbb{P}$ and $\times$ are operators in the mathematical language so they can be used to form set expresions, e.g., $\mathbb{P}(1..10)$. These operators also have a special status in that they are lifted to form type operators.

The power set $\mathbb{P}$ and set comprehension operators $\{ \mid \}$ are linked in following way: $\mathbb{P}$ can be used to construct a new type while $\{ \mid \}$ can be used to construct elements of that new type. Similarly for the cartesian product and pairing operators.

Associated with the constructors are some destructors. The set comprehension constructor is associated with the set destruction operator $\in$ transforming a set into a predicate. Likewise the pair constructing operator is associated with the two destructors prj_1 and prj_2.

This can be extended to the set of integers $\mathbb{Z}$ were the constructor is succ the destructor is pred and the corresponding set and type are $\mathbb{Z}$.

All this is summarized in the following table:

Constructor	**Destructor**	**Set operator**	**Type operator**
$\{ \mid \}$	$\in$	$\mathbb{P}$	$\mathbb{P}$
$\mapsto$	$\mathrm{prj}_1,\ \mathrm{prj}_2$	$\times$	$\times$
succ	pred	$\mathbb{Z}$	$\mathbb{Z}$

Constructors and destructors relationship are made more explicit in the following table:

Component	**Construction**	**Destruction**
$P(x)$	$\{x \mid P(x)\}$	$x \in \{x \mid P(x)\} \;\Leftrightarrow\; P(x)$
$o1,\ o2$	$o1 \mapsto o2$	$\mathrm{prj}_1(o1 \mapsto o2) = o1$ $\mathrm{prj}_2(o1 \mapsto o2) = o2$
n	$\mathrm{succ}(n)$	$\mathrm{pred}(\mathrm{succ}(n)) = n$

The following tables show the typing rules for the type construction operators and the corrsponding element construction operators:

Expression operator	**Type rules**
$\mathbb{P}(S)$	$\dfrac{\textbf{type}(S) = \mathbb{P}(\alpha)}{\textbf{type}(\ \mathbb{P}(S)\ )\ =\ \mathbb{P}(\mathbb{P}(\alpha))}$
$S \times T$	$\dfrac{\textbf{type}(S) = \mathbb{P}(\alpha),\quad \textbf{type}(T) = \mathbb{P}(\beta)}{\textbf{type}(\ S \times T\ )\ =\ \mathbb{P}(\alpha \times \beta)}$
$\mathbb{Z}$	$\textbf{type}(\ \mathbb{Z}\ )\ =\ \mathbb{P}(\mathbb{Z})$

Expression operator	Type rules
$\{x \mid P(x)\}$	$\dfrac{P(x) \implies \textbf{type}(x) = \alpha}{\textbf{type}(\ \{x \mid P(x)\}\ )\ =\ \mathbb{P}(\alpha)}$
$o1 \mapsto o2$	$\dfrac{\textbf{type}(o1) = \alpha,\quad \textbf{type}(o2) = \beta}{\textbf{type}(o1 \mapsto o2)\ =\ \alpha \times \beta}$
$succ(n)$	$\dfrac{\textbf{type}(n) = \mathbb{Z}}{\textbf{type}(\ succ(n)\ )\ =\ \mathbb{Z}}$

Extensionality relates equality of constructions to equality (or equivalence) of corresponding destructions. It is summarized in the following table. It essentially says that constructions as well as destructions are injective.

Equality of Construction	Equality of Destruction
$\{x \mid P(x)\} = \{x \mid Q(x)\}$	$P(x) \Leftrightarrow Q(x)$
$o1 \mapsto o2\ =\ p1 \mapsto p2$	$o1 = p1\ \wedge\ o2 = p2$
$\mathrm{succ}(n) = \mathrm{succ}(m)$	$n = m$

6.2 Defining new Algebraic Types

The Set Theory is extended with a new algebraic type theory in a straightforward fashion by defining a type construction operator (such as $\times$) and one or more element construction operators (such as $\mapsto$). The constructors for a new type may take arguments of that same type thus yielding an inductive type. For example the $cons(x, t)$ constructor for the inductive list type takes a list t as an argument. The inductive list type also has nil as a constructor and nil and $cons(x, t)$ are distinct. In the case of an inductive type, induction axioms can be provided.

Stated more explicitly, defining a new algebraic type (such as inductive lists) requires the following:

1. a single type construction operator that can be used both as a set expression operator and a type operators (like $\times$),
2. several new element constructors (like $\mapsto$),
3. extensionality axioms ensuring that constructed elements are uniquely determined by their constituents,
4. disjointness axioms ensuring that distinct constructors yield distinct elements,
5. induction axioms in the case of inductive types.

It is convenient (and conventional) to use a syntactic sugar to group the constructors together into a single definition of a new type. We illustrate such a syntactic sugar for lists and binary trees:

$$\begin{array}{rcl} list(\alpha) & ::= & nil \\ & | & cons(\ \alpha,\ list(\alpha)\ ) \end{array}$$

$$\begin{array}{rcl} tree(\alpha) & ::= & empty \\ & | & node(\ tree(\alpha),\ \alpha,\ tree(\alpha)\ ) \end{array}$$

The sugared definition for lists gives rise to three operators ($list$, nil and $cons$) and the sugared definition for trees also gives rise to three operators ($tree$, $empty$ and $node$). The $list$ and $tree$ operators can be used as type constructors as well as set operators. The nil and $cons$ operators are used as constructors for elements of type $list$, while the $empty$ and $node$ operators are used as constructors for elements of type $tree$.

The syntactic sugar gives rise to the following typing rules for these operators:

Expression operator	**Type rules**
$list(S)$	$\dfrac{\textbf{type}(S) = \mathbb{P}(\alpha)}{\textbf{type}(\ list(S)\ )\ =\ \mathbb{P}(list(\alpha))}$
$tree(S)$	$\dfrac{\textbf{type}(S) = \mathbb{P}(\alpha)}{\textbf{type}(\ tree(S)\ )\ =\ \mathbb{P}(tree(\alpha))}$

Expression operator	Type rules
nil	$\textbf{type}(\ nil_\alpha\ ) \ = \ list(\alpha)$
$cons(x, l)$	$\dfrac{\textbf{type}(x) = \alpha, \quad \textbf{type}(l) = list(\alpha)}{\textbf{type}(\ cons(x,l)\ ) \ = \ list(\alpha)}$
$empty$	$\textbf{type}(\ empty_\alpha\ ) \ = \ tree(\alpha)$
$node(t_1, x, t_2)$	$\dfrac{\textbf{type}(t_1) = tree(\alpha), \quad \textbf{type}(x) = \alpha, \quad \textbf{type}(t_2) = tree(\alpha)}{\textbf{type}(\ node(t_1,x,t_2)\ ) \ = \ tree(\alpha)}$

In the case of lists, $cons$ is an *inductive* constructor since it takes an argument of type $list(\alpha)$, while nil is a *base* constructor since it does not take an argument of type $list(\alpha)$. Similary $empty$ is a base constructor for trees and $node$ is an inductive constructor for trees.

The type and element constructors are well-defined for all well-defined arguments so have no additional well-definedness conditions. This is shown for the list constructors in the following well-definedness table:

Expression operator	Well-definedness
$\textbf{WD}(\ list(S)\ )$	$\textbf{WD}(S)$
$\textbf{WD}(\ nil\ )$	$\top$
$\textbf{WD}(\ cons(x,l)\ )$	$\textbf{WD}(x), \ \ \textbf{WD}(l)$

The syntactic sugar also gives rise to extensionality, distinctness and induction arguments. These are shown for lists in the following table of axioms:

	Axioms
$Extensionality$	$con(x,l) = cons(x',l') \implies x = x' \wedge l = l'$
$Distinctness$	$nil \neq cons(x,l)$
$Induction$	$P(nil) \wedge$ $(\ \forall x, l \cdot P(l) \implies P(cons(x,l))\ )$ $\implies (\ \forall l \cdot P(l)\ )$

A definition of a new algebraic type must contain at least one base constructor. It does not need to contain any inductive constructors. Examples of non-inductive algebraic types include a type constructor $sumtype$ that forms the discriminated union of two types and an enumerated type $direction$ that contains exactly four distinct values:

$$\begin{array}{rcl} sumtype(\alpha,\beta) & ::= & inj_1(\ \alpha\ ) \\ & | & inj_2(\ \beta\ ) \end{array}$$

$$\begin{array}{rcl} direction & ::= & north \\ & | & south \\ & | & east \\ & | & west \end{array}$$

6.3 Pattern-based Recursive Definitions

In Section 5, five different forms of operator definition were presented. The algebraic construction of types allows us to add a pattern-based recursive form of definition for operators where one or more arguments is an algebraic type. This is a special case of the conditional direct definition where each constructor of an algebraic type gives rise to a case and the constructor is used as a pattern. We illustrate a pattern-based recursive definition with the examples of $size$ and ap operators for lists:

Expression operator	Recursive direct definition
$size(l)$	$size(\ nil\ ) \quad \widehat{=} \quad 0$ $size(\ cons(x, l)\ ) \quad \widehat{=} \quad 1 + size(l)$
$map(f, l)$	$map(\ f,\ nil\ ) \quad \widehat{=} \quad nil$ $map(\ f, cons(x, l)\ ) \quad \widehat{=} \quad cons(\ f(x),\ map(f, l)\ )$

Expression operator	Type rules
$size(l)$	$\dfrac{\textbf{type}(l) = list(\alpha)}{\textbf{type}(\ size(l)\ ) = \mathbb{Z}}$
$map(f, l)$	$\dfrac{\textbf{type}(f) = \alpha \leftrightarrow \beta, \quad \textbf{type}(l) = list(\alpha)}{\textbf{type}(\ map(f, l)\ ) = list(\beta)}$

Expression operator	Well-definedness
WD$(\ size(l)\ )$	**WD**(l)
WD$(\ map(f, l)\ )$	**WD**$(f),\ $ **WD**$(l),\ \ f \in \alpha \nrightarrow \beta,\ \ l \in list(dom(f))$

6.4 Type Constructors as Set Operators

We have already stated that type constructors (such as $\mathbb{P}$, $\times$, $list$, $tree$) are also set expression operators. We have already seen the typing rules for these operators and we stated that their well-definedness depends only on the well-definedness of their arguments. We also need to provide a set theory definition of these operators. The power set and cartesian product operators are defined extensionally in the following table:

Expression operator	Extensional definition
$\mathbb{P}(S)$	$T \in \mathbb{P}(S) \iff T \subseteq S$
$S \times T$	$o_1 \mapsto o_2 \in S \times T \iff o_1 \in S \ \wedge \ o_2 \in T$

We use an extensional variant of the recursive definition form to define the inductive type constructors as follows:

Expression operator	Recursive extensional definition
$list(S)$	$nil \in list(S)$ $cons(x,t) \in list(S) \iff x \in S \wedge t \in list(S)$
$tree(S)$	$empty \in tree(S)$ $node(t_1 x, t_2) \in tree(S) \iff x \in S \wedge t_1 \in tree(S) \wedge t_2 \in tree(S)$

Monotonicity of the powerset and cartesian product operators follows directly from properties of sets:

$$\begin{aligned} S \subseteq S' &\implies \mathbb{P}(S) \subseteq \mathbb{P}(S') \\ S \subseteq S' \ \wedge \ T \subseteq T' &\implies S \times T \subseteq S' \times T' \end{aligned}$$

Monotonicity of the inductive type consructors is proved using the induction axioms for those types:

$$\begin{aligned} S \subseteq S' &\implies list(S) \subseteq list(S') \\ S \subseteq S' &\implies tree(S) \subseteq tree(S') \end{aligned}$$

For example, consider the inductive case in the proof for lists:

$$\begin{array}{ll} & cons(x, l) \in list(S) \\ \iff & \quad \text{“Definition of } list(S)\text{”} \\ & x \in S \ \wedge \ l \in list(S) \\ \implies & \quad \text{“ } S \subseteq S' \text{ ”} \\ & x \in S' \ \wedge \ l \in list(S) \\ \implies & \quad \text{“Induction hypothesis”} \\ & x \in S' \ \wedge \ l \in list(S') \\ \iff & \quad \text{“Definition of } list(S)\text{”} \\ & cons(x, l) \in list(S') \end{array}$$

7 Further considerations

We have shown how n-ary basic predicates and n-ary operators may be added to the language. The ability to add binders should also be explored. For example, it should be possible to add a summation binder such as

$$SUM(x).(\ P(x) \mid E(x)\ )$$

where P is a predicate constraining the values of bound variable x and E is an expression.

We have avoided making the syntax of predicate and operator delcarations precise. This needs to be explored further, including an exploration of succinct syntactic sugars. We already saw and example of a syntactic sugar for algebraic type declarations in Section 6.2. This is very similar to the approach taken in PVS [2]. PVS includes a syntactic sugar for destructors for algebraic types. For example, the head and tail destructors for lists are included as follows:

$$\begin{array}{lcl} list(\alpha) & ::= & nil \\ & \mid & cons(\ head : \alpha,\ tail : list(\alpha)\ ) \end{array}$$

This syntax is simultaneously defining five operators ($list$, nil, $cons$, $head$, $tail$).

The typing rules for operators may also benefit from a syntactic sugar such as the following:

$$\textit{symdiff}(\ \mathbb{P}(\alpha),\ \mathbb{P}(\alpha)\ )\ :\ \mathbb{P}(\alpha)$$

in place of

$$\frac{\textbf{type}(S) = \mathbb{P}(\alpha), \quad \textbf{type}(T) = \mathbb{P}(\alpha)}{\textbf{type}(\ \textit{symdiff}(S, T)\ ) = \mathbb{P}(\alpha)}$$

The unconstrained axiomatic form of operator definition can results on inconsistent definitions. A consitency proof obligation for axiomatic definitions coould be enforced but in many cases that may be difficult to prove. More schematic forms of axiomatic definition, such as the scheme for defining algebraic types, should be explored that may help to avoid incontencies.

References

[1] Jean-Raymond Abrial, Christophe Metayer, and Laurent Voisin. Rodin manual and language definition. http://deploy-eprints.ecs.soton.ac.uk/11/, 2007.

[2] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.