Technical Note: Building Abstract Mathematical Types in Event-B

James Snook[1]

ECS, University of Southampton, Southampton, U.K.
{jhs1m15

Abstract. The goal of this technical note is to demonstrate how Event-B can be used to build mathematical libraries in a way which maximises the sharing of theorems and proofs, therefore minimising the proof burden on the user. The approach taken is to construct abstract mathematical types such as monoids and demonstrate how concrete mathematical types like the naturals can inherit results from the abstract types. The result is that it is possible to build a library in this manner, however, without language features to support these notions the user is required to do a lot of additional manual work within the construction.

Keywords: Formal Methods; Event-B; Theorem Proving;

1 Introduction

The aim of this Technical note is to explore mathematical heritability using the Event-B language [1], extended with features from the Theory Plug-in [3]. The reason for looking into this work is due to a previous Event-B case study which modelled the expected flight paths of UAVs (unmanned aerial vehicles) [2][4]. To do this the real numbers were modelled axiomatically as a field, to create the axiomatic definition it was necessary to create two separate operators (addition and multiplication) and state the axioms for each of these operators separately. The axioms for each of the operators were the axioms of an abelian group. Not only was it necessary to state the axioms twice, but proofs that could be done on abelian groups had to be done twice once for each axiom. This technical note therefore looks at how abstract mathematical types can be created in such a way that they can inherit from each other. It also looks at how concrete types can inherit the results of the abstract types, aiding the development of concrete theories.

2 Operators

To construct abstract mathematical types such as monoids and groups it is first useful to have structures representing the building blocks of these types. This section describes the representation of generic operators (e.g., functions of the form $T \times T \rightarrow T$).

With the aim of later being able to construct a monoid, it is first useful to have a representation of an associative operator. There are several ways in which an associative operator could be defined using the Theory Plug-in, the simplest choice to do this within the Theory Plug-inis an axiomatic definition:

$AssocOp(x, y) : T$;
  `Axioms:`
    $x, y, z \cdot x \in T \land y \in T$
      $\Rightarrow AssocOp(AssocOp(x, y), z) = AssocOp(x, AssocOp(y, z))$ `//Associative`

This creates an abstract representation of an associative operator (e.g., there is no definition of how this operator works, it is only defined that it is associative). There are, however, problems with this representation, firstly there is no way of instantiating T to another type, so for example if it is desired to work to reason about associative operators on the natural numbers a new axiomatic definition needs to be given. Secondly there is no way to reuse this definition in later definitions. For instance to create an abelian operator (one which is both commutative and associative) there is no way of re-using the definition of the commutative or associative operator. Third, if a concrete operator is created (e.g., addition on the natural numbers) there is no way to relate this to the abstract definition. As the aim is to produce reusable definitions this way of defining operators is not helpful.

The alternative approach taken is to use the Event-B set comprehension syntax for creating subsets, along with the Theory Plug-inoperators in the following way:

$$AssocOp(t : \mathbb{P}(T)) \mathrel{\widehat{=}} \{op \mid op \in (t \times t) \to t \land \forall x, y, z \cdot x \in t \land y \in t \land z \in t \\ \land\ op(x \mapsto op(y \mapsto z)) = op(op(x \mapsto y) \mapsto z\}$$

A generic type t is introduced, the type of t is in the powerset of an Event-B generic type T, this allows the operator to be applied to subtypes (e.g., one may wish to reason about the naturals which are less than 10, these are not an Event-B type as they are a subtype of the naturals. Making $t : \mathbb{P}(T)$ allows the operator to be applicable to these types). The set theoretic statement says creates the set of all ops such that they are subsets of total functions on pairs of t to t ($t \times t \to t$), and they are associative $op(x \mapsto op(y \mapsto z)) = op(op(x \mapsto y) \mapsto z$. This definition solves the issues of the axiomatic definition. We can instantiate with any type we like by passing that type to the operator. The operator can be reused by future definitions, for instance the definition of the set of abelian operators is:

$$AbelianOp(t : \mathbb{P}(T) \mathrel{\widehat{=}} AssocOp(t) \cap CommOp(t)$$

($CommOp$ is defined extremely similarly to $AssocOp$ except the associative statement is replaced with one defining commutativity). There is no need to explicitly restate the definition of associativity or commutativity here. Concrete operators are able to be associated with the abstract operator. This is seen later in 6.

When proving with the interactive prover to use the definition of associative operators within abelian operators it is required that you expand the definition of abelian operators, and then expand the definition of associative operators. As the mathematical hierarchy builds up the definition that you need for a proof may get further and further away. Theories/Proof rules could be used to bring these definitions forward, however, they would need to be declared and proved at each level that one wished to use them.

3 Monoid

This section looks at how we can define the abstract monoid type. The definition of operators were a matter of creating a subtype, e.g., the generic type for an operator was $(T \times T) \times T$, and this was subtyped with additional constraints. A monoid on the other hand is a set with an associated identity and associative operator. An operator to represent a generic monoid can be constructed in the following way:

$$\begin{aligned} Monoid(t : \mathbb{P}(T)) \;\widehat{=}\; \{ & ident \mapsto op \mid ident \in t \\ & \wedge\, op \in AssocOps(t) \\ & \wedge\, \forall x \cdot x \in t \Rightarrow op(ident \mapsto x) = x \wedge op(x \mapsto ident) = x\} \end{aligned} \tag{1}$$

this can then be used within theorems in the following way:

$$\begin{aligned} & \forall t, ident, op, oe, x \cdot t \in \mathbb{P}(T) \wedge ident \mapsto op \in Monoid(t) \wedge oe \in t \wedge x \in t \\ & \quad \Longrightarrow\; (op(oe, x) = x \Leftrightarrow oe = ident) \end{aligned} \tag{2}$$

this theorem makes a generic monoid with the statement $ident \mapsto op \in Monoid(t)$, the theorem then goes onto state that an element of t acting as the identity (in this case oe) must be the identity. Several other theorems were declared about the monoid structure, including theorems about the uniqueness of the identity, and corollaries used to prove this.

The definition of monoids given in 1 can be further subtyped to construct new abstract types. For instance, commutative monoids can be defined with the following statement:

$$CommMonoid(t : \mathbb{P}(T) \widehat{=} \{CM \mid CM \in Monoid(t) \wedge prj2(CM) \in CommOp(t)\} \tag{3}$$

Here the operator part of the monoid is accessed using the $prj2$ function from Event-B (This gets the second part of pair, in this case the pair). To facilitate writing theorems it is useful to write operators to do this e.g.:

$$mon\_op(m : Monoid(T)) \widehat{=} prj2(m)$$

Given a monoid this operator will deconstruct it to get the operator part. A similar operator was created to access the identity. Whilst this case is simple so working out how to deconstruct the abstract type is also simple, as the abstract type becomes more complex deconstructing them also becomes more complex, making these 'deconstruction' operators more useful.

An alternative approach would be to make a datatype to represent all structures with an operator and an element. The advantage of this approach is that the construction of the datatype would automatically create functions to get the constituent parts of the monoid (i.e., the identity and the operator). The disadvantage is that the monoid would get a

new type which would not reflect its construction. There is also no way to add well-definedness conditions to a datatype; they are only constructed from Event-B types. As $AssocOp$ is not an Event-B type we cannot form a datatype using it. Instead the datatype would need to be formed using the supertype of AssocOps. An $OPERATOR$ could then be written to create a subset of the datatype which represented monoids. The need for these additional operators makes using datatypes a more complex solution than using the set theoretic syntax directly.

When declaring Event-B operators a proof obligation is generated asking the user to prove that the operator is well defined. These well-definedness proof obligations can be onerous to prove manually. Fortunately if these operators for building and deconstructing abstract types are well defined it should be possible to prove the well definedness obligations by simply expanding the definitions of the abstract types. Within the interactive prover a tactic was created to expand these definitions, causing the well definedness proofs to be discharged automatically.

At this point it would be convenient to define the power operator on the monoids. The simplest implementation of the power operator is recursive: $a^n = a \cdot a^{n-1}$ the Theories Plugin only supports recursive function definitions with recursive types. The built in representation of the naturals used by Event-B is not recognised by the Theory Plug-inas a recursive type, to resolve this the next section shows how the natural numbers can be described recursively within the Theory Plug-in.

4 Naturals

Within this section a recursive representation of the natural numbers is created. This concrete representation serves three purposes: The first is to demonstrate that concrete mathematical types can be represented within the Theory Plug-in. The second is to make a recursive representation of the naturals that can be used in recursive operators and to allow inductive proofs. An example of this was seen at the end of the last section. The third is to create a type and operators to demonstrate how concrete types can inherit from abstract types, this is demonstrated by showing that zero and addition form a monoid.

4.1 Naturals Definitions

The naturals here are defined as the Peano Naturals:

$$
\begin{array}{l}
Nat \ \widehat{=} \\
\quad \texttt{Constructors:} \\
\quad\quad zero \\
\quad\quad suc(prev : Nat)
\end{array}
$$

This definition says that a Nat can be constructed either with the $zero$ keyword, or with $suc()$, which requires a Natural number as an argument. Theory Plug-indatatypes automatically inductive theorems to datatypes.

From this definition several operators were created, for instance addition was defined as follows:

$$\begin{array}{l} nAdd(x : Nat, y : Nat) \widehat{=} \\ \quad cases[x] \\ \quad\quad zero \rightarrow y \\ \quad\quad suc(xs) \rightarrow suc(xs\ nAdd\ y) \end{array} \tag{4}$$

along with operators such as: decrement, subtraction, and divmod. These operators are all defined much as you might expect, during the construction of this theory a soundness issue[1] was discover which required some workarounds. The addition operator defined above 4 has been prefixed with *n*, this is because names in Event-B are global, and it is likely that in the future other types will want to define an addition operator (e.g., the integer numbers), this namespacing issue can be resolved by prefixing operators.

At this point we should have should have enough concrete structures to make use of the abstract monoid type defined in the last section. Unfortunately, within the Event-B environment the operators we have been defining are not first class members of the Event-B syntax, and cannot appear in expressions without their arguments. This means that a theorem such as:

$$zero \mapsto nAdd \in Monoid(Nat) \tag{5}$$

is not valid Event-B as $nAdd$ is declared without its arguments. To resolve this the nAdd operator can be wrapped within an Event-B lambda. Given it is likely that this technique will be used multiple times, this is done within an operator:

$$nAdd\_P \mathrel{\widehat{=}} \lambda x \mapsto y \cdot \top \mid x\ nAdd\ y$$

When it is desirable to pass addition as a function the $nAdd\_P$ operator can be used in the place of the $nAdd$ operator. The result is theorem 5 can be stated as:

$$zero \mapsto nAdd\_P \in Monoid(Nat) \tag{6}$$

This is now valid Event-B, and means what one may assume 5 meant. This technique can be used whenever an operator/function takes a function as an argument to allow an operator to be passed instead. For each of the operators defined on the Nat type a second operator was declared to encapsulate the operator as an Event-B function.

Proving theorem 6 does not automatically bring any of the work forward from the $Monoid$ definition to the Nat definition. For instance, theorem2 about the uniqueness of the identity is not automatically instantiated in the Nat, this theorem is only accessible by instantiating it with $zero$ and $addition$, this would then require a proof that $zero$ and $addition$ form a monoid (which can be done by referencing theorem6. Rather than do

[1] Within the Theories Plug-in associativity is special cased and allows the flattening of equations. If you indicate that an infix operator is associative, and create an associativity theorem you can use the automatic flattening of the equation to prove the theorem, and you can use the theorem to prove the associativity of the operator. This means all operators can be proved associative, even ones that are not.

this every time this theorem is needed the user can manually instantiate it with a new theorem:

$$\forall x, y \cdot x\ nAdd\ y = y \Leftrightarrow x = zero \tag{7}$$

This theorem can be proved using the instantiation steps outlined above. The theorem can then be used in later proofs without having to instantiate it from the monoid theorem. This technique was used multiple times (whenever a theorem from an abstract type was found to be useful in a proof on a concrete type).

5 Back to Monoids

Now that there is a recursive definition of the natural numbers it is possible to create a recursive operator to represent the power functions on the monoid type class. This was done with the following declaration:

$$\begin{array}{l} Pow(ident : T, op : AssocOps(T), a : T, p : Nat) \hat{=} \\ \quad cases[p] \\ \quad\quad zero \rightarrow ident \\ \quad\quad suc(ps) \rightarrow op(a \mapsto Pow(ident, op, a, ps)) \end{array} \tag{8}$$

Additional theorems can be written about this operator, e.g.:

$$x^p \cdot x^q = x^{p+q} \tag{9}$$

Having previously demonstrated that the Nat type forms a monoid it would be nice to inherit results about this operator. This can be done by declaring a new operator:

$$nTimes(x : Nat, y : Nat) \hat{=} Pow(zero, nAdd\_P, x, y) \tag{10}$$

This definition of multiplication allows theorems about the Pow operator to be easily inherited, however, using the $nTimes$ operator results in the user having to do extra expansions. An alternative approach is have an instantiated $nTimes$ operator, and prove it is the same as Pow operator (in a very similar way that was done previously on theorems). This has an additional overhead for the user, however, future proofs are then easier.

There are often several ways to write a function. An example of this is that the power operator on the monoids can be written using a method called exponentiation by squaring. This technique is often used within computing as it is considerably faster than simply using continued addition, and is simple to implement when numbers are represented in a binary manner.

There are advantages to having different implementation of the same function:

1. The second implementation may reflect the way the system that is being modelled works. Having an implementation which is the same as that of the modelled system means that the user does not have to justify the difference in the implementations.

2. For proofs involving concrete uses of an operator, the second implementation may finish in substantially fewer steps. This will take less time and use fewer of the computer's resources. Some proofs which require a lot of operator expansions may be impossible to complete using the previous implementation of the operator.
3. Some proofs may become easier when done with the second implementation. For example the concrete implementation of multiplication within the binary naturals may be more similar to that of multiplication using squaring. This makes it easier to prove equivalences.

If we implemented this operator ($sqPow$), we would require the following theorem to demonstrate its equivalence to the original Pow operator:

$$Pow\_P = sqPow\_P$$

Having demonstrated these two definitions are equivalent theorems from one operator can be moved to the other. Also within concrete implementations it is possible to use the different implementations interchangeably (i.e., to use whichever one is best suited to the proof that is being done).

6 Binary Numbers

Having explored The Peano naturals, the case study will now look at a different implementation of the natural numbers. The purpose behind the second representation is to demonstrate the reusability of the abstract types, and to demonstrate how results can be shared (and work reduced) via isomorphic relationships.

This section constructs an implementation of the binary numbers, this is an interesting representation of the natural numbers as it is a representation of numbers that is more like the base ten representation that humans generally use. It is also very similar to the the way that computers represent the natural numbers. A second advantage to this representation is that operations such as addition and multiplication happen in considerably fewer steps, this can simplify proofs where there are operations on large numbers.

The binary numbers are represented as lists of Boolean values, with $FALSE$ representing a zero and $TRUE$ representing a one. Within the Theory Plug-inlists are defined using the following datatype declaration:

$List \ \widehat{=}$
  Constructors
    nil
    $cons(head : T, \ tail : List(T))$

The representation chosen for the binary naturals has the least significant bit in the leading position (i.e., the head of the list is the least significant bit). This makes the definitions of operators such as addition and multiplication simpler. However, having the least significant bit first is counter intuitive as the decimal representation that everyone is used to has the most significant digit first.

Having defined the datatype to represent the binary numbers several other operators were defined (including many bitwise operators, and multiplication), as an example addition was constructed with the following definition:

$$
\begin{aligned}
&bnAdd(x : List(BOOL), y : List(BOOL)) \hat{=} \\
&\quad cases[x] \\
&\quad nil \rightarrow y \\
&\quad cons(xB, xs) \rightarrow if(xB = TRUE \wedge bnLSB(y) = TRUE) \\
&\qquad cons(FALSE, bnIncrement(xs\ bnAdd\ tail(y))) \\
&\quad else \\
&\qquad if(XB = FALSE \wedge bnLSB(y) = FALSE) \\
&\qquad\quad cons(FALSE, xs\ bnAdd\ bnShiftLeft(y)), \\
&\qquad else \\
&\qquad\quad cons(TRUE, xs\ bnAdd\ bnShiftLeft(y))
\end{aligned}
\tag{11}
$$

This definition of addition uses several helper functions, $bnLSB$ gets the least significant bit of the list, if the list is empty it returns $FALSE$ (zero), $bnIncrement$ increments the number by one, $bnShiftLeft$ is almost identical to the list $tail$ deconstructor (i.e., returns a list with the $head$ element removed) except when the value is nil it returns nil, this is used to continue the addition process on the rest of the list.

Whilst this definition of addition looks complicated it is in fact how addition is often taught, the least significant bits are added together, and the result is put in the least significant position, then the rest of the number is added together with an additional one if the addition of the least significant bit overflowed.

These operators gave enough of an implementation to be able to relate to the abstract monoid type, and the previously created naturals representation.

A representation of numbers in this fashion highlighted a problem that was not seen with the Peano naturals implementation. This is that numbers represented in this fashion can have additional bits that do not change the value of the numbers. This is directly analogous to us considering 00123 equal to 123. In the binary representation if the list ends in $FALSE$s (i.e., there are zeros at the most significant end of the list) these do not change the value of the number. In previous examples structural equality has been used, this will not work for the binary numbers as 00123 is not structurally identical to 123. Two ways to resolve this issue are explored below. First, an equivalence relation to use instead of structural equality. The second a subtype of 'List(BOOL)' is defined where the subtype has to have most significant bit set to one.

6.1 Equivalence Relation

To define an equivalence relation for binary numbers we need to define an operator, and demonstrate that it conforms to the axioms of an equivalence relation i.e. for an operator $\sim$:

Reflexivity $\forall a \cdot a \sim a$

Symmetry $\forall a, b \cdot a \sim b \Leftrightarrow b \sim a$
Transitivity $\forall a, b, c \cdot a \sim b \wedge b \sim c \Rightarrow a \sim c$

To do this an abstract definition of an equivalence relation was created. For example, here is the definition of an abstract reflexive relation:

$$Reflexivity(t : \mathbb{P}(T)) \widehat{=} \{refl | refl \in \mathbb{P}(t \times t) \wedge x \mapsto x \in refl\} \tag{12}$$

This abstract definition in Event-B does not quite represent a reflexive relation, what it really represents is a set where $x \mapsto x$ is a member of the set for all x. The reason for this is that in Event-B predicates are a separate syntactic category and there is no way to create an Event-B function of the form $T \rightarrow Pred$ (note that it is possible to create predicate operators, as mentioned earlier operators are not functions). It will be seen later how this representation can be used to represent a relation within Event-B. Symmetric and transitive relations were created following a similar pattern. Finally an equivalence relation was created with the following statement:

$$EquivRel(t : \mathbb{P}(T)) \widehat{=} ReflexRel(t) \cap SymmetricRel(t) \cap TransRel(t) \tag{13}$$

Several theorems about the equivalence relations were created, for instance $x \sim y \wedge \neg(y \sim z) \Rightarrow \neg(x \sim z)$, expressing this in the set theory syntax results in an expression that like this:

$$\begin{array}{l} \forall t, equ, x, y, z \cdot t : \mathbb{P}(T) \wedge equ \in EquivRel(t) \wedge x \in t \wedge y \in t \wedge z \in t \\ \quad \Rightarrow (x \mapsto y \in equ \wedge y \mapsto z \notin equ \implies x \mapsto z \notin equ) \end{array} \tag{14}$$

this theorem example shows how the $\in$ operator is used to turn the set notation into predicates for the purpose of theorems.

Along with several operators that have been presented before the definition of the equivalence relation uses the operators $bnIsZero$. This operator deconstructs a list checking that each value is $FALSE$ (i.e., that the list is either nil, or all zeros). The equivalence relation on the binary naturals is:

$$\begin{array}{l} bnEq(x : List(BOOL), y : List(BOOL)) \widehat{=} \\ \quad nil \rightarrow bnIsZero(y) \\ \quad cons(xB, xs) \rightarrow xB = bnLSB(y) \wedge xs \; bnEq \; bnShiftLeft(y) \end{array} \tag{15}$$

To inherit the results from the abstract definition it is required that this operator has a set syntax equivalent, this is achieved with with following statement:

$$bnEqSet() \widehat{=} \{x \mapsto y | bnEq(x, y)\} \tag{16}$$

It can now be shown that $bnEqSet$ is an instance of the $EquivRel$ type this is done with the following theorem:

$$bnEqSet \in EquivRel(List(BOOL)) \tag{17}$$

Once this is proved it is possible for the $bnEqSet$ to inherit results about the $EquivRel$ abstract type. As was seen before with the $Monoid$ type manually restating these proofs about the $bnEq$ operator. Proving them can be made easier using theorem 17, however, due to the lack of predicate functions this adds even more additional work.

The additional work of manually having to instantiate theorems reduces the benefit of having the abstract types available. However, the abstract types still serve a useful purpose of guiding the user through the proofs they need. For instance in the last example it was desired to prove that we had an equivalence relation, this requires proving reflexivity, symmetry and transitivity, the abstract definition encapsulated this information.

Finally, an aim of this section was to see how another type could be associated with the already constructed abstract types, in this case this would mean demonstrating that with the equivalence relationship and the binary naturals could form a monoid. However, the definition of the monoid was created based on structural equality, the result is it is not possible to relate these structures to the previously constructed monoid type. To allow this the monoid type would need to be redefined:

$$\begin{aligned} &MonoidEquiv(t : \mathbb{P}) \widehat{=} \\ &\quad \{equ \mapsto ident \mapsto op | equ \in EquivRel(t) \wedge ident \in t \wedge op \in AssocOp(t, equ) \\ &\qquad \wedge op(ident \mapsto x) \mapsto x \in equ \wedge op(x \mapsto ident) \mapsto x \in equ\} \end{aligned} \tag{18}$$

Reconstructing the monoids like this would include re-writing the associative operator type, and restating and proving the theorems about the monoid type.

6.2 Subtyping for Equality

Rather than constructing an equivalence relation we can look at the subclass of binary numbers which are normal. Normal, here, is defined as having no trailing zeros. We can use an operator to identify normal numbers:

$$\begin{aligned} &bn\_isNormal(a : List(BOOL)) \widehat{=} \\ &\quad \texttt{cases[a]} \\ &\qquad nil \rightarrow \top \\ &\qquad cons(aB, as) \rightarrow (as = nil \wedge aB = TRUE) \vee (as \neq nil \wedge bn\_isNormal(as)) \end{aligned}$$

From this definition we can use an operator to create a subset of $List(BOOL)$ which contains all the normal numbers:

$$Normal\_Bin \widehat{=} \{num \mid bn\_isNormal(num)\}$$

Once normality is defined we can start making statements about equality rather than equivalence. An interesting starting point is to look at how equality on the subtype relates to the equivalence relation, in certain conditions this can allow proofs about equivalence to be reused in the equivalent proofs about equality

$$\forall x, y \cdot x = y \Rightarrow x \; bn\_Eq \; y \tag{19}$$

$$\forall x, y \cdot bn\_isNormal(x) \wedge bn\_isNormal(y) \Rightarrow (x \; bn\_Eq \; y \Rightarrow x = y) \tag{20}$$

An effective strategy for working with subtypes is to prove that when an operator is applied to members of the subtype the result is a member of the subtype i.e., the operator is closed with respect to the subtype. An example of this would be the following theorem:

$$\forall x, y \cdot x \in NormalBin \wedge y \in NormalBin \implies x \; bnAdd \; y \in NormalBin \tag{21}$$

This sort of theorem allows the normality of numbers to be followed through an expression. This facilitates proofs because every number with equal value is also structurally identical, value and structure become the same. Unfortunately not every operator will produce structurally identical numbers. An example of this is subtraction, where the most significant bits of a number can cancel out leaving a series of trailing zeros. The solution to this problem is either to create a new operator which does not result in trailing zeros, in the case of subtraction this can be done by stopping the operations when the remains of the number are identical. However, in other cases the best approach may be to wrap the operator in a normalisation operator. In the case of the binary numbers an operator to remove the trailing $FALSE$s of a list was created for this purpose. It was also proved that applying this operator to any binary number resulted in a normal binary number. With this approach of subtyping it was possible to become a member of the abstract monoid type. This was done by proving isomorphic properties with the peano naturals, and is outlined in the isomorphisms section 7.

Both the subtyping and equivalence approaches to working with the binary numbers worked well, and there were trade offs for both of them. In the case of subtyping there was extra work demonstrating that operators that worked on the subtype were closed. Allowing normality to be proved throughout theorems. With the equivalence approach there was additional work demonstrating that the $bnEq$ operator was an equivalence operator. It was also harder to work with an equivalence relation than equality, this was due to a lack of support for equivalence relations within the interactive prover.

7 Isomorphisms

In this section isomorphisms between the peano naturals and the binary naturals are created. The aim of doing this is to simplify proofs on the binary naturals (although in some cases it may work the other way around).

When we construct inductive proofs on numbers we assume that the next number is the increment of the current number. This assumption is justifiable because we know that our number system, and its operators, are isomorphic to equivalent structures on the Peano numbers. Unless we have proved such an isomorphism between the Naturals and our binary numbers the only form of induction that can be used is the induction suggested by the datatype. Given a Boolean list x and an expression to prove Exp, induction takes the form:

1. Show the Exp is true where $x = nil$,
2. given Exp is true for $x = x\_tail$ show that Exp is true for $x = cons(x\_head, x\_tail)$, where x_head can take the value of TRUE or FALSE.

This is the same as, instead of having our inductive step as an increment, having the step as a multiplication by two, and maybe adding one. In many cases it makes proving results on the binary naturals harder than their equivalent proof on the peano naturals.

Not only does demonstrating an isomorphism between the Binary numbers and the Peano naturals make proving new theorems about the operator easier, it also means that we can inherit any proofs that we have already made about the equivalent operator on the other structure. If it is easier to prove a result on one type then it is worth writing the theorem within that type first.

This project has not tried to create an abstract representation of isomorphisms, although this would be useful work to develop.

7.1 Building the Isomorphism

To build an isomorphism the first step is to create a bijective function between the two structures, to do this two functions are defined in the following manner:

$$
\begin{array}{l}
bnToNat(a : List(BOOL)) \triangleq \\
\quad \texttt{cases[a]} \\
\qquad nil \ \rightarrow \ zero \\
\qquad cons(aB, \ as) \ \rightarrow \\
\qquad\quad if(aB = TRUE) \\
\qquad\qquad one \ nAdd \ (bnToNat(as) \ nAdd \ bnToNat(as)) \\
\qquad\quad else \\
\qquad\qquad bnToNat(as) \ nAdd \ bnToNat(as)
\end{array}
$$

$$
\begin{array}{l}
bnToBin(a : Nat) \triangleq \\
\quad \texttt{cases[a]} \\
\qquad zero \ \rightarrow \ nil \ \wr \ List(BOOL) \\
\qquad suc(xs) \ \rightarrow \ bnIncrement(bnToBin(xs))
\end{array}
$$

Theorems are added to prove that these functions are the inverses of each other:

$$
\begin{array}{c}
bnToNat(bnToBin(x) = x \\
bnToBin(bnToNat(x)) \ bnEq \ x
\end{array}
\tag{22}
$$

(The second statement is also true with equality if x is normal). In the remainder of this work equivalence will be looked at rather than subtyping, within the study both approaches were taken. To demonstrate that this is a bijection it was further required to show that every element in the peano naturals had an equivalent element in the binary naturals and vice versa.

To demonstrate that addition is isomorphic, it was useful to show that the 'increment' operator of the binaries was equivalent to the 'suc' operator of the naturals. This was done by proving the following two theorems:

$$\forall x \cdot bn\_toNat(bn\_increment(x)) = suc(bn\_toNat(x)) \tag{23}$$

$$\forall x \cdot bn\_toBin(suc(x)) = bn\_increment(bn\_toBin(x)) \tag{24}$$

Using these theorems it was then possible to demonstrate that that every element in peano naturals had an equivalent in the binary naturals and vice versa. Demonstrating that the functions formed a bijection. All that was then required to demonstrate an isomorphism for an operator was to prove that the operator had a homomorphic equivalent:

$$\forall x, y \cdot bn\_toBin(x\ nAdd\ y) = bn\_toBin(x)\ bnAdd\ bn\_toBin(y) \tag{25}$$

Given a bijection it is enough to prove a homomorphism in one direction. If there had been a abstract theory about isomorphisms, this result could have been included in that, instead the homomorphism in the other direction was proved independently. Having proved a bijection and a homomorphism the operators are proved to be bijective to each other.

7.2 Using the Isomorphism

Now that it has been demonstrated that addition on the Peano and Binary numbers are isomorphic, we can take proofs from the more complete theory (the Peano naturals) and bring them into the other. This requires re-writing all of the proof rules/theorems that we want to use on the type which is inheriting them, and then proving the theorems. These proofs all follow the same pattern, which can be demonstrated by showing that addition is commutative (here I have done it on normal numbers, the normality part of the proof is emitted). The theorem to prove is:

$$\forall x, y \cdot x\ bn\_Add\ y = y\ bn\_Add\ x$$

This is proved in the following manner:

1. Expand one side using the bijection:

$$bn\_toBin(bn\_toNat(x\ bn\_Add\ y)) = y\ bn\_Add\ x$$

2. Expand the inner function of the bijection using the homomorphism theorem:

$$bn\_toBin(bn\_toNat(x)\ nAdd\ bn\_toNat(y)) = y\ bn\_Add\ x$$

3. We can now use the property from the more complete theory (in this example to do a commutative swap around $nAdd$):

$$bn\_toBin(bn\_toNat(y)\ nAdd\ bn\_toNat(x)) = y\ bn\_Add\ x$$

4. Use the other homomorphism to expand the outer function of the bijection:

$$bn\_toBin(bn\_toNat(y))\ bn\_Add\ bn\_toBin(bn\_toNat(x)) = y\ bn\_Add\ x$$

5. Use the bijection this time to remove the bijective functions:

$$y\ bn\_Add\ x = y\ bn\_Add\ x$$

This process was repeated for many of the properties of the addition operator, including demonstrating that addition is commutative, and forms a monoid with a nil list. Each of the proofs for these was almost identical to the proof laid out above.

8 Conclusion

In conclusion building abstract mathematical theories such that the theories in the abstract types can be inherited by concrete types and future mathematical types is possible in Event-B. However, as Event-B does not have language mechanisms to support this sort of inheritance it required a lot of additional manual work by the user (e.g., restating, of theories, and bringing proofs forward from the abstract types). This additional work was very mechanical, e.g., to move theorems from an abstract type to a concrete type involved restating the theorem on the concrete type. To prove the theorem, required reusing the theorem on the abstract type and reusing the theorem that proved the concrete type was a member of the abstract type. Further difficulties were caused by predicates and operators being separate syntactic types within the Event-B language, is was possible to work around both of these issues, but again required additional effort by the user (again this was found to be very mechanical work).

References

1. Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.
2. Chris Bogdiukiewicz, Michael J. Butler, Thai Son Hoang, Martin Paxton, James Snook, Xanthippe Waldron, and Toby Wilkinson. Formal development of policing functions for intelligent systems. In *28th IEEE International Symposium on Software Reliability Engineering, ISSRE 2017, Toulouse, France, October 23-26, 2017*, pages 194–204. IEEE Computer Society, 2017.
3. Michael J. Butler and Issam Maamria. Practical theory extension in Event-B. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, volume 8051 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2013.
4. Toby Wilkinson, Michael Butler, Martin Paxton, and Xanthippe Waldron. A formal approach to multi-uav route validation. 2015.