Proposals for Mathematical Extensions for Event-B

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

21 April 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 rule library for predicates and operators.

(3) 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 [2]. 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 7 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
$sym(R)$	**type**$(R) \ = \ \alpha \leftrightarrow \beta$
$asym(R)$	**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 Rule libraries derived from axioms and theorems

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:

Theorem $[\alpha, \beta]$ *thm1*	$\forall R, S \cdot R \in \alpha \leftrightarrow \beta \ \wedge \ S \in \alpha \leftrightarrow \beta \wedge$ $sym(R) \ \wedge \ sym(S) \implies sym(R \cup S)$

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 rules derived from 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 rule form

$$\frac{sym(R), \quad sym(S)}{sym(R \cup S)}$$

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.

This approach works in the same way for theorems as it does for axioms [**?**, **?**]. For definitions more automation is possible as described below.

Rule 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 7 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:

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

6 Axiomatic Extension

In some cases it is pragmatic not to define an operator using one of the methods of Section 5. Instead we postulate 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 extension then takes the following form:

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

The axioms must be type consistent and well-defined under the well-definedness conditions for the introduced operators and those already existing. Except for well-definedness nothing is to be proved about the axioms. As usual, the risk of using axiomatic extension is that it can make a theory inconsistent. The guideline is that wherever possible we should use a definition when introducing an operator as this preserves consistency of the logic. We resort to an axiomatic definition when it is too inconvenient to do otherwise.

As an example of an axiomatic extension, 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 : $\quad mult(x, plus(y, z)) \ = \ plus(mult(x, y), mult(x, z))$

6.1 The Axiomatic Basis of Event-B

Event-B is based on a typed polymorphic first-order logic (TPFL) that includes an axiomatisation for a typed set theory and number theory. These axioms are added by axiomatic extension to the TPFL. For set theory this includes axioms for the empty set, extensionality, union, power set, separation, and infinity. For number theory this is essentially an axiomatisation of integer numbers plus natural number induction.

These axioms cover the existence and uniqueness of the terms defined in the core of Event-B.

7 Extending Set Theory with Algebraic Types

Before proposing an extension mechanism to it (section 7.2), let us review how our current Set Theory has been built (section 7.1).

7.1 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
$\{x \mid P(x)\}$	$\dfrac{P(x)\ \Longrightarrow\ \textbf{type}(x) = \alpha}{\textbf{type}(\ \{x \mid P(x)\}\ )\ =\ \mathbb{P}(\alpha)}$
$o1 \mapsto o2$	$\dfrac{\textbf{type}(o1) = \alpha,\quad \textbf{type}(o2) = \beta}{\textbf{type}(o1 \mapsto o2)\ =\ \alpha \times \beta}$
$succ(n)$	$\dfrac{\textbf{type}(n) = \mathbb{Z}}{\textbf{type}(\ succ(n)\ )\ =\ \mathbb{Z}}$

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**
$\{x\|P(x)\} = \{x\|Q(x)\}$	$P(x) \Leftrightarrow Q(x)$
$o1 \mapsto o2 \ = \ p1 \mapsto p2$	$o1 = p1 \ \wedge \ o2 = p2$
$\mathrm{succ}(n) = \mathrm{succ}(m)$	$n = m$

7.2 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**
$list(S)$	$$\frac{\textbf{type}(S) = \mathbb{P}(\alpha)}{\textbf{type}(\ list(S)\ )\ =\ \mathbb{P}(list(\alpha))}$$
$tree(S)$	$$\frac{\textbf{type}(S) = \mathbb{P}(\alpha)}{\textbf{type}(\ tree(S)\ )\ =\ \mathbb{P}(tree(\alpha))}$$

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

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**
WD($list(S)$)	**WD**(S)
WD(nil)	$\top$
WD($cons(x, l)$)	**WD**(x), **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:

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

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

7.3 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**
$size(l)$	$size(\ nil\ ) \ \widehat{=}\ 0$ $size(\ cons(x,l)\ ) \ \widehat{=}\ 1 + size(l)$
$map(f,l)$	$map(\ f,\ nil\ ) \ \widehat{=}\ nil$ $map(\ f, cons(x,l)\ ) \ \widehat{=}\ cons(\ f(x),\ map(f,l)\ )$

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), $f \in \alpha \nrightarrow \beta$, $l \in list(dom(f))$

7.4 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 constructors is proved using the induction schemas 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:

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

8 Binders

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

$$\mathit{SUM}\, x \cdot P(x) \mid E(x)$$

where P is a predicate constraining the values of bound variable x and E is an expression. A common technique for doing this uses the lambda notation

$$Sum(\lambda\, x \ \cdot \ P(x) \mid E(x))$$

Then a binder *SUM* could be introduced so that the first formula is interpreted as the second. The lambda notation of the Event-B mathematical notation would already suggest the syntax to be used, in general:

$$\mathit{BB}\, x \ \cdot \ P \mid E$$

where *BB* is the introduce binder. It would also be possible to base a binder on set comprehension. But it must be based on a binding concept already present in the notation so that introducing a binder is only syntactic sugar to improve readability. The syntax could also be different than that of lambda, but it should be fixed. Note that sum and *SUM* are operators in the above. They cannot be set-theoretic concepts because they introduce new syntax to the logic.

9 Further considerations

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 7.2. This is very similar to the approach taken in PVS [7]. PVS includes

a syntactic sugar for destructors for algebraic types. For example, the head and tail destructors for lists are included as follows:

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

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:

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

in place of

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

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.

10 Record Types

10.1 Field access syntax

A second syntax for function application could make record types easier to read:

$$x.f \quad = \quad f(x)$$

10.2 Explicit Record Types

see VDM

10.3 Algebraic Record Types

see Butler/Evans

10.4 Free Types as Record Types

A simple way to introduce record types without extending the notation is to use free type like record types based on the idea of destructors of Section **??**.

$$\begin{array}{rcl} msg(A) & ::= & empty \\ & | & req(\ code : A\ ) \\ & | & body(\ code : A\ ,\ block : BLOCK\ ) \\ & | & ack(\ code : A\ ) \end{array}$$

This approach would permit easily to deal with related record types. It gives without extra effort the sum type of several record types. Also recursion is already catered for. The different subsets of the sum type could be conveniently marked to permit lightweight use in formulas (e.g., $x \in Body$ would determine that x fits the schema $body(c, b)$ for some c and b):

$$\begin{array}{rcl} msg(A) & ::= & empty \\ & | & Req : req(\ code : A\ ) \\ & | & Body : body(\ code : A\ ,\ block : BLOCK\ ) \\ & | & Ack : ack(\ code : A\ ) \end{array}$$

Assignment to record types would be done in the following way (let $x \in Req$ and $bb \in BLOCK$):

$$x := body(x.code, bb)$$

Note that the parameter A in the declaration of msg can be any set. (The corresponding type is constructed using the type $\mathbb{P}(\alpha)$ of A. This is the simplest way to get a restriction on the values of a type. More restrictions could be added by defining subsets of msg. This holds also for recursive types. The following example is adapted from [**?**]:

$$\begin{array}{rcl} Tree & ::= & Leaf : leaf(\ \mathbb{N}\ ) \\ & | & Node : node(\ lt : Tree\ ,\ mv : \mathbb{N}\ ,\ rt : Tree\ ) \end{array}$$

Define a function $values \in Tree \to \mathbb{P}(\mathbb{N})$ by

$$\begin{array}{rcl} values(leaf(n)) & = & \{n\} \\ values(node(tl, n, tr)) & = & values(tl) \cup \{n\} \cup values(tr) \end{array}$$

Define $Ordered\_Tree$ by:

$$\begin{array}{rcl} Ordered\_Tree & = & Leaf \cup \\ & & \{node(tl, n, tr) | \\ & & \quad (\forall lv \cdot lv \in values(tl) \Rightarrow lv < mv) \wedge \\ & & \quad (\forall rv \cdot rv \in values(tr) \Rightarrow lv > mv)\} \end{array}$$

From the structure of the definition of set $Ordered\_Tree$ one could derive a suitable induction scheme to prove membership of a tree t in $Ordered\_Tree$.

Define a second subset $Balanced\_Tree$ by

$$\begin{array}{rcl} Balanced\_Tree & = & Leaf \cup \\ & & \{node(tl, n, tr) | \\ & & \quad height(tr) - height(tl) \in \{-1, 0, 1\}\} \end{array}$$

where $height \in Tree \rightarrow \mathbb{N}$ is defined by:

$$\begin{array}{rcl} height(leaf(n)) & = & 0 \\ height(node(tl, n, tr)) & = & 1 + max(height(tl), height(tr)) \end{array}$$

Balanced ordered Trees are characterised by the intersection

$$Ordered\_Tree \cap Balanced\_Tree$$

10.5 On Predicate Restrictions

todo: compare how the three variants would address this

11 Polymorphism

The standard way to extend first-order logic with typing and polymorphism is *polymorphic many sorted first order logic* (e.g., [4] or the appendix of [5]) also just called many sorted first order logic by mathematicians. It is the foundation of logic programming languages Gödel [5] and Mercury [9], and quite similar to the core of the Haskell type system.

Note that polymorphic many sorted first order logic is *not* a higher-order extension of FOL. It is actually a restriction of FOL, only accepting well-typed formulas. The model theoretic semantics is very similar to FOL, it just respects the type signatures. The inference rules from FOL are taken over.

Event-B deviates from this standard way by:

1. not giving a type signature to predicates
2. requiring that the type inference algorithm produces ground types for all expressions.

The first point probably has little implications, as implicitly the type inference algorithm is given the types of the built-in predicates (e.g., $POW(\alpha) \times POW(\alpha)$ for $\subseteq$) and the type inference will actually compute the concrete types for the various predicate calls.[1]

The motivation for the second point are (also) unclear to the authors. For the moment this aspect is hard-wired into the Rodin platform. Its purpose was probably to simplify the implementation of a typed abstract syntax tree in an imperative language such as Java, as no type variables need to be stored.

The drawbacks, however, are more numerous:

1. use of a non-standard formalism;
2. loss of referential transparency for users and refactoring tools, as the substitutivity of equals for equals does not hold in the untyped source language;

[1] Given the fact that Rodin accepts, e.g., the expression $\{x \mapsto y \mid x \subseteq y \wedge y \subseteq x\} = \{v \mapsto w \mid v \subseteq BOOL \wedge v = w\}$, it would seem that the tool actually does compute the types for the various predicates.

3. a more complicated formalisation of the type inference;

4. no polymorphic theorems.

Point 4 makes mathematical extension more cumbersome. We elaborate on the points 2–4 below.

11.1 Loss of Referential Transparency

In Haskell the empty list `[]` is a polymorphic constructor; it can be used as the base case for lists of integers or lists of any other datatype. In that respect it is similar to Event-B's empty set, which can be used as the base case for sets of integers or any other B datatype. In contrast to Haskell, however, Event-B does *not* allow the expression $\varnothing = \varnothing$. The reason is that the type inference cannot determine a ground type for the two appearances of $\varnothing$ in the predicate. Contrast this with a transcript of a Haskell session:

```
 $ ghci
GHCi, version 6.10.1: http://www.haskell.org/ghc/  :? for help
...
Prelude> let c = if []==[] then 1 else 0
Prelude> c
1
```

Note that this restriction of Event-B can also be found in Atelier-B.[2]

The ramifications of this restriction is that we, e.g., cannot substitute $x = \varnothing \wedge y = \varnothing$ in the predicate $x = y$ as the result is an incorrectly typed (according to Event-B) predicate expression.

In the internal "machinery" of Rodin this is not (or less of)[3] a problem: here all values are eplicitly typed. But for users, refactoring or automatic code generation tools this means a loss of referential transparency. For example, the standard rules for weakest precondition calculation [1, 8] cannot be applied as is:

- $[x := \varnothing]\ (x \neq \varnothing \Rightarrow card(x) > 1) = (\varnothing \neq \varnothing \Rightarrow card(\varnothing) > 1)$, giving rise to a type error in Event-B.
- $[x, y := \varnothing, \{\varnothing\}]\ (\{x\} \subset y \Rightarrow z = 1) = (\{\varnothing\} \subset \{\varnothing\} \Rightarrow z = 1)$ again giving rise to a type error in Event-B.

With the current input syntax of Rodin, refactoring or automatic code generation tools would need to capture predicates like $\varnothing \neq \varnothing$, $\varnothing \subseteq \varnothing$, $\{\varnothing\} = \varnothing$, $\{\varnothing\} \subset \varnothing$, $\varnothing \cup \varnothing = \varnothing$, $\varnothing \in \varnothing \rightarrow \varnothing$, ... or even $\{1 \mapsto \varnothing\} \neq \{2 \mapsto \varnothing\} \cup \{3 \mapsto \varnothing\}$ and $\{1 \mapsto 2\} \subset (\{1 \mapsto \varnothing, 2 \mapsto \varnothing\}; \{\varnothing \mapsto 2\})$ and filter them out to either true or false.

Expressions can also be problematic, if an expression drops a part of the argument, such as the projection functions. The current projection functions in Rodin are not yet

[2] It is unclear to the authors whether this is intentional or a side-effect of a limited type inference procedure.

[3] This problem occurred in earlier versions of Rodin. The static checker now generates an extended output of Event-B strings. It translates `x/={} => card(x)>1` to `x/=({}::typeof(x)) => card(x)>1`. The corresponding syntax is not allowed in the input notation.

problematic, as the user has to explicitly provide types: $prj1(\mathbb{N} \times POW(\mathbb{N}))(1 \mapsto \varnothing)$. This is going to change, however, at which point the expression $prj1(1 \mapsto \varnothing)$ and thus also the substitution $x := prj1(1 \mapsto \varnothing)$ will give rise to a type error. Here, a refactoring tool will have to catch expressions such as $prj1(1 \mapsto \varnothing)$ or $prj1(1 \mapsto (2 \mapsto \varnothing))$ and insert artificial constructs such as $prj1(1 \mapsto \varnothing \cap \mathbb{N})$ or $prj1(1 \mapsto (2 \mapsto \varnothing \cap \mathbb{N}))$.[4]

11.2 More Contrived Type Inference

Type inference and checking in standard polymorphism can be done efficiently using the Hindley-Milner type inference [6]. This can also be viewed as using resolution (i.e., unification) on a type theory expressed in the style of the following rule, defining a type rule for the intersection operator. Here, $e : \tau$ expresses the fact that the expression e can be of type τ.

$$\frac{\vdash S : POW(\tau) \quad \vdash V : POW(\tau)}{\vdash S \cap V : POW(\tau)}$$

In Prolog this can be encoded as the following clause:

```
type(inter(S,V),pow(T)) :- type(S,pow(T)),type(V,pow(T)).
```

Prolog execution can be used to perform type inference or type checking. See, e.g., a Prolog formalisation of a larger subset of Event-B in Appendix A. As can be seen, this results in full polymorphic type inference. ProB as of version 1.3 uses such a type inference and checking approach for classical B.

Moving to limited polymorphism In order to encode the limited polymorphism of Event-B, the mathematical formalisation of the type system needs to be adapted. The Prolog code also needs to be adapted to pass type parameters up, which can then be checked for groundess at the end (see, Appendix B). The outermost predicate of the Prolog type checker now becomes:

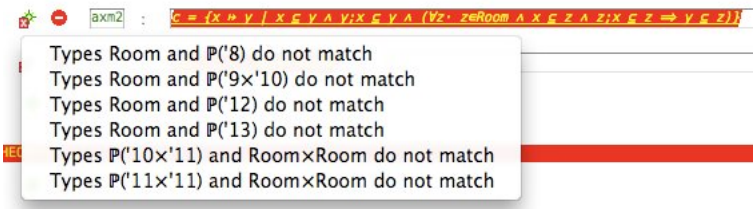
```
type_inference(X,OneSol) :- findall(T,type(X,T),AllTypes),
  (AllTypes=[OneSol] ->
    (ground(OneSol) -> true ; print('### Type not completely determined!'),nl)
          ;   print('### Mutliple typings allowed: '),print(AllTypes),nl,fail).
```

In Prolog this is still relatively straightforward. Note, however, that we use the Prolog built-in predicate `ground/1`. Formalising `ground` in pure logic is a bit more complicated and requires the addition of an additional virtual base type. If a type can create an instance containing this virtual base type, then the type is not ground. In Prolog one could formalise the negation of a non-ground type as follows:

```
non_ground_type(err).
non_ground_type(pow(X)) :- non_ground_type(X).
non_ground_type(A*B) :- non_ground_type(A) ; non_ground_type(B).
```

[4] When instantiating a bound variable x with $\varnothing$ it is not possible to type check $\varnothing$ by itself but it is necessary to bind it in a term with a variable x freed, e.g., in an equation $x = \varnothing$. With an earlier version of the Rodin proving interface the user also sometimes had to enter $\varnothing \cap \mathbb{N}$, say, instead of $\varnothing$.

In the Rodin reference on the Mathematical language type inference is formalised in terms of attribute grammars with inherited and synthesised attributes (see [3]). We would argue that this is a very implementation oriented description, not so easily comprehensible to many formal methods users. We believe "classical" typing inference rules to be more comprehensible. Also, possibly due to the more complicated nature of the type inference algorithm, the current Rodin type error messages can be incomprehensible to a normal user (and failing to locate the source of the type error in the formula):



The unification-based type inference checker of ProB gives four type errors on the corresponding example in classical B, highlighting the exact locations of the type errors:

```
CONSTANTS c
PROPERTIES
 c = {x,y | x<:y & (y;x) <: y & (!z.(z:Room & x<:z & (z;x) <: z => y<:z))}
```

The error associated with the first underlined occurence of z is as follows (i.e., the type checker was expecting a relation of type $POW(\alpha \times \alpha)$ but obtained an expression of type $Room$):

```
  type mismatch: Expected POW((_A*_A)), but was Room
/Users/leuschel/svn_root/TEX/rodin_papers/Test.mch
 ### Line: 6, Column: 49
```

11.3 No polymorphic theorems

We cannot prove $S \cup \varnothing = S$ in Rodin, let alone state it as an axiom, because the type of S, $POW(\alpha)$, is not ground. Especially in light of mathematical extensions such as sequences, trees, etc., this is potentially problematic. It will not be possible to generate generic proof rules — as used by the Rodin provers — in the language of the mathematical extensions, e.g., $\vdash\ E = E$ is a rule but $\forall E \cdot E = E$ is not a well-formed formula. This will eventually make it necessary to have two notations, one for expressing mathematical extensions and one for adding rules.

In Haskell we can define a predicate checking for an empty list

```
Prelude> let empty x = x == []
```

If this was not polymorphic it would have to give some type to `[]`, say T. So it would not be possible to use it with some other type S. A non-polymorphic approach seems not reasonable. In Haskell,

```
Prelude> empty []
True
```

In Event-B this would currently be impossible?

11.4 Solutions

In summary, we believe that the restriction of standard polymorphism in Event-B engenders a whole series of subtle problems, while providing no apparent benefit.

- Live with the existing Event-B de-facto standard, and formulate mathematical extensions using limited polymorphism.

  We believe this solution will hinder the Rodin tool's long term prospects. Also, solutions for refactoring tools will have to be found. One is to provide a simplifier procedure, which eliminates type-problematic tautologies ($\varnothing_\tau = \varnothing_\tau$), contradictions ($\varnothing_\tau \neq \varnothing_\tau$) and expressions ($prj1(1 \mapsto \varnothing)$) from Event-B formulas before a refactoring tool generates the user-readable and editable textual representation. Another solution is to allow users to add type annotations to constants, which the refactoring tool can make use of. Another solution is to extend the Rodin type inferencer to perform "defaulting", see below.

- Rewrite Rodin for full polymorphism, e.g,. based on polymorphic many sorted FOL.

  This may mean a large amount of implementation work. Note that for mathematical extensions, the AST needs to be extended (and thus probably rewritten) anyway. It should be examined to what extent such an extension of Rodin for full polymorphism can be done simultaneously with mathematical extensions and how much additional effort it would engender. Some of the current issues with the AST could also be cleaned up at the same time (hardwiring of some of the internal Rodin tools' procedures/intermediate datastructures into the AST, having the AST mutable, ...). An easy interface to functional/logic languages should also be provided (maybe Scala?).[5]

- Formulate the mathematical extensions using full polymorphism, while keepig the current Rodin core tool's limited polymorphism as is.

  When the mathematical extension tools need to generate legacy Rodin formulas, then Haskell-style defaulting can be used. Defaulting in Haskell replaces type variables by the unit type, in case a concrete type is needed (e.g., for the compiler).[6] In our case, we would need to introduce a family of default types

[5] Some of the aspects of Rodin are ideally suited for such languages, e.g., type inference, POG generation, static analysis, rule-based provers. The code for these in a functional/logic language would both be more succinct, maintainable and probably more efficient. E.g., a generic AST traversal can be written in 4 lines of Prolog; equivalent type-safe versions can also be written extremely succinctly in Haskell. The Rodin equivalent traversal of the AST takes several hundred lines of code, with ensuing difficulties for maintenance and comprehension. Our translator from the Rodin AST to Prolog requires1700 lines of Java code (that is with the new more succinct visitor of Rodin).

[6] This is also sometimes considered to be an ugly corner of Haskell. See, for example, `http://neilmitchell.blogspot.com/2009/02/monomorphism-and-defaulting.html`.

(deferred sets). If the maximum number of type variables inside the type of an expression is n (e.g., for $POW(\alpha \times \beta)$ we have $n = 2$), we need n deferred sets. One could then prove a meta-theorem that a polymorhpic formula is true after defaulting the types in Event-B iff it can also be proven in polymorphic many-sorted logic.

Similarly, to overcome the referential transparency issue, adding defaulting to the type inference algorithm could be used. E.g., the type inference could either add a new virtual type or always use BOOL for type variables.

- Live with the existing Event-B standard and formulate limited mathematical extensions for now.

  The idea is to not change the AST for the moment. Mathematical extensions would be implemented for the moment by using prefix-notation for new operators, thus not requiring changes (or only minimal changes) to the parser and AST. We would also use a prototype extensible rule-based prover into which the rules for the new extensions would be encoded.

  At a later stage, when it becomes clear which approach should be used for mathematical extensions, one of solutions 1-3 above would be adopted.

References

[1] Jean-Raymond Abrial. *The B-Book*. Cambridge University Press, 1996.

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

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools (Second Edition)*. Addison Wesley, 2007.

[4] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[5] Patricia Hill and John W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[6] Robin Milner. A theory of type polymorphism in programming. *Jcss*, 17:348–375, 1978.

[7] 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.

[8] Steve Schneider. *The B-Method: An introduction*. Palgrave Macmillan, 2001.

[9] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: An efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.

A Type inference for standard polymorphism

```
type(X,int) :- integer(X).
type('|->'(A,B),TA*TB) :- type(A,TA), type(B,TB). % maplet
type(inter(S,V),pow(T)) :- type(S,pow(T)),type(V,pow(T)). % intersection
type(inv(R),pow(B*A)) :- type(R,pow(A*B)). % inverse of a relation
type([],pow(_)). % empty set
type([H|Tail],pow(T)) :- type(H,T), type(Tail,pow(T)). % extension set
type(X+Y,int) :- type(X,int), type(Y,int). % addition

type(X=Y,pred) :- type(X,T),type(Y,T). % equality
type(X<Y,pred) :- type(X,int), type(Y,int). % less-than
type(in(X,S),pred) :- type(X,TX), type(S,pow(TX)). % set membership

type('&'(X,Y),pred) :- type(X,pred(TX)), type(Y,pred(TY)).
type(sym(R),pred) :- type(R=inv(R),pred(TR)). % symmetric relation predicate
% or type(sym(R),pred) :- type(R,pow(A*A)).

| ?- type([],R).
R = pow(_A) ?
yes
| ?- type(inter([],[3+7,4]),R).
R = pow(int) ?
yes
| ?- type([]=[3],X).
X = pred ?
yes
| ?- type([1]=[[]],X).
no

| ?- type(inv([]),R).
R = pow(_A*_B) ?
yes
| ?- type(sym([]),X).
X = pred ?
yes
| ?- type(inv(['|->'(1,[])]),R).
R = pow(pow(_A)*int) ?
yes
| ?- type(sym(inv(['|->'(1,[])])),R).
no
```

Provided there are no non-deterministic rules in the style of

```
type(X*Y,int) :- type(X,int), type(Y,int).
type(X*Y,pow(TX*TY)) :- type(X,pow(TX)), type(Y,pow(TY)).
```

we always get exactly one or no solution.

B Type inference for controlled polymorphism

```
type(X,int) :- integer(X).
type('|->'(A,B),TA*TB) :- type(A,TA), type(B,TB). % maplet
type(inter(S,V),pow(T)) :- type(S,pow(T)),type(V,pow(T)). % intersection
type(inv(R),pow(B*A)) :- type(R,pow(A*B)). % inverse of a relation
```

```
type([],pow(_)). % empty set
type([H|Tail],pow(T)) :- type(H,T), type(Tail,pow(T)). % extension set
type(X+Y,int) :- type(X,int), type(Y,int). % addition

% arguments of pred are only there to pass inner type variables
%    to the top and detect if polymorphism persists in inner nodes
type(X=Y,pred(T)) :- type(X,T),type(Y,T). % equality
type(X<Y,pred(int)) :- type(X,int), type(Y,int). % less-than
type(in(X,S),pred(TX)) :- type(X,TX), type(S,pow(TX)). % set membership

type('&'(X,Y),pred(conj(TX,TY))) :- type(X,pred(TX)), type(Y,pred(TY)).
type(sym(R),pred(TR)) :- type(R=inv(R),pred(TR)). % symmetric relation predicate

type_inference(X,OneSol) :- findall(T,type(X,T),AllTypes),
  (AllTypes=[OneSol]
    -> (ground(OneSol) -> true ; print('### Type not completely determined!'),nl)
    ;   print('### Mutliple typings allowed: '),print(AllTypes),nl,fail).
```