UNIVERSITY OF SOUTHAMPTON

FACULTY OF ENGINEERING AND PHYSICAL SCIENCE

School of Electronics and Computer Science

Constructing a New Language to Facilitate Mathematical Proofs In The Event-B Context

by

James H. Snook

Supervisor: Prof. Michael Butler, Dr. Thai Son Hoang

Examiner: Dr. Andy Gravell, Dr. Yamine Aït Ameur

November 23, 2020

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND PHYSICAL SCIENCE
School of Electronics and Computer Science
CONSTRUCTING A NEW LANGUAGE TO FACILITATE MATHEMATICAL PROOFS IN THE EVENT-B CONTEXT

by James H. Snook

There are many languages used for modelling systems, and verifying the consistency of specifications. Having a valid specification reduces the chances of finding major issues later in the development process. Many of these systems use a set theoretic syntax, which gives a powerful mathematical system for modelling discrete systems. This thesis focuses on the Event-B modelling system, although the results could be applied to other set theoretic modelling systems.

The aim of this thesis is to increase the number of mathematical theories available to the Event-B modeller, as this allows allows more accurate models to be created. The approach taken is to facilitate the definition of abstract mathematical types such as Monoids, Groups, and Rings in such a way that they can be built up hierarchically, and concrete mathematical types e.g., the naturals can use the results of the abstract types.

This work firstly shows how the current set theoretic language (Event-B) can be used to define abstract mathematical types, and have concrete mathematical types reuse the results of the abstract types. It also demonstrates many difficulties of defining these types using the set theoretic syntax. To resolve these issues a new language, $B^\sharp$, is proposed. This language is designed to facilitate the definition of abstract mathematical types, and translate to the current Event-B syntax. The major additions to the new language to allow the definition of mathematical types are type classes and subtypes.

At its core the $B^\sharp$ language is a HOL style language. This work demonstrates how a HOL style language can be translated into a set theoretic style language. This translation is then extended to translate all the features of the $B^\sharp$ language to the Event-B syntax. This allows theorems defined in the $B^\sharp$ language to be used by Event-B modellers.

A developer environment to define $B^\sharp$ theorems is built, and the translation from the $B^\sharp$ syntax to the Event-B syntax is implemented. This is used to define several mathematical types, and demonstrate the practicality of the approach.

The approach of adding a translation phase from a new syntax was found to be a safe (inconsistencies could not be added to proving mechanisms) and effective way of adding features to the Event-B toolset. Many additional features and improvements are suggested. This approach could be used in a similar manner to other theorem provers use of a meta language (ML) to safely add features without introducing inconsistencies.

Contents

Acknowledgements

Thanks to my supervisors, and Dr. Toby Wilkinson for their help and support with this work. Also to my wife for being such a fantastic support.

Chapter 1

Introduction

1.1 System Modelling with Set Theory

There are many languages used for modelling systems, and verifying the consistency of specifications e.g., Z [24], B [73], Event-B [57], VDM [49]. The aim of these systems is to remove errors during the specification of a system. As seen in [54], having a consistent specification reduces the chances of finding major issues later in the development process. Major issues would require a redesign of the system, or if not found, the system being produced with flaws. Other advantages of using formal methods are summarised in [26]. [40] argues that formal verification is necessary in both software development and mathematics. A similar sentiment is expressed in the QED Manifesto[88]; that all mathematical proofs should be machine verified. However, achieving this is proving difficult. Many of the previously mentioned systems use a set theoretic syntax (Z ,B, Event-B) which gives a powerful system for modelling discrete systems. It is important to trust the formal method being used. An approach to ensuring this is to formalise the method itself. For example, B and Event-B and Z have all been formalised [16][72][52] within Isabell/HOL [62].

This thesis is focused on easing the representation of mathematical structures within the Event-B modelling system. Unlike the QED manifesto the aim of defining mathematical theories is for their use in Event-B models. For example, a representation of real numbers can be used in an Event-B model of a physical system to represent time and space accurately. Whilst this thesis focuses on the Event-B modelling language, the approach taken could be applied to other similar set theoretic languages (e.g., Z and B). Event-B was chosen as its system is under continued development and use (e.g., with extensions such as [22] and [77], and use cases such as [90]). Previous work on the Event-B platform has also been done to allow mathematical extensions to the Event-B language (the Theory Plug-in [15]) which provides a good starting point for this work.

The Event-B language is a system modelling language focused on discrete systems; it is used to model physical systems, as well as software systems e.g., [13]. It is supported by the Rodin platform, an integrated development environment (IDE) designed to facilitate the construction of Event-B models. The Rodin platform is designed to be extensible, allowing new tools and language extensions to be added to the Event-B tool set without changing the core Event-B implementation. The ability to write extensions in this manner is valuable to the work in this thesis, and provided further incentive in the choice of working in Event-B environment.

1.2 The Theory Plug-in

A previous addition to the Rodin environment is the Theory Plug-in [15]. This brings to Rodin a method to write mathematical theories. These are not specific to a model and can be re-used in multiple independent models. The ability to define mathematical theories was achieved by adding an extension to the Rodin platform which extended the Event-B language adding the abilities to declare recursive datatypes, operators, and polymorphic types (which can be used within datatypes, operators, and theorems). The addition of operators allows for an axiomatic definition of the reals as has been done in [14]. The addition of recursive datatypes allows the definition of many types without the need to introduce additional axioms (which can cause contradictions). Polymorphic types can be instantiated within Event-B statements e.g., a list can be defined using a polymorphic type (a generic list), we can both define theorems about lists of naturals as well as theorems about the generic list. Theorems in which polymorphic types are used can be instantiated during proofs (the polymorphic type can be replaced with a type from the current context, if a suitable type is available). This allows the use of the theorem without having to re-declare it, or prove it for the current type.

By allowing Rodin users to define abstract and polymorphic types users can do all of the necessary proofs only on the abstract type. Before the Theory Plug-in it was possible to make such structures, but the structures were dependent on the type of the object (they were not polymorphic). Therefore they had to be redefined for each of the different sets where they were used.

The combination of Event-B's modelling and a theorem prover creates a powerful tool. As an example, a theory of two's-compliment arithmetic can be developed within the Theory Plug-in. This can then be used by a Rodin machine when modelling a piece of software. Using two's-complement arithmetic rather than an approximation, such as the integer numbers, would result in the user having to prove that their software specification is consistent with the correct arithmetic. This would include proving that overflow will not occur. This creates a more accurate model of the environment, and removes possible problems when building a specification.

1.2.1 Theorem Reuse in Mathematics

Event-B is a tool for modelling systems, including physical systems. Many physical systems require the modelling of continuous mathematics (e.g., real numbers, continuous functions etc. for modelling time and space). This was seen in [25][10][80]. For example, modelling the real numbers as a field. When looking at a mathematical theory, such as a field, it is useful to examine how it can be defined from simpler mathematical structures. The more complicated theory can then use results (theorems and proofs) from the simpler structures and defining the theory can be easier and faster. As an example of proof reuse, if we want to axiomatically define a field (which could be used as a representation of the real numbers) from scratch then we need to write all of the Abelian group axioms for addition, and write all of them again for multiplication (with the added condition of removing zero). This approach was taken to use Event-B to model the flight paths of unmanned aerial vehicles (UAVs), to ensure paths generated using a machine learning algorithm maintained a safe distance [10][90]. If we have built up mathematical structures instead of defining each of these axioms separately, it would be enough to say that a field is a ring where the multiplicative monoid forms an Abelian group without the additive identity. This definition of a field allows the field to use the results from rings. For instance, given a ring $(R, +, \cdot)$, with '$-$' giving the additive inverse of an element and 0, 1 being the additive and multiplicative identities respectively, the following are theorems:

For all $a, b \in R$:

$$a \cdot 0 = 0 \cdot a = 0 \tag{1.1}$$

$$-1 \cdot a = -a \tag{1.2}$$

$$a \cdot -b = -a \cdot b = -(ab) \tag{1.3}$$

$$-a \cdot -b = ab \tag{1.4}$$

$$1 = 0 \Leftrightarrow R \text{ is the zero ring} \tag{1.5}$$

These results are not difficult to prove, and all of these results are usefully inherited by more complex algebraic structures such as fields and the real numbers. The proofs for these results also rely on properties and proofs on previous structures, e.g., they rely on the uniqueness of identity elements, which can be proved for monoids.

Within mathematics this is described as a conservative extension (an extension of a theory which does not change the theorems that can be proved in the original theory). Within a modelling language having inheritance mechanisms which allow these extensions to be represented is useful. When we find similar structures, we want to be able to define these structures and use any results that apply to these structures. Often it will require the definition of a more abstract mathematical theory (one that only defines the mathematical structure), as in the first example.

Conservative extensions can take several forms. The simplest is where the type is extended with new axioms and operators, i.e., the new type is a subtype of the old type. With the new axioms we should be able to construct new proofs; this results in the proofs from the old type being a subset of the proofs in the new type. An example of this is Abelian groups inheriting from groups, with the additional axiom that the operator is commutative. This results in all Abelian groups being groups. Any proofs about groups can be applied to Abelian groups, however, not all proofs about Abelian groups can be applied to groups. Similarly we can define monoids from semi-groups with the addition of an identity.

New abstract types can be defined using several already existing abstract types. For instance when defining a ring we may want to declare addition and zero as a group and multiplication and one as a monoid. It is desirable for our tool to be able to define types in this manner.

1.2.1.1 Applying Abstract Types to Concrete Types

If we define a concrete theory e.g., defining the naturals (either with a datatype or axiomatically), we will want to be able to prove that parts of these theories conform to some of our abstract mathematical structures. We want to write a theorem saying that part of the theory forms a mathematical structure. Once we have proved this theorem, we want to be able to use the proofs and functions from the abstract mathematical structure. A simple example is defining the natural numbers, and writing a function for addition. We could then prove that zero and addition form a monoid, and use any results from the monoid theorem with the natural numbers.

1.2.1.2 Isomorphisms

Being able to swap between structures isomorphically reduces proof burdens considerably, as demonstrating that two structures are isomorphic would allow us to use proofs interchangeably from other structures.

1.3 Abstraction in Software

The examples in the previous section are focused on mathematics. Event-B developers are interested in this to allow them to better model physical systems. Event-B is also used to model software systems. Abstract types are becoming a common feature of modern programming languages. Examples of this are templates in C++ [83], protocols in Swift [45], and interfaces in Java [34]. These serve two purposes, first many functions can be written on the abstract type, and concrete types can inherit this work. Second, functions can be written directly about the abstract types. Any instance of the abstract type can then be used within this function.

A common example of this is having an abstract type for ordering; we will call *Ordered*. Ordering is a very common operation in software, e.g., for organising data. *Ordered* represents any type with a less than or equals operator ($\leq$). Within the abstract type it is possible to write all other ordering operators e.g., greater than $a > b = \neg a \leq b$ or equality $(a = b) = a \leq b \wedge b \leq a$. These do not have to be re-implemented in the concrete types. Given the *Ordered* type, it is possible to create an ordered list implementation which works for any concrete type that is an instance of the *Ordered* type by writing functions about the abstract type, rather than concrete types.

Being able to model these abstract types better allows us to model modern software languages which have these features.

1.4 Limitations of the Rodin platform

The set theory notation used by Event-B and the Theory Plug-in is capable of representing the desired mathematics (i.e., abstract mathematical types can be defined, and used by concrete types and other abstract types), however, the design of the language means that the user has to do considerable extra work to write these theories.

There is currently a standard library available from the Theory Plug-in wiki.[1] This library is fairly sparse compared to comparable libraries in Coq and Isabelle, and has not been constructed with mathematical extensions in mind. The result of this is that defining a new mathematical theorem will likely mean defining it from scratch. The language used by the Theory Plug-in is also not designed with mathematical extensions in mind, therefore, the user is required to do extra work when writing the mathematical theorems to allow theorems from abstract types to be used with the more concrete types that can be shown to form the abstract structure.

The Rodin platform tries to do all of the proof management for the user (storage, deletion, and some editing when a relied upon theory changes). Whilst this can be helpful it can also lead to unexpectedly losing proofs (and having to re-prove results). There is also no way of scripting the prover; you can not teach it a new tactic for a general style of proof, such as, teaching it how to use an isomorphism to prove a result.

1.4.1 PhD Aims

The general aims of the PhD are to provide support for extension of mathematical definition and proof to avoid unnecessary duplication of effort when developing mathematical theories. Allowing the construction of a significant library of extensible mathematical theories. This can then be used by Event-B modellers, increasing the number of systems that can be modelled, and the ease of modelling.

1.4.2 Contributions of this Thesis

In Chapter 3, a series of Event-B theories are defined which highlight the weaknesses of the current Event-B syntax for creating mathematical structures. A new language is defined which can be compiled to the current Event-B syntax, called $B^\sharp$. This language has dual design aims:

1. To be mappable to the current Event-B syntax used within in the Theory Plug-in.

[1] http://wiki.event-b.org/index.php/Theory_Plug-in

2. To have features to facilitate the definition of abstract mathematical types, and their reuse in concrete types.

To achieve this $B^\sharp$ adds several language features:

1. type classes
2. subtypes
3. unification of functions and operators to a single type
4. unification of predicates and the boolean type
5. methods, allowing functions to use elements from type classes without explicit declaration

A formal definition of the $B^\sharp$ syntax is given, followed by using a pseudo language to show the translation from the $B^\sharp$ syntax to the Event-B syntax.

Building on the formal definitions, an extension to the Rodin platform is created, allowing the development of mathematical theorems in the $B^\sharp$ language, and their translation to the Event-B syntax. This is used to define several mathematical theorems, which can then be used within the Event-B environment. This demonstrates the usefulness of the approach of a mapping step for adding new features in a consistent manner. Many additional features are suggested which could be added to the $B^\sharp$ language or $B^\sharp$ to Event-B mapping to facilitate the definition of mathematical types.

1.5 Structure of this Thesis

Chapter 2 is a literature review, to examine the way that mathematical structures have been built in other theorem provers, and how these provers prove theorems about the structures. In Chapter 3 a series of mathematical types are defined to examine the capabilities of the Theory Plug-in for defining an algebraic hierarchy, and examining how this can be applied to concrete mathematical objects. The Chapter concludes that it is possible to use the Event-B set theoretical syntax to represent these structures, however, the syntax was not designed with features to represent mathematical extensions, which makes representing these structures harder than in other theorem provers. Chapters 4 and 5 are an extension of the work I did in [78]. Chapter 4 proposes that a new language called $B^\sharp$ is created, which compiles to the current Event-B syntax. $B^\sharp$ is introduced by example, followed by a formal definition of the $B^\sharp$ abstract syntax. Chapter 5 gives examples of translation from $B^\sharp$ to Event-B. Chapter 6 introduces a language to describe the mappings between $B^\sharp$ and Event-B. It goes on to show how the core features of the $B^\sharp$ language are mapped to Event-B. Chapter 7 describes how $B^\sharp$ inference works; this is used to make pre-translation steps, where the $B^\sharp$ language is first simplified to its core features before being translated to Event-B. The structure of B$^\sharp$ files is also described. Chapter 8 describes the implementation of B$^\sharp$ and its IDE, the mathematical types developed, and the problems found within the implementation. Chapter 9 describes future work, including many additional features that could be consistently added using the translation approach. Chapter 10 concludes the work.

Chapter 2

Literature Review

To make it easier to define mathematical structures within the Rodin environment it is necessary to look at how these structures have been represented in other environments and to have an understanding of the current Event-B language so that it can be seen how similar notions can be implemented. This section will:

1. give a brief description of other languages relevant to this work
2. look at the features, type systems and proof mechanisms of other languages
3. describe the Event-B type and proof systems
4. describe the Rodin Theory Plug-in component, and how it extends the Event-B language

2.1 Type Theory and Theorem provers

This section gives a brief overview of some of the theorem provers currently available. A more detailed comparison of many provers can be found here [87], which compares the same proof in many different theorem provers (Note Event-B is not included as it was still in development when the book was written). There is also discussion of the type systems used within these theorem provers. For a more comprehensive overview of various type systems see [67].

2.1.1 Higher-Order Logic

Church's simply typed lambda calculus [17] (1940) provides the basis for many of the proof systems used today. Lambda calculus is the logic of functions. The rules of lambda calculus describe function application and equivalence. While functions take a single argument, this does not pose the restriction it initially appears to, as all multi-argument functions can be written as a sequence of functions that take only a single argument (known as currying) [20]. In untyped lambda calculus functions can be applied to any term, although may not produce anything useful. In simply typed lambda calculus every element is given a type, functions can then only be applied to elements of the correct type. There are a series of base types, and a type constructor →, given base types S and T a new type can be constructed $S \rightarrow T$. The type constructor can then also be used on this new type ($(S \rightarrow T) \rightarrow T$) allowing the construction of infinitely many types. A function which can be applied to type T resulting in type S has the type $T \rightarrow S$.

An early work in building a proof system using typed lambda calculus was LCF (logic of computable functions) [58]. This was based on the work of Scott [74] presented by Milner [59]. LCF has terms of the form of typed lambda calculus and formula typed as predicates. The LCF also includes a series of inference rules used within proofs and an outer language for adding axioms and goals to be proved.

Along with the development of the LCF language was the development of the ML functional programming language (ML for meta language) [32], which allowed proof rules to be added in a sound way (being checked by ML's type system). To reason about sets of types ML had a Hindley-Milner [42, 60] type system, which extends simply typed lambda calculus with type variables (parametric polymorphisms), often called generic types.

The HOL [33] language, which was developed from LCF, was originally developed for hardware verification. It has since been applied to other areas, such as an implementation of floating point numbers [39] in HOL Light [41], allowing the verification of functions in floating point libraries. The HOL language is an implementation of higher order logic extended with Hindley Milner style type variables. This means that within HOL predicates are no longer a separate syntactic category to expressions. As specified in [33] the defining features of HOL are as as follows:

1. Variables can range over functions and predicates (hence 'higher order').
2. The logic is typed.
3. There is no separate (from expressions) syntactic category of formulae (terms of type bool fulfil their role).

HOL languages also inherits the Hindley-Milner type systems used in later versions of LCF. A general principal within HOL languages is that the logical core of the language is small, and that all proofs need to go through the small logical core. This approach is facilitated using the ML meta language allowing the addition of additional features safely which are proved using the logical core.

Since the initial development of HOL (HOL88) there have been many descendents of the language and all of these have extensive libraries built on top of the core code. The ones that are still currently in use are:

HOL4 [76] which is the direct descendent of the original HOL system (via several other versions that are no longer in development).

HOL Light [41, 38], an implementation designed to minimise the use of computer resources, at the time allowing its use on standard personal computers. Along with its light core, it introduced several new ways to do proofs that were not available in other HOL systems at the time such as proving theorems in a backward fashion using tactics.

HOL Zero [2], an implementation designed to maximise trustworthiness. It has a simple inference kernel (core), and its pretty printer is designed to print formula in an unambiguous manner ensuring the user proved what they thought they had. The source code for HOL Zero is extremely well documented in an attempt to ensure there are no consistency bugs within the code.

Isabelle/HOL [62] is an implementation of HOL built on the Isabelle platform [66][1], which provides a generic infrastructure for building deductive systems with a focus on theorem proving in higher order logic. The aim is to make developing such systems easier than writing them in pure ML. Several logics have been developed using the Isabelle platform including Isabelle/HOL. Unlike the previously mentioned implementations of higher order logic Isabelle/HOL is not a descendant of HOL88, leading to larger syntactical differences to other HOL systems in both developing and proving theorems. Isabelle/HOL provided additional logic automation (inherited from Isabelle e.g., first order theorem proving tools) that were not available in other HOL systems at the time Isabelle/HOL was developed.

PVS (Prototype Verification System) [64] is a strongly typed higher order logic, with extensions (e.g., predicate subtyping, and parameterisation, which are covered in detail in the section below on language features). In contrast to the HOL family of languages the core of PVS does not attempt to take a minimalistic approach, e.g., the previously mentioned extensions. This increases the chances that there are inconsistencies in the core of the language. However, there are benefits to implementing proof automation and the speed (and usability) of the language.

2.1.2 Martin-Löf Languages

Higher order logic is not the only form of type theory used to in proof systems. Martin-Löf type theory [55] (intuitionistic type theory) , is widely used, including by Coq [8], NuPRL [3] and Agda [11]. Martin-Löf type theory treats propositions as types, and when a proposition is made, its type is calculated. Evidence for the proposition is a member of the type calculated for the proposition i.e., the type calculated for a proposition is the type of its proofs, and any instance of this type is a proof of the proposition. The advantage of this system is that proofs become functions within the language, and these functions can be used as algorithms within a program. For example given:

$$\forall m, n \in \mathbb{N} \cdot \exists p, r \in \mathbb{N} \cdot r < p \land m \times p + r = n$$

[1]The reference manual for which can be found at https://isabelle.in.tum.de/doc/isar-ref.pdf

a proof object of this statement results in a function that given two integers will calculate a p and r such that $m \times p + r = n$.

To allow this it is necessary to have several additional type constructors that are not present in the higher order logic systems, for example a type needs to be given for the proposition:

$$\forall x \in A \cdot B(x)$$

The type given is $\sum(A, B)$, the generalised Cartesian product of $B(x)$ (where $B(x)$ is a new type, for each member $x \in A$). A member of $B(x)$ is a proof function such that for all x in A, $b(x)$ is a proof of $B(x)$.

Within Martin-Löf type theory one can use type variables within expressions. Unlike in Hindley-Milner style type systems these type variables are themselves typed. Any type can not contain itself, which leads to the question of the type of *Type*, the solution is a hierarchy of types, so *Type* has the type $Type_1$ and the type of $Type_i$ is the type $Type_{i+1}$.

Martin Löf type theory diverges from classical mathematics by rejecting the law of the excluded middle i.e., $P \vee \neg P = \top$. It is instead necessary to demonstrate that there is either a proof for P or a proof for $\neg P$. The allows paradoxes caused by statements such as ”this statement is false” to be resolved, as it is no longer required that this statement is either provably true, or provably false. Despite this divergence from classical mathematics large mathematical libraries have been developed such as The CoRn library for Coq [19].

2.1.3 Algebraic Specifications

Algebraic specifications [35] were designed to give descriptions of data types in an implementation independent manner. An abstract datatype is defined with a series of operations and a series of axioms to describe the operations. There is also the notion of sufficient completeness for the axioms, that is, whether or not the axioms provided are sufficient to give semantics to each of the operations of the datatype.

There are many current specification languages e.g., OBJ3 [30], CASL (common algebraic specification language) [4], CafeOBJ [23] (which is more focused towards theorem proving). Algebraic specification languages have been used for a wide range of applications, including generating runnable code from specifications to hardware specifications and verification [29]. There are many parallels between features in modern algebraic specification languages and the previously described languages. Common feature in modern algebraic specification languages (including CASL and those based on OBJ3 (BOBJ [31], CAFEOBJ, Maude [18])) include subtyping (including with multiple inheritance), and parameterised programming [84], where one algebraic specification depends on another object determined by an abstract specification.

2.1.4 Focalize

Focalize [37] is a modelling environment used primarily for modelling software. The Focalize language is constructed from basic building blocks called species. These act in a similar way to type classes in other languages, where a series of elements (signatures) are grouped together with a set, and given definitions, e.g., a species could be defined with a signature for equivalence, this would define a setoid.

Uniquely amongst the languages looked at here Focalize adopts some aspects of object oriented design e.g., species can have methods, if one species inherits from another, methods can be overridden. The result of this overriding is that proofs which use methods need to be invalidated if the method is overwritten. This is handled by the Focalize compiler.

2.1.5 ACL2

ACL2 [44] (A Computational Logic for Applicative Common Lisp) was created with the specific aim of being a programming language [51] where the models constructed in verification projects can be executed. This is achieved by taking a subset of common Lisp [44], axiomatising the primitives and introducing inference rules. This allows programs to be written in the subset of Lisp, and proofs to be made about these programs. Lisp is an untyped language with partial functions. The result of this is that functions can be called with arguments on which they are not defined. To resolve this issue ACL2 introduces the notion of guards which check that arguments are in the domain of a function. Guards have been identified for each of the common Lisp primitives within ACL2. ACL2 has been used in many Software and Hardware verification projects many of which are outlined in [44].

2.2 Language Features

This section will look at language features that are present in the languages in the previous section which facilitate the definition of mathematical libraries.

2.2.1 Datatypes

Datatypes (or recursive datatypes) are a common feature in many languages e.g., HOL4, HOL light, Isabelle/HOL, Coq, etc. Datatype definitions allow types to be defined by a series of generator operators, and any element is only a member of the type if it can be defined by the generator operators. It is necessary to have some base operators (atoms) which do not take an argument, for this definition to work. Datatypes allow proof by induction (specifically structural induction). Early work on data types was presented by Burstall [12] and Hoare [43]. Along with the constructor operators datatypes also have destructors, allowing elements of the datatypes to be split into their constituent parts. Language support for datatypes stops the user from having to explicitly state induction axioms for a given type saving the user work and removing the possibility of making mistakes restating axioms.

The existence of datatypes also allows the writing of recursive functions using pattern matching to match cases of a recursive type. As an example the the naturals can be defined with the following datatype:

```
$Nat \triangleq$
  Constructors:
    $zero$
    $suc(prev : Nat)$
```

Here the constructors are $zero$ and $suc(prev : Nat)$, and the destructor is $prev$. This definition, along with structural equality (two elements are equal if they are constructed using an identical series of constructors) is enough to satisfy Peano's natural axioms (along with the induction principle given by the inductive datatype). Addition can then be defined as a recursive function:

```
$nAdd(x : Nat,\ y : Nat) \triangleq$
  cases[x]
    $zero \rightarrow y$
    $suc(xs) \rightarrow suc(xs\ nAdd\ y)$
```

When x matches $zero$ the result is y. When $suc(xs)$ is matched (where xs is an element in Nat) the result is $suc(xs\ nAdd\ y)$.

Many languages extend the notion of datatypes to (co)inductive datatypes using an approach based on least and greatest fixed points of monotonic functions [65] e.g., Isabelle/HOL, Agda, Coq, BOBJ. Extending datatypes in this way allows the definition of mutually recursive datatypes, and infinite datatypes (given with definitions of co-recursive datatypes). By allowing users to prove operators are monotone rather than having syntactic constraints, a wider range of datatypes can be defined.

2.2.2 Predicate Subtypes and Dependent Types

Predicate subtypes [70] are types which are defined by restricting a type with a predicate. For example, the subtype of integers less than 30 could be declared in the following way:

$$T = \{x : \mathbb{Z} | x < 30\}$$

As noted in [70] and [30] allowing predicate subtypes allows many functions to be specified as a total functions on subtypes rather than as partial functions. In many cases it is possible for the type system to determine whether a call to a function that has subtype arguments is well defined, rather than the user having to manually demonstrate this. Type inference on subtypes is generally undecidable. To allow better reasoning about subtypes, functions require a return type. If the return type is a subtype then the type system may infer a weaker type (a less constrained type) as the result of the function. In this case the system will generate a proof obligation to demonstrate the return type is correct. Predicate subtypes are included in Obj3 [30], PVS [64], and theorem provers which use Martin-Löf type theory.

Martin-Löf based type systems take the notion of predicate subtypes further with dependent types, these allow a type to depend on another variable. Extending the example above we could have a type as follows:

$$T[n : \mathbb{N}] = \{x : \mathbb{N} | x < n\}$$

Here T is a family of subtypes of natural numbers, parameterised by n. This paper [46] describes how dependent types can be translated into higher-order logic. The approach is to define predicate functions to simulate the dependent types, for example $m : T[n : \mathbb{N}]$ is simulated by the function $\lambda m : \mathbb{N}, n\mathbb{N} | m < n$ within theorems this function can then be used in place of the dependent type information. [46] uses this approach systematically, and uses the ML language to add syntactic and proof features to HOL to handle these types.

2.2.3 Type Classes

Type classes are a mechanism by which a set of types can be grouped by the properties of the type. The type class describes the properties that a type needs to have to be a member of the type class. The properties in the type class can be sufficient to prove additional properties of the type class (e.g., given a monoid type class it would be possible to prove the uniqueness of the identity element). Type classes can also be used to constrain parametric polymorphisms (type variables), allowing generic functions to be written about types constrained to members of the type class. For example a class, called Setoid, could be defined (a type with an equivalence relation) and from this it would be possible to write a not-equivalent function:

$$not\_equiv\langle T : Setoid\rangle(a : T, b : T)\ \neg(a\ T.eq\ b)$$

Here the type variable T is constrained to types that have an equivalence relation (e.g., T must be a member of the $Setoid$ type class). The equivalence relation associated with T can then be used (in the example $T.eq$ is used to access this).

Types shown to have the properties of a given type class can use the theorems relating to the type class, and have the constrained parametric functions applied to their elements without further work.

It is worth noting the similarities between type classes and algebraic specification languages such as OBJ3 based languages and CASL, which utilise parameterised programming. Algebraic specification languages view datatypes as many sorted algebras e.g., sets equipped with operations. They provide an abstract specification for these types, while type classes fulfil this role in purely functional languages. Parameterised programming [84] allow one type to depend on another, provided the other type conforms to a specification. Given an algebraic specification of a datatype D another datatype E can be parameterised with (depend upon) this datatype $E[D]$ and within the definition of E properties of D can be referred to. Any datatype that conforms to the algebraic specification of D can be used in the place of D. This resembles type classes specifying properties of types, and being used to constrain parametric polymorphisms. The resemblance is strong enough that the system for type checking OBJ3, order sorted unification [56] can also be used for type inference on type classes [63].

Type Classes were originally proposed by Walder and Blott [85] and have since been used in many languages such as Haskell [50], Coq [79] and Isabelle [36] [61]. It is of interest that the original implementation of type classes in [85] is entirely implemented using a preprocessor on a Hindley-Milner typed language, although it uses features of programming languages (such as strings and dictionaries). Isabelle/HOL has an implementation of type classes added to the language based on order sorted unification [56] [63]. It was later shown that type classes could emerge from higher-order logic [86] demonstrating their soundness in HOL systems.

Type classes also have a first class implementation in Coq, as described in [79], where it is shown that they emerge from Coq's underlying logic with minimal changes. [19] describes an extensive library of abstract mathematical types built using the type classes within Coq.

2.2.4 Partial Functions

Partial functions are functions which are not defined for every value within the type of their domain, e.g., in Event-B a partial functions is written $S \pfun T$ where S and T are sets. A member of this type does not have to be defined for every value of S. In HOL style languages and Martin-Löf languages partial functions are not first class members of the language, instead they need to be modelled within the language. Partial functions appear as first class members of VDM (Vienna Development Method) [48] a specification language which uses the logic of partial functions LPF [7], Event-B [1], B [73] (the formal method from which Event-B evolved) and Z [24] (A formal method based on set theory and first-order predicate logic, closely related to B and Event-B). An approach to modelling partial functions in other languages with total functions is to introduce an option type in the following manner:

$$\textbf{datatype}\langle a\rangle\ option = Some\ a \mid None$$

A partial function that would return a type T would instead return the type $option\langle T\rangle$ with the value $None$ if the arguments to function result in an undefined result. In [71] Event-B's syntax is embedded in HOL using this technique. Another approach is to model a partial function with a similar total function, e.g., passing zero as the result of the division function when the denominator is zero.

As mentioned previously languages which have predicate subtypes e.g., OBJ3 [30] and PVS [64] can take a different approach, it is often possible to rewrite a partial function as a total function on the correct subtype, e.g., in the case of division the denominator could be the subtype of $\mathbb{R}/0$.

2.3 Modelling In Event-B

In order to understand modelling in Event-B it is useful to have a brief overview of how a user models a system using the Event-B language, and how it is demonstrated that the model is consistent. A more in depth introduction can be found in [47] or [1].

This section gives an overview of modelling in Event-B. In order to demonstrate that a model is consistent it is necessary that various properties of the model are proved to be true. The Event-B language does this by generating proof obligations based on the model. As modelling in Event-B is described the proof obligations associated with various parts of the model will be discussed. To aid this discussion it is helpful to know how proofs are reasoned about in Event-B.

2.3.1 Proofs

Proofs constructed in Event-B are done by applying a series of proof rules. These are described here using proof-theoretic semantics [68, 27]. Proof-theoretic semantics is built on units called sequents. When describing a proof there is a goal (a predicate that needs to be proved), and a set of assumptions/hypotheses which are assumed to be true. A sequent expresses this information. Given a goal G and a set of hypotheses H a sequent is written:

$$H \vdash G \tag{2.1}$$

Given the set of hypothesis H, G should be shown to be true.

To demonstrate that a sequent is valid a series of proof rules can be applied. A proof rule is a relationship between sequents. Given a set of sequents **A** (called the antecedent), a sequent **C** (called the consequent) and a name n a proof rule is written:

$$\frac{\mathbf{A}}{\mathbf{C}}\,n \tag{2.2}$$

This reads as, given proofs of the sequents in A, n yields a proof of the sequent C. As an example:

$$\frac{\mathbf{P} \vdash \mathbf{R} \quad \mathbf{Q} \vdash \mathbf{R}}{\mathbf{P} \vee \mathbf{Q} \vdash \mathbf{R}} \tag{2.3}$$

This says that to demonstrate that the hypotheses **P** or **Q** give R it is necessary to show that both **P** and **Q** independently give **R**

To prove a sequent S, a proof rule is chosen with S as the consequent, S is proved when each of the sequents in the antecedents is proved. Some proof rules may have an empty antecedent, for example, given an expression **E**:

$$\frac{}{\mathbf{E} = \mathbf{E}} \tag{2.4}$$

This is an axiom of the system (i.e., it is assumed to be true). A proof of a sequent can then be viewed as a tree of chosen proof rules. If the proof is valid all of the branches will end with an axiom (meaning no more sequents need to be proved). The complete list of Event-B proof rules can be found in [1].

When a proof obligation (generated by an Event-B model) is a sequent. Attempts can then be made to prove this sequent using the Event-B proof rules.

2.4 Model Structure

Event-B models are split into two different categories, contexts and models.

Contexts contain static statements, they define constants and carrier sets (types). The constants and carrier sets can then be further defined by adding axioms using the Event-B mathematical language. For example, a carrier set *COLOUR* can be defined, and the constants *red* and *green* can be made members of the type with the following axiom:

$$red \in COLOUR \wedge green \in COLOUR$$

Defining a type axiomatically in this manner bears similarity to algebraic specifications [35], where datatypes (a set with a set of operations) are described by an abstract specification, as a series of axioms which are implementation independent. A major difference is that Event-B does not have a notion of datatypes, so that functions are not encapsulated within a type but separate entities. In Event-B there are no restrictions on axioms (datatypes in algebraic specifications require the axioms to not contradict each other and to describe the datatype to a level of sufficient completeness, enough to describe meaning to all the operations of the datatype). Event-B does not enforce any form of completeness, or the axioms to be consistent, instead, it is up to the model developer to ensure the axioms are consistent, and that they have enough axioms to complete their proof (each additional axiom should be justified). Note, as the axioms are not checked in any way the user can introduce contradictions e.g., $\mathit{TRUE} = \mathit{FALSE}$ allowing them to prove anything. Often one can use a model checker to find instances satisfying the axioms to demonstrate consistency.

Event-B machines are where the system behaviour is specified. They can see and use constants and carrier sets from contexts. Machines are made up of events, variables and invariants. Events provide the mechanism for state change (variables can be changed to different values, but the type of the variable does not change). Events are made up of the following components:

Parameters: Allow the events to change the state of the machine based on the possible values of parameters

Guards: A series of predicates that specify under what conditions the event can happen, this includes restricting values that parameters can take. These predicate statements can be made up of variables in the machine, context, and the parameters.

Actions: These describe the changes that will apply to variables.

Invariants are predicates that the variables should satisfy in all reachable states. It is required that this is proved (which may require the user to perform manual proof steps). The result is that for every event *evt* a proof obligation is generated for every invariant *Inv* of the form:

$$\begin{array}{l} \mathit{Axioms\ and\ theorems} \\ \mathit{Invariants\ and\ theorems} \\ \mathit{Guards\ of\ the\ event} \\ \mathit{before\text{-}after\ predicate} \\ \vdash \\ \mathit{Inv\ modified} \end{array} \tag{2.5}$$

The proof obligation will be called *evt-Inv*. Here the *before-after predicate* describe state change made by an event, e.g., an event could change a variable $x := x + 1$, the *before-after predicate* would write this as $x' = x + 1$ where x' is the after value of x. *Inv modified* is then *Inv* except with the modified state e.g., if the invariant was $x \in \mathit{Int}$ the *Modified invariant* would become $x' \in \mathit{Int}$. A second proof obligation is generated to check the feasibility of the change made by the *before-after predicates*, i.e., it is checked that given the invariants, theorems, and guards there is a possible valid change of state:

$$\begin{array}{l} \textit{Axioms and Theorems} \\ \textit{Invariants} \\ \textit{Guards of the Event} \\ \vdash \\ \exists v' \cdot \textit{before-after predicate} \end{array} \tag{2.6}$$

In future proof obligations will be described, but not formalised. The formal definitions of proof obligations can be found in [1].

2.4.1 Refinement

Event-B supports refinement of machines. This allows the user to start with an abstract model and incrementally refine it closer to the system the user is trying to model. The refined model can add new variables, extend events, add new invariants, and define new events. The refined model must only act in ways that are consistent with the abstract model. What this means is that events must only occur when the abstract event can occur (guard strengthening). When a concrete event occurs the state change created by the event must be consistent with state change of the abstract event. Concrete machines can see the state of the abstract machines. The concrete machine may want to use new variables to define a more concrete model. To allow this the new state has to be related to the old state (i.e., concrete variables must have a specified relationship to abstract ones). This is done by creating new invariants that describe the relationship between the abstract variables and the concrete ones. These invariants are known as gluing invariants. Concrete events may want to use different parameters to the abstract events. Similarly it may be desirable to change parameters in refined events, to do this whilst maintaining the link to the abstract events predicates need to be added to relate the abstract parameters to the concrete parameters and state. These predicates are called witnesses.

Where necessary the proof obligations are expanded to include the witnesses, and the invariants from the abstract model, e.g., these are added to *evt-Inv* in (2.5). A proof obligation is also added which requires the user to demonstrate the abstract guard is true whenever the concrete guard is. This is an incomplete description of the generated proof obligations. For a complete description see Chapter 5 in [1].

An example from [1] is an island where access is restricted to only a certain number of cars. There is a single lane bridge to the island, cars on the bridge count as on the island. The initial (most abstract) model has the following elements:

1. A constant d for the maximum number of cars (declared in a context, all other elements are defined in a machine).
2. A variable n for the current number of cars on the island (initialised to zero).

3. An invariant $n \leq d$
4. An event to increment cars on the island, ML_IN ($n := n + 1$), guarded with $n < d$
5. An event to decrement cars on the island, ML_OUT ($n := n - 1$), guarded with $n > 0$

This model is then refined to model the traffic on the bridge separately and further refined to model traffic lights on the bridge.

A machine can only refine at most one other machine. The refinement adds two new features to the machines:

1. Relating new variables to old ones: In the example above the first refinement introduces three new variables to model the bridge, The variable a, cars on the bridge travelling towards the island, b cars on the island, and c cars on bridge leaving the island. The gluing invariant is $a + b + c = n$ i.e., the cars on the bridge count as on the island. This is an example of a gluing invariant.
2. Refining Events: Each event in the abstract machine can be refined by one or more event in the concrete machine, refined events in the above example include:
   - cars moving onto the bridge (incrementing a) the direct refinement of ML_IN.
   - cars moving from the bridge to the island (decrementing a and incrementing b)
   - cars moving from the island to the bridge (decrement b increment c)
   - cars moving off the bridge (decrement c) the direct refinement of ML_OUT

These refined events also have updated guards for when they can be applied (e.g., the bridge is only one lane so an invariant is added saying either a or c is zero. An example of a guard needed to prove this is that a cannot be increment if $c \neq 0$). The proof obligations discussed above are generated to ensure that refined events in the concrete machine do not contradict the abstract events they refine, and that gluing invariants are not violated.

2.5 The Event-B Mathematical Language

The Event-B language has a typed set theory syntax. Expressions within the Event-B language are made up variables (members of types), type and set variables, and operators. Operators are typed using generic types, and the operator is constrained to a type pattern, rather than specific types (e.g., the union operator $S \, \mathrm{union} \, T$ requires that S and T are both subsets of the same complete type). In standard Event-B (the Theory Plug-in extends the Event-B language as will be seen in 2.6) it is not possible to define new operators.

The rest of this section explores the Event-B language in more detail, starting by looking at the type system, and then looking at operators available within Event-B.

2.5.1 The Type System

With Event-B there are three ways to define a new type:

1. The introduction of a basic type (carrier set).
2. Using the power set operator $\mathbb{P}$ (this also acts as an operator on sets as expected), e.g., $\mathbb{P}(\mathbb{Z})$ gives the type of sets of integers (including the type of $\mathbb{Z}$).
3. Using the Cartesian product of two types e.g., $\mathbb{Z} \times BOOL$

Expressions written in the Event-B language have a type created by one of these rules. The Rodin platform has a type inference engine, it is often the case that types do not need to be explicitly given as they can be inferred from other constraints.

In the thesis by Schmalz [72] (where the Event-B logic is embedded in Isabelle/HOL) it is argued that this type system "closely resembles" higher order logic as in [33] for the following reasons:

- it is possible to quantify sets and sets of set, similar to the ability to quantify over functions in higher order logic
- the logic is typed
- there is a difference between predicates and expressions, however, as Event-B has a *BOOL* type and a *bool* operator that transforms a predicate into the *BOOL* type e.g., $bool(\top) = TRUE$ [2] with these additions the *BOOL* type can be used to represent predicates, transforming to predicates as necessary (formula can be transformed to the *BOOL* type using the *bool* function, and the *BOOL* type can be transformed back to predicates by appending $= TRUE$) [3]. Resulting in a simple work around for the user to use "predicate" variables.
- Event-B has type variables referred to as carrier sets.

Unlike the type system in higher order logic, Event-B's logic has support for partial functions. This is achieved by the use of well definedness predicates, that state when an operator can be used. Much of the work done by Schmalz is representing Event-B's partial functions within Isabelle/HOL.

The Event-B type system makes Event-B more like HOL style languages than axiomatic set theory. For example, in the early 20th century, Zermelo–Fraenkel set theory (ZF set theory) was developed [91] [75]. ZF set theory overcomes the problems with naive set theory (Cantor's set theory [82]) such as Russell's paradox, by only allowing sets to contain sets excluding themselves. Unlike Event-B, ZF set theory allows the union of any two sets to form a new set (in Event-B this

[2] Notice that the *bool* function is accepting a predicate as an argument, which is notionally not allowed in Event-B
[3] Making predicate variables first class would require considerable extra work on the interactive prover

is only allowed if the elements of the set are from the same type). Von Neumann-Bernays-Gödel (NBG) set theory [28] adds the notion of classes to ZF set theory. This addition is a conservative extension to ZF set theory. Classes allow collections of sets defined by a formula. Every set is a class, any class that is not a member of another class is a proper class. Classes allow reasoning about things bigger than sets e.g., we can have the class of ordinals (well ordered sets), as the class of well ordered sets is well ordered it would contain itself, so cannot be a set.

2.5.2 Operators

We have already seen the Event-B operators that are able to define new types. This section will give a description of the other Event-B operators. Event-B operators are largely inherited from set theory (although there are also a collection of integer operators). The operators can be split into two groups, predicate operators and expression operators. A complete list of all the operators can be found here [69].

Of particular interest to this thesis are the lambda operator and the operators for handling subsets.

The lambda operator allows the creation of functions, which given an argument expand to an expression (like functions in simply typed lambda calculus). Unlike in simply typed lambda calculus, lambda abstractions in Event-B can be partial. The syntax for the lambda operator is:

$$\lambda x \cdot P | E$$

P is a predicate restricting the values of x, E is an expression which can include the parameter x (the lambda expression is only well-defined when $P(x)$ is true, and E is well-defined for x satisfying P). Given a function application $(\lambda x \cdot P|E)(y)$ instances of x in E will be replaced by y with the usual substitutions to stop free variables in E conflicting with y. The type of x is inferred from P and E. If it can not be inferred then the user may need to add more information to the P expression e.g. $\cdots \wedge x \in \mathbb{N}$.

Event-B has an unusual syntax for pairs, to define a pair the $\mapsto$ symbol is used, e.g., a lambda with two parameters would have the following form:

$$\lambda x \mapsto y \cdot P | E$$

The pair operator ($\mapsto$) is left associative so $E \mapsto F \mapsto G$ is $(E \mapsto F) \mapsto G$.

Event-B has a wide range of operators for sets including the operators $\subset$, $\subseteq$ $\in$, $\mathbb{P}(\dots)$. There is also support for set comprehension given a set F and a predicate P, $\{F|P\}$ is the set of all values of F which satisfy P ($\{e_1, e_2, \dots, e_n\}$ can also be used to define sets from a series of elements providing all of the elements have the same Event-B type). Despite many of the same operators working on types and sets in Event-B sets are not types, for example the in the expression $x \in \mathbb{N}$

(x is in the naturals) the type of x will be inferred as as $\mathbb{Z}$ (the integer type) as in Event-B the naturals are a subset of the integers, not a type in their own right. A result of this is the need to add explicit setting predicates e.g., given a function $F \in \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, the following expression will never be provable regardless of the function:

$$\forall x, y \cdot F(x \mapsto y) = F(y \mapsto x) \tag{2.7}$$

As $\mathbb{N}$ is not a type, Event-B's inference mechanism infers the type of x and y to be integers, and the function F is not defined on negative numbers. To resolve this the user is required to explicitly define which subsets elements are members of within the expression:

$$\forall x, y \cdot x \in \mathbb{N} \land y \in \mathbb{N} \Rightarrow F(x \mapsto y) = F(y \mapsto x)$$

An unusual feature of the Event-B language is that operators are not first class entities within the language, so can only be used within expression with their arguments. For instance given a functions $f \in (\mathbb{Z} \times \mathbb{Z}) \times \mathbb{Z} \rightarrow \mathbb{Z}$. Invoking $f(*\mapsto 7)$ would cause a syntax error, as $*$ is the multiplication operator, and would expect arguments. This can be resolved by encapsulating the operator in a lambda abstraction $f((\lambda x \mapsto y \cdot \top | x * y) \mapsto 7)$.

2.5.3 Well-Definedness

When an expression is written in Event-B, a proof obligation is generated to demonstrate that the expression is well defined. This is the conjunction of the well-definedness conditions for each operator used, instantiated with the variables from the equation. For instance, if we have a function f and an expression E, and we write the new expression f(E) the following well-definedness (WD) proof obligation will be generated:

$$E \in dom(f) \land f \in S \nrightarrow T$$

This is generated because the function operator has a well-definedness clause stating that it is only well defined for some subset of S (accessed by the dom operator). Without WD conditions there would need to be some other mechanism to deal with arguments for which the function are not defined e.g., the optionals discussed in 2.2.4.

2.6 The Theory Plug-in Additions to Event-B

The Theory Plug-in [15] is a plug-in to the Rodin toolset to facilitate the creation of mathematical types. It extends the Event-B language to allow the user to define new operators. It also extends the type system with the ability to write datatypes, recursive operators can be written to act on these datatypes. This creates a powerful additional tool in the Rodin toolset. Datatypes allow new types to be defined in a consistent manner without the need for user added axioms on the type. Along with the language features of operators and datatypes, the Theory Plug-in also adds theorems and allows the user to define proof rules. Proof rules are statements that can be used within the interactive prover.

The datatypes added by the Theory Plug-in are very similar to the datatype feature already described earlier in this Chapter 2.2.1 and are not given further description. The rest of this section gives a more in depth description of the other features added by Theory Plug-in.

2.6.1 Operators

Operators can be split into two major groups, axiomatically defined operators and operators that expand to a given expressions (constructive operators).

When defined constructive operators have a set of typed parameters, these parameters are used in a set of expressions. The constructive operator can then be called with a set of arguments (Event-B expressions of the correct type). These arguments are substituted into the operators expression in place of the parameters. Constructive operators can be further split into two definitions recursive and non-recursive operators.

Non-recursive operators have a single expression within their definition, when called with arguments the operator is expanded using substitution to generate an Event-B expression. For example:

$$squared\_nat(x : Nat) \hat{=} x \ ntimes \ x$$

Given an expression E of type *Nat*:

$$squared\_nat(E) = E \ ntimes \ E$$

Operators can take type variables as arguments, allowing the definition of new type operators e.g.:

$$triple(t : \mathbb{P}(T)) \hat{=} t \times t \times t$$

Using the set syntax it is possible to use operators to make dependent subsets, similar to the dependent subtypes of Martin-Löf languages e.g.,

$$\mathit{finiteNat}(n : \mathbb{N}) \mathrel{\hat{=}} \{x \mid x \in \mathbb{N} \land x \leq n\}$$

However, as previously noted this definition will not create an Event-B type.

Recursive operators work by case matching with the constructors of the recursive type. Each constructor of the recursive type has an associated expression. When called with arguments the expression associated with the matched case is expanded. When the matched constructor is atomic (a constructor that has no parameters) the expanded statement cannot reference the operator that is being defined. If the matched constructor is non-atomic (has parameters) the operator being defined can be referenced within the expansion. As an example, consider the following recursively defined operator:

```
$nAdd(x : Nat,\ y : Nat) \mathrel{\hat{=}}$
  cases[x]
    $zero \rightarrow y$
    $suc(xs) \rightarrow suc(xs\ nAdd\ y)$
```

Here *nAdd* is referenced within the definition of *nAdd*, however, it could not have been called in the *zero* case as *zero* is an atomic constructor.

Operators can also be defined axiomatically within the Theory Plug-in. The result of this is very similar to the axiomatic description of functions in standard Event-B.

The Theory Plug-in also adds the notion of a predicate operator, this is an operator that takes a series of expressions and returns a predicate.

Finally the Theory Plug-in adds a conditional operator, given an Event-B predicate P and two expressions e_1 and e_2, the expression $\mathit{COND}(P, e_1, e_2)$ represents e_1 if P otherwise it represents e_2. In effect this is a convenience operator, as this could have been defined as an operator using the *BOOL* type:

$$\begin{array}{l} \mathit{ite}(P : \mathit{BOOL}, e_1 : T, e_2 : T) \\ \quad \mathit{axiom} : \mathit{ite}(\mathit{TRUE}, e_1, e_2) = e_1 \land \mathit{ite}(\mathit{FALSE}, e_1, e_2) = e_2 \end{array}$$

However, the user would have been required to convert arguments from predicates to expressions as necessary. Note that if the *BOOL* type had been defined as a Theory Plug-in datatype with the atomic constructors *TRUE* and *FALSE* the *ite* operator could have been a recursive operator (as described above) on the *BOOL* type (when a datatype only has atomic constructors recursive operators reduce to case based operators) and an axiomatic definition would not have been necessary.

From this point forward when the Event-B language is referred to, it is assumed that it is the Event-B language with the Theory Plug-in extensions.

2.6.2 Theorems

Theorems are named predicate expression which the user can declare. These expressions can then be used in the same way as axioms in future proofs. A proof obligation is generated, the user needs to prove these statements using previously defined operators and theorems. As soon as theorems have been declared they can be used within the proof of future theorems. This allows the user to prove theorems in the order they desire. For example, a user may be in the middle of a proof when they realise they need an additional corollary to ease the proof, this can be declared immediately as a theorem, allowing the user to continue with their proof, then the corollary can be proved afterwards (allowing the user to concentrate on the initial proof).

As expected quantification is allowed within theorems:

$$\begin{aligned}&\forall x,y \cdot x \in List(T) \land y \in List(T)\\ &\quad \Rightarrow list\_length(appendList(x,y)) = list\_length(x)\ pAdd\ list\_length(y)\end{aligned}$$

Here T is a generic type. When this theorem is use within the proof of another theorem, the type T needs to be instantiated with a type available in the context of the theorem being proved. This can be a datatype (e.g., the natural numbers *pNat*) or another generic type. The instantiation type is chosen by the user. It is normally obvious which type to choose. For instance, if the theorem being proved is a theorem about lists of integer numbers, then the integer numbers type should be chosen. Having instantiated the types the theorem becomes a sequent within the current proof. The variables introduced by the universal quantifier can then be instantiated with variables from within the proof context (multiple times if necessary).

2.7 The Rodin Platform

Modelling in Event-B is facilitated by the Rodin [47] tool set. The Rodin toolset is an extensible modelling environment built on top of the Eclipse platform [89]. At its core is the implementation of the Event-B, the proving mechanisms, and Event-B editors. Many additional tools have been created for the Event-B toolset including code generators, additional proof tactics (such as using external provers to automatically discharge proofs), and graphical state machine editors.

There are a variety of Event-B text editors which the user can use to write their models. The editors can have very different forms, this is made possible as Rodin does not save Event-B files in plain text. Instead the Event-B representation is stored as an abstract syntax tree, this can then be displayed and edited in multiple different ways by different editors. The user can change the editor that they use at any point. When an Event-B file is saved the changes are parsed and the Rodin tool generates any necessary proof obligations.

When a proof obligation is generated an attempt will be made by the Rodin tool to automatically discharge it. This is done by the invocation of a series of tactics. A tactic is a set of proof rules.

The tactic can use control flow to apply the rules e.g., they can be applied in a loop. The user can edit the tactics that are automatically applied and the order in which they are applied. Additional proof tactics can be added programmatically. This must be done with care as an inconsistent proof tactic would allow anything to be proved.

There have been several proof tactics added, such as the SMT (Satisfiability modulo theories) proof tactics. SMT solvers work by finding a value that satisfies a series of constraints. If a value can be found that satisfies the given constraints the system is described as satisfiable. The other results that can be obtained are unsatisfiable (there is no value that will solve these constraints) and undecided (no value was found that will solve these constraints). Note that unsatisfiable is also a useful result, to demonstrate that $f(x) < 5$ in SMT we instead show that $f(x) \geq 5$ is unsatisfiable. This type of constraint solving is useful in resolving types and in the case of Event-B well-definedness constraints. There are many SMT solvers available e.g., Z3 [21] and CVC4 [5]. There is also a common language that many SMT solvers use to express constraints called SMT-LIB [6]. The SMT proof tactic in Rodin translates Event-B to the SMT-LIB syntax, and then runs one of the SMT solvers (as selected by the user) to see if the SMT solvers can resolve the proof obligation. This process is designed to make proving easier. This is very different to the purpose of this thesis which is to ease the definition of theories in Event-B by translating from a HOL style language.

If the tactic fails the Rodin tool makes it clear to the user that there are proof obligations that need to be proved. The user can select proof obligations and guide the proof using the interactive provers, which allows the user to manually apply certain proof rules and tactics. Operators have a series of associated proof rules, when these proof rules are available, the operator is highlighted, and the proof rule can be run by clicking on the operator (if there is more than one associated proof rule, the desired proof rule is selected from a list). As a trivial example we may have the following proof obligation:

$$\begin{array}{c} x = y + 1 \\ \vdash \\ y + 1 = x \end{array} \tag{2.8}$$

Within the interactive prover the user can click on the $=$ in the list of hypothesis, and select “Apply equality from left to right” or “Equality from right to left”. This will replace x with $y + 1$ in other expression, completing the proof. A second type of proof rule is available, which allows new hypothesis to be inferred from existing hypothesis, for example given the following hypothesis:

$$\begin{array}{c} f \in S \rightarrow T \\ \blacktriangledown g \in T \rightarrow U \end{array} \tag{2.9}$$

The ▼ can be clicked on by the user to infer the existence of a new total function $h \in S \to U$, which is added to the hypothesis.

Event-B in its standard form does not give a sound way to extend the language with new proof rules, to do this requires changing or extending the Rodin toolset using Java. This has a strong potential to introduce unsound rules, as the rules that are added have no checks other than those made by the person adding the rule. In contrast HOL users can add proof rules using their meta-language (ML) and ML's type system ensures that only logically sound methods can be used when creating proof rules, ensuring their soundness [33] i.e., Due to the type system in ML it is only possible to construct new proof rules from already available tactics. This allows HOL based languages to have small cores which can be extended consistently.

2.7.1 Theory Plug-in Proof Rules

The Theory Plug-in directly adds two new proof rules to the interactive prover, to facilitate proofs with datatypes. These allow case analysis and induction to be used on variables which are instances of datatypes. The user can click on any variable which is a member of a datatype and select which of these to apply. The approach of case analysis is that if you can prove the hypothesis for every different case a variable can take then it must be true. Therefore, the user is required to prove the hypothesis for each of the constructors in the datatype. When induction is selected the user is required to prove the hypothesis for each of the atomic constructors, followed by proving the hypothesis for the non-atomic constructors, unlike case analysis the inductive hypothesis is added to the list of available hypothesis for non-base constructors.

The Theory Plug-in also adds a mechanism for adding proof rules. It allows the addition of two types of proof rules, re-write rules and inference rules. To define a re-write rule the user provides an expression which the interactive prover can match, and the expression that it can be re-written as. This will generate a proof obligation to demonstrate that these two expressions are equivalent. Inference rules allow the interactive prover to make an inference based on the current hypothesis. The difference between these and re-write rules is that the user only needs to demonstrate that the first expression implies the second expression, not that it is equivalent to it.

Declaring proof rules allows the use of the operator's properties within the interactive prover. To define these proof rules we need to declare metavariables. The metavariables for the associative re-writes are as follows:

`Metavariables`

`op`	:	$\mathbb{P}(\mathbb{P}(T) \times \mathbb{P}(T) \times \mathbb{P}(T))$
`a`	:	$\mathbb{P}(T)$
`b`	:	$\mathbb{P}(T)$
`c`	:	$\mathbb{P}(T)$

And the Proof rule states:

```
Given
  $op(a \mapsto op(b \mapsto c))$
When
  $op \in AssocOps(\mathbb{P}(T))$
Rewrite As
  $op(op(a \mapsto b) \mapsto c)$
```

When declaring proof rules in the Theory Plug-in the metavariables must be Event-B types rather than sets. The result of this is that typing must be done in the well-definedness ($when$) part of the proof rule definition. In the proof rule declared above, it may appear that a, b, and c, could be members of different sets, which would mean the proof rule is not true. However, as $AssocOps$ is only well defined when a, b, and c are members of the same set, the proof rule is provable.

2.8 Conclusion

The languages discussed in this section have much in common, particularly in the structure of the languages. In the HOL and Martin-Löf Languages functions/datatypes are declared, then theorems are declared about these. In the case of Event-B the role of functions is replaced by operators. Unlike functions operators are not first class members of the syntax, they cannot be used within expressions without their arguments. This inability to use operators within an expression without its arguments makes it harder to develop theories which make claims about groups of operators, or define operators which take other operators as arguments. The advantage of operators is that proof rules can be written about them, which can be used directly in the interactive prover.

The tools for proving in Event-B work differently from many of the other proof systems. Generally the approach to proving is to provide a proving language, the user needs to learn this language, and write a script to prove theorems. Whereas, the Event-B language has an interactive prover. This allows the user to explore different approaches by clicking on elements of an expression, and seeing which rules can be applied. This saves the user needing to learn a new language, and series of commands. The disadvantage of this approach is that proofs are less reusable. It is difficult to copy and edit relevant sections of a proof, when proving a similar result.

Event-B was designed as a system modelling language, with a strong emphasis on refinement, unlike many of the other languages discussed which were designed as generic theorem provers. The interactive theorem prover was designed to facilitate proofs for these types of systems. The result of this is that Event-B and its theorem prover will not be found on any list of top generic theorem provers.

Unlike the other proving languages, Event-B has a set theoretic syntax at its core (although the other proving languages have set syntax as an emergent feature, i.e., they represent sets using predicates which are true for members of the set they wish to represent, and false otherwise.)

Other proving languages are backed by a meta language (ML language), which allows new features to be added to the languages in a safe and consistent way. In contrast to add new features to the Event-B language these have to be added using Java, and there is no guarantee of their consistency. This feature of a meta language has facilitated adding new features to provers to allow mathematical types to be defined more easily. Examples of this are type classes, co-recursive datatypes, and set theoretic syntax. These features in other languages have allowed the creation of large mathematical libraries. Having a consistent way to add these features to the Event-B environment would allow mathematical types to be more easily defined. These in turn could be used by Event-B modellers to model a more diverse range of systems.

Many HOL style languages have large libraries of mathematical types demonstrating the power of these systems. It is mathematical libraries like these that we would like to make available to the Event-B modeller, to extend the range of systems which can be modelled. Given the similarities between HOL style languages and Event-B discussed in Section 2.5.1 a translation from a HOL style language to Event-B will facilitate the definition of mathematical types usable by the Event-B modeller. Event-B has advantages over HOL in the area of modelling discrete systems, particularly in the area of refinement. It is therefore not desirable to move away from Event-B, a translation of HOL syntax to Event-B would not require this. The benefits of translating Event-B to a HOL style language for the purpose of proof facilitation has already been seen in [72].

Chapter 3

Constructing Mathematical Theories in the Theory Plug-in

3.1 Introduction

The aim of the case study is to look at the capabilities of Event-B and the Theory Plug-in for defining mathematical objects in a hierarchical and heritable manner. This study also looks at ways in which this process could be made easier, both in defining theories, and making proofs about them.

Within this chapter the Theory Plug-in is used to define some basic mathematical structures.[1] In the future these mathematical structures, and the theorems and proofs about them will be described as theories. Within these theories there will be examples of the types of mathematical extension discussed above. Issues which arise when using the Theory Plug-in are also examined, and solutions to these issues demonstrated.

Abstract (and reusable) mathematical types and concrete types are represented. It is shown how the abstract and concrete types can relate to each other. Relating concrete mathematical types via isomorphisms is also looked at, to see how this relationship could reduce work for the user.

3.2 Operators

To define 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 $T \times T \rightarrow T$).

[1]The source code for these mathematical structures can be found in the *Case Study Release Branch* of the following repository: `https://github.com/JSN00k/RodinMathCase-Study`.

With the aim of later being able to represent 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-in is an axiomatic definition:

$AssocOp(x, y) : T;$

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

This defines 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 define 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 defined (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 defining subsets, along with the Theory Plug-in operators in the following way:

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

A generic type parameter 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 defines 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:

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

(*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 3.7.

When proving with the Rodin 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.

3.3 Monoid

This section looks at how we can define the abstract monoid type. The definition of operators is a matter of creating a subtype, e.g., the generic type for an operator was $(T \times T) \times T$, and this is 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 defined in the following way:

$$\begin{aligned} Monoid(t : \mathbb{P}(T)) \triangleq \{ & ident \mapsto op \mid ident \in t \\ & \land op \in AssocOps(t) \\ & \land \forall x \cdot x \in t \Rightarrow op(ident \mapsto x) = x \land op(x \mapsto ident) = x \} \end{aligned} \tag{3.1}$$

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

$$\begin{aligned} & \forall t, ident, op, oe, x \cdot t \in \mathbb{P}(T) \land ident \mapsto op \in Monoid(t) \land oe \in t \land x \in t \\ & \quad \implies (op(oe, x) = x \Leftrightarrow oe = ident) \end{aligned} \tag{3.2}$$

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 3.1 can be further subtyped to define new abstract types. For instance, commutative monoids can be defined with the following statement:

$$CommMonoid(t : \mathbb{P}(T) \triangleq \{CM \mid CM \in Monoid(t) \land prj2(CM) \in CommOp(t)\} \tag{3.3}$$

Here the operator part of the monoid is accessed using the $prj2$ function from Event-B, which gets the second part of a pair e.g., if CM is a pair of the form $ident \mapsto op$, then $prj2(ident \mapsto op)$ gives op. To facilitate writing theorems it is useful to define operators to do this e.g.:

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

Given a monoid this operator will deconstruct it to get the operator part. A similar operator was defined 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.

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 defining 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 defined 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 Theory Plug-in 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-in as a recursive type, to resolve this the next section shows how the natural numbers can be defined recursively within the Theory Plug-in.

3.4 Naturals

Within this section a recursive representation of the natural numbers is defined. 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 define a type and operators to demonstrate how concrete types can use inherit results from abstract types, this is demonstrated by showing that zero and addition form a monoid.

3.4.1 Naturals Definitions

The naturals here are defined with the following datastruture:

```
Nat ≙
  Constructors:
    zero
    suc(prev : Nat)
```

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. The inductive hypothesis does not need to be explicitly stated as this is automatically introduced by the Theory Plug-in datatypes. This along with structural equality gives the datastructure axioms almost identical to Peano's natural axioms (with the exception that Peano's original axioms started at one not zero). Given this I will refer to this representation of the natural as the Peano naturals in the rest of this chapter.

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

$$\begin{aligned}&nAdd(x : Nat, y : Nat) \hat{=}\\&\quad cases[x]\\&\qquad zero \rightarrow y\\&\qquad suc(xs) \rightarrow suc(xs\ nAdd\ y)\end{aligned} \tag{3.4}$$

We also defined 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[2] was discover which required some workarounds. The addition operator defined above (3.4) 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:

$$zero \mapsto nAdd \in Monoid(Nat) \tag{3.5}$$

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:

$$nAdd\_P \hat{=} \lambda x \mapsto y \cdot \top \mid x\ nAdd\ y \tag{3.6}$$

[2]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 define 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.

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 3.5 can be stated as:

$$zero \mapsto nAdd\_P \in Monoid(Nat) \tag{3.7}$$

This is now valid Event-B, and means what one may assume was intended by (3.5) . 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 (3.7) does not automatically bring any of the work forward from the *Monoid* definition to the *Nat* definition. For instance, theorem (3.2) about the uniqueness of the identity is not automatically instantiated in the *Nat*. This theorem is only accessible by instantiating it with $zero$ and $addition$, which would then require a proof that $zero$ and $addition$ form a monoid (which can be done by referencing theorem (3.7)). Rather than do this every time this theorem is needed the user can manually instantiate it with a new theorem:

$$\forall x, y \cdot x \; nAdd \; y = y \Leftrightarrow x = zero \tag{3.8}$$

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).

3.5 Back to Monoids

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

$$\begin{aligned}
&Pow(ident : T, op : AssocOps(T), a : T, p : Nat) \triangleq \\
&\quad cases[p] \\
&\quad\quad zero \rightarrow ident \\
&\quad\quad suc(ps) \rightarrow op(a \mapsto Pow(ident, op, a, ps))
\end{aligned} \tag{3.9}$$

Along with this operator, the *Pow_P* operator was added. This wraps the *Pow* operator in the same way as was done for the *nAdd* operator 3.6. Additional theorems can be written about the *Pow* operator, e.g.:

$$x^p \cdot x^q = x^{p+q} \tag{3.10}$$

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

$$nTimes(x : Nat, y : Nat) \hat{=} Pow(zero, nAdd\_P, x, y) \tag{3.11}$$

This definition of multiplication allows theorems about the *Pow* operator to be easily used. 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 define a function. An example of this is that the power operator on the monoids can be written using a technique called exponentiation by squaring. Instead of calculating a^{13} by multiplying a together 13 times we calculate a, a^2, a^4 (by doing a^{2^2} etc), then multiply together the correct combination of a^{2^n} to get the final result $a^8 \cdot a^4 \cdot a = a^{13}$. This technique is often used within computing as it is considerably faster than simply using continued application of an operator and is simple to implement when numbers are represented in a binary manner (the set bits in the binary number match the a^{2^n}s which need to be multiplied together).

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

1. The alternative definition 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 alternative definition 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 alternative definition. 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.

An operator *sqPow* was defined using exponentiation by squaring, along with *sqPow_P* which wrapped the *sqPow* operator in a lambda (as was done for the *nAdd* operator 3.6). Then the following theorem was defined 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).

3.6 Binary Numbers

Having explored the Peano naturals, we now look at an alternative 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 defines an implementation of the binary numbers, which 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 are resolved 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-in lists are defined using the following datatype declaration:

```
List ≙
  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 defined with the following definition:

$$\begin{array}{l}
bnAdd(x : List(BOOL), y : List(BOOL)) \hat{=} \\
\quad cases[x] \\
\quad nil \rightarrow y \\
\quad cons(xB, xs) \rightarrow if(xB = TRUE \land bnLSB(y) = TRUE) \\
\qquad cons(FALSE, bnIncrement(xs\ bnAdd\ tail(y))) \\
\quad else \\
\qquad if(XB = FALSE \land bnLSB(y) = FALSE) \\
\qquad\quad cons(FALSE, xs\ bnAdd\ bnShiftLeft(y)), \\
\qquad else \\
\qquad\quad cons(TRUE, xs\ bnAdd\ bnShiftLeft(y))
\end{array} \tag{3.12}$$

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 defined 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, but 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.

3.6.1 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 $\forall a, b, c \cdot a \sim b \land b \ \sim \ c \Rightarrow a \sim c$

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

$$Reflexivity(t \ : \ \mathbb{P}(T) \hat{=} \{refl | refl \in \mathbb{P}(t \times t) \land x \mapsto x \in refl\} \tag{3.13}$$

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 define an Event-B function of the form $T \rightarrow Pred$ (note that it is possible to define predicate operators, but, 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 are defined following a similar pattern. Finally an equivalence relation is defined with the following statement:

$$EquivRel(t \ : \ \mathbb{P}(T)) \hat{=} ReflexRel(t) \cap SymmetricRel(t) \cap TransRel(t) \tag{3.14}$$

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

$$\begin{aligned} &\forall t, equ, x, y, z \cdot t \ : \ \mathbb{P}(T) \land equ \in EquivRel(t) \land x \in t \land y \in t \land z \in t \\ &\Rightarrow (x \mapsto y \in equ \land y \mapsto z \notin equ \implies x \mapsto z \notin equ) \end{aligned} \tag{3.15}$$

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

An alternate approach to representing relations would be instead to use the set of functions that map to the Event-B *BOOL* type. E.g., The definition of the reflexive relation would instead be:

$$\{refl \mid refl \in t \times t \rightarrow BOOL \land \forall x \cdot x \in t \Rightarrow refl(x \mapsto x) = TRUE\} \tag{3.16}$$

To change these to Event-B predicates you would use a statement such as $refl(x \mapsto y) = TRUE$. This was the initial approach taken to defining relations, however, this was changed after advice from colleagues within the Event-B community. It was found that the tools used within Event-B proving worked considerably better using the newly suggested approach resulting in less work for the user. Many of the proofs which required manual interaction with the prover in the initial implementation were discharged automatically in the newer implementation.

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:

$$\begin{aligned}&bnEq(x : List(BOOL), y : List(BOOL)) \widehat{=} \\ &\quad nil \rightarrow bnIsZero(y) \\ &\quad cons(xB, xs) \rightarrow xB = bnLSB(y) \wedge xs \; bnEq \; bnShiftLeft(y)\end{aligned} \tag{3.17}$$

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:

$$bnEqSet() \widehat{=} \{x \mapsto y \mid bnEq(x, y)\} \tag{3.18}$$

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

$$bnEqSet \in EquivRel(List(BOOL)) \tag{3.19}$$

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 it is necessary to manually restate these proofs about the *bnEq* operator. Proving them can be made easier using theorem 3.19, 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 defined 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 defined based on structural equality, the result is it is not possible to relate these structures to the previously defined monoid type. To allow this the monoid type would need to be redefined:

$$\begin{aligned}&MonoidEquiv(t : \mathbb{P}) \widehat{=} \\ &\quad \{equ \mapsto ident \mapsto op \mid equ \in EquivRel(t) \wedge ident \in t \wedge op \in AssocOp(t, equ) \\ &\qquad \wedge op(ident \mapsto x) \mapsto x \in equ \wedge op(x \mapsto ident) \mapsto x \in equ\}\end{aligned} \tag{3.20}$$

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

3.6.2 Subtyping for Equality

Rather than defining 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:

```
$bn\_isNormal(a : List(BOOL) \mathrel{\hat{=}}$
    cases[a]
      $nil \rightarrow \top$
      $cons(aB, as) \rightarrow (as = nil \land aB = TRUE) \lor (as \neq nil \land bn\_isNormal(as))$
```

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

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

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

$$\forall x, y \cdot x = y \Rightarrow x\ bn\_Eq\ y \tag{3.21}$$

$$\forall x, y \cdot bn\_isNormal(x) \land bn\_isNormal(y) \Rightarrow (x\ bn\_Eq\ y \Rightarrow x = y) \tag{3.22}$$

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:

$$\forall x, y \cdot x \in NormalBin \land y \in NormalBin \implies x\ bnAdd\ y \in NormalBin \tag{3.23}$$

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, that is, 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 define 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 defined 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 3.7.

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. Proving operators were closed with respect to normality, allows normality to be easily demonstrated throughout expressions. 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.

3.7 Isomorphisms

In this section isomorphisms between the Peano naturals and the binary naturals are defined. 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 define 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 use 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.

We have not tried to define an abstract representation of isomorphisms, although this would be useful work to develop.

3.7.1 Defining the Isomorphism

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

```
$bnToNat(a : List(BOOL)) \hat{=}$
  cases$[a]$
    $nil \rightarrow zero$
    $cons(aB, as) \rightarrow$
      $if(aB = TRUE)$
        $onenAdd(bnToNat(as)\ nAdd\ bnToNat(as))$
      $else$                                          (3.24)
        $bnToNat(as)\ nAdd\ bnToNat(as)$

$bnToBin(a : Nat) \hat{=}$
  cases$[a]$
    $zero \rightarrow nil ⦂ List(BOOL)$
    $suc(xs) \rightarrow bnIncrement(bnToBin(xs))$
```

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

$$\begin{aligned} &bnToNat(bnToBin(x)) = x \\ &bnToBin(bnToNat(x))\ bnEq\ x \end{aligned} \tag{3.25}$$

(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:

$$\forall x \cdot \mathit{bnToNat}(\mathit{bn\_increment}(x)) = \mathit{suc}(\mathit{bnToNat}(x)) \tag{3.26}$$

$$\forall x \cdot \mathit{bnToBin}(\mathit{suc}(x)) = \mathit{bn\_increment}(\mathit{bnToBin}(x)) \tag{3.27}$$

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, thus 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:

$$\forall x, y \cdot \mathit{bnToBin}(x\ nAdd\ y) = \mathit{bnToBin}(x)\ bnAdd\ \mathit{bnToBin}(y) \tag{3.28}$$

Given a bijection it is enough to prove a homomorphism in one direction. If there had been an 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.

3.7.2 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 less complete theory, and then proving the theorems. These proofs all follow the same pattern, which can be demonstrated by showing that addition is commutative (here it is shown 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:

$$\mathit{bnToBin}(\mathit{bnToNat}(x\ bn\_Add\ y)) = y\ bn\_Add\ x$$

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

$$\mathit{bnToBin}(\mathit{bnToNat}(x)\ nAdd\ \mathit{bnToNat}(y)) = y\ bn\_Add\ x$$

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

$$bnToBin(bnToNat(y)\ nAdd\ bnToNat(x)) = y\ bn\_Add\ x$$

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

$$bnToBin(bnToNat(y))\ bn\_Add\ bnToBin(bnToNat(x)) = y\ bn\_Add\ x$$

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.

3.8 Ordering

As described in 1.3, ordering is a common example of the use of an abstract type within computer science. By having a less than operator, we can write functions for other comparison operators and use the less than operator to make other types such as ordered lists. To define the less than relation we first define anti-symmetry:

$$\begin{gathered}AntiSymmetry(t : \mathbb{P}(T)\hat{=}\{anti \mid anti \in \mathbb{P}(t \times t) \land \\ \forall x, y \cdot x \in t, y \in t \Rightarrow x \mapsto y \in anti \Rightarrow y \mapsto x \notin anti\}\end{gathered} \tag{3.29}$$

A less than relation can then be defined as:

$$lt(t : mathbbP(T))\hat{=}AntiSymmetry(t) \cap TransRel(t) \tag{3.30}$$

This can then be used to define other ordering operators, such as greater than or equal:

$$geq(t : \mathbb{T}, op : lt(t), x, y)\hat{=}x \mapsto y \notin op \tag{3.31}$$

The less than operator can be used to write functions which can be used to create ordered lists:

$$\begin{aligned}
&insert(a : T, l : List(T), lessT : lt(T)) \hat{=} \\
&\quad match[l] \\
&\qquad nil \rightarrow cons(a, nil) \\
&\qquad cons(h, t) \rightarrow COND(a \mapsto h \in lessT, cons(a, l), cons(h, insert(a, t)))
\end{aligned} \tag{3.32}$$

Rather than using *cons* to construct lists, *insert* can be used and the created list will be ordered. *insert* also allows the easy definition of a function to order a list by ordering the tail of a list and inserting the head.

This definition of ordering will not work with equivalence relations, to reduce the complexity of the example.

3.9 Additional Features

The main focus of this case study has been the expressiveness of the Event-B language, looking at the structures to allow the sharing of proofs across types. This has been done with an aim on developing tools to easy the definition of these types. The expressiveness of the language is not the only feature that was used within the development of these theorems. This section looks at the other features of the Theory Plug-in that aided in the development of the case studies.

3.9.1 Proof Rules

An extensive use of the Theory Plug-in proof rules feature was used. This included proof rules to help with typing, for example:

$$zero \mapsto pAdd\_P \in Monoid(pNat)$$

This type of proof rule proved very useful when inheriting proofs from the abstract *Monoid* type, as the generated proof obligation would require this knowledge, without this proof rule a theorem would have to be instantiated every time this result was needed (this is extra work for the user).

Proof rules were also used to automatically deconstruct abstract types, for example:

$$mon\_ident(ident \mapsto op) = ident$$

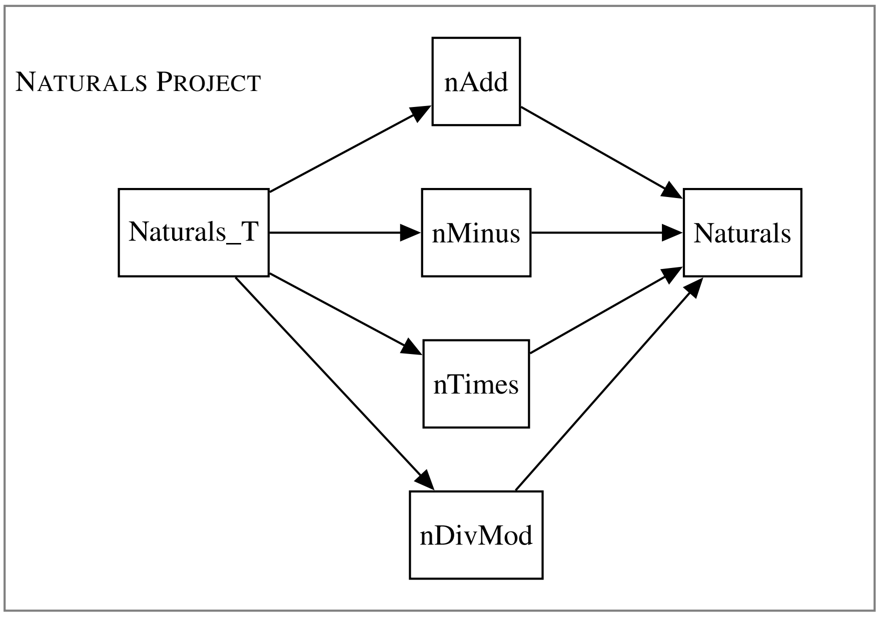
This caused *mon_ident* to automatically expand whenever that was possible again saving the user from having to manually do trivial proof steps. Many theorems that were defined had equivalent proof rules to ease use within the interactive prover, an example of this is:

$$x\ pAdd\ suc(y) = suc(x\ pAdd\ y)$$

the trivialness of this statement is in a sense what makes it so useful, it comes up all the time, so it saves a lot of time not having to reprove the result or instantiate a theorem. Many of the proof rules were initially written as theorems, it was a frustration that they need to be entirely restated to make them become proof rules. There were also many theorems that could not be adequately expressed as proof rules due to restrictions on proof rules, such as the typing of proof rules has to be total Event-B types.

3.9.2 Theory Construction

The theory import mechanism of the Theory Plug-in allows the import of any given theory from a given project. Once a theory is imported every theory which is imported by the imported theory is also imported. This also makes every symbol from all the imported theories visible. The result of this is that operators must be uniquely named, a prefix system was used within this case study. Circularly importing files is not allowed, the system for checking this within the Theory Plug-in is unreliable, so it is up to the user to ensure this never happens. This also results in theories needing to be split in multiple files, for instance, in the case study the naturals need to import the monoids and to make the *Pow* operator, the monoids need to import the naturals, this is resolved by splitting the definition of the naturals and monoids across more than one file. Proof rules declared within a file are only accessible from other files, these are very useful within proofs, this encourages the user to make theory files small so the results within the files can easily be used. To balance these constraints theories were constructed in the following way:



This structure allows the whole of the theory to be imported easily (in this example by importing the *Naturals* file), but also allows smaller files to be developed as part of the theory. There may be several layers in the middle of these structures.

3.10 Lessons learnt From the Case Study

The Theory Plug-in and Event-B language are powerful tools for defining mathematical concepts. The case study demonstrates that the set theoretic approach allows all of the forms of mathematical extension and isomorphism that were set out in Section 1.2.1. As a tool the interactive prover allows the user to explore proving in a way that is not possible in the other languages looked at (which require one to learn an additional language for proving theorems).

Whilst representing the desired mathematics was possible, there are areas where the user was required to do substantial amounts of additional work due to the language design:

1. Rather than being able to use the subset as a type the user is required to use the subset's super type, and then write additional statements to say when the operator/proof rules can be used,
2. Concrete operators can not be passed as a type, causing the user to define a new operator returning a functional representation of the original operator,
3. Operators defined axiomatically can not be instantiated,
4. Demonstrating that a concrete structure is a subset of an abstract structure does not result in theorems and proofs being inherited. Instead it is necessary to re-write the theorems, and prove them (although the proofs are trivial),
5. After demonstrating isomorphisms there is still considerable work to move proof rules and theorems to the new type,
6. Every operator had to be named uniquely. There are cases where, due to accidentally re-using the same name, sections of code need to be redefined and proved.
7. Predicates being a separate syntactic categories created considerably more work in defining and instantiating abstract theories about relations.

All of these problems are solvable within the current Event-B language; however, they require the user to do extra work. Many of these problems are caused by Event-B's type system being relatively weak, and operators not being first class types. Changing the Event-B language is constrained by its more general use in the Rodin tool, and the way the language is interpreted. This makes it difficult to add complex structures to the language; instead, the user needs to build these structures.

Chapter 4

Introducing $B^\sharp$

In the previous chapter several difficulties were highlighted using the current tools and language to define abstract mathematical types. These issues could be resolved by adding additional operators and proof rules, and much of this additional work is entirely mechanical, and therefore could be automated. As a result this chapter proposes a language called $B^\sharp$ (B Sharp). The aim of this language is to make it easier to develop abstract mathematical types which are usable within the Event-B environment. To achieve this $B^\sharp$ will generate Event-B operators, theorems and proof rules. To ensure the consistency of the generated Event-B, proofs will be done on the generated Event-B using the current tools (rather than a new system for proving directly on $B^\sharp$).

This chapter introduces the core of the $B^\sharp$ language initially by demonstration. It goes on to give a formal definition of the language. Finally other features of the language are introduced, and discussed e.g., file and project structures.

4.1 $B^\sharp$ by Example

4.1.1 Datatypes

$B^\sharp$ allows the definition of types via datatypes, which are very similar to the datatypes of Event-B. Here is an example of the definition the Peano naturals:

$$\begin{aligned}&\textbf{Datatype}\ \mathit{pNat}\\ &\quad \mid \mathit{zero}\\ &\quad \mid \mathit{suc}(\mathit{prev} : \mathit{pNat})\ \underbrace{\{\ldots\}}_{\text{Type Body}}\end{aligned} \tag{4.1}$$

The primary difference between this and an Event-B datatype is the “Type Body” part of the expression. Within the “Type Body”, functions and theorems related to the datatype can be declared. This works as an import and namespacing mechanism. The *pNat* type can be imported by other B$^\sharp$ files, then *zero* and *suc* (and any other functions) within the *pNat*'s type body are available within these files. If there is a name clash e.g., another type such as the integers has also been imported, the clash can be resolved by prefixing, e.g., instead of using *zero*, *pNat.zero* would be used.

As in Event-B datatypes can introduce parametric types, for example, the definition of a list is:

$$\begin{array}{l} \overbrace{\textbf{Datatype}}^{\text{keyword}}\ \overbrace{List}^{\text{name}}\ \underbrace{\langle T\rangle}_{\text{parametric context}} \\ \text{constructors}\left\{\begin{array}{l} \mid nil \\ \mid cons(\underbrace{head : T, tail : List\langle T\rangle}_{\text{destructors}}) \\ \end{array}\right. \\ \underbrace{\{\ldots\}}_{\text{Type Body}} \end{array} \tag{4.2}$$

Again this is very similar to an Event-B datatype declaration. A list can be constructed with either the *nil* constructor, or the *cons* constructor. A list constructed with *cons* can be deconstructed with the *head* and *tail* destructors (undoing the *cons* constructor).

4.1.2 Functions

Functions can be declared within a type body. For instance:

$$\overbrace{add2}^{\text{function Name}}\ \underbrace{(x : pNat)}_{\text{arguments}} : \overbrace{pNat}^{\text{return type}}\ \underbrace{suc(suc(x))}_{\text{Function Definition}}$$

declares a function that adds two to any naturals.

A function definition can appear anywhere within the body of a type, but is not allowed to be declared outside of a type body. This is used for namespacing in the same way as with the datatype constructors.

The return type of a function generates a proof obligation to show that, given the types of the function arguments, the type returned by the function is the return type e.g., for the function above the following proof obligation is generated:

$$\forall x : pNat \cdot add2(x) \in pNat \tag{4.3}$$

In cases with complete types (as above) this will be trivially proved by type-checking. However, in cases with subtypes (e.g., all the naturals less than ten), manual proof steps may be required.

Functions allow both direct and recursive definitions, for example, the function for addition is defined recursively as follows:

$$\begin{aligned}
&add(x, y : pNat) : pNat\\
&\quad \textbf{match}\ x\ \{\\
&\qquad |\ zero : y\\
&\qquad |\ suc(xs) : suc(xs\ add\ y)\\
&\quad \}
\end{aligned} \tag{4.4}$$

B♯ functions only support primitive recursion, as this is the only recursion supported by the Theory Plug-in. The focus is on adding type class like features to the Rodin environment and demonstrate that additional features can be added through a translation phase. In the future, it should be possible to use the translation phase to add support for general recursive functions, similar to the way it was done in Isabelle/HOL [53]. Recursive functions in the Theory Plug-in must produce results all of the same type. The only addition to this added by B♯ is the user is required to prove that recursive functions produce the return type specified by the user (which may be a subtype so not checked by the Event-B) type checker.

B♯ functions can introduce parametric types to be used within the function, for instance:

$$\begin{aligned}
&curry2\ \overbrace{\langle T\rangle}^{parametrictypes}\ (f : T \times T \to T, x : T) : T \to T\\
&\quad \lambda y : T \cdot f(x, y)
\end{aligned} \tag{4.5}$$

This function simulates currying for functions with two arguments. As functions are first class members of the B♯ language previously defined functions can be used as arguments to the *curry*2 function, for example the *curry*2 function could be called like this:

$$curry2\langle pNat\rangle(add, suc(suc(zero))) \tag{4.6}$$

generating an alternative to the *add*2 function. The condition that a function has to be declared within a type body seems restrictive for the *curry*2 function, however, the class bodies are also used for namespacing (see 4.3). It is therefore still necessary that *curry*2 is declared in a type body.

4.1.3 Class Declarations

Class declarations allow reasoning about sets of a generic type. For example the following **Class** declaration allows reasoning about the set of all reflexive relations:

$$\overbrace{\textbf{Class}}^{\text{keyword}}\,\underbrace{\mathit{ReflexRel}}_{\text{name}}\;\overbrace{\langle T\rangle}^{\text{parametric types}}\;\underbrace{[r]}_{\text{Instance Name}}\;:\;\overbrace{T\times T\to \mathit{Bool}}^{\text{supertypes}}\;\underbrace{\textbf{where}}_{\mathit{keyword}}\;\overbrace{\forall x : T\cdot r(x,x)}^{\mathit{constraints}}\underbrace{\{\dots\}}_{\text{type body}} \tag{4.7}$$

This statement declares a new class called *ReflexRel*, the class of all relations (defined by the supertype) where all elements are related to themselves (defined in the constraints section). The 'Instance Name' gives the name of a generic instance of the type, used within the **where** statement and the type body. This can be seen in the constraints of the above declaration, where r is an instance of the *ReflexRel* type.

B$^\sharp$ also allows multiple supertypes, for instance, the declaration of an equivalence relation is:

Class $\mathit{EquivRel}\langle T\rangle[e] : \mathit{ReflexRel}\langle T\rangle,\ \mathit{SymmetricRel}\langle T\rangle,\ \mathit{TransRel}\langle T\rangle$ {
  Theorems {
    TransInverse:

$$\forall x,y,z : T\cdot e(x,y)\wedge\neg e(y,z)\Rightarrow\neg e(x,z); \tag{4.8}$$

    TransRewrite:

$$\forall x,y,z : T\cdot e(x,y)\Rightarrow(e(x,z)\Leftrightarrow e(y,z)); \tag{4.9}$$

  }
}

This says that an equivalence relation is a reflexive, symmetric and transitive relation.

This example also shows how to add theorems to a type body. Theorems are declared within a **Theorems** block, and class bodies can have multiple theorem blocks. Theorems are named allowing them to be easily identified during proving. Within the theorems the generic instance e from the class declaration can be used. Having a generic equivalence relation means that the equivalence relation does not have to be explicitly quantified over in every theorem, rather, any theorem that references the instance name will implicitly quantify over a generic equivalence relation. As the theorems are declared in the *EquivRel* type it is expected that they are about equivalence relations.

B$^\sharp$ also allows quantifiers to have a parametric context, the result is that the above theorem (4.8) could have been written as follows:

$$\forall\langle T, eq : \mathit{ReflexRel}\langle T\rangle\rangle\ x,y,z : T\cdot eq(x,y)\wedge\neg eq(y,z)\Rightarrow\neg eq(x,z) \tag{4.10}$$

This makes the quantification over the type explicit, at the expense of removing the focus from the theorem expression.

4.1.3.1 Required Elements

Along with declaring reasoning about sets of types, the class statements also allow reasoning about sets which have "*required elements*". As an example here is a *Setoid* declared within B$^\sharp$:

$$\textbf{Class } \mathit{Setoid}\langle T\rangle[S] : \mathbb{P}(T)\,\overbrace{(\mathit{equ} : \mathit{EquivRel}\langle S\rangle)}^{\text{require element}}\{\} \tag{4.11}$$

This is the set of all sets which have an associated equivalence operator. The associated elements are declared in the required elements. For a type to be a member of the *Setoid* class it is required to have an associated equivalence relation represented by *equ*. Classes which have required elements are referred to as type classes, as they behave in a similar way to type classes in other languages, e.g., Coq [8], Isabelle/HOL [62], Haskell [50].

Within B$^\sharp$, it is possible to restrict generic types to members of the class. As an example, we can define an operator based on equivalence, rather than equality:

$$\begin{aligned} &\textbf{Class } \mathit{baseOp}\langle T : \mathit{Setoid}\rangle[bOp] : T \times T \to T \\ &\qquad \textbf{where } \forall x, y, z : T \cdot T.equ(x, y) \\ &\qquad\qquad \Rightarrow T.equ(bOp(x, z), bOp(y, z)) \land T.equ(bOp(z, x), bOp(z, y))\{\} \end{aligned} \tag{4.12}$$

Here T must be a member of the *Setoid* type class, the equivalence relation associated with T is accessed using *T.equ*. As can be seen within the **where** statement. The above definition defines the class of total functions, such that when they are supplied with equivalent arguments (based on T's equivalence relation), they evaluate to equivalent results.

From *baseOp* more constrained operators can be defined, e.g., a commutative operator:

$$\begin{aligned} &\textbf{Class } \mathit{CommOp}\langle T : Setoid\rangle[cOp] : \mathit{baseOp}\langle T\rangle \\ &\qquad \textbf{where } \forall x, y : T \cdot T.equ(cOp(x, y), cOp(y, x))\{\} \end{aligned} \tag{4.13}$$

Here *baseOp* is instantiated with the polymorphic type T. To do this T must be a *Setoid*, T is declared this way as a parametric type ($\langle T : Setoid\rangle$).

As with the classes previously seen, type classes can inherit from previously declared type classes:

$$\begin{aligned} &\textbf{Class } \mathit{Monoid}[M] : SemiGroup\,(ident : M) \\ &\qquad \textbf{where } \forall x : M \cdot equ(op(x, ident), x) \land equ(op(ident, x), x)\{\ldots\} \end{aligned} \tag{4.14}$$

The supertype of the *Monoid* class is a *SemiGroup* which means that the *Monoid* class is a *SemiGroup* which additionally also requires an associated element, *ident*. Type classes can use the required elements from their supertypes, in this case an associative operator *op*, from the *SemiGroup* and *equ* inherited from *Setoid* (the supertype of the *SemiGroup*). This can be seen in the **where** statement.

Within the body of the *Monoid* class there are several theorems declared and proved about the uniqueness of the identity. Within these theorem declarations the elements of the type classes can be used e.g.:

$$\forall x : M \cdot equ(op(x, ident), ident) \Leftrightarrow equ(x, ident) \tag{4.15}$$

Quantification is implicit for classes with required elements.

As with theorems, functions can use the elements from the type class. When a function does reference a member of the type class it will be referred to as a method. For example, within the monoid class the following method is declared:

$$\begin{aligned}
&raiseToL(x : M, p : pNat) : M\\
&\quad \textbf{match}\ p\ \{\\
&\qquad |zero : ident\\
&\qquad |suc(ps) : op(x, raisetoL(x, ps))\\
&\quad \}
\end{aligned} \tag{4.16}$$

This method makes reference to the *ident* and *op* elements of the monoid class. In this case the return type M is not a complete type, so the proof obligation about return types will need to be proved manually (using an inductive step on p). These method and theorem declarations can be equivalently written using a function with a polymorphic context:

$$\begin{aligned}
&raiseToL\langle M : Monoid\rangle(x : M, p : pNat) : M\\
&\quad \textbf{match}\ p\ \{\\
&\qquad |zero : M.ident\\
&\qquad |suc(ps) : M.op(x, raisetoL\langle M\rangle(x, ps))\\
&\quad \}
\end{aligned} \tag{4.17}$$

When methods are used within the type class that declared them (or one of its subtypes) it is not necessary to make explicit reference to the type class, e.g., $\forall x : M, p : pNat \cdot raiseToL(x, p) \ldots$ is a valid call. Outside the type body of the monoids, the type needs to be provided explicitly e.g., $raiseToL\langle T\rangle(x, p)$ where T must be a monoid.

Whilst this function can be used in the same way as the method definition, when a concrete type becomes an instance of the Monoid type class the method would be inherited by the concrete type, if declared in the functional way it would not be.

Given a generic instance of a monoid M the method can be accessed using $M.raiseToL(x, p)$. If we are in the body of the monoid declaration or one of its subtypes then the *M.* can be dropped as this will be inferred. This declaration is equivalent to calling $raiseToL\langle M\rangle(x, p)$ on the functional declaration.

4.1.4 Available Types in B$^\sharp$

Class and **Datatype** declarations, and type constructors are the only ways in which types can be added within B$^\sharp$. This does not result in a restriction over the current Event-B. In Event-B types are declared and then axiomatically defined. Types like this can be created in B$^\sharp$ using the *Class* declaration. This is achieved by subtyping a generic type and adding required elements and *where* constraints. Using the required elements we can associate elements and functions with a type. Examples of this were seen in the *Monoid* declaration 4.14, where the identity element was added. Within the *SemiGroup*, an associative operator would have been added to the required elements using this statement $op : AssocOp(SG)$ (where *SG* is the instance name of the *SemiGroup* type. The **where** statement can then be used to add axioms (beyond those made by typing) to the type and its elements.

The **Class** declaration can also be used to define functions between types. As an example:

$$\begin{aligned}&\textbf{Class}\ STo TBaseFunc\langle S : Setoid, T : Setoid\rangle[bOp] : S \rightarrow T\\ &\quad\textbf{where}\ \forall x, y : S \cdot S.equ(x, y) \Rightarrow T.equ(bOp(x), bOp(y)\{\}\end{aligned} \tag{4.18}$$

This also allows for axiomatic definitions of functions, using the **where** conditions. An example would be a function between two *Monoids* where the function must map the identity of one monoid to another.

Finite types can be created using a **Datatype** with only base constructors, e.g.:

$$\begin{aligned}&\textbf{Datatype}\ Colour\\ &\quad \mid RED\\ &\quad \mid GREEN\\ &\quad \mid \ldots\\ &\quad \{\ldots\}\end{aligned} \tag{4.19}$$

Declaring finite types in this manner allows functions to be written using case matching. An alternative approach would be to use a **Class** declaration to define a finite type axiomatically. This approach has the disadvantage of meaning that functions also have to be axiomatically defined, and it is harder to do case analysis in proofs (a tactic automatically available on datatypes).

New types can be created by restricting datatypes, for example:

$$\textbf{Class } \mathit{ModArith}[\mathit{MA}] : \mathit{pNat} \textbf{ where } \forall x : \mathit{MA} \cdot \mathit{lessThan}(x, \mathit{suc}(\mathit{suc}(\mathit{suc}(\mathit{suc}(\mathit{suc}(\mathit{zero})))))) \{\dots\} \tag{4.20}$$

This creates the type of natural numbers less than 5 (The *lessThan* is assumed to be a function that is true if the first argument is numerically less than the second). Functions can then be written on this to create the naturals modulo 5.

We can use B$^\sharp$ to define types with an ordering relation:

$$\textbf{Class } \mathit{Ordered}[O] : \mathit{Setoid}(\mathit{lt} : \mathit{ltRel}(O)) \tag{4.21}$$

This commonly occurs within software engineering to allow sorting of values and data. This type can be used to define other ordering functions such as greater than. For example, we can define an ordering operator on the naturals and show that it is a member of this type class to make the other ordering operators available within the naturals.

Another possible type would be an ordered list. To achieve this, a function to check the order of a list would first have to be defined:

$$\begin{aligned}
&\mathit{isOrdered}(l : \mathit{List}\langle T\rangle, \mathit{lt} : \mathit{ltRel}(T)) : \mathit{Bool}\\
&\quad \textbf{match } l\{\\
&\qquad \mid \mathit{nil} : \top\\
&\qquad \mid \mathit{cons}(h,t) : \mathit{lt}(h, \mathit{head}(t)) \wedge \mathit{isOrdered}(t)\\
&\quad \}
\end{aligned} \tag{4.22}$$

Note that a *COND* statement checking the tail is not empty in the *cons* match statement has been omitted for brevity. The type of ordered lists would be created with a class declaration in the following way:

$$\textbf{Class } \mathit{OrderedList}[\mathit{OL}] : \mathit{List}\langle T\rangle \textbf{ where } T \in \mathit{Ordered}, \mathit{isOrdered}(\mathit{OL}, T.\mathit{lt}) \{\dots\} \tag{4.23}$$

Functions for inserting and merging ordered list could then be added.

This class could then be used in the creation of a model of units:

$$\textbf{Class } \mathit{Units} : \mathit{pNat}\ (l : \mathit{OrderedList}\langle \mathit{pNat}\rangle)\{\dots\} \tag{4.24}$$

Each value in the ordered list represents a different unit. Addition and multiplication functions could then be defined such that addition works when the lists are equal, and multiplication multiplies the values, and merges the lists. This example could be further extended by having a rational type for numbers and fractional type of ordered lists (an ordered list for both the numerator and denominator).

An effect of the translation to Event-B (discussed in Chapter 6) is that B$^\sharp$ classes become Event-B sets with axioms about their elements. The result of this is that if you make a **Class** declaration with contradictory axioms, the definition of the set will only be satisfiable with the empty set. This is in contrast to Event-B axiomatic definitions where contradictory axioms cause a consistency problem in the whole system.

The ability to axiomatically define types, and create inductive datatypes, means that any theories that can be defined in Event-B can also be defined in B$^\sharp$. The lack of co-recursive datatypes in Event-B is carried forward into B$^\sharp$; the result of this is that without first constructing a theory of co-recursive datatypes B$^\sharp$ is not able to represent infinite co-recursive types such as streams.

4.1.5 Instances

Instances allow concrete types to become members of type classes, this allows the type to be used in the cases where the polymorphic declaration is constrained to a type class. For example, in B$^\sharp$ we can assert that *pNat*, and equality (=) as the equivalence relation satisfy the constraints of the *Setoid* class, with the following statement:

$$\textbf{Instance}\ \mathit{Setoid}\langle \mathit{pNat}\rangle(=) \tag{4.25}$$

The theorems of the $Setoid$ class can are then inherited by the *pNat* type.

The instance statement itself makes equality the equivalence relation associated with the *pNat* type, changing the *pNat* type. The *pNat* type can now be used within B$^\sharp$ as a *Setoid* e.g., we saw in equation 4.13 *baseOp* instantiated with T where T was declared as a member of the *Setoid* class. Due to the instance statement above 4.25 *pNat* can be used as a $Setoid$ in the same way i.e., $\mathit{baseOp}\langle pNat\rangle$. A proof obligation is generated with the following form:

$$[=] \in \mathit{EquivRel}\langle \mathit{pNat}\rangle)$$

(In B$^\sharp$ an infix function is wrapped in square brackets to make it act identically to an prefix function.) This particular proof obligation is proved automatically in Rodin.

In many cases it is not desirable to make a change to the type, for example, there are several monoids on the natural numbers e.g., zero and addition, and one and multiplication both form monoids. It is not clear that either of these should be the default monoid. To resolve this B$^\sharp$ has the notion of a named instances as can be seen below in (4.26).

An **Instance** declaration instantiates the theorems from the type class into the conforming class, this makes finding the theorem, and using it considerably easier within the interactive prover (as it is now a theorem on the expected type). As an example:

$$\textbf{Instance } \mathit{CommMonoid}\langle \mathit{pNat}\rangle(add, zero)\ \overbrace{\mathit{addMon}}^{\text{Inst Name}}\ \overbrace{(times = \mathit{raiseToL})}^{\text{method renaming}} \tag{4.26}$$

defines an instance called $addMon$ that can be used wherever a member of the $CommMonoid$ type is expected (or one of its super types). As an example of theorem instantiation, a theorem with the following form is defined from theorem (4.15):

$$\forall x : pNat \cdot add(x, zero) = zero \Leftrightarrow x = zero \tag{4.27}$$

Methods are also instantiated, and the instantiated methods can be renamed. This is what the 'method renaming' section does, above this in effect creates the following function declaration:

$$\begin{aligned}
&times(x : pNat, p : pNat) : pNat\\
&\quad \textbf{match}\ p\ \{\\
&\qquad |zero : zero\\
&\qquad |suc(ps) : add(x, times(x, ps))\\
&\quad \}
\end{aligned} \tag{4.28}$$

Methods which do not have an entry in the method renaming section are accessed with a combination of the instance name and their original name as such $instName.funcName$, e.g., the instance of the *Monoid* $raiseToR$ can be accessed as $addMon.raiseToR$, as no specific renaming was given.

4.2 Extensions

At the top level of a file **Datatype**s and **Class**es can be declared, and other statements appear within the type bodies of these statements. This system on its own is too restrictive, as it does not allow types to mutually reference each other (types can only reference types declared above their current position). This can be seen as a problem in the *pNat* and *Monoid* example. To declare the $raiseToL$ operator the *Monoid* type has to know about the *pNat* type, for add and $zero$ to become an instance of the *Monoid* type class the *pNat* type needs to know about the *Monoid* class. This is resolved by the **Extend** syntax:

$$\begin{aligned}
&\textbf{Extend}\ pNat\ (monoids)\{\\
&\qquad \textbf{Instance}\ CommMonoid\langle pNat\rangle(add, zero)\ addMon\ (times = raiseToL)\\
&\qquad \ldots\\
&\quad\}
\end{aligned} \tag{4.29}$$

An **Extend** statement allows the original type body to be extended, statements within the type body of the extend statement act like they are in the type body of the extended type (e.g., in this case they act like they are in the type body of the *pNat* type). They can also reference classes/datatypes that have been declared above them. This resolves the circular referencing problem.

4.3 Naming, Scope, and File Structure

Up to this point it has been assumed that all B$^\sharp$ has been declared in a single file, and that all elements such as functions and classes have been uniquely named. However, the language is designed to allow development across multiple files, and allow certain names to be reused.

4.3.1 Scope

B$^\sharp$ contains the notion of scoping. This goes some way to allow names to be reused. Whilst this has not been made explicit, the notion of scoping has been used in previous examples, e.g., the *EquivRel* theorems (4.8) the class instance name can be used, because the theorems are in the scope of the class.

B$^\sharp$ contains three levels of scope these are:

1. function/lambda/quantifier scope, variables introduced in functions/lambdas/quantifiers must be unique from each other and are only available within the associated expression (this follows the same rules as Event-B).
2. Class scope parameters introduced as required elements are available anywhere in the classes type body, the type body of subtypes, and the type body of extensions.
3. File scope, all class names, datatype names, constructor/destructor names and function names are available for use anywhere within the file.

4.3.2 Naming

In general it is important that two elements in the same scope do not have the same name, this is also in general advantageous to the user, and anyone else looking at the developed theories e.g., it would be confusing to have two classes both named *Monoid* even if the language could disambiguate them based on context. As such the general rule is that names within a given scope must be unique. $B^\sharp$ functions, constructors and destructors have different scoping rules to other elements. They are declared within Class scope, and within the scope of a class they must be named uniquely, however, they can be used within the File scope. For example, the following theorem is declared in an extension to the *Monoid* type:

$$
\begin{array}{l}
\textbf{Extend}\ \mathit{Monoid}\ (\mathit{raiseTheorems})\{\\
\quad \textbf{Theorems}\{\\
\qquad \mathit{RaiseToL\ add\ Rule}\ :\\
\qquad\quad \forall x\ :\ M, p, q\ :\ pNat \cdot op(raiseToL(x,\ p), raiseToL(x, q)) = raiseToL(x,\ add(p, q));\\
\quad \}\\
\}
\end{array}
\tag{4.30}
$$

The *add* function from the *pNat* type is used within this theorem despite the theorem being in a different scope to the scope of the *pNat* type (where *add* is declared).

This causes a problem where more than one type may have a function with the same name, e.g., the rational numbers will also have an *add* function. This is resolved by functions having full names, the full name of the *add* function is *pNat.add*, *add* is an abbreviation of this, and when there is only one *add* function in scope the abbreviation can be used. When there are two functions with the same abbreviated name in scope, an error is given and the full name is asked for.

4.3.3 Imports and Project Structures

In $B^\sharp$, code is split into packages, then files. Packages are achieved using the file structure, and have the name of the folder that contains all of the related files. For example, there could be a *Monoid* folder which would contain the files:Monoid.bs, *CommMonoid.bs*, *SemiGroup.bs* etc. Splitting the $B^\sharp$ code up this way makes it move navigable, and easier for the user to find the classes that they want.

The notion of imports is used to bring elements from the scope of another file into the current file. An **Import** statement imports a specific top level element, for example:

$$\textbf{Import}\ \overbrace{\mathit{Monoid}}^{\text{package}}.\underbrace{\mathit{Monoid}}_{\text{file}}.\overbrace{\mathit{SemiGroup}}^{\text{Top Level Element}} \tag{4.31}$$

This brings the *SemiGroup* type and all functions declared within its type body into the current scope everywhere below the **Import** statement.

To understand a full example, we can imagine that the definition of *pNat* is now in its own file, in a project called *Naturals*. Now the *raiseTheorems* extension (4.32) theorem declared in a *Monoid* file will not automatically have access to the *pNat* type, or addition so it will need to be imported in the following way:

$$
\begin{array}{l}
\textbf{Import}\ \mathit{Naturals.pNat.pNat}\\
\textbf{Extend}\ \mathit{Monoid}\ (\mathit{raiseTheorems})\{\\
\quad \textbf{Theorems}\{\\
\qquad \mathit{RaiseToL\ add\ Rule}\ :\\
\qquad\quad \forall x\ :\ M, p, q\ :\ pNat \cdot op(raiseToL(x,p), raiseToL(x,q)) = raiseToL(x, add(p,q));\\
\quad \}\\
\}
\end{array}
\tag{4.32}
$$

The **Import** statement makes the *pNat* class available, and any functions declared within the type body of the *pNat* class (such as *add*).

4.4 B$^\sharp$ Language Formal Presentation

This section will formalise the syntax of the language introduced in the last section. The syntax is formalised in an abstract manner i.e., syntactical separators such as keywords, and commas are not included. Having this formalisation will later allow the translation to Event-B to be described. As many of the language definitions are mutually dependent, when introducing a language feature it is often necessary to reference a feature that has not yet been described (e.g., functions reference expressions, and expressions reference functions, and both can not be introduced at once).

To present the syntax it is first useful to establish some conventions, this section will present B$^\sharp$'s syntax in a similar fashion to BNF (Backus-Naur form) with the following conventions:

- Id: a unique identifier (e.g., a string)
- $pattern?$: optional
- $pattern*$: zero or more repetitions
- $pattern+$: One or more repetition
- $(pattern)$: grouping
- $pat_1 \mid pat_2$: choice

(*pattern*, pat_1 and pat_2 are place holders for an expression built using the rules above).

Finally, there are occasions where elements that have already been defined need to be referenced (i.e., it is not desired to follow the rule for that element, but, to reference an already defined member of the element using its identifier). Elements that are references rather than rules will be subscripted with Id, e.g., $datatype_{Id}$ means the rule expects a datatype identifier, not to follow the datatype rule.

The language presented is a simplified form of B$^\sharp$ with many of the elements which are in effect syntactic sugar not presented e.g., infix functions are not presented, instead they are assumed to be prefix functions with two arguments. This allows the language to be presented more clearly, and simplifies the presentation of the translation to Event-B later. The language is initially presented as if all elements are contained within a single file. The form of the file is:

$$file ::= (datatype \mid class \mid extend)* \tag{4.33}$$

This says that a file is made up of any number ($*$) of *datatype*, *class* and *extend* declarations in any order.

To better understand the formalisation datatypes are formalised below. After this the whole abstract language is presented, with some further comment.

4.4.1 Datatypes

Previously the example of a list datatype was shown (4.34), with had the following definition:

$$\begin{gathered}
\overbrace{\textbf{Datatype}}^{\text{keyword}}\ \overbrace{List}^{\text{name}}\ \underbrace{\langle T\rangle}_{\text{parametric context}} \\
\text{constructors}\left\{\begin{array}{l} \mid nil \\ \mid cons(\underbrace{head : T, tail : List\langle T\rangle}_{\text{destructors}}) \end{array}\right. \\
\underbrace{\{\ldots\}}_{\text{Type Body}}
\end{gathered} \tag{4.34}$$

The abstract syntax for this statement is as follows:

$$datatype ::= Id\ \ paraType*\ \ constructor+\ \ typeBody \tag{4.35}$$

Referring back to the list example, the identifier (*Id*) is the name *List*, the parametric context is a list of parametric types (zero or more *paraType*s, in this case *T*), there are then a list of one or more *contructor*s and a *typeBody*.

The *paraType*s which make up a parametric context can be optionally constrained by a type class, this is what the *typeConstr* rule in the definition below allows:

$$paraType :: Id \;\; typeConstr? \tag{4.36}$$

Parametric types can only be constrained to a single type class, however, new type classes can be defined to amalgamate previous type classes, resulting in this not being a restriction.

Finally, it is necessary to see how the constructors are made up:

$$\begin{aligned} constructor &::= Id \;\; deconstructor* \\ deconstructor &::= Id \;\; constrType \end{aligned} \tag{4.37}$$

A constructor is made up of a series of destructor statements, each of which is made up of a series of typed elements.

Datatypes require a constructor that does not reference the datatype in which they are declared, this is called a base constructor. For example, in the declaration of the *List* type it is required that at least one of the constructors does not reference the *List* type, this is fulfilled by the *nil* constructor, which has no destructors so does not reference any types. $B^\sharp$ does not support co-datatypes as it maps these directly to the current Event-B datatypes which do not support these e.g., streams are not supported.

4.5 Complete Formalisation

Having given an example of how the concrete syntax seen previously in the examples relates to the abstract syntax the rest of the abstract syntax is presented below in figure 4.1.

$$
\begin{aligned}
file &::= (import \mid datatype \mid class \mid extend)* \\
import &::= datatype_{Id} \mid class_{Id} \mid extend_{Id} \\
datatype &::= Id \;\; paraType* \;\; constructor+ \;\; typeBody \\
paraType &::= Id \;\; (class_{id} \;\; constrType*)? \\
constructor &::= Id \;\; deconstructor* \\
deconstructor &::= Id \;\; constrType \\
constrType &::= unaryType \;\; (constructor \;\; constrType)? \\
unaryType &::= typeBracket \mid paraType_{Id} \mid typeConstr \mid instance_{Id} \\
typeConstr &::= (datatype_{Id} \mid class_{Id}) \;\; constrType* \\
typeBracket &::= constrType \\
typeBody &::= (function \mid theorem \mid instance)* \\
function &::= Id \;\; paraType* \;\; parameter* \;\; constrType \;\; expression \\
parameter &::= Id \;\; constrType \\
theorem &::= Id \;\; expression \\
class &::= Id \;\; paraType* \;\; instName \;\; constrType+ \;\; parameter* \;\; typeBody \\
instName &::= Id \\
expression &::= functionCall \mid quantifier \mid lambda \\
functionCall &::= funcName \;\; constrType* \;\; expression* \\
functionName &::= parameter_{Id} \mid deconstructor_{Id} \mid function_{Id} \mid constrType \\
&\qquad \mid functionCall \mid lambda \mid class_{Id} \\
quantifier &::= quantType \;\; paraType* \;\; parameter* \;\; expression \\
lambda &::= paraType* \;\; parameter* \;\; expression \\
extend &::= Id \;\; (class_{Id} \mid datatype_{Id} \mid extend_{Id}) \;\; typeBody \\
instance &::= class_{Id} \;\; constrType* \;\; expression* \;\; Id?
\end{aligned}
$$

Figure 4.1: $B^\sharp$ language[1]

It is worth drawing attention to a few elements of the syntax. Firstly in the *constrType* definition (which represents types constructed using type constructors e.g., $T \times List\langle T \rangle$), *constructor* is not

[1]Instead of having separate rules for universal and existential quantifiers the *quantifier* rule uses the *quantType* rule which is assumed to define the quantifier as one of these.

defined. $\text{B}^\sharp$'s type constructors are equivalent to Event-B's set constructor operators (including the functional types e.g., ($T \times pNat \to T$ is a type). As described further in the translation section below there is a one to one mapping between the $\text{B}^\sharp$ constructors and the Event-B set constructors.

In the definition of a function call there are many elements which can be used as a function name. This is because functions can be passed as arguments to other functions, allowing function parameters to be used as function names if they have a functional type. Deconstructors are treated as functions which take an instance of the type they were declared in as their argument. Finally, function calls do not require any parametric context (the *constrType**) or arguments (*expression**), when there is no parametric context or arguments the *functionCall* rule acts as a wrapper for any element that can appear within expressions (which includes elements like types).

Chapter 5

Translating B$^\sharp$ to Event-B by Example

This chapter shows examples of the translation from B$^\sharp$ to Event-B. This is done using examples to give the reader a strong idea of what the translation is doing. In the next chapter these translation steps are shown using functions in a pseudo language designed to be expressive enough to describe the translations.

Along with creating objects usable by the Event-B modeller, the translation has additional goals:

1. Generate Event-B which resembles the original B$^\sharp$: Proofs will be done in the Event-B environment and the generated Event-B must be recognisable to the user for this purpose.
2. Generating efficient Event-B: When proving, Event-B works better with some structures than others, for example Event-B reasons better about sets than functions for *Bool*s. As discussed in Section 3.6.1, the choice of Event-B representation can make a difference to the difficulty of the proofs. The correct choice of representation can result in proofs being discharged entirely automatically.

These additional aims are discussed during the example translations given below.

5.1 Examples of Translation

This section goes through the B$^\sharp$ examples shown in Chapter 4 and shows how these examples are mapped to Event-B.

5.1.1 Datatypes

The Event-B code generated by B$^\sharp$ datatype declarations closely resembles the B$^\sharp$ from which it is generated. The most significant difference is in the naming of the datatype constructors.

Within Event-B names are global. The result of this is in Event-B if you define two elements with the same name, they cannot be visible and used within the same context (i.e., you can only import one of the files). To resolve this when Event-B is generated the B♯ tool prefixes elements which could result in a naming clash. The result is the datatype used to declare the B♯ *pNat* type (4.1), generates the following Event-B:

$$
\begin{aligned}
&\textbf{Datatype}\ \mathit{pNat} \\
&\quad \mid \mathit{pNat\_zero} \\
&\quad \mid \mathit{pNat\_suc}(\mathit{pNat\_prev} : \mathit{pNat})
\end{aligned}
\tag{5.1}
$$

In B♯ datatype and class names are required to be unique.

This strategy of prefixing elements to avoid name clashes within Event-B is used extensively. The prefix is the name of the type in which the element is declared.

5.1.2 Functions

When translating a B♯ function to Event-B two Event-B operators are generated. One gives a direct definition of the operator using parameters in an expression. When given arguments the operator can be expanded into an expression by substituting parameters for arguments. Theorems about the operator can then be stated, and proved with the expansion. The second operator allows the direct definition operator to be passed as an Event-B functional type (a passing operator). It takes no arguments, and the operator's expression wraps the calling operator in a lambda. This technique was seen in the Event-B case study in 3.4.1.

The mapped functions again need to be prefixed with the name of the class they are declared in. In the examples of functions from Section 4.1.2, the Event-B operator generated to evaluate from the B♯ functions is very similar to the B♯ function declarations. For instance, the Event-B definition of addition is almost identical to the B♯ example (4.4):

$$
\begin{aligned}
&\mathit{pNat\_add}(x : \mathit{pNat}, y : \mathit{pNat}) \mathrel{\widehat{=}} \\
&\quad \textbf{cases}[x] \\
&\qquad \mathit{pNat\_zero} \rightarrow y \\
&\qquad \mathit{pNat\_suc}(xs) \rightarrow \mathit{pNat\_suc}(xs\ \mathit{pNat\_add}\ y)
\end{aligned}
\tag{5.2}
$$

The differences are that the B♯ typing is expanded ($x, y : \mathit{pNat}$ becomes $x : \mathit{pNat}, y : \mathit{pNat}$) and calls to Event-B operators need to be prefixed. The reason that the expansion is so similar is because the definition expressions are simple (there are examples of Event-B generated by more complex expressions below). The passing operator generated by the translation for addition is:

$$
\mathit{pNat\_add\_P}() \ \widehat{=}\ \lambda x \mapsto y \cdot x \in \mathit{pNat} \land y \in \mathit{pNat} \mid \mathit{pNat\_add}(x, y) \tag{5.3}
$$

This operator wraps the evaluation operator in an Event-B lambda expression, the *pNat_add_P* operator can therefore be passed like a function. When translating functions within expressions the correct Event-B operator is chosen based on the context (if the function has arguments the evaluation operator is used, otherwise the passable operator is used).

Finally an Event-B theorem is generated, to demonstrate the operator returns the return type specified in the B$^\sharp$ statement. In cases where the return type is a complete type this theorem will be proved automatically:

$$\forall x, y \cdot x \in pNat \land y \in pNat \Rightarrow pNat\_add(x, y) \in pNat \tag{5.4}$$

In the B$^\sharp$ examples we also saw the *curry*2(4.5) function which had a parametric context, this maps to the following Event-B operators:

$$curry2(T : \mathbb{P}(T\_EvB), f : T \times T \to T, x : T) \mathrel{\hat{=}} \lambda y \cdot y : T \mid f(x \mapsto y) \tag{5.5}$$

$$curry2\_P(T : \mathbb{P}(T\_EvB)) \mathrel{\hat{=}} \lambda f \mapsto x \cdot f \in T \times T \to T \land x \in T \mid curry2(T, f, x) \tag{5.6}$$

In the first of these equations (5.5) the parametric type introduced in the B$^\sharp$ definition becomes the first argument of the generated operator with the statement $T : \mathbb{P}(T\_EvB)$. In Event-B parametric types are introduced at the file level rather than for specific functions/classes (this allows axiomatic definitions of types, which is not supported in B$^\sharp$). *T_EvB* is an Event-B parametric type. B$^\sharp$ parametric types map to subsets in Event-B, hence, using the powerset of *T_EvB*, this allows reasoning about non-complete types (such as the binary naturals seen in the Event-B case study).

Note also that as f is an Event-B functional type, rather than an operator, the arguments need to be passed using the Event-B maplet syntax ($\mapsto$) for pairs. Whenever a parameter is treated as a function the B$^\sharp$ argument list is automatically mapped to the Event-B maplet syntax. Only when a function is called directly is the operator argument syntax used (a comma separated list).

The second generated Event-B operator allows the *curry*2 function to be instantiated with a type. For instance, if another B$^\sharp$ function had a parameter $f : ((pNat \times pNat \to pNat) \times pNat) \to (pNat \to pNat)$, then $curry\langle pNat\rangle$ could be passed as this argument (as this is the type of the curry2 function in B$^\sharp$). In Event-B $curry\langle pNat\rangle$ is translated to $curry2\_P(pNat)$.

5.1.3 Class Declarations

Class declarations are mapped to Event-B operators which construct Event-B sets. For instance the Event-B generated by the *ReflexRel* declaration (4.7) is as follows:

$$\begin{aligned} \mathit{ReflexRel}(T : \mathbb{P}(T\_EvB)) \;\widehat{=}\; \{r|r \in \mathbb{P}(T \times T) \\ \wedge \forall x \cdot x \in T \Rightarrow x \mapsto x \in r\} \end{aligned} \tag{5.7}$$

The operator above is used to type elements in the Event-B representation. For instance a B$^\sharp$ statement such as $r : \mathit{ReflexRel}\langle pNat\rangle$ would translate to $r \in \mathit{ReflexRel}(pNat)$ within Event-B. Syntactically this is similar to the original B$^\sharp$ statement, except that in Event-B predicates are a separate syntactic category. To resolve this a relation is expressed in Event-B as a restricted set of pairs. The B$^\sharp$ Boolean statement $r(a, b)$, translates to the Event-B predicate $a \mapsto b \in r$.

5.1.4 Quantifiers and Theorems

B$^\sharp$ quantifiers translate to their Event-B equivalents, there are however two additional areas which complicate the translation. Firstly B$^\sharp$ quantifiers have a typed parameter list, secondly B$^\sharp$ allows quantifiers to have parametric contexts. Examples of both of these were seen in example (4.10):

$$\forall\langle T, equ : \mathit{ReflexRel}\langle T\rangle\rangle\; x, y, z : T \cdot equ(x, y) \wedge \neg equ(y, z) \Rightarrow \neg equ(x, z)$$

Which translates to the following Event-B:

$$\begin{aligned} &\forall T, equ \cdot T \in \mathbb{P}(T\_EvB) \wedge equ \in \mathit{ReflexRel}(T) \Rightarrow (\forall x, y, z \cdot x \in T \wedge y \in T \wedge z \in T \\ &\quad\Rightarrow x \mapsto y \in equ \wedge \neg(y \mapsto z \in equ) \Rightarrow \neg(x \mapsto z \in equ)) \end{aligned} \tag{5.8}$$

Here it can be seen that the types in the parametric context are quantified over separately. This makes instantiation of the parametric types easier when proving. The typed B$^\sharp$ parameters are first declared, then the generate expression is prefixed with explicit typing information for the parameters. It can be seen from this that the B$^\sharp$ parametric type, and parameter information is considerably more concise than the Event-B representation. The initial B$^\sharp$ expression is also more concise due to the combination of predicates and Booleans.

In the examples theorem (4.8) it was seen that parametric types and generic instances declared in the class, could be used within the expression. If these types are used within the expression then B$^\sharp$ adds them to the parametric context of the quantifier before mapping to Event-B e.g., (4.8) becomes identical to (4.10) before being mapped to Event-B. This inference increases the conciseness of the B$^\sharp$ representation, without a loss of clarity as the theorems are declared within the type body of a type class, having the inferred elements available is expected.

Whilst the verbosity of the Event-B statement makes the generated Event-B difficult to read, it is not expected that the user will interact directly with the Event-B theorem, instead they will interact with it via the interactive prover where quantification is stripped away. For example, the interactive prover will present the following to the user:

Hypotheses:
$e \in EquivRel\_T(T)$
$x \in T$
$y \in T$
$z \in T$
$x \mapsto y \in e$
$\neg\, y \mapsto z \in e$
Goal:
$\neg\, x \mapsto z \in e$

This is very similar to the original B$^\sharp$ statement (4.8), except for the previously discussed difference in the representation of relations.

5.1.5 Required Elements

When a class declaration has required elements as in the case of the *Setoid* declaration (4.11), the generated Event-B needs to be able to represent the required elements, and this is done by pairing the required elements with the supertype. The generated Event-B operator for *Setoid* class is as follows:

$$Setoid(T : \mathbb{P}(T\_EvB)) \triangleq \{S \mapsto equ \mid S = T \wedge equ : EquivRel(T)\} \tag{5.9}$$

Along with typing statements, Event-B operators are generated to deconstruct elements of the type set, to access the required elements, these use the Event-B's $prj1$ and $prj2$ operators to get the members of pairs constructed with the $\mapsto$ operator. For example, the setoid type has a single required element, the deconstructor to access this element has this definition:

$$Setoid\_equ(T : \mathbb{P}(T\_EvB), S \in Setoid(T)) \triangleq prj2(S) \tag{5.10}$$

These Event-B operators are then used within the generated Event-B expressions. The Event-B generated from the *CommOp* (4.13) statement is as follows:

$$\begin{aligned}
&CommOp(T1 : \mathbb{P}(T\_EvB), T : Setoid(T1)) \mathrel{\hat{=}} \\
&\quad \{cOp \mid cOp \in baseOp(T1,T) \land \\
&\qquad \forall x, y \cdot x \in T1 \land y \in T1 \Rightarrow cOp(x \mapsto y) \mapsto cOp(y \mapsto x) \in Setoid\_equ(T1,T)\}
\end{aligned} \tag{5.11}$$

The arguments to the *CommOp* operator allow a generic setoid to be passed as an argument. The type has to be passed as a separate argument due to Event-B not allowing subtyping. Within (4.13) statement *T.equ* is used to access the equivalence relation of the setoid, in Event-B this becomes $Setoid\_equ(T1,T)$ using the Event-B deconstructor. Again we see here that the source B$^\sharp$ (4.13) is a lot less verbose than the generated Event-B.

When theorems are declared about type classes, the Event-B mapping constructs a generic version of the type class in Event-B, which can then be used within the theorem expression. For example the Event-B generated from identity theorem on the $Monoid$ type (4.15) is:

$$\begin{aligned}
&\forall T, equ, op, ident \cdot T \in \mathbb{P}(T\_EvB) \land T \mapsto equ \mapsto op \mapsto ident \in Monoid(T) \Rightarrow \\
&\quad (\forall x \cdot x \in T \Rightarrow (op(x \mapsto ident) \mapsto ident \in equ \Leftrightarrow x \mapsto ident \in equ))
\end{aligned} \tag{5.12}$$

Defining the generic *Monoid* in this manner results in the theorem expression being very similar to the B$^\sharp$ declaration (again with the exception of the representation of the relations, and the Event-B typing statement). The result is that when interacting with the interactive prover, the theorem looks very similar to the theorem written in the B$^\sharp$ syntax.

5.1.6 Methods

Methods (functions which reference the containing type) have a similar naming issue to the theorems discussed above. This is resolved in the same way as theorems, if the type class is used then a transformation is done on the B$^\sharp$ to add the type class to the parametric context. So *raiseToL* shown in (4.16) is first transformed to (4.17), then mapped to the following Event-B:

$$\begin{aligned}
&raiseToL(T : \mathbb{P}(T\_EvB), M : Monoid(T), x : T, p : pNat) \mathrel{\hat{=}} \\
&\quad match\ p \\
&\qquad \mid zero : Monoid\_ident(T, M) \\
&\qquad \mid suc(ps) : Monoid\_op(T, M)(x \mapsto Monoid\_raiseToL(T, M, x, ps))
\end{aligned}$$

B$^\sharp$ allows arguments to parametric contexts without arguments being given to parameters, e.g., M.*raiseToL* is valid B$^\sharp$, and is the *raiseToL* function for the *Monoid* M. To facilitate this in

Event-B a second operator is generated, which is similar to the first-class operator generated for functions, and has the following definition:

$$\begin{aligned} &raiseToL\_P(T : \mathbb{P}(T\_EvB), M : Monoid\_T(T)) \\ &\quad = \lambda x \mapsto p \cdot x \in T \land p \in pNat \mid raiseToL(T, M, x, p) \end{aligned} \tag{5.13}$$

Given a *Monoid* this definition returns a lambda expression to compute the *raiseToL* operator for that monoid. Given a B$^\sharp$ statement such as $raiseToL\langle T\rangle$, this operator is used as the passable function, the type T is expanded to generate the Event-B type arguments. The operators generated here would be identical to the operators generated by function equivalent (4.17), however, the difference in declaration does effect the mapping when generating Event-B from instance statements.

5.1.7 Instances

When an instance is declared in B$^\sharp$ it is necessary to generate a proof obligation showing that the concrete type is a member of the type class. After this each of the theorems and methods from the type class (and its supertypes) are instantiated within the concrete type. In principle, these do not need to be proved as they have been proved in the abstract type, it is enough to prove concrete elements form the abstract type.

Note that only functions and theorems that use an implicit reference to the abstract type are instantiated, if a theorem/function makes no reference to the abstract type it is unclear what instantiation should happen.

To prove that a concrete type is a member of a type class a new Event-B theorem is generated stating that the concrete elements form the abstract type, i.e., an 'instance' theorem. As an example the theorem for this generated by statement (4.26) is as follows:

Theorems:
addMon in CommMonoid
$pNat \mapsto \{x \mapsto y \mid x = y\} \mapsto pNat\_add\_P \mapsto zero \in CommMonoid(pNat)$

The Event-B expression for the instantiated theorems is often considerably simpler than the abstract Event-B expression. For instance the identity theorem on the monoids (5.12) becomes:

$$\forall x \cdot x \in pNat \Rightarrow (pNat\_add(x, zero) = zero \Leftrightarrow x = zero) \tag{5.14}$$

This is easier to work with in the interactive theorem prover than directly using the theorem on the monoids. Directly using the monoid theorem would require instantiation of the monoid theorem and the instantiation of instance theorem with every use.

Generated theorems do not in principle need proving, although proving them is a simple process of instantiating original theorem from the type class with the instance. This results in hypothesis identical to the proof obligation. The approach of using abstract types is only useful if there are multiple concrete types which can reuse the results from the abstract types. There are many other examples of monoids in mathematics, for example, it is also possible to define a commutative monoid with one and addition, where the instantiated identity theorem becomes:

$$\forall x \cdot x \in pNat \Rightarrow (pNat\_times(x, one) = one \Leftrightarrow x = one) \tag{5.15}$$

Again this would lead to the theorems and functions associated with the monoid type class being instantiated for this new monoid.

Chapter 6

Core Translations

6.1 Introduction

This chapter focuses on the translations from B$^\sharp$ to Event-B. To allow this, an abstract representation of Event-B elements is given. Then a language is defined to describe the translation between the B$^\sharp$ and Event-B syntaxes.

Once the tools to describe the translation have been shown, translations of the core elements of B$^\sharp$ are shown. As seen in the chapter on translation examples, Chapter 5, there were instances where the translation were first done using inference to change one B$^\sharp$ statement to another, these translations are not included in the core translations, they are described in Chapter 7.

The section on core translations is split into three broad sections:

1. Defining functions useful within the translations
2. Translating declarations: e.g., the declaration of a new function
3. Translating expressions

The later two sections are dependent on each other i.e., declarations contain expressions, and expressions use the elements defined within a declarations (a function call in an expression uses a declared function).

B$^\sharp$ and Event-B have many similar data structures, e.g., they both have a datastructure called *datatype*. To make it clear to which datastructures are being referred, Event-B datastructures will be subscripted with *evB* ($datatypes_{evB}$ is the Event-B datastructure).

6.2 Event-B Abstract Syntax

To show the translation from B$^\sharp$ to Event-B it is necessary to have an abstract representation of the Event-B elements generated by the translation. In [57] the concrete syntax for Event-B elements is described. Based on this description the figure 6.1 below shows the abstract elements. This uses the same BNF style syntax as was used when presenting the B$^\sharp$ language elements.

6.3 Translation Language

In order to define the translation from B$^\sharp$ to Event-B, we introduce a *translation language* that operates on the syntax of the source and target language. A pseudo language is chosen to show the translation as it allows the translation functions to be easily understood and close in format to the implementation of the translation from B$^\sharp$ to Event-B. In Chapter 8, we see that the implementation translates B$^\sharp$ statements directly to concrete Event-B statements. We present the translation rules as transformations on data structures that represent the syntactic structures in the source and target languages.

The translation is done in a single pass of the abstract syntax tree with each element having a function which describes how it is translated. Many of the functions to translate top level elements (e.g., type bodies) are performed entirely by calling the translation function on their containing elements. These translation functions are omitted. Within the core translation, theorems and datatypes translate to Event-B theorems and datatypes, and functions and classes translate to Event-B operators. Each function translates to two Event-B operators, an operator which generates a lambda statement, so it can be used without arguments within an expression, and an operator which can be expanded by substituting parameters with arguments. Each B$^\sharp$ **Class** declaration translates to an operator which defines the class in Event-B, and a series of operators to deconstruct the class (one operator for each required element).

6.3.1 Data Structures

Each of the rules in the abstract syntaxes shown in 4.1 and 6.1 give rise to a data structure in the translation language. For example, in the B$^\sharp$ syntax the function syntax has the following definition:

$$datatype ::= Id \;\; paraType* \;\; constructor+ \;\; typeBody \tag{6.1}$$

This gives rise to a data structure named *datatype*. The names of the destructors for this data structure are the same as the rule names for the elements, e.g., given a function d : *datatype*, *d.typeBody* will access the *typeBody* of d.

$$
\begin{aligned}
\mathit{datatype} &::= \mathit{Id}\;\; \mathit{cType}_{id}\;\; \mathit{constructor}+\\
\mathit{constructor} &::= \mathit{Id}\;\; \mathit{destructor}\,*\\
\mathit{destructor} &::= \mathit{Id}\;:\; \mathit{set-expr}\\
\mathit{op} &::= \mathit{Id}\;\; \mathit{param}+\;\; \mathit{expr}\\
\mathit{pred-op} &::= \mathit{Id}\;\; \mathit{param}+\;\; \mathit{pred-expr}\\
\mathit{param} &::= (\mathit{set-var})\;\; \mathit{set-expr}\\
\mathit{expr} &::= \mathit{set-comp} \mid \mathit{lambda} \mid \mathit{quantifier} \mid \mathit{pair-expr} \mid \mathit{op-call} \mid \mathit{bool-func}\\
&\quad \mid \mathit{bracket} \mid \mathit{set-expr}\\
\mathit{set-expr} &::= \mathit{set-unary}\;\; (\mathit{set-op}\;\; \mathit{set-expr})?\\
\mathit{set-unary} &::= \mathit{type} \mid \mathit{power-set} \mid \mathit{op-call} \mid \mathit{set-comp} \mid \mathit{set-var}_{Id} \mid \mathit{set-bracket}\\
\mathit{type} &::= \mathit{cType}_{id} \mid \mathit{datatype-constr}\\
\mathit{datatype-constr} &::= \mathit{datatype}_{id}\;\; \mathit{set-expr}*\\
\mathit{set-var} &::= \mathit{Id}\\
\mathit{dom} &::= \mathit{set-expr}\\
\mathit{ran} &::= \mathit{set-expr}\\
\mathit{unary-var} &::= \mathit{Id}\\
\mathit{var} &::= \mathit{set-var} \mid \mathit{unary-var}\\
\mathit{power-set} &::= \mathit{set-expr}\\
\mathit{set-comp} &::= \mathit{pair-params}\;\; \mathit{pred-expr}\\
\mathit{set-bracket} &::= \mathit{set-expr}\\
\mathit{lamda} &::= \mathit{pair-params}\;\; \mathit{pred-expr}\;\; \mathit{expr}\\
\mathit{quantifier} &::= \mathit{quant-type}\;\; \mathit{pair-params}\;\; \mathit{pred-expr}\\
\mathit{pair-params} &::= \mathit{ident}\;\; \mathit{pair-params}?\\
\mathit{ident} &::= \mathit{Id}\\
\mathit{pair-expr} &::= \mathit{expr}\;\; \mathit{pair-expr}\\
\mathit{op-call} &::= \mathit{op}_{id}\;\; \mathit{expr}+\\
\mathit{pred-op-call} &::= \mathit{pred-op}_{Id}\;\; \mathit{expr}+\\
\mathit{func-call} &::= (\mathit{lambda} \mid \mathit{op-call} \mid \mathit{funcCall} \mid \mathit{param}_{id} \mid)\;\; \mathit{pair-expr}\\
\mathit{pred-expr} &::= \in \;\mid\; \cap \;\mid\; \wedge \;\mid \mathit{pred-op-call} \;\mid\; \Rightarrow \;\mid\; \ldots\\
\in &::= \mathit{expr}\; \mathit{set-expr}, \quad\; \cap ::= \mathit{pred-expr}+, \quad\; \wedge ::= \mathit{pred-expr}+,\\
\Rightarrow &::= \mathit{pred-expr}\;\; \mathit{pred-expr}\\
\mathit{bool-func} &::= \mathit{pred-expr}
\end{aligned}
$$

Figure 6.1: Event-B Abstract Syntax

Within the translation it is necessary to construct data structures (mostly the Event-B data structures that are the output of the translation). To facilitate understanding, data structures are constructed using the concrete syntax of the B$^\sharp$ and Event-B languages. For example to create an Event-B set comprehension data structure the translation language would use the following:

$$\{pair{-}params \mid pred{-}expr\} \tag{6.2}$$

Where $pair{-}params$ and $pred{-}expr$ are correctly typed expressions within the translation language (i.e., Event-B datatypes).

In the *datatype* definition above (6.1), some of the rules will generate multiple elements. Within the translation language these becomes ordered lists of those elements (e.g., *paraType** will generate a list of *paraType*s due to the *). Within the translation language to signify an element is a list rather than an individual element it is highlighted in bold (d.**paraType**). For example, $\mathbf{d} : List\langle datatype\rangle$ means that **d** is a list of *datatype*s. In the abstract syntax whenever a rule is suffixed with $*$ or $+$ a list is constructed within the datatype. Within the translation language single elements are automatically encapsulated in Lists when required.

In (6.1) the $paraType$ element is optional (Recall in the abstract syntax $*$ is zero or more, and *?* is zero or one). These expressions can be omitted when constructing a datatype. If these destructors are used and there is no value then they have the value nil.

6.3.2 Translation Functions

The translation rules are defined in translation language as functions which at the top level will take B$^\sharp$ datatypes and produce Event-B datatypes (there are also intermediate functions which take B$^\sharp$ datatypes and produce other B$^\sharp$ datatypes). Given a list of typed B$^\sharp$ and Event-B elements $t_1 : Type_1, \ldots, t_n : Type_n$, a function in the translation language will have the following form:

$$functionName(t_1 : Type_1, \ldots, t_n : Type_n) \rightarrow Type_{return}\{expression\} \tag{6.3}$$

Given a correctly typed set of expressions $e_1 : Type_1, \ldots e_n : Type_n$ a function call is made with the following syntax:

$$functionName(e_1, \ldots, e_n) \tag{6.4}$$

In the translation language the return type is to show the user what type to expect the function to return. It should always match the type of the element returned by the function.

As a convenience, functions can take a list of arguments in place of a single argument, this causes the function to work on each element of the list, and return a list. In effect given the $apply$ map which applies a function to each element of a list, when a single element function f is called with a list $\mathbf{l}$ the result is $apply(f, \mathbf{l})$ (a list with f applied to each element of $\mathbf{l}$).

Function names can be overloaded (i.e., a function name can be used more than once), and this is resolved by using the type of the argument to disambiguate the function which should be called. This reduces conditional statements in the translation language, as the language will choose the correct function based on type rather than this needing to be stated explicitly.

The expression within a translation language function is made up of function calls, conditionals, match statements and let statements. These statements all use a fairly standard syntax which the reader should be familiar with from other languages.

Because the naming rules in Event-B are different to the B$^\sharp$ naming rules, it is necessary to generate free variables in Event-B when translating. To resolve this there are a series of special functions. Functions suffixed with Id generate unique variable names within Event-B using the arguments to the function. When an Id function is called with the same arguments it will produce the same Event-B identification. The generated identifier is seen as an implementation detail and is discussed further in the implementation Chapter 8.

6.3.3 Let Statements

As in other languages, *let* statements are used to assign a name to the result of an expression. They also allow a syntax for multiple assignment, this works in the following way:

```
1    $evBEvalOp(f : function) \rightarrow op_{evB}\{$
2      $let\ fName\langle T_1 : TC_1, \dots T_i : TC_i\rangle(p_1 : P_1, \dots, p_j : P_j) : RetType\ \ expression = f$
3      $\dots$
4    $\}$
5
6
```

Line 2 has the effect of making the following statements:

$$let\ fName = f.Id$$
$$let\ T_1 : TC_1, \dots, T_i : TC_i = f.\mathbf{paraType}$$
$$let\ p_1 : P_1, \dots p_j : P_j = f.\mathbf{parameter}$$
$$let\ RetType = f.constrType$$
$$let\ expression = f.expression$$

The values on the left of the multiple assignment match the constructors of the abstract type (in this case f). Lists are unwrapped, and $\dots$ are used to represent missing elements. This multiple assignment syntax has the benefit of relating the abstract type to the concrete syntax. This results in mapping functions being simpler to understand.

6.4 Mapping Declarations

6.4.1 Class Declarations

An initial example of the mapping is how to map a class declaration in B$^\sharp$ to an Event-B operator, e.g., such as in the setoid example (5.9). The following function, *typeOp*, maps a B$^\sharp$ class declaration to an Event-B operator declaration:

```
$typeOp(c : class) \rightarrow Op_{evB}\{$
    $let\ c = \textbf{Class}\ C\ \langle S : Type_S, \dots, T : Type_T\rangle[N] : sType_1, \dots, sType_i$
          $(v_1 : S_j, \dots v_i : S_j)\ \textbf{where}\ P_1; \dots; P_k; \{\dots\}$
    $let\ opName = evBTypeId(C)$    $//\ C,\ The\ class\ name\ becomes\ the\ op\ name$
    $//\ Each\ parametric\ type\ can\ generate\ several\ Event\text{-}B\ types$
    $//\ s_1 : S_{evB1}, \dots, s : S_{evBk}\ are\ the\ elements\ constructed\ by\ evBParam(S : Type_s)$
    $let\ s_1 : S_{evB1}, \dots, s : S_{evBk}, \dots, t_1 : T_{evB1}, \dots, t : T_{evBl} \dots$
        $= evBParam(S : Type_s), \dots, evBParam(T : Type_T)$                          (6.5)
    $let\ SName = superTypeId(c)$    $//\ SName\ is\ an\ identifier\ for\ supertype$
    $let\ v_1 : evB(S_1), \dots, v_i : evB(S_i) = evBParam(v_1 : S_1), \dots, evBParam(v_i : S_i)$
    $return \left( \begin{array}{l} opName(s_1 : S_{evB1}, \dots, s : S_{evBi}, \dots, t_1 : T_{evB1}, \dots t : T_{evBi}) \\ \quad \hat{=} \{SName \mapsto v_1 \mapsto \cdots \mapsto v_i \\ \qquad \mid SuperTypeName \in evB(sType_1) \cap \cdots \cap evB(sType_1) \\ \qquad \wedge\ v_1 \in evB(S_1) \wedge \cdots \wedge v_i \in evB(S_i) \\ \qquad \wedge\ evBPred(P_1) \wedge \cdots \wedge evBPred(P_i)\} \end{array} \right)$
$\}$
```

The first let statement decomposes the argument c into its constituent elements. The second statement generates an Event-B name for the operator (in practice the name for the operator can simply be the name of the class, as in B$^\sharp$ this has to be unique). Next Event-B parameters for the Event-B operator are mapped from the B$^\sharp$ parametric context. This is done with the *evBParam* function described below (Section 6.4.2). Each of these calls can generate multiple Event-B parameters which are added to a single list of B$^\sharp$ parameters. Next Event-B equivalents to the required elements are constructed. Finally an Event-B set comprehension expression is defined to represent the class. Most of this is constructed from elements which are already translated above, except for the translation of the **where** predicates, $P_1 \dots P_k$, which is done inline. This is done with the *evBPred* function which compiles B$^\sharp$ expressions to Event-B predicates. The B$^\sharp$ expressions must be of type *Bool*. In essence each part of element of the class statement is translated into an Event-B component, which are then put together into an Event-B operator which defines a set comprehension statement. We will now look at how each of the individual components are translated.

6.4.2 Mapping Parametric Types and Parameters

This section will look at the mappings of parametric types and parameters, these are looked at together because they both produce Event-B parameters. The function used to generate Event-B parameters is called *evBParam* and is overloaded so can be called with B$^\sharp$ parametric types, or B$^\sharp$ parameters/required elements.

When a parametric type (*paraType*) is mapped to Event-B parameters the types that are needed to define the class of the *paratype* are inferred. For example the definition of an associative operator is as follows:

$$AssocOp\langle T : Setoid\rangle \dots \tag{6.6}$$

This is equivalent to:

$$AssocOp\langle S, T : Setoid\langle S\rangle\rangle \dots \tag{6.7}$$

This is a useful process when the **paraType**s of the constraining class do not have any required elements. However if the *paraType*s are constrained to type classes there is no syntactic way to access the required elements. For example, if a function on associative operators was declared:

$$opFunc\langle A : AssocOp\rangle \dots \tag{6.8}$$

Then the following is inferred:

$$opFunc\langle T, S : Setoid\langle T\rangle, A : Assoc\langle S\rangle\rangle \dots \tag{6.9}$$

The problem is that without the type S being explicitly defined there is no way to access the equivalence relation of S. In these situations the user will need to declare the function in the following way:

$$opFunc\langle S : Setoid\langle T\rangle, A : Assoc\langle S\rangle\rangle \dots \tag{6.10}$$

In the *typeOp* function (6.5) the parameters of the type class were also turned into a list Event-B parameters (using the *evBParam*(c.**paramType**) function).

The *evBParam* function first expands the parametric types as in (6.9), and these expanded types are then easily turned into Event-B parameters. Base types (types with no type class constraint) become a subset of a complete Event-B type i.e., T in B$^\sharp$ becomes $T : \mathbb{P}(T_{evB})$. Constrained classes are defined using the type operators constructed by (6.5) e.g. $S : Setoid$ becomes $S : Setoid(T)$ where T has already been defined when expanding the types, and $Setoid(T)$ is the typing operator defined for the *Setoid* type class. This is a recursive process so when we need to use a type constructor all the types for this type constructor have already been defined. The definitions for the functions can be found in Appendix A and Section A.1.

Converting B$^\sharp$ typed elements such as parameters and required elements is much simpler as each typed element in B$^\sharp$ maps to a single typed element in Event-B. The process is to generate an identifier for the Event-B parameter, and type it by converting the B$^\sharp$ type into an Event-B type. This is done with the following function:

```
$evBParam(p : parameter) \rightarrow param_{evB}\{$
 $// \ p : Type$
  $return\ evBId(p) : evB(p.contrType)\ //\ p_{evB} : TypeExpr$
$\}$
```
(6.11)

This completes the function definitions to generate an Event-B typing operator (6.5) from a B$^\sharp$ class. However, this is not the only function constructed by a B$^\sharp$ class declaration. Operators are also constructed to deconstruct an instance of a type class to access the required elements. This was seen in example (5.10). For each of the class's required elements (*Parameters*), the following function is used to generate an Event-B deconstructor operator:

```
$classDecon(c : Class, r : Parameter) \rightarrow Op_{evB}\{$
  $let$ **Class** $C\ \langle S_1 : TC_1, \dots, T_i : TC_i\rangle[N] : sType_1, \dots, sType_i$
       $(v_1 : S_1, \dots v_i : S_i)$ **where** $P_1; \dots; P_i; \{\dots\} = c$
  $let\ T_1 : Type_1, \dots, T_i : Type_i = evBParam(S_1 : TC_1, \dots, T_i : TC_i)$
  $let\ opName = evBId(c, r)$ // Name of the deconstructor op based on class and element.
  $let\ typeConstrName = evBId(c)$ // Name of the Event-B type op constructed by (6.5)
  $let\ cInstName = evBID$ // A unique ID, the name doesn't matter
  $return\ \left(\begin{array}{l} opName(T_1 : Type_1, \dots, T_i : Type_i, cInstName : typeConstrName(T_1, \dots, T_i)) \\ \quad \hat{=}\ getterExpr(c, r, classInstName) \end{array}\right)$
$\}$
```
(6.12)

The Event-B operator defined takes sufficient arguments to generate an abstract instance of the class, and the *getterExpr* then deconstructs the abstract instance to get the correct element. When a deconstructor is used in B$^\sharp$ (e.g., $S.equ$ where S is a *Setoid*) the instance of the class (S) contains enough information to generate all of the arguments to the Event-B deconstructor operator. The definition of the *getterExpr* function is:

```
$getterExpr(c : Class, r : parameter, n : expr_{evB}) \rightarrow expr_{evB}\{$
  $let\ r_1 : R_1 = r$
  $let$ Class $C\ \langle S_1 : TC_1, \dots, T_i : TC_i\rangle[N] : sType_1, \dots, sType_i$
      $(v_1 : V_1, \dots v_i : V_i)$ where $P_1; \dots ; P_i; \{\dots\} = c$
  if $r_1 : R_1 \notin \{v_1 : V_1, \dots, v_i : V_i\}$ // r isn't a required element of $c$ so must be an element of a superclass
    $return\ getterExpr(sType_1, r, prj1(n))$
  elif $r_1 : R_1 \in \{v_1 : V_1, \dots, v_i : V_i\}$
    $return\ prj2(\underbrace{prj1(\dots prj1(}_{c.parameter.indexof(param)\ times}(prj2(n)))))$
}
```

$$(6.13)$$

The Event-B representation of a type class has the form $superType \mapsto (mappedRequiredElements)$. If the parameter r is not a required element of the type c but is instead a required element of its super type, the function above is applied to the super type instead. This is done by applying *getterExpr* to $prj1(n)$ (the representation of the supertype). Otherwise $prj2(n)$ (the *mappedRequiredElements*) contains the parameter r. Its location depends solely on where the required element appeared in the declaration of the class $prj1$ is applied the correct number of time to find this. Finally , unless the element returned by the $prj1$s is the first element in the type class, a final $prj2$ needs to be applied to get the unique element.

6.4.3 Datatype Declarations

Due to the similarities between datatypes in B$^\sharp$ and Event-B the translation rules are simple:

```
$evB(d : datatype) \rightarrow datatype_{evB}\{$
    $let\ name = evBId(d)$
    $let\ S_1, \ldots, S_i = evBParam(d.\mathbf{paramType})$ $//\ d\langle S_1, \ldots, S_i \rangle$
    $let\ a, \ldots, z = d.\mathbf{constructor}$ // The B$^\sharp$ data type constructors
    $let\ d_1^a, \ldots, d_i^a = a.\mathbf{destructor}$ // The destructors for constructor $a$
    $\vdots$
    $let\ d_1^z, \ldots, d_i^z = z.\mathbf{destructor}$
    $return\ \left( name\langle S_1, \ldots, S_i \rangle \hat{=} \right.$
        Constructors:
            $evBId(a)(evBId(d_1^a) : evB(d_1^a.constrType), \ldots, evBId(d_i^a : evB(d_i^a.constrType)))$
            $\vdots$
            $\left. evBId(z)(evBId(d_1^z) : evB(d_1^z.constrType), \ldots, evBId(d_i^z : evB(d_i^z.constrType))) \right)$
$\}$
```

(6.14)

Each B$^\sharp$ element is translated into their corresponding Event-B element, and they then construct an Event-B datatype in the same manner as a B$^\sharp$ datatype. This process is made simpler by parametric types in the B$^\sharp$ datatype being constrained to only base types. B$^\sharp$ does not allow datatypes to be constructed with a non-base type. If you want to reason about a datatype with a non-base type, a class declaration can be used to subclass a datatype constraining the base type. A future version of B$^\sharp$ could automate this process.

6.4.4 Function Declarations

In the examples it was seen that when mapping a function to Event-B two Event-B operators are required, one as an expandable operator, and the other a passable operator. The process for tranlsating to both these types of operators will be looked at. The function for generating the expandable operator is called *evBEvalOp* and the function for generating the passable operator is called *evBPassableOp*.

The translation function to generate the callable operator has the following definition:

```
$evBEvalOp(f : function) \rightarrow op_{evB}\{$
    $let\ fName\langle T_1 : TC_1, \dots T_i : TC_i\rangle(p : P_1, \dots p_i : P_i) : RetType\ \ expression = f$
    $let\ name = evBEvalID(f)$
     // Get parameters from the parametric context
    $let\ s_1 : S_1, \dots, s_i : S_i = evBParam(T_1 : TC_1), \dots evBParam(T_j : TC_j)$
     // Get parameters from the B$^\sharp$ parameters
    $let\ a_1 : A_1, \dots, a_i : A_i = evBParam(p_1 : P_1), \dots evBParam(p_i : P_i)$
    $return\ \left(\begin{array}{l} name(s_1 : S_1, \dots s_i : S_i, a_1 : A_1, \dots, a_i : A_i) \\ \quad \hat{=}\ evBExpr(f.expression) \end{array}\right)$
$\}$
```
(6.15)

Both the parametric types and the function parameters become Event-B operator parameters. It is noticeable that at this point in the translation the return type of the B$^\sharp$ function is not used. It is seen later (Section 7.1.1) that this defines an Event-B theorem that the function does return the return type. Later it is seen that this is used to generate an Event-B theorem.

The passable operator is defined with the following mapping function:

```
$evBPassableOp(f : function) \rightarrow Op_{evB}\{$
    $let\ fName\langle T_1 : TC_1, \dots T_i : TC_i\rangle(p : P_1, \dots p_i : P_i) : RetType\ \ expression = f$
    $let\ name = evBPassableId(f)$
    $let\ evalName = evBEvalId(f)$
    $let\ s_1 : S_1, \dots s_i : S_i = evBParam(T_1 : TC_1), \dots evBParam(T_i : TC_i)$ // Operator parameters
    $let\ q_1 : Q_1, \dots q_i : Q_i = evBParam(p_1 : P_1), \dots, evBParam(p_j : P_j)$
    $return\ \left(\begin{array}{l} name(s_1 : S_1, \dots, s_i : S_i)\ \hat{=} \\ \quad \lambda q_1 \mapsto \cdots \mapsto q_i \cdot q_1 \in Q1 \wedge \dots q_i \in Q_i \mid evalName(t_1, \dots t_i, p_1, \dots, p_i) \end{array}\right)$
$\}$
```
(6.16)

Here only the parametric types become arguments to the Event-B operator. The expression returned by the operator is an Event-B lambda which takes the function parameters, and calls the callable operator.

Translation is slightly different when mapping B$^\sharp$ functions that return booleans. This is due to separation of predicate types within the Event-B language. When translating the expandable operator the Event-B operator must be declared as a predicate operator, and must be compiled through a different process that returns an Event-B predicate statement. This is done by changing function *evBEvalOp* (6.15) so the expression is compiled with *evBPredExpr(f.expression)* instead of $evBExpr(f.expression)$. The difference in expression mappings can be seen in Section 6.5.

The predicate mapping for the passable operator replaces the lambda expression from the expression operator with a set comprehension expression. i.e. the lambda expression in (6.16) is replaced with:

$$\{p_1 \mapsto \cdots \mapsto p_i \mid p_1 \in P1 \land \cdots \land p_i \in P_i \land evalName(t_1, \ldots, t_i, p_1, \ldots p_i) \tag{6.17}$$

The result of this is that the return type of the operator is a set. In B$^\sharp$ this is a function and can be called e.g. $f(a, b, c)$, the predicate result is obtained with the following expression $a \mapsto b \mapsto c \in f$. When proving this will reduce to a call to the operator defined by $evBEvalOp$.

6.4.5 Instances

Instances are mapped in the following way:

$$\begin{aligned}
&evB(i : instance) \rightarrow expr_{evB}\{\\
&\quad let\ \textbf{Instance}\ \ Class\langle T_1 \ldots T_i\rangle\ \ (E_1, \ldots, E_i)\ \ Name = i\\
&\quad let\ baseClass = evB(Class\langle T_1, \ldots, T_i\rangle)\\
&\qquad return\ baseClass \mapsto evBExpr(E_1) \mapsto \cdots \mapsto evBExpr(E_i)\\
&\}
\end{aligned} \tag{6.18}$$

In the examples of Chapter 4 it was seen that instances did considerably more than this. However, the generation of theorems and operators from instances statements uses large amounts of B$^\sharp$ to B$^\sharp$ mapping and inference which is covered in the next chapter (Chapter 7).

6.4.6 Mapping Types

Due to the similarities between Event-B sets and B$^\sharp$ types, mapping B$^\sharp$ types is largely a matter of mapping the unary types, and mapping the *constructor* to an Event-B set constructor. For each B$^\sharp$ *constructor* there is an equivalent Event-B set constructor making most of the mapping simple. It is assumed that $evB(c : constructor)$ returns the equivalent Event-B $set{-}op_{evB}$ (set constructor). The B$^\sharp$ *Bool* type is mapped to the Event-B $BOOL$ type, however, function types that return booleans are instead mapped to Event-B sets, e.g., the B$^\sharp$ type $a : S \times T \rightarrow Bool$ becomes $a : \mathbb{P}(S \times T)$. Now, if a is called in an expression $a(s, t)$ then it is evaluated in Event-B as $s \mapsto t \in a$ (this evaluation will be seen in the Expression Mapping section below 6.5). This allows predicate functions to be succinctly represented in Event-B. The mapping functions are as follows:

```
$evB(c : cosntrType) \rightarrow set{-}expr_{evB}\{$
   $//\ constrType ::= unaryType\ \ (constructor\ \ constrType)?$
   $//\ e.g., (T \times T) \rightarrow T$
  if $c.constructor =$ ‘ $\rightarrow$ ’ $\wedge\ constrType = Bool$
    $return\ \mathbb{P}(evB(c.unaryType))\ \ //\ \mathbb{P}(T \times T)$
  elif $c.constructor \neq$ ‘ $\rightarrow$ ’ $\vee\ constrType \neq Bool$
    $let\ \bowtie\ = evB(c.constructor)$  // Set constructor equivalent of B$^\sharp$ type constructor
    $return\ evBType(c.unaryType) \bowtie evB(c.constrType)\ \ //\ T_{evB} \times T_{evB} \rightarrow T_{evB}$
$\}$
```
(6.19)

The focus on total function constructors (→) is considered because this is the type of B$^\sharp$ functions. There is no issue in principal to extending this translation route to other functional constructors.

The *unaryType* expressions are more complex to translate, as, in B$^\sharp$ the unary type may be an instance of a type class (either an *instance*, a constrained *paraType*, or a constructed class). When these types appear in a set expression it is the set the type class is associated with that must be found. The Event-B representation stores the set information within its construction. This can be seen in (6.18) and (6.5) where an Event-B type is constructed as the first part of the paired lists which define the Event-B type. More generally the supertype of the current type is represented as the first element of the Event-B construction. The result is that to get the Event-B base type we can recurse up the supertypes until we reach a concrete supertype. We know when the concrete supertype will have been reached, as we can simultaneously recurse up the B$^\sharp$ type hierarchy, until the B$^\sharp$ base type is reached (a constructed type). The *baseTypeForExpr* function is defined to find this type. It recursively descends the B$^\sharp$ type structure while also finding the supertypes in Event-B (the first part of the representation). Once the B$^\sharp$ supertype is not a type class the Event-B expression will be the base class. This is done using the $baseTypeForExpr(e : expr_{evB}, c : class) \rightarrow set{-}expr_{evB}$, the definition of which is given in Appendix A Section A.2.

This function can then be used to get the Event-B type representation from B$^\sharp$ parameters and instances:

```
$evBType(p : parameter) \rightarrow set{-}expr_{evB}\{$
  $p_1 : Class = p$
  $return\ baseTypeForExpr(evBID(p), Class)$
$\}$
```
(6.20)

The function for getting the Event-B instance is near identical and can be found in A.2.

Due to the similarities between Event-B and B$^\sharp$ datatypes getting an Event-B type representation for a B$^\sharp$ datatype is simple:

```
$evBType(d : datatype, \mathbf{params} : List\langle constrType\rangle) \rightarrow set{-}expr_{evB}\{$
$let\ d\langle p_1, \ldots, p_i\rangle = d\langle \mathbf{params}\rangle$
   $return\ evBId(d)\langle evBType(p_1), \ldots, evBType(p_i)\rangle$
$\}$
```
(6.21)

Simply construct the Event-B equivalent to the datatype with the types list.

Finally we need to be able to get the Event-B type for a constructed class e.g., *Setoid*$\langle T\rangle$ where T is a type in the local context. To do this T has to be substituted into the *Setoid*s base type. This is done by descending the B$^\sharp$ type tree until the base type is found, then substituting into the type expression the types from the parametric arguments list with the types in the classes parametric context (when it was defined). B$^\sharp$ allows multiple supertypes, however, they must have the same type structure so it does not matter which supertype is used, and we choose to use the first supertype. The function is defined in the following way:

```
$evBSetForClassConstr(c : class, \mathbf{parametricArgs} : List\langle constrType\rangle) \rightarrow set{-}expr_{evB}\{$
   $let\ sType = head(c.constrType)$ // The First Supertype of the class
   if $sType\ isA\ class$
      $return\ evBSetForClassConstr(sType, \mathbf{parametricArgs})$
   elif $\neg(sType\ isA\ class)$
      // In the supertype class expression replace the $paraTypes$ with the parametric Args
      $return\ evB(substituteInArgs(sType, c.\mathbf{paraType}, \mathbf{parametricArgs}))$
$\}$
```
(6.22)

Getting the Event-B type when a class is constructed in B$^\sharp$ is done with the function (6.22). Subclassing in Event-B can only add additional constraints and required elements, it does not change the type of the elements represented within the type class.

There is a second situation in which it is required to build Event-B types from class constructors, where it is desired to get the type including the representation of the required elements. To do this the *typeOp* (constructed by (6.5)) is used to do the typing. This presents two additional difficulties, firstly, subtypes can be passed in the place of the exact required types, so the parameter type and the argument type may not match identically. The second is caused by the type inference done on parametric contexts discussed in 6.4.2. Continuing the example, if there is an instance of a *Setoid*, S : *Setoid*, the associative operators can be defined with *AssocOp*$\langle S\rangle$, as additional typing was inferred in the operator declaration. The inferred type also needs to be added as an argument e.g., as seen in 6.7 the *AssocOp* constructor actually needs *AssocOp*$\langle S, T : Setoid\langle S\rangle\rangle$.

So when the *AssocOp* constructor is called with an instance of the *Setoid* class it is necessary to also have a type for the first argument (the base type of the *Setoid* instance).

To resolve the first issue (the argument type not matching the parameter type), a function called *evBClassReprForClass* is used to recursively step up the B$^\sharp$ type hierarchy until the correct type is found, whilst simultaneously getting the super types of the Event-B representation. Once the right B$^\sharp$ type is found we know we have the correct Event-B type. The definition of this function is in A.3.

To resolve the inferred types it is enough to notice that when a type is created, the required inferred types are declared within the definition. To understand this it is useful to see an example. Given an instance of a setoid *pNat_Setoid*, and a call $AssocOp\langle pNat\_Setoid\rangle$ the types inferred are exactly the *constrType*s used in the instance declaration to define *pNat_Setoid* (i.e.:$Setoid\langle pNat\rangle$). Doing this inference results in the call being equivalent to
$AssocOp\langle pNat, pNat\_Setoid\rangle$. A series of *getInferredTypes* are declared to get the inferred types from the different unary types in B$^\sharp$. e.g.,:

```
$getInferredTypes(i : instance, c : class) \rightarrow List\langle expr_{evB}\rangle\{$
    $let$ $\textbf{Instance}\ Class\langle T1 \dots T_i\rangle\ (E_1, \dots, E_i)\ \ Name = i$
    $let$ $S_1, \dots, S_i = getInferredTypes(i.constrType)$
    $let$ $cRepr = evBClassReprForClass(i, c)$ // Event-B instance of c          (6.23)
    $return$ $S_1, \dots, S_i, cRepr$ // Type arguments for an operator
$\}$
```

Other functions of this form can be found in Appendix A, Section A.4.

These mapping functions are used whenever a parametric context is filled with type arguments.

6.5 Expression Mapping

There are two separate routes for mapping expressions depending on the Event-B context. If the Event-B context requires an expression then *evBExpr* is called otherwise *evBPred* is called. This can be seen in the mappings above where *evBPred* or *evBExpr* are chosen depending on context. *evBPred* can only be called on expressions in B$^\sharp$ that return a *Bool*. However *evBExpr* can be called on any expression.

6.5.1 Booleans to Predicates and Vice Versa

As seen earlier when mapping a function that returns the *Bool* type, it is transformed into an Event-B set (using set comprehension). Similarly parameters which are functional types which returned *Bool* are mapped to the Event-B power set of the function's domain type. B$^\sharp$ lambda constructs have a similar mapping to the passable functions constructed, so that they produce Event-B sets when required by the Event-B type system:

```
$evBLambda(l : lambda) \rightarrow expr_{evB}\{$
    let $\lambda\langle T_1 : TC_1, \ldots, T_i : TC_I\rangle\; p_1 : P_i, \ldots p_i : P_i \cdot expr = l$
    if $returnType(expr) \neq Bool$
        return $evBLambdaExpr(l)$                                                  (6.24)
    elif $returnType(expr) = Bool$
        return $evBLambdaPred(l)$
$evBLambdaExpr(l : lambda) \rightarrow lambda_{evB}\{$
    let $\lambda\langle T_1 : TC_1, \ldots, T_i : TC_I\rangle\; p_1 : P_i, \ldots p_i : P_i \cdot expr = l$
    $s_1 : S_1, \ldots s_i : S_i = evBParam(T_1 : TC_1), \ldots, evBParam(T_t : TC_i)$
    $q_1 : Q_1, \ldots, q_i : Q_i = evBParam(p_1 : P_1, \ldots, p_i : P_i)$           (6.25)
    return $\lambda\; s_1 \mapsto \cdots \mapsto s_i \mapsto q_1 \mapsto \cdots \mapsto q_i \cdot s_1 \in S_1 \wedge \cdots \wedge p_i \in P_i \mid evBExpr(expr)$
$\}$
$evBLambdaPred(l : lambda) \rightarrow set{-}comp_{evB}\{$
    let $\lambda\langle T_1 : TC_1, \ldots, T_i : TC_I\rangle\; p_1 : P_i, \ldots p_i : P_i \cdot expr = l$
    $s_1 : S_1, \ldots s_i : S_i = evBParam(T_1 : TC_1), \ldots, evBParam(T_t : TC_i)$
    $q_1 : Q_1, \ldots, q_i : Q_i = evBParam(p_1 : P_1, \ldots, p_i : P_i)$           (6.26)
    return $\{s_1 \mapsto \cdots \mapsto q_i \mid s_1 \in S_1 \wedge \cdots \wedge q_i \in Q_i \wedge evBPred(l.expression)\}$
$\}$
```

As seen above whenever a B$^\sharp$ parameter (or a lambda expression) has a type of a function to a *Bool* it becomes a set expression (rather than an Event-B function). Normally when a parameter is of a functional type and is called as a function, it is translated to an Event-B expression in the following way:

```
$evBExpr(p : parameter, \mathbf{args} : List\langle expression\rangle) \rightarrow func\text{-}call_{evB}\{$
    let $p_{evB} = evBId(p)$
    let $arg_1 \mapsto \cdots \mapsto arg_n = evBExpr(\mathbf{args})$                     (6.27)
    return $p_{evB}(arg_1 \mapsto \cdots \mapsto arg_i)$
$\}$
```

The functional parameter can be called as an Event-B style functional type.

If the parameter is a function type that returns a *Bool*, the *evBId* call will give a parameter represented by a set rather than an Event-B functional type, so the above compilation will not work. For example, if we have a parameter $p : T \rightarrow Bool$ in B$^\sharp$, and an expression of $p(t)$ (where $t : T$) then the mapping $func{-}call_{evB}(evBId(p), evBExpr(t))$ will not produce valid Event-B (a typing error will exist). To solve this an $\in_{evB}$ expression is used instead, e.g. the function call is mapped to $evBExpr(t) \in evBId(p)$. This works fine when translating as a predicate expression, however, it can also be translated as an expression. In this case the Event-B *bool* operator is used to convert it into a boolean $bool(evBExpr(t) \in evBId(p))$. An identical process is used to convert lambda calls as necessary (replacing p with the lambda expression).

```
$evBParamCallBoolPred(p : parameter, \mathbf{args} : List\langle expression\rangle) \rightarrow pred\text{-}expr_{evB}\{$
    $let\ p_{evB} = evBId(p)$
    $let\ arg_1 \mapsto \cdots \mapsto arg_i = evBExpr(\mathbf{args})$                    (6.28)
    $return\ arg_1 \mapsto \cdots \mapsto arg_i \in p_{evB}$
$\}$
```

The *evBParamCallBoolExpr* function simply wraps the result of the *evBParamCallBoolPred* in an Event-B *bool* operator. Again this process is identical when calling lambdas. B$^\sharp$ named functions (functions that are not lambdas) can also return booleans. As seen previously they become Event-B predicate operators. When a named function is called it becomes an Event-B predicate operator call. Again, if it is required to return an expression rather than a predicate it is wrapped in the Event-B bool function.

It is possible that a B$^\sharp$ function will be called that returns a *Bool* and Event-B expects an expression. The Event-B operator constructed by the B$^\sharp$ function will return an Event-B predicate. If an expression is required, the Event-B *bool* operator is used to turn the predicate into an expression.

If a Datatype is created over the B$^\sharp$ *Bool* type (e.g., $List\langle Bool\rangle$) the mapped deconstructors may return the Event-B *Bool* type. If this is required to be an Event-B predicate, it is transformed into a predicate by appending $= TRUE$ to the result of the mapped expression.

From this point on it is assumed that these translations between B$^\sharp$ boolean and Event-B predicates/*Bool*s are made automatically when required. These are all the additional translations that are necessary to map B$^\sharp$ booleans to Event-B. Functional types that do not return a *Bool* cannot be converted into an Event-B predicate.

6.5.2 Function Calls

The two routes of translating expressions (*evBExpr* and *evBPred*) are near identical with the exception of the function call i.e., replacing *Expr* with *Pred* in the function names. This section therefore only presents functions to map B$^\sharp$ expressions to Event-B expression. The small differences in the mapping of predicate expressions are explained along the way.

$$
\begin{array}{l}
\mathit{evBExpr}(\mathit{fc} : \mathit{functionCall}) \rightarrow \mathit{expr}_{\mathit{evB}}\{\\
\quad \mathit{match}\ \mathit{fc.functionName}\\
\qquad \left.\begin{array}{l} \mid \mathit{functionCall}\\ \mid \mathit{lambda}\\ \mid \mathit{parameter}\end{array}\right\} \left(\begin{array}{l} \mathit{let}\ \mathit{funcNameExpr} = \mathit{evBExpr}(\mathit{fc.functionName})\\ \mathit{let}\ \mathit{arg}_1 \mapsto \cdots \mapsto \mathit{arg}_j = \mathit{evBExpr}(\mathit{fc}.\mathbf{expression})\\ \mathit{return}\ \mathit{funcNameExpr}(\mathit{arg}_1 \mapsto \cdots \mapsto \mathit{arg}_j)\end{array}\right)\\
\qquad \mid \mathit{deconstrutor} : \left(\begin{array}{l} \mathit{let}\ \mathit{deconName} = \mathit{EvBId}(\mathit{deconstructor})\\ \mathit{let}\ \mathit{arg} = \mathit{evBExpr}(\mathit{fc}.\mathbf{expression})\ \text{// Deconstructors take an instance of the datatype}\\ \mathit{return}\ \mathit{deconName}(\mathit{arg})\end{array}\right)\\
\qquad \mid \mathit{function} : \left(\begin{array}{l} \mathit{let}\ \mathit{funcName} = \mathit{evBEvalId}(\mathit{function})\\ \mathit{let}\ \mathit{TArg}_1, \ldots, \mathit{TArg}_i = \mathit{getInferredTypes}(\mathit{fc}.\mathbf{ctx}, f.\mathbf{function}.\mathit{constrType.class})\\ \mathit{let}\ \mathit{arg}_1, \ldots, \mathit{arg}_j = \mathit{evB}(\mathit{fc}.\mathbf{expression})\\ \mathit{return}\ \mathit{funcName}(\mathit{TArg}_1, \ldots, \mathit{TArg}_i, \mathit{arg}_1, \ldots, \mathit{arg}_j)\end{array}\right)\\
\}
\end{array}
\tag{6.29}
$$

There is also an *evBPredExpr*, which is called when Event-B requires a predicate instead of an expression. This is almost identical except for the changes discussed in Section 6.5.1, and $\mathit{let}\ \mathit{funcName} = \mathit{evBEvalId}(\mathit{function})$ becomes $\mathit{let}\ \mathit{funcName} = \mathit{evBPredId}(\mathit{function})$.

$B^\sharp$ functions can appear without arguments. If there are no arguments then the constructed passable operator is used (i.e., $\mathit{let}\ \mathit{funcName} = \mathit{evBEvalId}(\mathit{function})$ becomes $\mathit{let}\ \mathit{funcName} = \mathit{evBPassableId}(\mathit{function})$. Other than this the call is the same. The only exception to this is if the $B^\sharp$ function does not expect any arguments, in this case the *evBEvalId* is always used. Only $B^\sharp$ functions that return *Bool* can produce Event-B predicates, the result of this is that when translating a $B^\sharp$ function call to a predicate expression it is simply necessary to call the Event-B operator with the correct arguments.

Unlike in object oriented languages, there is no notion of being able to mutate a class or datatype. The result is that if you have a class with a parametric context, and you create an instance of the class with the supertype and the subtype, the class created with the subtype is a subtype of the class created with the supertype (covariance). This is ensured by the translation to Event-B sets. Functions take covariant arguments (you can call a function with a subtype), and give out contravariant results (they are only guaranteed to return the supertype). This is not enforced in $B^\sharp$. The place where this could become problematic is checking if the result of one function can be the argument to another function. When the $B^\sharp$ is translated to Event-B the corresponding Event-B generates well definedness proof obligations to check the arguments of functions. The result is that if the type information of the argument says it is a supertype of the function parameter, the user will be required to prove that the argument is actually the more constrained type required by the parameter.

6.5.3 Quantifiers

B$^\sharp$ quantifiers map to their equivalent Event-B quantifier ($evB(q : quantType)$ returns the Event-B equivalent of the $quantType$). When a quantifier has a parametric context, an additional universal quantifier is constructed to quantify over the parametric types. This results in the following translation function:

```
$evB(q : quantifier) \rightarrow quantifier_{evB}$ {
    $let\ \Lambda\langle T_1 : TC_1, \dots T : TC_i\rangle p_1 : P_1, \dots, p_j : P_j \cdot predExpr = q$
    $let\ s_1 : S_1, \dots s_i : S_i = evBParam(T_1 : TC_1), \dots, evBParam(T_i : TC_i)$
    $let\ q_1 : Q_1, \dots, q_j : Q_j = evBParam(p_1 : P_1), \dots, evBParam(p_j : P_j)$          (6.30)
    $let\ \Lambda_{evB} = evB(q)$  // $\Lambda_{evB}$ is either $\forall$ or $\exists$
    $return\ \forall s_1 \mapsto \dots \mapsto s_i \cdot s_1 \in S_1 \wedge \dots s_i \in S_i$
        $\Rightarrow \Lambda q_1 \mapsto \dots \mapsto q_j \cdot q_1 \in Q_1 \wedge \dots \wedge q_j \in Q_j \rightarrow evBPred(q.expression)$
}
```

In the case that there are no parametric types in the B$^\sharp$ quantifier, the first universal quantifier in the return statement is not generated.

6.5.4 Theorems

Theorem declarations in B$^\sharp$ and Event-B are almost identical, except in B$^\sharp$ the expression must be of type *Bool* rather than being a predicate expression. The result of this is that the mapping is very simple:

```
$evB(t : theorem) \rightarrow theorem_{evB}$ {
    $let\ TheoremName = evBId(t)$
    $let\ thmExpression = evBPred(t.expression)$                    (6.31)
    $return \begin{pmatrix} TheoremName : \\ thmExpression \end{pmatrix}$
}
```

This demonstrates all of the mapping functions required to map a B$^\sharp$ expression into an Event-B expression.

6.6 Conclusion

This chapter has demonstrated how the core elements of $B^\sharp$ are translated into the Event-B language. The largest difference between the $B^\sharp$ language and the Event-B language is the type class definitions. It was shown how the type class definitions become Event-B sets, which were encapsulated in Event-B operators. This allows easy reuse of the definitions e.g., groups can easily reuse the definitions in Monoids etc. It also allows one type to depend on another e.g., the type of associative operators depending on the setoid type allowing the use of the equivalence relation. To allow type class definitions to be separated into their constituent parts deconstructor operators were also defined.

These type class definitions also created differences between the $B^\sharp$ and Event-B type systems, e.g., the *Monoid* type class is a subtype of the *Setoid* type class therefore could be passed in the place of a *Setoid*. This is not the case with passing the Event-B representation of a type class to an Event-B operators.. It was shown how the Event-B representation of the type class could be deconstructed to find the correct representation to pass to operators. A similar approach was necessary for instances of type classes.

Many elements of expressions translated readily to their Event-B equivalents. Getting associated variables from type classes was done using the operators generated to deconstruct type classes. A major difference between the $B^\sharp$ and Event-B language was the representation of predicates and booleans. This was resolved by having a separate translation route for $B^\sharp$ function types that returned a boolean. This also made it necessary to change the translation when these functional types were called.

Chapter 7

Inference and Structure

In the previous chapter the core of B$^\sharp$ was mapped to Event-B. However, this is only part of the work done during the mapping from B$^\sharp$ to Event-B. B$^\sharp$ statements contain additional meaning which has not been encoded during the expression mapping. For instance, due to B$^\sharp$ allowing reasoning about non-complete types, functions have a user defined return type. This return type information needs to be encoded into the Event-B mapping. This chapter will look at how this additional information is encoded within Event-B by generating additional Event-B statements. These additional Event-B statements are used to prove that constraints declared in B$^\sharp$ are satisfied. E.g., when a user declares a return type to a function it is necessary to show that the function really does return this type. This becomes necessary in B$^\sharp$ as subtying is not decidable. Therefore, an Event-B theorem is generated requiring the user to show the return type is what they claim.

This chapter then goes on to look at how the structure of B$^\sharp$ files influences the structure of the generated Event-B files, e.g., how B$^\sharp$ method of importing files is translated to Event-B file inclusion.

7.1 Generated Statements

The approach taken in this section is to generate additional B$^\sharp$ statements and then assume that these are mapped to Event-B using the mapping functions in the previous section. The statements are not generated using the concrete B$^\sharp$ syntax, instead they are generated within the abstract syntax and are not visible to the user. They are considered an intermediate step within the translation. This approach is taken because it is simpler to generate new B$^\sharp$ statements from B$^\sharp$ than it is to generate Event-B.

In many cases the generated statements are B$^\sharp$ theorems, e.g., a theorem that demonstrates that a function returns the type it claims to. These bare similarities to proof obligations generated by Event-B. The reason for these theorems is to facilitate proving (often discharging the proof obligations generated by Event-B). For example, in Event-B we may have a function *F1*, where one of the arguments is the result of another function *F2*. Event-B will generate a well-definedness statement that *F2* returns the correct type for the parameter of *F1*. Already having a theorem about the returned type of *F2* will in many cases make discharging this proof obligation trivial. Ideally this process could be automated which is discussed further in 9.3. If instead, we generated Event-B proof obligations we would not be able to use the result the same way in later proofs.

7.1.1 Function Typing

Function typing is in effect the user stating a theorem that the function returns the type they believe it does. As subtypes are not in general decidable it can be necessary for the user to prove that the function returns the expected type. This is done by generating a typing theorem in B$^\sharp$ using the following translation function:

$$
\begin{aligned}
&funcTheorem(f\ :function) \rightarrow theorem\ \{\\
&\quad let\ TypeTheoremName = thmID(d)\ //\ \text{A B}^\sharp\text{ identifier for the theorem}\\
&\quad let\ fName\langle T_1\ :\ TC_1, \dots T_i\ :\ TC_i\rangle(p\ :\ P_1, \dots p_j\ :\ P_j)\ :\ RetType\ \ expression = f\\
&\quad return\ \begin{pmatrix} TypeTheoremName: \\ \quad \forall\langle T_1\ :\ TC_1, \dots, T_i\ :\ TC_i\rangle p_1\ :\ P_1, \dots p_j\ :\ P_j \\ \quad\quad \cdot fName\langle T_1, \dots, T_i\rangle(p_1, \dots, p_j) \in RetType \end{pmatrix}\\
&\}
\end{aligned}
\tag{7.1}
$$

This defines a theorem which quantifies over the parametric types, and the parameters. These become the arguments to the function, and an ∈ expression is used to make sure the type of the function with these arguments is the type the user provided. This is then converted into Event-B using the function from the previous chapter.

7.1.2 Class *paraType* Inference

The *typeBody* and *instName*, and *extend* ((4.7)) constructs were not included in the core mappings Chapter 6. Their purpose within the B$^\sharp$ language can be explained entirely as B$^\sharp$ to B$^\sharp$ translations/inference. To understand the strength of the instance declarations it is important here to understand this type inference. As discussed in example (4.7) classes introduced instance variables, which can be used as a generic instance of the class anywhere within the class body (or one of its extensions). If the instance variable is referenced within an expression, the function or theorem is re-written to introduce an additional *paraType* to represent this instance.

For example within the *Group* type body there is the following theorem:

$$\begin{aligned}&IdentUnique :\\ &\quad \forall x, y : G \cdot equ(op(x, y), y) \Leftrightarrow equ(x, ident)\end{aligned}$$

This references the containing *Group* type in several ways firstly the explicit reference to *G*, which is the groups instance name. Then there are the references to *equ*, *op* and *ident* which are the associated elements of the *Group*. To translate this the implicit reference is made explicit in the following way:

$$\begin{aligned}&IdentUnique :\\ &\quad \forall \langle G : Group\rangle x, y : G \cdot G.equ(G.op(x, y), y) \Leftrightarrow G.equ(x, ident)\end{aligned}$$

The type class $G : Group$ is added to the parametric context of the expression.

This *paraType* will be referred to as 'the class paratype'. The class paratype is generated with the following function:

$$\begin{aligned}&makeInstParaType(C : class) \rightarrow paraType\{\\ &\quad return\ C.instName : C\\ &\}\end{aligned} \tag{7.2}$$

Functions and theorems appear only within class bodies, however, they may not have any references to the class or the classes associated elements. The first step of the process is to search the theorem and function expressions for references to the type class. This is done with the *findParaType* function which recurses over the expression tree to see if the class paratype occurs, if the class *paraType* is found then the the function/theorem is altered to allow its use, using one of the following functions:

$$\begin{aligned}
&addClassParaToFunc(f : function, C : class) \rightarrow function\{\\
&\quad \textit{let}\ fName\langle T_1 : TC_1, \dots T_i : TC_i\rangle(p : P_1, \dots p_j : P_j) : RetType\ \ expression = f\\
&\quad \textit{let}\ I : C = makeInstParaType(C)\\
&\quad \textit{return}\ fName\langle I : C, T_1 : TC_1, \dots T_i : TC_i\rangle(p : P_1, \dots p_j : P_j) : RetType\ \ expression = f\\
&\}
\end{aligned} \tag{7.3}$$

$$\begin{aligned}
&addClassParaToTheorem(C : class, t : theorem) \rightarrow theorem\{\\
&\quad \textit{let}\ thmName : expr = t\\
&\quad \textit{let}\ I : C = makeInstParaType(C)\\
&\quad \textit{return}\ \forall\langle I : C\rangle\ expr\\
&\}
\end{aligned} \tag{7.4}$$

addClassParaToFunc prepends the class *paraType* to the functions list of **paraType**s, the *addClassToTheorem* function adds a universal quantifier to add the class *paraType* to the theorem. If the expression already starts with a universal quantifier the *classParaType* can be added to this rather than creating a new quantifier. Once this substitution has been done, the functions and theorems can be mapped using the core mapping rules.

In the examples the associated elements could be referenced directly within expressions, e.g., operators within the monoid class could use the monoid op directly within their expressions. In practice this is handled by scoping, and the statement $op(a, b)$ is treated identically to $M.op(a, b)$ (this is covered in more detail in 7.2). The result of this translation is that if an associated element is referenced within an expression, then the type class has been implicitly referenced and the *classParaType* needs to be added to expressions using the one of the functions above.

7.1.3 *paraType* Substitution

When mapping instance statements to Event-B the theorems and functions from the type class are instantiated during the mapping (this is covered in more detail in the section below 7.1.4). To facilitate this mapping B$^\sharp$ allows paratypes to be substituted with instance types. As an example:

$$addMon\_raiseTo : pNat = raiseToL\langle M := addMon\rangle \tag{7.5}$$

This substitution is expanded to give:

$$
\begin{array}{l}
addMon\_raiseTo(x : addMon, p : pNat) : addMon \\
\quad \textbf{match}\ p\ \{ \\
\quad\quad |\ zero : zero \\
\quad\quad |\ suc(p) : addMon\_raiseTo(x, addMon\_raiseTo(x, p)) \\
\quad \}
\end{array}
\tag{7.6}
$$

A function $subParaType(e : expression, \mathbf{p} : List\langle paraType\rangle, \mathbf{c} : List\langle constrType\rangle) \rightarrow expression$ Is used to replace *paraType*s with *constrType*s within an expression. The expression is descended recursively, and if a *paraType* is found that is in the list **p**, it is replaced by the equivalent *constrType* in **c** (the *constr* type with the matching index in the list). This can be used to define theorems and functions that are instantiated with the instance:

```
$instFunc(f : function, \mathbf{p} : List\langle paraType\rangle, \mathbf{c} : \langle constrType\rangle) \rightarrow function\{$
  $let\ oldFuncName\langle S_1 : STC_1, \dots S_i : STC_i T_1 : TC_1\rangle(p_1 : P_1, \dots p_j : P_j) : RetType\ \ expr = f$
  $let\ instFuncId = instFuncId(f, \mathbf{p}, \mathbf{c})$
  $let\ p_1 : P_{sub1} \dots p_j : P_{subj} = p_1 : subParaType(P_1, \mathbf{p}, \mathbf{c}), \dots$
  $let\ newRetType = subParaType(RetType, \mathbf{p}, \mathbf{c})$
  $let\ newExpr = subParaType(expr, \mathbf{p}, \mathbf{c})$
  $return\ instFuncId\langle T_1 : TC_1, \dots, T_i : TC_i\rangle(p_1 : P_{sub1}, \dots, p_j : P_{subj})\ newRetType\ \ newExpr$
$\}$
```
(7.7)

Any parametric type that has a concrete replacement is removed from the parametric context of the function and, where the parametric type was referenced in the rest of the function declaration, it is replaced by the concrete type. The same process is used to instantiated expressions in theorems where parametric types are introduced by quantifiers.

When substituting *paraTypes* in expressions there may be function calls made which could be replaced by a previously instantiated function call. Where possible B$^\sharp$ will use an instantiated function, rather than letting the instantiation happen in Event-B. Using a previously instantiated function eases proofs using the interactive prover, as instantiation and well-definedness proofs are avoided. If an abstract function is used, then to use a theorem about the function we must use Event-B instantiation to instantiate the theorem with the correct types. A series of proof obligations are generated to ensure that the types used in the Event-B instantiation match the types of any function parameters where they are used as arguments. This results in the user having to do additional work to discharge these proof obligations. If there is a previously available B$^\sharp$ instantiation the well-definedness proof obligations about the parametric type will not be present as the parametric type will not appear in the Event-B representation (it is removed by the instantiation). The example language used thus far is not sufficient to describe the method of finding instantiated functions succinctly, instead this is described below using set theory.

Within the translation language the substitution functions took a list of *paraType* and a list of *constrType*, this was really doing the job an unordered map (*map*) defining a one to one relationship between *paraTypes*s and *constrTypes*:

$$map \subset paraType \times constrType$$

When the function f is instantiated a new function is created using the *map* to substitute the *paraType*s in the function expression with the associated *constrType*s. The newly instantiated function it is given a name based on the name of the original function and the instantiation map. The result is that if the function is used in an expression with the same instantiation map, the already instantiated function can be used.

If a substitution with a map *sub_map*, is happening on a function call with f : *function* and $\mathbf{c}$: $List\langle constrType\rangle$, it is desirable to use an instantiated function where possible. To do this we note there are two maps involved in the function call, the one between the functions parametric context f.**paraType** and the *constrType*s in the call. These will be called *para_map*, and the substitution map *sub_map* respectively. The goal is to generate a map that can be used to find an instantiated function $inst\_map$. $inst\_map$ is generated in the following way:

$$\begin{aligned} restr\_map &= para\_map \rhd dom(sub\_map) \\ inst\_map &= \{x, y | x \in dom(restr\_map) \land y \in sub\_map(para\_map(x))\} \end{aligned} \tag{7.8}$$

If an instantiated function f_{inst} is found for a function *fc* then the function call is modified to the following (i.e., the instantiated function is used):

$$\begin{aligned} &functionCall(functionName = f_{inst}, \\ &\qquad \mathbf{constrType} = fc.\mathbf{constrType} - dom(para\_map), \\ &\qquad \mathbf{expresion} = fc.\mathbf{expression}) \end{aligned} \tag{7.9}$$

Note the subtraction of the domain of *para_map* is not recursive, so only removes *constrType*s if they are an exact map for the parametric type in *para_map*.

Once this transformation has been made the generated function call then has the normal substitution rules applied to it.

This system of finding an already instantiated function is not perfect, as it does not deal with partial matches e.g., given f instantiated with $f\langle S ::= A, T\rangle$ (T is not instantiated) and a function call $f\langle A, B\rangle(\dots)$ the map generated by the call is $S ::= A, T ::= B$ only partially matches the substitution map so it is not instantiated. However, in B$^\sharp$ instantiation is currently only used when an instance of a type class is created, and this method of instantiation works well in this case as there is only one value in the instantiation map.

7.1.4 Instance Statements

Along with creating the Event-B representation of an instance a B$^\sharp$ instance statement will generate many other statements. The most important of which is a theorem that the defined instance is an instance of the type class. The expression ($pred{-}expr_{evB}$) for this theorem is generated with the following mapping:

$$\begin{aligned} & instIn(i : instance) \rightarrow pred{-}expr_{evB}\{ \\ & \quad let\ \textbf{Instance}\ Class\langle T_1, \ldots, T_i\rangle\ (e_1, \ldots e_i)\ IName = i \\ & \quad return\ evB(i) \in evB(Class\langle T_1, \ldots, T_i\rangle) \\ & \} \end{aligned} \tag{7.10}$$

This is then wrapped in an Event-B theorem.

To ease proofs within the interactive theorem prover it is useful to have the theorems and functions from the B$^\sharp$ type class instantiated for the instance (these are instantiated in B$^\sharp$ then translated to Event-B). The instantiation reduces the complexity of theorems and methods making them easier to use within proofs. To instantiate these theorems and proofs the concepts introduced in the previous two sections are used (the class *paraType* inference 7.1.2 and substitution 7.1.3). In each of the *typeBody*s associated with the instantiated type class (the type body of the type class, its supertypes, and any extensions which are in scope) all of the contained theorems and functions are instantiated with the mapping of the inferred class parameter to the instance. The instantiated statements are then mapped to Event-B using the core mappings 6.4. The function typing theorems are also generated for the mapped function. Provided the theorem generated to show the instance is a member of the type class is provable, all of the additional theorems defined are true by construction, and do not need to be proved. These theorems were proved on the abstract type class, which proved that the theorems were true for all members of the type class. The initial theorem generated by the instance statement required the user to show that the instance is a member of the type class. As the theorems being instantiated are true for all instances of the type class, and the user has had to prove that we have an instance of the type class, the instantiated theorems must be true.

In the previous section 7.1.3 the issue of trying to find the best instantiated function to use within a function call was discussed. The instantiation done within an instance statement resolves this issue as a single *paraType* is mapped to an instance. The result is that an instantiated function can either be found or not found (if it does not exist), if it is found then it is used.

7.2 Naming, Scoping and Importing

As seen in the examples 4.3 B$^\sharp$ has a notion of scopes. A scope defines for how long a named element is visible, and every B$^\sharp$ element that itself contains named declarations (declarations with *Id*s) generates a new scope. For example, given the function declaration:

$$function ::= Id\ paraType{*}\ parameter{*}\ constrType\ expression \tag{7.11}$$

The function element contains named elements *paraType* and *parameter*. The names of these elements become part of the functions scope, and the declared elements can be used within the function after their declaration. However, they cannot be used outside of the function declaration. This allows another function declaration to reuse the same names of the *paraType*s without any conflict or confusion. This extends to the general rules of scoping in B$^\sharp$:

1. Elements which contain other named elements create a new scope, this scope ends when the rule for the element ends.
2. When a named element is declared its identifier is added to the scope in which it is declared
3. Identifiers are not visible outside the scope they are declared in
4. Every declaration in a scope must have a unique identifier (this does not include declarations within embedded scopes).
5. Subtypes and *Extend*s extend the scope of the classes they subtype/extend. This means that names from the supertype/extended class cannot be reused.

These rules allow different classes to have required elements with the same names, or different functions to have parameters with the same name.

An exception to this rule are function declarations, constructors and destructors (which can be viewed as special functions). These declarations are added to scope of the *file* element, not the element they are declared in. This allows them to be used outside of the *datatype*/*class*/*extend* in which they were declared. This also means that their name must be unique within the file scope. As outlined in 4.3 it is expected that implementations will use inference to generate the function identifier from the function name and class name, allowing function names to be reused as expected. This is covered in detail in the implementation Chapter 8

B$^\sharp$ theories can be declared in multiple files. This results in it being necessary to reference theories declared in other files. The system used within B$^\sharp$ is to allow the import of individual top level elements (i.e., *datatype*/*class*/*extend* from another file). The result of this import is similar to redeclaring that top level element in the current file. Imports only include the imported top level element they import, they do not include other declarations from the same file, this is important for namespacing. Only elements from the imported top level element are added to the namespace of the current file. To import a top level element an *import* statement is used. It uses an identifier to reference a *datatype*/*class*/*extend* from another file. The result of this is to import all of the file level statements from the imported elements e.g., if a datatype is imported then the datatype identifier, the constructor and destructor identifiers, and the function identifiers are all added to the current file scope. This allows them to be used within the current file for defining new theories.

7.3 Conclusion

The focus of this chapter was how non-core elements of $B^\sharp$ were translated to core elements of $B^\sharp$. Along with this it was seen how additional $B^\sharp$ statements were generated, representing theorems about the type systems, and instances. Finally a description of scoping in $B^\sharp$ was given.

A large part of translating non-core elements to core elements was dealing with type bodies. Within expressions in type bodies the user can use elements from the type class they are within. To translate to a core Event-B representation it was shown how the implicit reference to the type class was made explicit, by adding the type class to the function/quantifiers parametric context.

Many additional $B^\sharp$ statements were generated. Many of these were translating implicit theorems in $B^\sharp$ into explicit theorems. This included the return type of functions, and the typing of **Instance** statements. Along with this **Instance** statements also generated many additional $B^\sharp$ theorems where the theorems of their type classes were instantiated. The result of this was that these theorems will not need to be instantiated when proving using the interactive prover.

Chapter 8

Implementing B$^\sharp$

To demonstrate the feasibility of the B$^\sharp$ language an implementation of the translations introduced in the previous chapters was produced. This chapter discusses that implementation, including:

1. Tools used by the implementation
2. The concrete syntax used within the implementation
3. Additional features of the implementation
4. The mathematical types written within B$^\sharp$ and how successful the translations were.
5. The lessons and improvements learnt from the implementation

8.1 Implementation Tools

This section briefly outlines the tools and techniques used to build a tool to develop mathematical theories in B$^\sharp$.

The B$^\sharp$ implementation was written using a tool called Xtext [9]. Xtext is a tool for creating domain specific languages, and associated IDEs. It was chosen due to its close integration with Eclipse and its feature set (allowing many features of modern IDEs to be added with relative ease). Xtext takes a grammar definition written using an extended BNF style syntax. From this it generates a parser, allowing statements in the defined language to be transformed into an abstract syntax tree. In practice this abstract syntax tree is a tree in EMF (Eclipse Modelling Framework) [81]. The abstract syntax tree is easier to reason about than a raw text file, and is used to implement many of the features of the B$^\sharp$ tool, including the translation to Event-B. Many of the other tools within the Rodin environment use a similar approach of generating an EMF tree to ease reasoning. This includes the standard Event-B syntax, but not the extended syntax used by the Theory Plug-in. Having an abstract syntax tree as the stored representation allows other tools to translate directly to the Event-B syntax. For example UML-B [77] generates an EMF tree, which is then translated directly to the EMF which used by standard Event-B with a series of translation rules. This EMF tree is then used to display the text which the user sees. Once Xtext has generated an abstract syntax tree it is up to the user to write a program to use the abstract syntax tree, in our case to define Event-B structures.

Along with the creation of an abstract syntax tree Xtext also automatically adds many IDE features such as syntax highlighting, error highlighting, and content assist (suggestions for the next elements, and autocompletes). These features can be programmatically customised to suit the language being constructed.

An alternative approach would have been to create an editor for the Event-B syntax, then to have a feature for adding additional syntax elements using inference rules (in a similar way to ML). The advantages of the approach taken using Xtext, is that from the language definition it creates a code editor which plugs directly into the Rodin platform with the features outlined above. It is also easier to read new definitions in Xtext than using inference rules. The reason that the Xtext syntax was not used within Chapter 6, is that it includes the concrete syntax definitions which are not relevant to the translations there.

The features outlined above are used in the rest of the chapter to construct an IDE for the B$^\sharp$ language, and translate it to Event-B.

8.2 Concrete Syntax

This section formalises the concrete syntax already seen in the examples in Chapter 4. It first introduces the BNF style syntax used to describe the concrete representation. It then presents an example of how to read the syntax. Finally it shows the formalised concrete syntax for the core of the B$^\sharp$ language.

The concrete syntax is described using a similar syntax to the one used to describe B$^\sharp$'s abstract syntax in Chapter 4. In the concrete syntax it is also necessary to introduce separators, scope markers and keywords to allow the language to be parsed in a unique way. Without these separators it is in general impossible for a parser to tell the difference between the end of one declaration and the beginning of the next. Syntax elements are introduced using single quotes, as an example in the implementation identifiers are constrained to only contain certain characters, this is defined with the following rules:

$$\begin{aligned}
asciiChar &::= \text{‘}a\text{’} \mid \ldots \mid \text{‘}z\text{’} \mid \text{‘}A\text{’} \mid \ldots \mid \text{‘}Z\text{’} \\
asciiNum &::= \text{‘0’} \mid \ldots \mid \text{‘9’} \\
name &::= asciiChar\ \ (asciiChar \mid asciiNum \mid \text{‘\_’})*
\end{aligned}$$

The *asciiChar* rule tells the parser that the next thing to expect is a ascii letter (in long form it says that the next character must be an a or a b, $\ldots$, or a Z). Similarly the *asciiNum* tells the parser to expect a number. The *name* rule says that a *name* must start with a letter, and can then continue with any number of letters or numbers. The *name* rule replaces the abstract notion of an identifier in the abstract syntax. It defines a *name* as a single *asciiChar*, followed by any number of letters, numbers or underscores.

Within the concrete syntax it is necessary to reference elements that have already been created, this was seen in the abstract syntax where the subscript *Id* was used to show the element was an instance rather than an instruction to call the rule. In the concrete syntax this is replace by wrapping the rule *name* in square brackets. When this is done it tells the parser to expect the name of an already declared element (of the correct type), e.g., *[function]* tells the parser to expect the name of an already declared function. It does not tell the parser to expect the *function* rule. This syntax was chosen due to its similarity to the Xtext syntax for defining a language.

A common pattern within the rules is to have a list of elements with a separator. An example of this is the *paraArgs* rule:

$$\textit{paraArgs} \; ::= \; `\langle\text{'} \;\; \textit{constrType} \;\; (`,\text{'} \;\; \textit{constrType})* \;\; `\rangle\text{'} \tag{8.1}$$

This rule declares that the parser should expect a *constrType* optionally followed by a comma and another *constrType*, which can be repeated many times.

It is useful before all of the rules to see an example of a declaration rule:

$$\begin{aligned} \textit{datatype} \; ::= `\textbf{Datatype}\text{'} \;\; &\textit{name} \;\; \textit{parametricContext}? \\ &\textit{constructor}+ \;\; \textit{typeBody} \end{aligned} \tag{8.2}$$

This rule states that to declare a *datatype* one must start with the keyword **Datatype**, followed by providing a *name* (using the previously discussed *name* rule), followed by a parametric context, one or more *constructor*s, and a type body.

The rest of the rules for the core B$^\sharp$ syntax are described in the following Figure 8.1:

$$
\begin{aligned}
\textit{asciiChars} &::= \text{`}a\text{'} \mid \ldots \mid \text{`}z\text{'} \mid \text{`}A\text{'} \mid \ldots \mid \text{`}Z\text{'}\\
\textit{asciiNums} &::= \text{`0'} \mid \ldots \mid \text{`9'}\\
\textit{name} &::= \textit{asciiChars}\ \ (\textit{asciiChars} \mid \textit{asciiNums} \mid \text{`\_'})*\\
\textit{file} &::= [\textit{datatype}] \mid [\textit{class}]\\
\textit{datatype} &::= \text{`}\textbf{Datatype}\text{'}\ \ \textit{name}\ \ \textit{parametricContext}?\\
&\quad \textit{constructor}+\ \ \textit{typeBody}\\
\textit{parametricContext} &::= \text{`}\langle\text{'}\ \ \textit{paraType}\ \ (\text{`,'}\ \ \textit{paraType})*\ \ \text{`}\rangle\text{'}\\
\textit{paraType} &::= \textit{name}\ \ (\text{`:'}\ \ [\textit{class}])?\\
\textit{constructor} &::= \text{`}|\text{'}\ \ \textit{name}\ \ \textit{destructor}\ \ (\text{`,'}\textit{deconstrutor})*\\
\textit{destructor} &::= \textit{name}\ \ \text{`:'}\ \ \textit{constrType}\\
\textit{constrType} &::= (\text{`('}\ \ \textit{type}\ \ (\text{`}\times\text{'}\ \ \textit{constrType})?\ \ \text{`)'}) \mid (\textit{type}\ \ (\text{`}\times\text{'}\ \ \textit{constrType})?)\\
\textit{type} &::= [\textit{paratype}] \mid ([\textit{datatype}] \mid [\textit{class}])\ \ \textit{paraArgs}?\\
\textit{paraArgs} &::= \text{`}\langle\text{'}\ \ \textit{constrType}\ \ (\text{`,'}\ \ \textit{constrType})*\ \ \text{`}\rangle\text{'}\\
\textit{typebody} &::= (\textit{function} \mid \textit{theoremBody})*\\
\textit{function} &::= \textit{name}\ \ \textit{paramList}?\ \ \textit{constrType}\ \ \textit{expression}\\
\textit{paramList} &::= \text{`('}\textit{parameter}\ \ (\text{`,'}\ \ \textit{parameter})*\text{`)'}\\
\textit{parameter} &::= \textit{name}\ \ \text{`:'}\ \ \textit{constrType}\\
\textit{theoremBody} &::= \text{`\{'}\ \ \textit{theorems}*\ \ \text{`\}'}\\
\textit{theorems} &::= \textit{thmName}\ \ \text{`:'}\ \ \textit{expression}\text{`;'}\\
\textit{class} &::= \text{`}\textbf{Class}\text{'}\ \ \textit{name}\ \ \textit{parametricContext}?\ \ \text{`['}\ \ \textit{instName}\ \text{`]'}\\
&\quad (\text{`:'}\ \ \textit{constrType}\ \ (\text{`,'}\ \ \textit{constrType})*)?\ \ \textit{paramList}\ \ \textit{where}?\ \ \textit{typeBody}\\
\textit{where} &::= \text{`}\textbf{where}\text{'}\ \ \textit{expression}\ \ (\text{`;'}\ \ \textit{expression})*)?\\
\textit{instName} &::= \textit{name}\\
\textit{functionCall} &::= \textit{funcName}\ \ \textit{argList}?\\
\textit{argList} &::= (\text{`('}\textit{expression}\ \ (\text{`,'}\ \ \textit{expression})*\text{`)'})\\
\textit{expression} &::= \textit{functionCall} \mid \textit{quantifier} \mid \textit{lambda}\\
\textit{functionName} &::= [\textit{parameter}] \mid [\textit{deconstructor}] \mid [\textit{function}]\ \textit{paraArgs}?\\
&\quad \mid [\textit{constrType}]\\
\textit{quantifier} &::= (\text{`}\forall\text{'} \mid \text{`}\exists\text{'})\ \textit{parametricContext}?\ \ \textit{paramList}?\ \ \text{`}\cdot\text{'}\ \ \textit{expression}\\
\textit{lambda} &::= \text{`}\lambda\text{'}\ \ \textit{parametricContext}?\ \ \textit{paramList}?\ \ \text{`}|\text{'}\ \ \textit{expression}\\
\textit{extend} &::= \text{`}\textbf{Extend}\text{'}\ \ ([\textit{class}] \mid [\textit{datatype}] \mid [\textit{extend}])\ \ \text{`('}\ \ \textit{name}\ \ \text{`)'}\ \ \textit{typeBody}\\
\textit{instance} &::= \text{`}\textbf{Instance}\text{'}\ \ [\textit{class}]\ \ \textit{parametricContext}\ \ \textit{argList}\ \ \textit{name}
\end{aligned}
$$

Figure 8.1: B$^\sharp$ language

For a complete description of the B$^\sharp$ syntax see Appendix B, or the B$^\sharp$ implementation code. [1]

In the complete description there are some small differences in the description of the syntax for practical reasons. Xtext does not allow left recursion so to resolve this Xtext has a syntax for reordering the generated syntax tree (making the expression syntax more complicated). In Xtext elements need to be named so they can then be identified in the generated abstract syntax tree, e.g., in Xtext the *parametricContext* rule would become:

$$\begin{gathered}\mathit{ParametricContext}\ :\\ \text{‘}\langle\text{’}\mathit{paraType}{+}{=}\mathit{paraType}\ (\text{‘,’}\ \mathit{paraType}{+}{=}\mathit{paraType}){*}\ \text{‘}\rangle\text{’}\end{gathered} \tag{8.3}$$

Within the abstract syntax tree this causes a list named *paraType* to be constructed referencing the *paraType*s.

8.3 Translation

Given the concrete syntax Xtext generates a parser. Using this B$^\sharp$ statements are parsed and an abstract syntax tree is generated in the form of an EMF tree. An EMF tree is a series of Java objects which reference each other with named references. Looking at the previous example (8.3) a Java object is created called *ParametricContext*, which contains a reference to a list named *paraType*, which contains references to objects of type *paraType*. The objects in the EMF tree match up with the datatypes used to describe the translation in chapters 6 and 7. Ideally these translations could be applied directly to the EMF tree to generate Event-B abstract syntax tree. Unfortunately the Event-B syntax used in the Theory Plug-in does not also use an EMF representation. The internal Event-B representation used in the Theory Plug-in does not generate a textual representation (so if B$^\sharp$ were translated directly to this representation the results would not be examinable by the user). Therefore the decision was taken to produce textual Event-B statements, and allow the current Rodin toolset to parse these and construct the internal representation. The result of this is that Event-B theories generated are visible and modifiable by the user. However, any modification to Event-B theory will be written over the next time a modification is made to the B$^\sharp$ theory. Possible changes to the translation process to stop this over writing are discussed in Section 9.1. This is akin to translating to the concrete syntax presented in [57]. For example the code to translate a quantifier uses code similar to the following Java methods:

```
static String evBWithoutParaType() {
  List⟨evBAbsParams⟩ absParams = paramsUtil.evBReprParams(parameter);
  String result = quantString(quantType);
  result += paramsUtil.pairedVars(absParams);
  result += "·";
  result += paramsUtil.setContainmentPred(params);
```

[1] https://github.com/JSN00k/BSharpSource

```
    result += "⇒";
    result += expression.evBPred();
  }

  String evB() {
    if (parametricContext != null) {
      return evBWithoutParaType();
    } else {
      List⟨evBAbsParams⟩ absParaTypes = paramsUtil.evBReprParaType(
   paraTypes);
      String result = "∀";
      result += paramsUtil.pairedVars(evBAbsParams);
      result += "·";
      result += paramsUtil.setContainmentPred(params);
      result += "⇒";
      result += evBWithoutParaType();
      return result;
    }
  }
```

These methods are comparable to the abstract translation function (6.30). There is an immediate syntactic difference. The code is contained within a Java Class, and uses object oriented methods. So the quantifier does not need to be passed into a function (as the method is in the quantifier class). The types which make up the quantifier are directly referenceable (e.g., we can use *parameter* instead of *q.parameter*).

Chapter 7 introduced a series of B$^\sharp$ to B$^\sharp$ translations, used to simplify translations and inference. When implementing this the Java code looks less like the translation functions. This is because the abstract syntax tree for the translated expression is created directly, rather than generating concrete B$^\sharp$ syntax (which is used to create datatypes in the translation language).

8.3.1 Naming

There are several areas in the translations from B$^\sharp$ to Event-B where Event-B identifiers need to be defined. These identifiers need to be unique within the Event-B scope which they are generated in. A simple approach is taken in the current implementation, and inappropriate naming of variables can result in naming clashes within the generated Event-B. The general rule for naming is that an Event-B element will have the same name as the B$^\sharp$ element. This works well in many situations where there is a one to one relationship between Event-B elements and B$^\sharp$ elements. There are places where a single B$^\sharp$ statement generates several Event-B statements. In these cases it is necessary to create new variable names based on the current names.

The most notable example of generating multiple elements is parametric types in B$^\sharp$. As seen in the examples of translating a commutative operator 4.13 and 5.11 the B$^\sharp$ parametric types become several Event-B types. This expansion of the parametric types is done using the translation rule A.1. The approach taken to naming is that any additional parametric type defined is given an index. The final parametric type then has the same name in Event-B as it does in B$^\sharp$. This is seen in the example where $\langle T : \mathit{Setoid}\rangle$ becomes Event-B parameters $T1 : \mathbb{P}(T\_EvB), T : \mathit{Setoid}(T1)$. Whenever a parametric type is used an Event-B parametric type will be required and this is given the same name as the B$^\sharp$ type with *_EvB* appended (as can be seen in the example). Recall that Event-B parametric types cannot be used directly as this would not allow subtyping. The current implementation does not check whether or not there are any conflicting B$^\sharp$ variable names in scope. Therefore if the user has named a variable the same as a type with an index the generated Event-B will not be valid. This problem will be clear to the user when looking at the generated Event-B.

When writing functions in B$^\sharp$ two Event-B operators are generated, an expandable operator, and a passable operator (one that can be passed as an Event-B function). As seen in 5.2 using mappings described in 6.4.4. Naming functions produces two issues. One, in Event-B the function names are global and therefore all have to have unique names. Two, two Event-B operators are generated by a single function declaration.

Recall that all functions in B$^\sharp$ are declared within the type body of a class/datatype, (or the *Extension* of one). Classes and datatypes require globally unique names. This resolves the function naming issue as generated Event-B operators can be prepended with the class/datatype name. This guarantees their uniqueness. It also matches nicely with the long form of function names. Recall that if a function cannot be uniquely identified it has a long name e.g., in B$^\sharp$ the *Nat add* function and *Int add* function may both be visible, to resolve this the *Nat add* function can be referenced as *Nat.add* (similarly *Int.add*). These long names match nicely with the Event-B operator name (*Nat_add*).

To resolve the issue of two operators being generated the expandable operator is given the name as described above, and the passable operators has *_P* appended. This presents a possible danger for the user if they have a B$^\sharp$ function called *foo* and one called *foo_P*, so the user needs to avoid such clashes.

Finally there are several situations where theorems are generated. Functions generate theorems that state the return type is the type declared. Given a function name *foo* the generated theorem is called *foo typing theorem*.

Instance statements generate many theorems, the first is a proof that the elements in the instance statement really generate the instance. Given an instance statement of the form:

$$\textbf{Instance}\ \mathit{CommMonoid}\langle Nat\rangle([=], add, zero)\ \mathit{addMon} \tag{8.4}$$

The name of the typing theorem is *addMoninCommMonoid*.

Instance statements also instantiate theorems from the type they are instances of, e.g., in the example above the theorems associated with CommMonoid are instantiated as theorems about *Nat*. The name of the *Nat* theorem is prefixed with the name of the **Instance** so with the example above the theorem names will be prefixed with *addMon*. This is important because more than one instantiation may be made with theorems which have the same name. e.g.:

$$\textbf{Instance}\ \mathit{CommMonoid}\langle Nat\rangle([=], times, one)\ timesMon \tag{8.5}$$

If nothing was done to change the theorem names two, lots of theorems would be generated with the same names.

8.4 Additional Features and Extensions

Whist implementing B$^\sharp$ it was possible to add many additional features to B$^\sharp$ and its tools which were not part of the specification laid out in Chapters 6 and 7. These further facilitate the construction of mathematical types.

8.4.1 Syntax Aware IDE

8.4.1.1 Autocomplete

The B$^\sharp$ tool supports the user with autocompletion. At any point when developing theorems the user can press a key combination, and the IDE will give a list of possible tokens that could appear in the current location. If the user has started typing the token the IDE will only suggest tokens which match the string they have currently started typing.

This is particularly useful with Event-B/B$^\sharp$ where unicode characters are used within the syntax. The autocomplete syntax can be used to choose the unicode syntax elements (such as ∀). The most used benefit of autocomplete is the ability to complete function names e.g., starting to type *ra* then activating autocomplete will give options to complete with *raiseToR* and *raiseToL*, this reduces the amount of time it takes to type statements into B$^\sharp$. Along with this it allows the instant look up of relevant functions, for example, if you activate autocomplete without having typed any characters a list of relevant functions and syntax elements will be displayed.

8.4.1.2 Syntax Errors

When a token is typed that cannot go in the current location, it is underlined by the IDE to mark that it is an error. When the user hovers the cursor over the line with the error a list of possible correct tokens will be displayed so the user knows how to fix these problems.

To generate these features in Xtext it is necessary to have a well defined syntax, and $B^\sharp$ elements to be programmatically scoped correctly. As this is necessary for the compilation of the $B^\sharp$ language these features are in effect given for free.

8.4.2 Imports

In the specification the $B^\sharp$ import syntax imports a single class/datatype/extension. This is extended in the implementation to allow multiple classes to be imported at once, and classes from projects to be imported more easily. The first addition is the **From** command, this allows the importing of multiple files from the same project e.g.,:

$$\textbf{From}\ \ \mathit{Monoid}\ \ \textbf{Import}\ \ \mathit{SemiGroup.SemiGroup}\ \mathit{Monoid.Monoid} \tag{8.6}$$

It is not necessary to restate the project in every import.

The second addition is the $*$ syntax, this allows all classes/datatypes/extensions from a file to be imported in a single command:

$$\textbf{From}\ \mathit{Monoid}\ \ \textbf{Import}\ \ \mathit{SemiGroup}.*\ \ \mathit{Monoid}.* \tag{8.7}$$

Here everything from the *SemiGroup* and *Monoid* files are imported, not just the *SemiGroup* and *Monoid* classes. This additional syntax is optional, as sometimes it is necessary for scoping, or mapping reasons to only import the correct classes/datatypes/extensions.

8.4.3 Infix Functions

Infix functions with precedence are added as part of the implementation. The following declaration is an example of this:

$$\begin{array}{l} \mathit{add}(x, y : \mathit{pNat}) : \mathit{pNat}\ \mathrm{INFIX}\ 100 \\ \quad \mathit{match}\ x\ \{ \\ \qquad |\ \mathit{zero} : y \\ \qquad |\ \mathit{suc}(xs) : \mathit{suc}(xs\ \mathit{add}\ y) \qquad \} \end{array} \tag{8.8}$$

As *add* takes only two arguments it can be declared as an infix function. This is done with the INFIX statement, followed by the precedence of the operator. The integer value at the end is the precedence. Higher precedence functions bind more tightly. For example given a $times$ with precedence 200, and the expression $a\ add\ b\ times\ c$ this is treated as $a\ add\ (b\ times\ c)$. When generating Event-B appropriate brackets are added (as user declared Event-B infix operators do not have a notion of precedence).

A practical issue occurs with infix functions. Due to the way they are parsed they expect to have an argument on their left and right. This causes an issue when one tries to pass them as a function (parsing can become non-deterministic). To resolve this issue, if a infix function (including ones inherited from Event-B) is passed as a variable, it must be wrapped in parenthesis e.g., the *add* function is passed as [*add*].

8.4.4 Inferring Supertypes in Instances

Within the implementation when declaring an **Instance** the user is not required to give all the operators which make up the instance if these can be inferred. For example, if in the *pNat* class a default *Setoid* instance has been declared, equivalence relation from this can be used in future **Instance** statements:

$$\textbf{Instance}\ \mathit{CommMonoid}\langle \mathit{pNat}\rangle(\mathit{add}, \mathit{zero})\ \mathit{addMon} \tag{8.9}$$

Here the *Setoid* associated with *pNat* is used to infer that the equivalence relation is [=]. This approach has an additional benefit: the **Instance** statement knows that the theories about *Setoid*s have already been instantiated, so they do not need to be re-instantiated when generating the Event-B from the *addMon* instance.

8.4.5 If/Else Statements

The B$^\sharp$ language adds additional syntax for doing conditional statements. Along with the *COND* function inherited from Event-B (see Section 2.6.1), B$^\sharp$ adds if/else statements, for example:

$$\begin{aligned}
&\mathit{if}\,(\mathit{denom} = \mathit{zero})\ \{\\
&\quad \mathit{count}\\
&\}\ \mathit{else}\ \{\\
&\quad \mathit{suc}(\mathit{count})\\
&\}
\end{aligned} \tag{8.10}$$

This is a syntactic restructuring of the *COND* function, and can be used in any place where the *COND* function is e.g., *COND(boolExpr, expr1, expr2)* becomes *if (boolExpr) expr1 else expr2*.

The aim of this is to produce more readable B$^\sharp$ code that can be better understood. The *COND* function is not replaced and can still be used. The *COND* expression is lazily evaluated. During proof, once the value of the conditional can be resolved to true or false, the *COND* expression is replaced by the relevant expression (*expr1* if true and *expr2* if false).

As an example the *divMod* function on the natural numbers was declared in the following way:

```
divMod(n, d, count : pNat) : pNat × pNat
  match n {
    | zero : (zero, zero)
    | suc(ns) :
      if suc(ns) = d {
        (suc(count), zero)
      } else {
        if suc(ns) minus d = zero {
          (count, suc(ns))
        } else {
          divMod(suc(ns) minus d, d, suc(count))
        }
      }
  }
```
(8.11)

8.5 Type Classes Implemented in B$^\sharp$

In order to evaluate the advantages of the B$^\sharp$ language, and to test the tool, a series of mathematical type classes were defined. The types were based on the types used as examples in Chapters 4 and 5. The following theory files were created:

1. Relations: Definition of Reflexive (*ReflexRel* type class), Symmetric (*SymmetricRel*, and Transitive (*TransRel*) relations, and the definition of a *Setoid* type class.
2. SemiGroup: Definition of a *SemiGroup* (a set with an associative operator) including a series of theorems about identities.
3. Monoid: *Monoid* definition, theorems about the identity, definitions of the *RasieTo* operators, and theorems about them.
4. CommMonoid: Definition of the *CommMonoid* type class. A monoid where the operator is also commutative. Theorem that the raise to operators are then equivalent.

5. pNat: A data structure (*pNat*) is used to define the naturals (*zero*, *suc(x)* and structural equality resulting in axioms identical to Peano's natural axioms). Various operators are defined (addition, subtraction, division). Zero and addition is shown to be an instance of the *Monoid* type class. Generating multiplication. Theorems about multiplication are added (distributivity, associativity). One and multiplication are shown to also be an instance of a *Monoid*.

6. Group: The *Group* type class is defined a *Monoid* where each element has an inverse. Various group theory theorems are declared.

7. AbGroup: The *AbGroup* (Abelian Group) type class is defined as a subtype of the *Group* type class and the *CommMonoid* type class.

8. Ring: The *Ring* type class is defined, making use of the *AbGroup* and *Monoid* type classes

A concise description of the types is given in Chapters 4 and 5. The types defined in the implementation have many more functions and theories than those described. See Appendix C for the complete set of B$^\sharp$ theories. [2]

The basis of the developed abstract theories is the equivalence relations, so this was the first set of theories defined. Using the **Class** syntax a *Reflexive*, *Symmetric*, and *Transitive* relation were defined. These were then used together to define an equivalence relation 4.1.3. These structures served to test class declarations where there were no required elements. The declaration of the *Equivalence* relation tested multiple inheritance of these structures (it inherits from all three of the previous declarations). Within the *Equivalence* class theorems were declared to make using the transitive relation easier. The theorems relied on the automatic quantification of the class, which resulted in significantly simpler B$^\sharp$ expression in comparison to their Event-B counterparts.

The *Transitive* relation was then used to define the *Setoid* class (a set with an associated equivalence relation). This demonstrated the usefulness of classes with associated elements.

Having defined a *Setoid*, operators could then be built. This cannot be done without the notion of equivalence as operators require that if they are called with equivalent arguments, they give an equivalent results. A *baseOp* class was defined to encapsulate this definition as shown in 4.12. This tested the notion of having constrained parametric types ($\langle T : Setoid \rangle$).

Having defined operators, a series of monoid and group classes were defined: *SemiGroup, Monoid, CommMon* (a commutative monoid), *Group*, and *AbGroup*. For example, the definition of the Abelian group is:

$$\textbf{Class}\ AbGroup : Group, CommMonoid\ \{\} \tag{8.12}$$

[2]The source for these and the Event-B they generated, can be found at https://github.com/JSN00k/BSharpCaseStudies.

This definition tested multiple inheritance between classes with associated elements. The definitions and theorems of the *Group* and the *CommMonoid* classes were available to the *AbGroup* class and its instances.

The final abstract class defined was the *Ring* class with the following definition:

$$\begin{aligned}
&\textbf{Class } Ring[R] : Setoid\,(G : AbGroup\langle R\rangle, M : Monoid\langle R\rangle)\ \textbf{where}\\
&\quad \forall x, y, z : R \cdot equ(M.op(x, G.op(y, z)), G.op(M.op(x, y), M.op(x, z)));\\
&\quad \forall x, y, z : R \cdot equ(M.op(G.op(y, z), x), G.op(M.op(y, x), M.op(z, x)));\ \{\\
&\}
\end{aligned}\tag{8.13}$$

This generates the following Event-B:

$$\begin{aligned}
&Ring(T : \mathbb{P}(T\_EvB) \mathrel{\hat{=}} \{R \mapsto (G \mapsto M) | R \in Setoid(T) \land G \in AbGroup(T) \land M \in Monoid\\
&\quad \land \forall x, y, z \cdot x \in T \land y \in T \land z \in T \Rightarrow\\
&\qquad Monoid\_op(T, M)(x \mapsto AbGroup\_op(T, G)(y \mapsto z))\\
&\qquad\quad \mapsto AbGroup\_op(T, G)(Monoid\_op(T, M)(x, y)\\
&\qquad\quad \mapsto Monoid\_op(T, M)(x, z)) \in Setoid\_equ(T, R)\\
&\quad \land \forall x, y, z \cdot x \in T \land y \in T \land z \in T \Rightarrow\\
&\qquad Monoid\_op(T, M)(AbGroup\_op(T, G)(y \mapsto z) \mapsto x)\\
&\qquad\quad \mapsto AbGroup\_op(T, G)(Monoid\_op(T, M)(y, x)\\
&\qquad\quad \mapsto Monoid\_op(T, M)(z, x)) \in Setoid\_equ(T, R)
\end{aligned}\tag{8.14}$$

This is a good example of how the complexity in Event-B increases with complex statements, whilst B$^\sharp$ manages this complexity better.

Having defined abstract classes a concrete implementation of the natural numbers was implemented as in (4.1). This demonstrated the use of the **Datatype** constructor in B$^\sharp$. Having defined a representation of the naturals, it was then possible to define the *raiseTo* functions on the *Monoid* type. To do this an **Extend** statement was used to add to the *Monoid* class. This imported the *pNat* type, and defined the *raiseTo* functions as shown in 4.16. This demonstrated the use of **Extend** statements and B$^\sharp$ methods (elements such as *op* and *ident* did not have to be explicitly declared in the B$^\sharp$ definitions).

Finally **Instance** statements were used to define addition and multiplication monoids as described in (4.26). This generated the *times* and *power* functions on the natural type. It also generated instantiated theorems from the *CommMonoid*, *Monoid* and *SemiGroup* types.

To demonstrate conformance to an **Instance** statement it was necessary to demonstrate that the elements which made up an instance conformed to the relevant type class. As an example, to demonstrate that zero and addition form a monoid, it is necessary to show that addition is associative, and that zero acts as the identity under addition. The result is that the defined theories have more structure than if theorems are added in an entirely ad-hoc manner as required.

8.6 Generated Event-B

The translation of B$^\sharp$ to Event-B worked well and generated Event-B theories that would be expected. The instance statements were found to be very effective, where theorems inherited from type classes were instantiated for concrete classes. In total, B$^\sharp$ generated 77 Event-B theorems and 121 proof obligations. Of these proof obligations, 108 were proved; 30 of which were considered true by construction (could be proved simply by instantiating the type class). These instantiated theorems were then used within many of the remaining proofs, saving the user from having to re-prove or re-declare them. The remaining 13 proof obligations were not proved due to difficulties with the current implementation of the interactive theorem prover which got exponentially slower with deeper inheritance. Examples of this were theories on AbGroups, Groups (Group<-Monoid<-SemiGroup<-Setoid) and Rings made use of the AbGroup type. This slow down currently restricts the usefulness of B$^\sharp$ to simple type classes. The cause, at least in part, is the generation of massive (and very receptive) well-definedness proof obligations. Because of this, these issues are not seen in other languages, which use their type systems rather than generating these. It may be possible to resolve these issues with a change to the way Event-B generates these proof obligations (currently the simple approach of conjoining all well-definedness proof obligations is used). It has been suggested that the use of the *prj1* and *prj2* operators also contribute to these problems, and therefore another solution may be to find a different translation which avoids these operators. This problem could also be partially mitigated by B$^\sharp$ adding additional proof rules which would act as a shortcut for the interactive prover. Further discussion of this can be found in Section 9.3.

As all proofs are done in Event-B, the system used to handle proofs is the Event-B system. This system is to store proofs in a separate file from the theory, and link the proofs and proofs obligations using a naming scheme. The result of this is that when a B$^\sharp$ statement is changed only the theory file is changed, and the Event-B proofs file remains the same. Therefore, unless you change the name of the B$^\sharp$ declaration (e.g., the theorem name), the current proof will be preserved. This matches the current Event-B behaviour, however, there are problems with the current Event-B system and possible improvements to this system are suggested in Section 9.3.3.

8.7 Comparison With Other language

The core of the B$^\sharp$ language is most similar to the core of HOL style languages. In general, these have small cores (kernels) and additional features are then added which translate to the core of the language. The result of this is that it is only necessary to trust that the small kernel of the language is consistent. B$^\sharp$ takes a similar approach to this, where we translate the core of B$^\sharp$ to Event-B and other features can be added using B$^\sharp$ to B$^\sharp$ translations. Unlike Isabelle/HOL, B$^\sharp$'s core does not have support for let statements, function currying, or function overloading via type inference. As seen in the section 4.5 example, it is possible to simulate function currying and let statements can be simulated using function declarations with no arguments. To allow function overloading in a similar way to HOL additional work needs to be done on B$^\sharp$'s inference, so correct functions can be automatically chosen.

Whilst the core of B$^\sharp$ is similar to HOL style languages most implementations of HOL have had a long time for theories to be defined and features to be added. The result is that these more mature languages have many additional features that are not yet available in B$^\sharp$. Examples of this were seen in Chapter 2, e.g., co-inductive datatypes, mutually recursive functions, and set representation. Along with adding syntax, these features in HOL also have additional proof tactics which facilitate proving. Possible ways of adding new features and proof tactics are discussed in Chapter 9.

Class declarations in B$^\sharp$ are unlike their equivalents in HOL languages as they translate directly to Event-B rather than the HOL like core of B$^\sharp$ (they are a core feature of the B$^\sharp$ language). This ability to translate directly to Event-B may make adding other features to B$^\sharp$ easier than in HOL languages e.g., set notation.

The work on B$^\sharp$ demonstrates how features can be added safely to the Event-B environment to make representing mathematical theories easier. There are, however, still many features that could be added to the language to facilitate the construction of new theories.

8.8 Lessons and Conclusions

The implementation of a B$^\sharp$ tool demonstrated its feasibility as a language. It showed that using a translation phase was a consistent and viable way to add language features.

There were many lessons were learnt from the implementation of the B$^\sharp$ tool. In many cases, these have already fed back into the previous work and are not included here. However, there are some outstanding lessons that do need to be commented on. These issues were not critical to demonstrating the feasibility of a translation phase and have therefore been left for possible future implementations.

8.8.1 Inheritance Syntax

There are several places in the B$^\sharp$ language where one class inherits from another (such as **Class** declaration statements and **Instance** statements). These statements need to be extended to allow inheritance from concrete instances. As an example when developing B$^\sharp$ classes the following class was declared:

$$\textbf{Class } \mathit{NatCommMonoid}[M] : \mathit{CommMonoid}\langle \mathit{pNat} \rangle \tag{8.15}$$

This declaration produced the expected Event-B definitions. However, the aim of declaring this class was to write and prove a theorem stating the inherited $raiseToL$ function is associative on this class.

Unfortunately the declared theorem proved not to be true. To be true there would need to be restrictions on the equivalence relation being used. To make this declaration true the following statement could be used:

$$\textbf{Class } \mathit{NatCommMonoid}[M] : \mathit{CommMonoid}\langle \mathit{pNat} \rangle \textbf{ where } M.\mathit{equ} = [=] \tag{8.16}$$

Alternatively a more general syntax would be:

$$\textbf{Class } \mathit{NatCommMonoid}[M] : \mathit{CommMonoid}, \mathit{pNat.Setoid} \tag{8.17}$$

This statement extends the notion of inheritance to include instances allowing an abstract type to have concrete components. Much of the implementation for this is already done within the inference used in **Instance** statements.

This change would also apply to the syntax of **Instance** statements. For example the following declaration:

$$\textbf{Instance } \mathit{CommMonoid}\langle \mathit{pNat} \rangle\ (add, zero)\ \mathit{addMon} \tag{8.18}$$

Would become:

$$\textbf{Instance } \mathit{CommMonoid} : \mathit{pNat.Setoid}\ (add, zero)\ \mathit{addMon} \tag{8.19}$$

Previously additional information was being inferred from the parametric context. This does not scale, for instance, for classes with two parametric types (e.g., a class representing a homomorphism), it is no longer possible to do this inference as there is no individual instance to choose from. When a concrete supertype is given instead the parametric context can always be inferred.

Finally in the statement above 8.19 it is possible to infer that we need a setoid from the *pNat* class, so it can be reduced to:

$$\textbf{Instance } \mathit{CommMonoid} : \mathit{pNat}\ (add, zero)\ \mathit{addMon} \tag{8.20}$$

and B$^\sharp$ will use the default setoid of the *pNat* class.

The initial translation is very useful for representing and generating statements that are complex to express in Event-B such as Rings (Section 8.5).

8.8.2 Type Class Typing

Variables can be typed as an instance of a type class, this was seen in the definition of a ring 8.21 e.g., $G : AbGroup$. This approach worked, however, with these definitions the multiplicative part of the ring was $M.op$ and the additive part was $G.op$. These operators names are not descriptive, and it would be better for the user to be able to give the members of the type classes explicit names. For example the following syntax could be used:

$$\begin{aligned}
&\textbf{Class } Ring[R] : Setoid\ ((add, zero) : AbGroup\ R, (times, zero) : Monoid\ R \textbf{ where}\\
&\quad \forall x, y, z : R \cdot equ(times(x, add(y, z)), add(times(x, y), times(x, z)));\\
&\quad \forall x, y, z : R \cdot equ(times(G.op(y, z), x), add(times(y, x), times(z, x)));\ \{\\
&\}
\end{aligned} \tag{8.21}$$

This would also allow a change to the mapping to Event-B, e.g., it could generate an Event-B typing statement with the following form:

$$\{ \ldots equ \mapsto add \mapsto zero \ldots \mid \ldots equ \mapsto add \mapsto zero \in AbGroup(T) \ldots \} \tag{8.22}$$

Note that the new syntax is using multiple inheritance (as described previously 8.8.1). However, previously the classes are comma separated. Here this would conflict with the comma for separating variables. To allow this, supertypes could be separated with white space (rather than a comma).

8.8.3 Infix Functions

It was found that infix functions could not be easily used within other statements. This is because allowing infix functions to be passed as functional types causes issues parsing expressions, i.e., it is difficult to decide if an infix function is being called as a function, or being passed as an expression to another function. It may be possible to parse the expression tree correctly by taking more account of typing information. For now the solution employed is to add additional syntax so that if a infix function is being passed as a function it is first wrapped in square brackets e.g., when passing equals as an equivalence relation the following expression is used [=].

8.8.4 Obsolete Theorems

Whilst defining mathematical classes it was found that some theorems had the effect of stopping other theorems being useful. As an example, in the *CommMonoid* class there is a theorem that *raiseToL* = *raiseToR*. As these are effectively the same function, any instance of a commutative monoid only needs one of them defined (and only needs theorems about one of them). Currently instances of the *CommMonoid* class will instantiate both *raiseTo* functions, and the theorems about them. The negative side effect of this is that when proving it is harder to find the desired theorem due to the clutter. In a future release a syntax could be added to identify such theorems to resolve this issue.

8.8.5 Conclusion

The implementation of a B$^\sharp$ tool demonstrated its feasibility as a language. As seen in Section 8.5, B$^\sharp$ definitions are considerably more concise than the equivalent Event-B, which also makes them easier to comprehend. The implementation generated valid Event-B statements which had the expected meaning, given the B$^\sharp$ statements. . The major factors in this were the explicit typing of B$^\sharp$ variables, predicates not being a separate syntactic category, class bodies giving methods, and implicit quantification in theories.

As all of the proving is done within Event-B, the meaning of B$^\sharp$ statements is entirely defined by the translation rules to Event-B. A translation is considered correct if it produces the expected Event-B. To test the effectiveness and correctness of the translation, many B$^\sharp$ types were defined and translated to Event-B as outlined in Section 8.5.

The ability, in B$^\sharp$, to write higher order functions, demonstrated with the *curry2* function (4.5) should facilitate the embedding of logics within the B$^\sharp$ language, and so making it easier to define a theory of Event-B in a similar way to [72].

A major incentive of the B$^\sharp$ language was to reduce repeated work via inheritance and instantiation of theorems from abstract classes to concrete classes. This was achieved using the B$^\sharp$ instance statements. As seen in Section 8.6. These statements worked as expected in the classes implemented using the B$^\sharp$ tool. This saved considerable effort re-writing and proving theorems when defining B$^\sharp$ models.

In general when proving theorems within the interactive theorem prover the statements that needed to be proved looked a lot like the B$^\sharp$ statements. This is because the standard automatic tactics in the interactive theorem prover instantiate the Event-B typing information. This reduces the hypothesis to one that looks much like that declared in B$^\sharp$. For example, the following theorem is declared in the $EquivRel$ class:

$$\begin{aligned}&\text{Trans Rewrite} :\\ &\quad \forall x, y, z : T \cdot e(x, y) \Rightarrow e(x, z) \Leftrightarrow e(y, z);\end{aligned} \tag{8.23}$$

When this theorem is seen by the user in the interactive prover it looks like this:

$$\begin{aligned}&e \in \mathit{EquivRel}(T)\\ &x \in T\\ &y \in T\\ &z \in T\\ &x \mapsto y \in e\\ &\vdash\\ &x \mapsto z \in e \Leftrightarrow y \mapsto z \in e\end{aligned} \tag{8.24}$$

The view in the prover and the B$^\sharp$ statement match well, however, the user does need to know how B$^\sharp$ predicates are represented in Event-B. There are larger areas of difference. For instance, when a B$^\sharp$ method is used, the class it is declared in is an inferred parameter of the function, whereas in Event-B this is made explicit. So the following B$^\sharp$ statement:

$$\mathit{raiseToL}(x, p) \tag{8.25}$$

Becomes the following Event-B statement:

$$\begin{aligned}&T \mapsto equ \mapsto op \mapsto ident \in \mathit{Monoid}(T)\\ &\vdash\\ &\mathit{raiseToL}(T, T \mapsto equ \mapsto op \mapsto ident, x, p)\end{aligned} \tag{8.26}$$

within in the prover. This statement is less similar to the B$^\sharp$ that generated it, and requires the user to understand that a method is a function which takes the class as the first argument.

The work done here shows that an approach of adding a translation phase, allows new language features to be added in a safe and consistent manner (similar to that achieved using a meta language in other proving systems). This opens the possibility of adding many new features to the B$^\sharp$ language, and its translation, increasing the ease with which systems can be modelled using the Event-B toolset.

It is the author's opinion that the more concise representation of mathematical statements, along with the IDE features, make developing mathematical theories in the B$^\sharp$ tool considerably easier than the equivalent development using the Event-B syntax. The inheritance and instantiation features of B$^\sharp$ allowed the extension of theorems, saving a considerable amount of work when defining them. Theorems were instantiated which may have been neglected when similar mathematical types are declared in Event-B, easing proofs of future theorems. The language having features designed for reuse and inheritance encouraged theorems to be designed using these principles, resulting in better structured mathematical theorems as seen in Section 8.5.

Chapter 9

Future Work

The approach of having a stage of compilation allows features to be safely added to the B$^\sharp$ tool in a consistent manner. This chapter explores some of the additional features that could be added to the B$^\sharp$ implementation. These features are focused on the following areas:

1. Allowing B$^\sharp$ theories to be extended using the current Event-B tools.
2. Easing proofs, by having the B$^\sharp$ tools interact with the Event-B prover.
3. Increasing the usability of B$^\sharp$ theories within the Event-B language.

The aim of these additions is to aid the development of theories in B$^\sharp$, and increase the number of problems the tool can be applied to.

9.1 Extending B$^\sharp$ With Event-B

The current approach to the B$^\sharp$ tooling is that when a B$^\sharp$ file is saved, new Event-B files are automatically generated. These files include all of the mapped B$^\sharp$ statements from the saved files. The new Event-B files simply replace any previous Event-B theory files. This approach works reasonably well because proofs are not stored in the current Event-B files, but in a separate file of their own, and are related to the Event-B via the names of Event-B objects. This means that regenerating a file and statements (even with additional new statements) will not remove the proofs which the user has already done. However, if the user has opened the Event-B file and added any statements, these will be removed the next time B$^\sharp$ generates this Event-B file because these statements are not declared in the B$^\sharp$ syntax. This approach is problematic because not all features of Event-B are currently available in B$^\sharp$. For example, in Event-B it is possible to add proof rules (statements which can be used directly by the interactive prover, both automatically, and interactively by the user).

The aim then is to make it so when Event-B is generated by B$^\sharp$, B$^\sharp$ only replaces statements it should, and does not remove other statements. The approach suggested is one used within the Eclipse Modelling Framework (EMF) [81]. EMF has a similar problem where an abstract syntax tree of Java classes is generated from a model described graphically using Ecore. When generating new Java, EMF marks generated statements (e.g., classes and methods) with a tag ("@generated") in the comment above the statement. The user can then change the generated Java, adding new methods and classes. The user can also change generated statements by simply deleting the tag, before editing the Java. EMF knows when generating Java that it is only allowed to change statements that it previously generated (i.e., have the generated tag above them).

This approach will work well with the way that Event-B is generated by the B$^\sharp$ tool. Each of the generated Event-B declaration statements (operators, theorems, and types) have tightly associated comments in which the the tag can be added. This approach leaves the user with the maximum ability to make changes to the Event-B theories. However, this also means that the B$^\sharp$ and Event-B theories may not correspond entirely. An alternative approach is to programmatically mark generated theories, and make it so the user can only add statements (i.e., generated statements cannot be changed). This would ensure that B$^\sharp$ and Event-B theories correspond, but restricts the freedom of the user to change the generated Event-B.

9.2 Reporting Event-B errors in B$^\sharp$

Given there is no requirement to demonstrate that a B$^\sharp$ translation generates valid Event-B, there is nothing to stop someone creating a B$^\sharp$ translation that does not always generate valid Event-B. Having a mechanism by which problems in Event-B can be reported back and displayed within B$^\sharp$ would be helpful. With the current mechanism for translating this can only be achieved in a very broad manner, e.g., if the expression for a theorem has a syntax error, this could be recognised in Event-B during compilation and the theorem could be marked as problematic in B$^\sharp$. It would be better to have finer grained error reporting. To allow this would require the translation to translate directly to the Event-B abstract syntax tree rather than the concrete Event-B syntax. This could be achieved by having an EMF representation of the Theory Plug-in Event-B syntax (as there is for the standard Event-B syntax), then errors within specific Event-B nodes could be propagated back to the B$^\sharp$ node which initially generated the Event-B.

9.3 Facilitating Proofs

Up to this point the focus has been on generating Event-B which represents abstract mathematical types. Having generated the Event-B it then becomes necessary to discharge the proof obligations associated with the generated Event-B. This is done using the interactive prover. This section will look at different ways in which the Event-B tool can help to facilitate the proving process.

9.3.1 Proof Rules

In Chapter 3 the ability to add proof rules (rules which can be used by the interactive prover) was used to facilitate proofs. In principal $B^\sharp$ could generate whole series of proof rules which would greatly ease proofs. In practice due to the implementation of the proof rules within the Theory Plug-in, and bugs within the Theory Plug-in implementation, a feature generating proof rules has not yet been implemented.

This section will look at the proof rules which can be generated by the $B^\sharp$ tool. These proof rules fall into two broad categories: Proof rules which the tool can generate automatically (with no user interaction) (such as typing rules) and those where the user explicitly tells $B^\sharp$ to generate a proof rule (e.g., the user may want to turn a theorem into a proof rule).

9.3.1.1 Automatically Generated Rules

Anytime an Event-B operator is used within an expression the well definedness proof obligations for that operator are generated. Some of these well definedness proof obligations are generated automatically by the Event-B system. For example, when an Event-B operator is declared its parameters can be typed as members of subsets. When this operator is used a well definedness proof obligation is generated where the arguments to the operator need to be shown to be in the correct subset for the associated parameter. This results in the user having to prove statements such as:

$$\begin{aligned}
&M \in Monoid(T)\\
&x \in T\\
&y \in T\\
&\vdash\\
&x \mapsto y \in dom(Monoid\_op(T, M))
\end{aligned} \tag{9.1}$$

And:

$$\begin{aligned}
&T \mapsto equ \mapsto op \mapsto ident \in Monoid(T)\\
&x \in T\\
&p \in pNat\\
&\vdash\\
&Monoid\_raiseToL(T, T \mapsto equ \mapsto op \mapsto ident, x, p) \in T
\end{aligned} \tag{9.2}$$

These proof obligations are type checking proofs, and can be greatly eased by adding typing proof rules. In the top example 9.3 the reason this proof obligation is not automatically discharged is because the the type of $Monoid\_op(T, M)$ is not immediately available within the interactive prover. To resolve this B$^\sharp$ could automatically generate typing proof rules. In this case the following rule would make 9.3 instantly provable:

$$\frac{M \in Monoid(T)}{Monoid\_op(T, M) \in \mathbb{P}(T \times T \rightarrow T)} \tag{9.3}$$

Using this proof rule in a forward direction makes the type of the getter immediately available, causing any well definedness proof obligations about the type to be immediately dischargeable. The B$^\sharp$ tool is already able to generate the Event-B type of a getter, making this type of proof rule easy to generate within the current code base.

When dealing with abstract classes in the interactive prover it is necessary to be able to access the elements of the type class. The result is that at some point the type class definition needs to be expanded e.g., if there was the statement $M \in Monoid(T)$ to access the operator it would need to be expanded to $equ \mapsto op \mapsto ident \in Monoid(T)$. This results in the following proof rules being useful:

$$\frac{T \in \mathbb{P}(T\_EvB), equ \mapsto op \mapsto ident \in Monoid(T)}{equ \in EquivRel(T)} \tag{9.4}$$

and

$$\frac{T, equ \mapsto op \mapsto ident \in Monoid(T\_EvB)}{op \in \mathbb{P}(T \times T \rightarrow T)} \tag{9.5}$$

The first of these proof rules 9.4 is a shortcut, it allows the definition of the *EquivRel* to be accessed without having to expand the *Monoid* type, then the *SemiGroup* and *Setoid* (the deeper into a class hierarchy, the more steps are needed to access the earlier elements). Along with being a shortcut for the user this also acts as a shortcut for the automatic provers giving a higher chance typing proof obligations will be discharged automatically.

The second proof rule 9.5 gives the type of the element without having to manually deconstruct the elements, this has the same effect as 9.3 allowing the type to be accessed immediately.

Another set of typing proof rules can be generated allowing access to the return types of functions:

$$\frac{T \in \mathbb{P}(T\_EvB), M : Monoid(T), x \in T, p \in pNat}{Monoid\_raiseToL(T, M, x, p) \in T} \tag{9.6}$$

This proof rule will also be generated when functions are instantiated:

$$\frac{x \in pNat, p \in pNat}{times(x, y) \in pNat} \tag{9.7}$$

Note that because *pNat* is an Event-B datatype (not a set) the current Event-B toolset will already infer the type of this function correctly. This is only true for Event-B types, it does not in general hold for sets. It is often the case that the instantiated type will be a set in Event-B (not a type), at which point this proof rule will automatically discharge the associated typing statements.

These additional proof rules start to make B$^\sharp$ types act as types within the interactive prover. The user will still have to do some proofs (as it is not generally possible to infer typing of subtypes), but these will be significantly reduced.

9.3.2 Proof Rules From Expressions

Along with the automatically generated proof rules described above, it is desirable to allow theorems to become proof rules in the Event-B environment (in Event-B this requires restating the theorem as a proof rule, and reproving the theorem). However, it is not desirable to have all theorems become proof rules. Firstly, proof rules can be applied automatically, and if all theorems become proof rules then there is a high chance the automatic provers will get stuck in a loop (applying a proof rule to get to one state and then applying another which returns the proof to the original state). Secondly, having too many proof rules will start to make proving harder, as the user will need to spend time finding the appropriate proof rule, where one proof rule is almost always the desired one.

In Event-B when adding a proof rule the user can specify if the rule can be applied automatically by proving tactics. There is also the notion of a rewrite rule. Given two expressions that are equivalent, i.e., $E_1 \Leftrightarrow E_2$ they allow the user to click on the first expression E_1 and it to be re-written as the second expression. These work from left to right. They need to be redeclared the other way round (and reproven) to allow E_2 to be rewritten to E_1. Finally, 'inference' rules can be added, these take the form:

$$\frac{V, E_1, \dots E_n}{I} \tag{9.8}$$

where V are typed variables, E_n are expressions (where the variables can be used) and I is an inferred expression. When the user uses this type of proof rule the expressions E_n remain in the hypothesis, the new expression I is added to the hypothesis.

The aim is then to allow the user to specify what type of Event-B rule to generate from a B$^\sharp$ statement. Along with theorems it is also useful to allow **where** expressions from **Class** declarations to become proof rules. These are effectively the axioms of a class.

There are then three different forms of B$^\sharp$ statement which can become proof rules:

$$\forall V \cdot E_1 \wedge \cdots \wedge E_n \Rightarrow I \tag{9.9}$$

$$\forall V \cdot E_1 \Leftrightarrow E_2 \tag{9.10}$$

$$\forall V \cdot E_1 \wedge \ldots E_n \tag{9.11}$$

To allow the user to direct what type of proof rule should be generated from the B$^\sharp$ statements additional keywords are added to the language. The keywords are *AXIOM*, *INFER*, *REWRITE*, *AUTO*, *FORWARD*, *BACKWARD*, and *BOTH*. We will now see how these can be used.

In the case of the third statement 9.11, if the statement is followed by the *AXIOM* keyword ($E_1 \wedge \cdots \wedge E_n$ *AXIOM* then the 9.11 proof rule is generated. In Event-B this is generated as an automatic inference rule, where the expression $E_1 \wedge \cdots \wedge E_n$ has to appear in the goal. The result of this proof rule is that if the *goal* of the current proof is $E_1 \wedge \cdots \wedge E_n$ then the proof is complete and automatically discharged.

When a statement of the form of 9.9 appears, an inference rule can be generated. To make this happen the user appends the word *INFER* at the end of the statement. This will generate a proof rule of the form seen in 9.8. To make this as efficient in Event-B as possible it is better if the user splits $E_1 \wedge \cdots \wedge E_n$ up into a list of expression $E_1, \ldots, E_n$ which matches 9.9. By default this proof rule will not be applied automatically by the Event-B tactics. To allow the Event-B tactics to apply this automatically the *AUTO* word can be added after *INFER*.

When there are expressions of the form 9.10 the *INFER* keyword can still be used, however, in this case the direction of inference is unknown. This can be specified with the *FORWORD* ($E_1 \Rightarrow E_2$) or *BACKWARD* ($E_2 \Rightarrow E_1$) keywords, or *BOTH* (which generates both the forward and backward versions). These then work identically to the previously described inference proof rules. Expressions like this can also generate rewrite proof rules. This is achieved by replacing the *INFER* keyword with the *REWRITE* one (note that because rewrites are directional it is still necessary to give the direction (*FORWARD* etc.).

As proof rules are base on axiomatic parts of type classes, and theorems (which already require proofs) additional proofs for the proof rules are not necessary. Ideally B$^\sharp$ will generate a proof of the proof rule by instantiating the already defined theorem.

Concrete types will inherit proof rules from abstract types, in the same way as is done for theorems. As theories get more complex this may result in too many proof rules being generated, and this design may need to be revisited.

9.3.3 Proof Scripts

In many of the other theorem prover languages looked at, proving is done using a second language to write a proof script rather than an interactive prover. As noted in Chapter 2, there are advantages to each of these approaches. Whilst it is not desirable to lose the advantages of the interactive prover, having the advantages of scripting proofs would be helpful. Specifically, it would allow proofs of similar theorems to be largely shared (making only small manual adjustments to the proofs). It could also be used to make Event-B models more shareable. Finally proof scripting could allow direct interaction with the core proof rules of Event-B, currently these are only available by applying a proving tactic (an indirect approach).

The initial approach to adding proof scripts would be to create a list of interactions with the interactive prover. For example, given the theorem:

suc out of addition:

$$\forall x, y \cdot x \in pNat \land y \in pNat \Rightarrow x\ pNat\_add\ pNat\_suc(y) = pNat\_suc(x\ pNat\_add\ y) \tag{9.12}$$

The following interactions were made with the interactive prover:

```
Default auto tactic profile
apply induction to datatype identifier x
    inductive base case (x = zero):
        expand pNat_add [0]
        expand pNat_add
    inductive step case (x = suc(p_prev))
        expand pNat_add [0]
        expand pNat_add
```

There are several problems to resolve with this script. The first line of the script applies the “Default auto tactic profile”. Tactics are entities which can be created and edited by the user. In this case this is the standard tactic which is available when Rodin is downloaded. On other peoples machines the tactic may have been edited, which may break this proof script. To resolve this each project will store the used tactics as text files. When a tactic is referenced the correct file will be found in the project, and this is the tactic that will be used. A second problem exists where additional prover plug-ins may have been installed, which are unavailable on other peoples machines. For example there could be a proof step to run the SMT solvers and if another user does not have these installed the proof will then fail to be reproduced. As with the tactics the state of the Rodin platform on which the proof was performed should be made explicit. This could be achieved by each project having a list of installed plug-ins, and version numbers. Having this as explicit information would also help with trust of proofs. For instance if a bug came to light in an earlier version of a prover plugin, it would make it clear that the proof needed to be re-run to validate it.

The biggest problem with the script above is the current state of the proof is not communicated (e.g., we cannot see the current goal of the proof). This could be achieved by making the B$^\sharp$ editor interact with the interactive prover, when proving the interactive prover would run the entire proof script, generating a proof tree. The interactive prover would then display the position in the proof tree based on the location of the cursor in the B$^\sharp$ window. As the user is editing a proof script they would then be able to see the state of the proof within the interactive prover. Ideally the user would be able to make an action within the interactive prover and this would be instantly added to the proof script in the B$^\sharp$ environment.

It is useful to see an example of where having a proof script would help. In the *Monoid* class the following theorem is generated:

$$\begin{aligned}
&T \mapsto equ \mapsto op \mapsto ident \in Monoid(T)\\
&x \in T\\
&\vdash\\
&Monoid\_raiseToL(T, T \mapsto equ \mapsto op \mapsto ident, x, p) \in T
\end{aligned} \tag{9.13}$$

Which can be proved with the following script:

```
 Default auto tactic profile
In hyp[0] ($T \mapsto equ \mapsto op \mapsto ident \in Monoid(T)$) expand Monoid
In hyp[1] ($T \mapsto equ \mapsto op \in SemiGroup$ expand SemiGroup
In hyp[4] ($op \in AssocOp(T)$) expand AssocOp(T)
In hyp[4] ($op \in baseOp(T,\ T \mapsto equ)$) expand baseOp
    apply induction to datatype identifier p
        expand Monoid_raiseToL
        expand Monoid_ident
    inductive step case (p = suc(p_prev))
        expand Monoid_raiseToL
        expand Monoid_op
```

An identical theorem is defined for the typing of *Monoid_raiseToR*, in order to prove this second theorem the proof script for *Monoid_raiseToL* could be copied and pasted, and *Monoid_raiseToL* replaced with *Monoid_raiseToR*.

In the script above there are several expansions in the hypothesis. These resolve typing and will not be necessary when the proof rules from section 9.3.1.1 have been added.

Finally with proof scripts it would be possible to allow functions within the scripts. These could be implemented with text replacement before the script is run by the interactive prover. In the example above this could be used to make it so a single function with the argument of the function name could be used to prove both the *Monoid_raiseToR* and the *Monoid_raiseToL*.

9.4 Generating Abstract Event-B Types

In Chapter 2 it was seen that the current approach in Event-B is to define a parametric type and a series of operators and then define these axiomatically. So to define a Ring in Event-B a parametric type called *Ring* would be created, then the operators would be created and axiomatically defined in the following way:

$$
\begin{aligned}
&zero() : Ring\\
&one() : Ring\\
&add(x : Ring, y : Ring) : Ring\\
&mult(x : Ring, y : Ring) : Ring\\
&axioms :\\
&\quad \forall x, y, z \cdot add(add(x, y), z) = add(x, add(y, z))\\
&\quad \forall x, y \cdot add(x, y) = add(y, x)\\
&\quad \forall x \cdot add(x, zero) = x\\
&\quad \forall x \cdot \exists minusX \cdot add(x, minusX) = zero\\
&\quad \forall x, y, z \cdot mult(mult(x, y), z) = mult(x, mult(y, z))\\
&\quad \forall x, y, z \cdot mult(x, add(y, z)) = add(mult(x, y), mult(x, z))\\
&\quad \forall x, y, z \cdot mult(add(y, z), x) = add(mult(y, x), mult(z, x))
\end{aligned}
\tag{9.14}
$$

Despite the problems with this form of definition (the repetition, lack of extensibility and heritability), it works well with the current Event-B toolset because the *Ring* is an Event-B type. Using the representation generated from B$^\sharp$ is harder because it creates an Event-B set rather than type. When modelling it is easier to work with an Event-B type, as the Event-B tools are better able to reason about Event-B types (e.g., there will be fewer well definedness theorems, and the tools are better able to infer types).

The Event-B definition above 9.14 has the same axiomatic definition as the B$^\sharp$ definition 8.21, with associated elements replaced by operators, and the **where** statements in the *Group* and *Monoid* replaced by axioms. Where possible a concrete type could be created (e.g., a concrete implementation of the rational numbers is an instance of a ring), this can be difficult and time consuming. An alternative solution is to add a second form of instantiation to the B$^\sharp$ tool, which directly builds the abstract Event-B type from the abstract B$^\sharp$ type. This allows the abstract B$^\sharp$ types to be used directly within Event-B. It would also resolve the issues of the Event-B types extensibility. If a monoid is desired in Event-B then the *Monoid* type can be instantiated. As has been seen this is extended in B$^\sharp$ to create the *AbGroup* type. If this type is desired in Event-B it can be instantiated instead. Because the B$^\sharp$ can be extended the Event-B does not have to be.

9.4.1 Operators From Simple Types

As described above associated elements from $B^\sharp$ type classes become operators. First we will look at how elements with simple types are translated into operators. Here by simple type we mean the elements are not typed with a $B^\sharp$ type class, e.g., types such as T or $T \times T \rightarrow T$. These are split into two categories, functional types (any type of the form $\tau_1 \rightarrow \tau_2$ where τ_1 and τ_2 are type expressions) and non-functional types (any element typed in a different way).

A non-functional associated element of type τ will become an Event-B operator which has no parameters, and has a return type of τ. e.g., the *Monoid* (4.14) type class has an associated element $ident : M$, this would be represented by the Event-B operator:

$$ident() : T \tag{9.15}$$

A functional type e.g., $\tau_1 \rightarrow \tau_2$ becomes an operator with parameters generated from τ_1 and a return type of τ_2. The parameters are generated by splitting τ_1 on the inner product, i.e., given τ_1 of the form $\tau_3 \times \cdots \times \tau_n$ the parameters are $\tau_3, \ldots, \tau_n$. As an example the associated element $op : T \times T \rightarrow T$ becomes the Event-B operator:

$$op(p_1 : T, p_2 : T) : T \tag{9.16}$$

A difficulty with this approach is that in $B^\sharp$ T may be a constrained parametric type (e.g., we have seen $T : Setoid$). This is resolved by instantiating the constraining type first (note that when mapping, T represents a set of the type for T and all the associated elements of T so $T.equ$ can be mapped to the generated *equ* operator). The example used here is a poor example as the *Setoid* type is treated as a special case. Whenever a complete Event-B type is instantiate the *Setoid* type is ignored as the equivalence relation becomes structural equality and it is therefore not needed.

9.4.2 Operators From Non-Simple Types

Many associated elements are typed using $B^\sharp$ classes rather than type constructors. Examples of this is the *op* in the *SemiGroup* definition is typed $op : AssocOp\langle SG\rangle$), or in the *Ring* definition there is an associated operator typed $M : Monoid\langle R\rangle$. These can be split into two difference groups, ones typed with simple classes (classes without associated elements), and ones typed with type classes (classes with associated elements).

9.4.2.1 Typed With Simple Classes

When an element is typed with a simple type class it is initially treated in the same way as typing the associated element with the base type of the class (the implementation of B$^\sharp$ already has functions to get this base type). So the type declaration $op : AssocOp\langle SG\rangle$ becomes the same as $op : SG \times SG \rightarrow SG$. This would then generate an operator in the same way as described above, where SG is the type being used when defining the $SemiGroup$ type. Having constructed the Event-B operator, the **where** statements and the theorems associated with it are mapped to Event-B axioms. This is done by mapping the expressions of the **where** and theorem statements. Expression mapping is described below 9.4.3

9.4.2.2 Typed With Type Classes

When an associated element is typed with a type class, the whole of the type class is mapped to Event-B. To distinguish the operators mapped from the associated elements type class and the type class that is being mapped, the elements from the associated element types name are changed. e.g., in the case of $M : Monoid\langle R\rangle$ the operators generated for M will all have the prefix M so M_ident and M_op will be created. The names of the axioms will also gain the same prefix.

When a type class is in the process of being mapped from B$^\sharp$ to Event-B a map of associated element names in B$^\sharp$ to the operator names in Event-B is maintained. When possible the operators will have the same names as the associated elements. In the example above when the monoid is mapped the mapping becomes slightly more interesting. The monoid Ms map is $\{ident : M\_ident, op : M\_op, \ldots\}$. This mapping is used when compiling expressions to find the correct Event-B operators.

9.4.2.3 Functions

Functions associated with type classes are mapped to Event-B operators with the same parameters and return type of the function. There is no need to map the definition of the function. The functions are only defined by the theorems about the functions.

9.4.3 Expressions

Once the operators for type classes have been generated the **where** and theorem expressions are mapped to axioms. In principal theorems could be mapped to theorems, however as these have already been proved in the standard B$^\sharp$ mapping they do not need to be re-proved so it is easier to map them as axioms.

As an example in the *Group* class there is the following theorem:

$$\forall x, y : G \cdot equ(op(x, y), y) \Leftrightarrow equ(x, ident) \tag{9.17}$$

If x acts as the identity on any element it is the identity. This is mapped to the following axiom in Event-B:

$$\forall x, y \cdot op(x, y) = y \Leftrightarrow x = ident \tag{9.18}$$

This is achieved by first mapping the B$^\sharp$ to the simplified form:

$$\forall \langle G : Group \rangle x, y : G \cdot G.equ(G.op(x, y), y) \Leftrightarrow G.equ(x, G.ident) \tag{9.19}$$

Now this is compiled in the same way as B$^\sharp$ statements have been compiled previously. Except as G is a complete B$^\sharp$ type it does not need to be quantified over so this disappears, and whenever an associated element is mapped (e.g., $G.op$) the map of associated elements is used to find the Event-B operator. Finally equ is special cased to become equality on complete Event-B types.

In the $Ring$ type class the following theorem was declared:

$$\forall x : R \cdot equ(M.op(x, G.ident), G.ident) \wedge equ(M.op(G.ident, x), G.ident) \tag{9.20}$$

(Multiplication by zero gives zero). This is expanded to:

$$\forall \langle R : Ring \rangle x : R \cdot equ(R.M.op(x, R.G.ident), R.G.ident) \wedge equ(M.op(R.G.ident, x), R.G.ident) \tag{9.21}$$

In this case when $R.M.op$ is called, $R.M$ finds the monoid mapping for M, so $R.M.op$ finds the Event-B operator associated with the M element. Which results in the expected expression being generated.

The complete mapping for the B$^\sharp$ $Ring$ to an Event-B abstract type is:

$$
\begin{aligned}
&G\_ident() : Ring\\
&G\_op(p_1 : Ring, p_2 : Ring) : Ring\\
&G\_raiseToL(p_1 : Ring, p_2 : pNat) : Ring\\
&G\_raiseToR(p_1 : Ring, p_2 : pNat) : Ring\\
&M\_ident() : Ring\\
&M\_op(p_1 : Ring, p_2 : Ring) : Ring\\
&M\_raiseToL(p_1 : Ring, p_2 : pNat) : Ring\\
&M\_raiseToR(p_1 : Ring, p_2 : pNat) : Ring \qquad (9.22)\\
&axioms :\\
&\quad \forall x, y, z \cdot G\_op(G\_op(x, y), z) = G\_op(x, G\_op(y, z))\\
&\quad \text{All other } AbGroup \text{ typing, where statements, and theorems as axioms}\\
&\quad \forall x, y, z \cdot M\_op(M\_op(x, y), z) = M\_op(x, M\_op(y, z))\\
&\quad \text{All other } Monoid \text{ typing, where statements and theorems as axioms}\\
&\quad \forall x, y, z \cdot M\_op(x, G\_op(y, z)) = G\_op(M\_op(x, y), M\_op(x, z))\\
&\quad \text{All other Ring typing, where statements and theorems as axioms}
\end{aligned}
$$

Creating a *Ring* in B$^\sharp$ and instantiating it means that the axioms of the components of the ring, and the theorems about these components do not have to be restated and reproved. This is a considerable saving of effort for the Event-B modeller.

Before implementing this form of instantiation the syntax change suggested in 8.18 (e.g., in the B$^\sharp$ definition of the *Ring* having $(add, zero) : AbGroup$ instead of $G : abGroup$). This change could propagate through to the Event-B instantiation, giving nicer names to the associated elements.

9.5 Additional Operators

In addition to the suggested form of instantiation in the previous section (Section 9.4), there are many additional keywords which could be added to the B$^\sharp$ tool to aid both the development of mathematical theories and Event-B models. For example, a keyword for homomorphisms and isomorphisms could be added. Given an abstract type (e.g., a Group), an isomorphism keyword could create a new type class which related the required elements of two Groups with a function between the Groups. This function would be constrained with **where** statements such that it represented a group isomorphism. Additionally, an instance statement could be used to show that two concrete groups are isomorphic, allowing B$^\sharp$ to transfer theorems and proofs between the two isomorphic types.

A further possibility would be to add keywords to allow $B^\sharp$ to contain Event-B model contexts and syntax. This could be achieved by adding *context* and *machine* keywords, which could use the recently developed XEvent-B[1] tool to generate Event-B models. This would allow $B^\sharp$ to also add new features to standard Event-B e.g., reasoning about composition and concretisation.

9.6 Translating to Other Provers

The translation has demonstrated how a HOL style language can be translated to the typed set theory syntax of Event-B. A natural question to ask is then is possible to use this similarity to bring results from a more mature theorem prover such as Isabelle/HOL or HOL light to Event-B. A couple of reasons why we may want such a translation is to bring features from another language to $B^\sharp$ or to help with proofs of theorems generated by $B^\sharp$.

If the translation from the core of $B^\sharp$ to Event-B were to be formalised, and a similar translation to a more mature HOL style language were to be demonstrated, then the more mature HOL style language could be used to prove the $B^\sharp$ theorems and the proofs could be trusted within Event-B. To achieve this, it would be necessary to formalise the semantics at the core of $B^\sharp$, followed by formalising the translation to HOL and Event-B. At this point, proofs about $B^\sharp$ done in HOL could be used within the Event-B environment. A disadvantage of this is that any future features added to $B^\sharp$ which translate directly to Event-B (rather than the $B^\sharp$ core) would not be able to do proofs in HOL until similar work had been done to demonstrate semantic alignment (which may be impossible if the translation does not always generate valid Event-B).

Demonstrating an equivalence between the core of $B^\sharp$ and the core of a HOL style language would give the potential to bring features from the HOL style language into $B^\sharp$. Features in HOL style languages are added by translating the new features into the small core of the language. A way to bring these features to $B^\sharp$ could be to create $B^\sharp$ to $B^\sharp$ translations which copy the additional features in HOL languages. Another alternative is to add a new feature which mirrors the HOL feature to $B^\sharp$. This new feature could be implemented by a translation to the HOL language, then using HOL to reduce it to the core of the HOL language and translating it back to $B^\sharp$. The resulting $B^\sharp$ statements would then be translated to Event-B using the translation described in Chapter 6. A difficulty with this approach is that the HOL features will also add proving features which will not be available in the Event-B prover, which may make proofs about these statements more difficult.

9.7 Implementing Future Work

As a final note on the future work, given the current mappings of $B^\sharp$ the generated theorems cause the current Rodin toolset to become slow as the generated theorems increase in complexity. Some

[1]https://sourceforge.net/projects/rodin-b-sharp/files/Plugin_XEvent-B/

of the changes, such as adding more proof rules will help with this situation, as these proof rules will help to reduce the number of hypothesis the Rodin system has to deal with at any given time. However, before continuing development on the $B^\sharp$ tool effort should be put into optimising the performance of the Event-B tool for use with statements generated by $B^\sharp$, and fixing bugs within the Theory Plug-in which increase the complexity of current proofs.

Chapter 10

Conclusion

The primary aim of this thesis was to improve the method by which mathematical types can be defined within the Rodin toolset, and made available to the Event-B modeller. This has been achieved by the creation of a new language called B$^\sharp$, which had many elements in common with a HOL-style language. Several additional features were included within B$^\sharp$ such as type classes, type bodies (used to associate theorems and functions with the classes), and instantiation (used to associated concrete types to type classes). Due to the additional features of the new language developing heritable mathematical theorems within B$^\sharp$ was considerably easier than within Event-B.

To make the theorems developed in B$^\sharp$ available to the Event-B modeller a series of mappings from B$^\sharp$ to Event-B were declared. Theorems within B$^\sharp$ could then be translated to Event-B. This also acted to allow the theorems in B$^\sharp$ to be proved using the Rodin interactive prover (i.e., the theorems could be proved within the translation), removing the need to construct a specific prover for B$^\sharp$. The B$^\sharp$ instantiation caused theorems and functions within the type classes to be recreated as theorems and functions within concrete types. This automatic instantiation made proving within the interactive prover considerably easier as theorems did not have to be instantiated at the times of the proof. Instantiation at the time of the proof has a second disadvantage, in that you need to re-instantiate in each different proof.

To test the feasibility of the B$^\sharp$ language an IDE was created to allow development of B$^\sharp$ theories. This IDE also contained a compiler to create Event-B theories from the B$^\sharp$ ones. Along with the translation many additional IDE features were added, such as syntax aware completion and error detection. This eased the creation of theories, as many problems with declarations were noticed faster than in the equivalent development in Event-B. Several B$^\sharp$ theories were defined. It was found that creating heritable theories within B$^\sharp$ was considerably more concise than creating the equivalent theorems within Event-B, due to the additions of the following features:

1. type classes
2. subtypes

3. unification of functions and operators to a single type
4. unification of predicates and the boolean type
5. methods, allowing functions to use elements from type classes without explicit declaration

Along with a feature allowing the instantiation of functions and theorems using instances, further benefits, and the evidence for them, was summarised in Section 8.8.5.

In Section 3.10 of Chapter, we saw a list of problems with developing mathematical theories in Event-B. The table below summarises how these issues were solved within $B^\sharp$:

Rather than being able to use the subset as a type the user is required to use the subset's super type, and then write additional statements to say when the operator/proof rules can be used.	Within $B^\sharp$ elements can be typed using any class or datatype. The classes allow the creation of subtypes which become subsets in Event-B. This includes constraining parametric types as seen in the definition of *CommOp* (4.13), with translation example (5.11) and mapping rules (6.4.2). This resolves the issue of long subsetting statements (3.2). $B^\sharp$'s parametric type representation was seen in the translation of the *Ring* definition 8.21. Where $B^\sharp$ is much more concise and readable than the generated Event-B. As seen in Section 5.8, a similar process is used when translating typed parameters to Event-B subsets for functions, quantifiers, lambdas, and class declarations.
Concrete operators can not be passed as a type, causing the user to define a new operator returning a functional representation of the original operator.	This was resolved by $B^\sharp$ translating to two operators, the additional operator being defined in such a way that it could be used in Event-B expressions without arguments. When translating, $B^\sharp$ chooses the correct Event-B operator based on context. This was seen in example 5.2 with translation example given in 5.1.2 and mapping shown in 6.4.4 and 6.5.2. This means the user no longer has to declare additional operators or wrap operators in lambdas as was seen in Section 3.4.
Operators defined axiomatically can not be instantiated.	In $B^\sharp$, rather than axiomatically defining a type, **Class** declarations are used. As seen in 4.1.4 abstract types can be used in the place of axiomatic definitions and can be instantiated.

Demonstrating that a concrete structure is a subset of an abstract structure does not result in theorems and proofs being inherited. Instead it is necessary to rewrite the theorems and prove them (although the proofs are trivial).	$B^\sharp$ resolved this using the **Instance** keyword, which used a translation to Event-B to instantiate the abstract classes with concrete types. This resolves the issue discussed in Section 3.4. The user no longer has to manually redeclare theorems, reducing work when defining theories. The example, translation, and mapping can be found at 4.26, 5.1.7 and 7.1.4 respectively.
After demonstrating isomorphisms, there is still considerable work to move proof rules and theorems to the new type.	It is possible to represent isomorphisms using $B^\sharp$'s type class e.g., we can define a type of monoid isomorphisms. A potential way of generalising this approach is discussed in 9.5, such that a new isomorphism type class does not have to be defined for every abstract class.
Every operator had to be named uniquely. There are cases where, due to inadvertently re-using the same name, sections of code need to be redefined and reproved.	This issue was solved in $B^\sharp$ using a different scoping and import system to Event-B. As described in Section 4.3, the ability to re-use function names allows them to be named more intuitively (e.g., we can have add functions on naturals and integers).
Predicates being a separate syntactic categories created considerably more work in defining and instantiating abstract theories about relations.	In $B^\sharp$, predicates are not a separate syntactic category. This resolved problems discussed in 3.6.1. This can be seen by comparing the Event-B theorem 3.15 with the equivalent $B^\sharp$ theorem 4.8. Examples of the $B^\sharp$ syntax can be found in Section 5.1.3, with the mapping in Section 6.5.1.

Along with the improvements above, the additional features added by the Theory Plug-in to the Event-B language (datatypes and parametric types) are maintained. $B^\sharp$ allows the axiomatic definition of types using **Class** statements (Section 4.1.4). The mapping of these types generates constrained Event-B subsets. Unlike the axiomatic types in the Theory Plug-in, these cannot generate consistency errors, however, they are also harder to use directly when modelling in Event-B. A possible solution of another translation is suggested in 9.4. This would generate axiomatically defined Event-B types from the $B^\sharp$ classes.

The approach of having a compilation stage in development of $B^\sharp$ could be extended to consistently allow many other features to be added to $B^\sharp$. Any feature that is added to $B^\sharp$ is given meaning by its translation to Event-B. If a new feature is inconsistent it will generate inconsistent Event-B, and the user will not be able to prove the generated Event-B. The translation stage gives the ability to consistently add features in a similar fashion to the way other languages use a meta language (ML). Many additional possible features are outlined in the previous chapter.

Finally the translation from $B^\sharp$ to Event-B showed how a HOL style language could be translated to a language based on typed set theory (such as Event-B). Using similar techniques it may be possible to translate theorems from a more mature theorem proving language such as Isabelle/HOL to Event-B. To get the full benefit from such a translation it would be necessary to show some form of semantic equivalence rather than reconstructing the proofs within the $B^\sharp$ environment.

Appendix A

Additional Translation Functions

This appendix contains additional translation functions not contained within chapter 6 or 7.

A.1 Expanding Parametric Types

In section 6.5 it is described how parametric types constrained to type classes need to be expanded to allow compilation to Event-B. The following recursive functions are used to genetate the additional types required in Event-B.

First a function is defined that can recursively expand the constrained *paraType*:

```
$expandParaType(p : paraType) \rightarrow List\langle paraType\rangle\{$
$// T : TypeClass\langle T_{expr1}, \ldots ., T_{expri}\rangle$
$// Id\ (class\ contrType*)?$
  $let\ T : TypeClass\langle T_{expr1}, \ldots ., T_{expri}\rangle = p$
  if $(TypeClass = nil)$ // There is no contraining $TypeClass$
    $return\ bSharpId(p)$ // Generates a base bsharp type, easily translatable to Event-B
  melif $(TypeClass \neq nil)$
     // p.class is a type class with a list of parametric type$\langle cp_1 : TC_1 \ldots cp_i : TC_i\rangle$
    $let\ A : AC, \ldots, Z : ZC = TypeClass.paraType$
    $let\ a_1, \ldots, a_i : AC\langle a_1 \ldots\rangle, \ldots, z_1, \ldots, z_i : ZC\langle z_1 \ldots\rangle$
          $= expandParaType(A : AC), \ldots, expandParaType(Z : ZC)$
    $return\ \{a_1, \ldots, a_i : A_i, \ldots, z_1, \ldots, z_i : Z_i, p : p.class\langle a_1, \ldots, a_i, \ldots, z_1, \ldots z_i\rangle\}$
$\}$
```
(A.1)

Note that the list of types generated in the penultimate line contains base types (with no contraints) e.g., a_1, and types contrained by type classes e.g., $a_i : A_i\langle a_1 \dots \rangle$. This expansion results in type statements that match those required by Event-B. The B$^\sharp$ base types can be turned fairly directly into Event-B types (a type T becomes an argument $T : \mathbb{P}(T\_evB)$. The operators generated by 6.5 can be uesd to contruct the other types (as in $S : Setoid\langle T \rangle$). This is done with the following translation function:

```
$evBArgForParaType(p : paraType) \rightarrow [param]\{$
  $let\ p : Class\langle T_1, \dots, T_i \rangle = p$
  if $Class = nil$
    $return\ evBId(p) : \mathbb{P}(evBTypeId(p))$ // $p : \mathbb{P}(P\_evB)$
  elif $Class \neq nil$                                              (A.2)
    $let\ evBOpName = evBId(Class)$ // Event-B class constructor
  // Create the Event-B type with class constructor
    $return\ evBId(p) : evBOpName(evBId(a_1), \dots evBId(a_i)$
$\}$
```

Due to the expansion done using A.1 the types $a_1 \dots a_i$ have already been contructed and therefore no longer have contraining types of their own (i.e., they are not of the form $a : A$).

Putting this together when a B$^\sharp$ object has a parametric context (e.g., class declarations) each parameter in the context is turned into Event-B parameters with the following translation function:

```
$evBParam(p : paraType) \rightarrow List\langle [param] \rangle\{$
$let\ T : Class = p$                                                 (A.3)
  $return\ evBArgsForParam(expandParaType(T : Class))$ // $T_1 : T_{evB1}, \dots T_i : T_{evBi}$
$\}$
```

The types are first expanded, this results in a series of types that can be converted easily to Event-B sets using the set construction operators already generated.

A.2 Unary Types

Given an Event-B representation of a class, and the B$^\sharp$ definition of the class it is then possible to get the Event-B set from the expression as described in 6.4.6. Here an Event-B representation of a class is deconstructed to find the set type that the type class is associated with.

```
$baseTypeForExpr(e : [expr], c : class) \rightarrow [set{-}expr]\{$
  $let\ sType = head(c.\mathbf{constrType})$ // The first supertype of the class
  if $sType\ isA\ class$
    if $c.\mathbf{parameter} \neq nil$ // The class has required elements
      $return\ baseTypeForExpr(prj1(e), sType)$
    elif $c.\mathbf{parameter} = nil$ // The class has required elements
      $return\ baseTypeForExpr(e, sType)$                    }        (A.4)
  elif $\neg(sType\ isA\ class)$
    if $c.\mathbf{parameter} \neq nil$
      $return\ prj1(e)$
    elif $c.\mathbf{parameter} = nil$
      $return\ e$
```

The functions is slightly more complicated than described within the chapter as. This is because type classes which add no new required elements have an identical Event-B expression to their superclass, so the first element of the Event-B expression is not seperated from the Event-B expression instead the whole Event-B expression is used.

Along with $evBType(p : parameter)$ function (6.20), there are near identical functions for instance and bracket types:

```
$evBType(i : instance) \rightarrow [set{-}expr]\{$
  $return\ baseTypeForExpr(evB(i), i.class)$                    (A.5)
}
$evBType(b : typeBracket) \rightarrow [set{-}expr]\{$
  $return\ (evBType(b.constrType))$ // Bracketed in Event-B      (A.6)
}
                                                                (A.7)
```

A.3 Finding The Correct Supertype

When in B$^\sharp$ a type argument is passed in place of one of its supertypes it is necessary to get the Event-B representation of the supertype (rather than the representation of the current type. The following functions recursively climb the B$^\sharp$ type structure to find the correct representation:

```
$evBTypeForClassInternal(e : [expr], c : class, d : class) \rightarrow [expr]\{$
  if $c = d$
    return $e$ // $e$ already represents the correct class
  elif $c \neq d$
    if $c.\mathbf{parameters} \neq nil$                                              (A.8)
      return $evBTypeForClassInternal(prj1(e), head(c.\mathbf{constrType}), d)$
    elif $c.\mathbf{parameters} = nil$
      return $evBTypeForClassInternal(e, head(c.\mathbf{constrType}), d)$
$\}$
```

```
$evBClassReprForClass(p : paraType, d : class) \rightarrow [expr]\{$
  return $evBTypeForClassInternal(evBID(p), p.class, d)$                       (A.9)
$\}$
```

```
$evBClassReprForClass(i : instance, d : class) \rightarrow [expr]\{$
  return $evBTypeForClassInternal(evB(i), i.class, d)$                         (A.10)
$\}$
```

The approach taken is to climb the class hierarchy until the current class c matches the desired class d, when the current class matches the desired type then the current Event-B [expr] is the correct representation of the class. When the classes do not match, if the current class introduces new parameters then the Event-B supertype is represented by $prj1$ of the current type. If no required elements are added then Event-B representation of the super type is the same as the Event-B representation of the current type.

A.4 Getting Inferred Types for Type Constructors

As described in Section 6.4.6. When a class constructor is called with type arguments it is neccessary to infer additional types from the arguemnts. These functions are used to do that inferrence for various different unary elements.

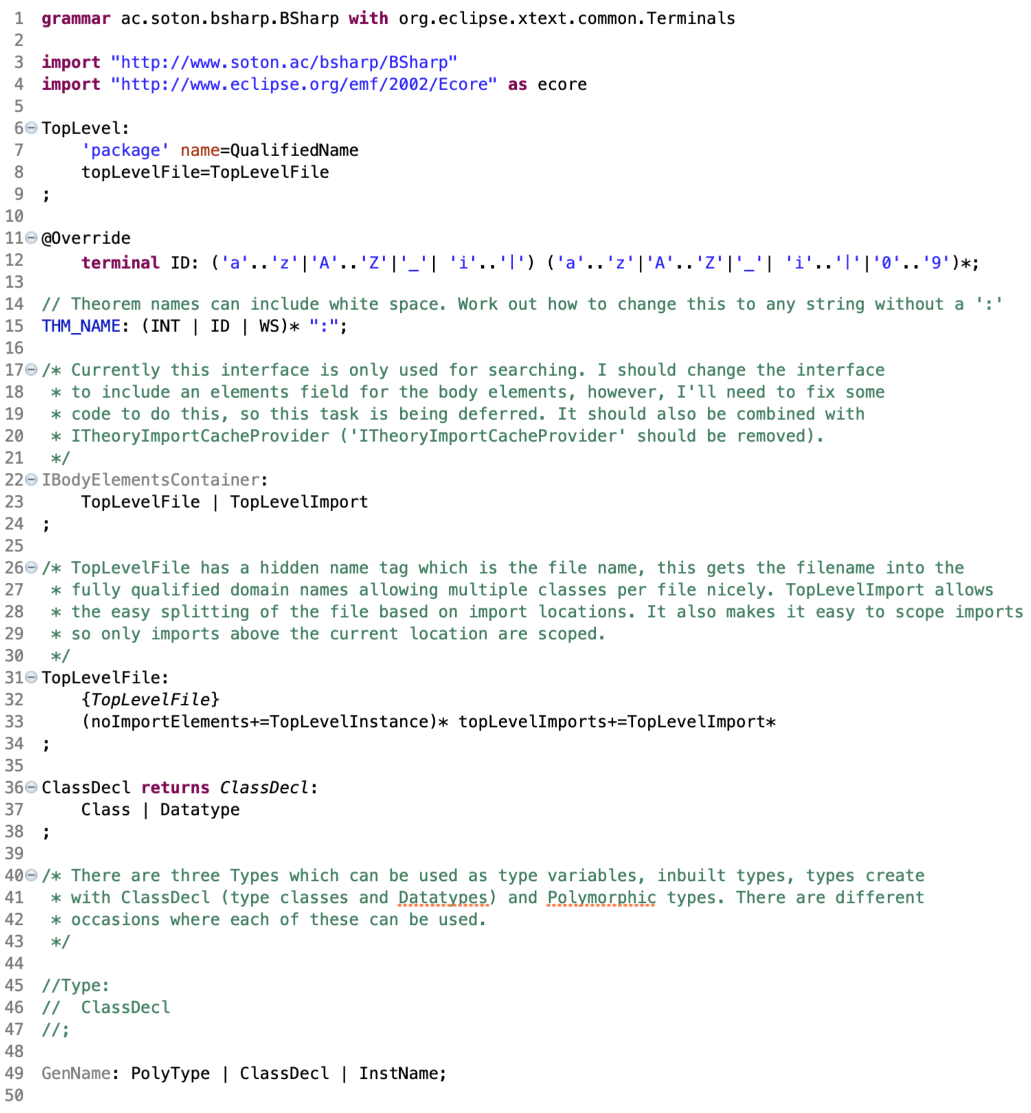
```
$getInferredTypes(p : paraType, c : class) \rightarrow List\langle[expr]\rangle\{$
    $let\ r : tClass\langle T_1, \dots, T_n\rangle = p$
    $let\ S_1, \dots, S_i = getInferredTypes(T_1, \dots, T_i)$                                  (A.11)
    $let\ cRepr = evBClassReprForClass(p, c)$ // Event-B instance of c
    $return\ S_1, \dots, S_i, cRepr$  // Type arguments for an operator
$\}$
$getInferredTypes(t : typeConstr, c : class) \rightarrow List\langle[expr]\rangle\{$
    $let\ Type\langle T_1, \dots, Ti\rangle = t$
    if $Type\ isA\ datatype$
        $let\ dtName = evBId(Type)$
        $return\ dtName\langle evB(T_1), \dots, evB(T_i)\rangle$
    elif $Type\ isA\ class$                                                                   (A.12)
        $let\ T_1, \dots, T_i = getInferredTypes(T_1, \dots T_i), Type.\mathbf{paraType}.class)$
        $let\ typeOpName = evBTypeId(t.class)$
        $return\ T_1, \dots, T_i, typeOpName(T_1, \dots, T_i)$
$\}$
$getInferredTypes(b : typeBracket, c : class) \rightarrow List\langle[expr]\rangle\{$
    $return\ getInferredTypes(d.constrType, c)$                                                (A.13)
$\}$
$getInferedTypes(ct : constrType, c : class) \rightarrow [expr]\{$
    $return\ evBType(ct)$                                                                      (A.14)
$\}$
```

The function to get inferred types on *constrType* is lowest precedence, so where possible one of the other functions will be called first (as *constrType* could be a wrapper for one of the other types).

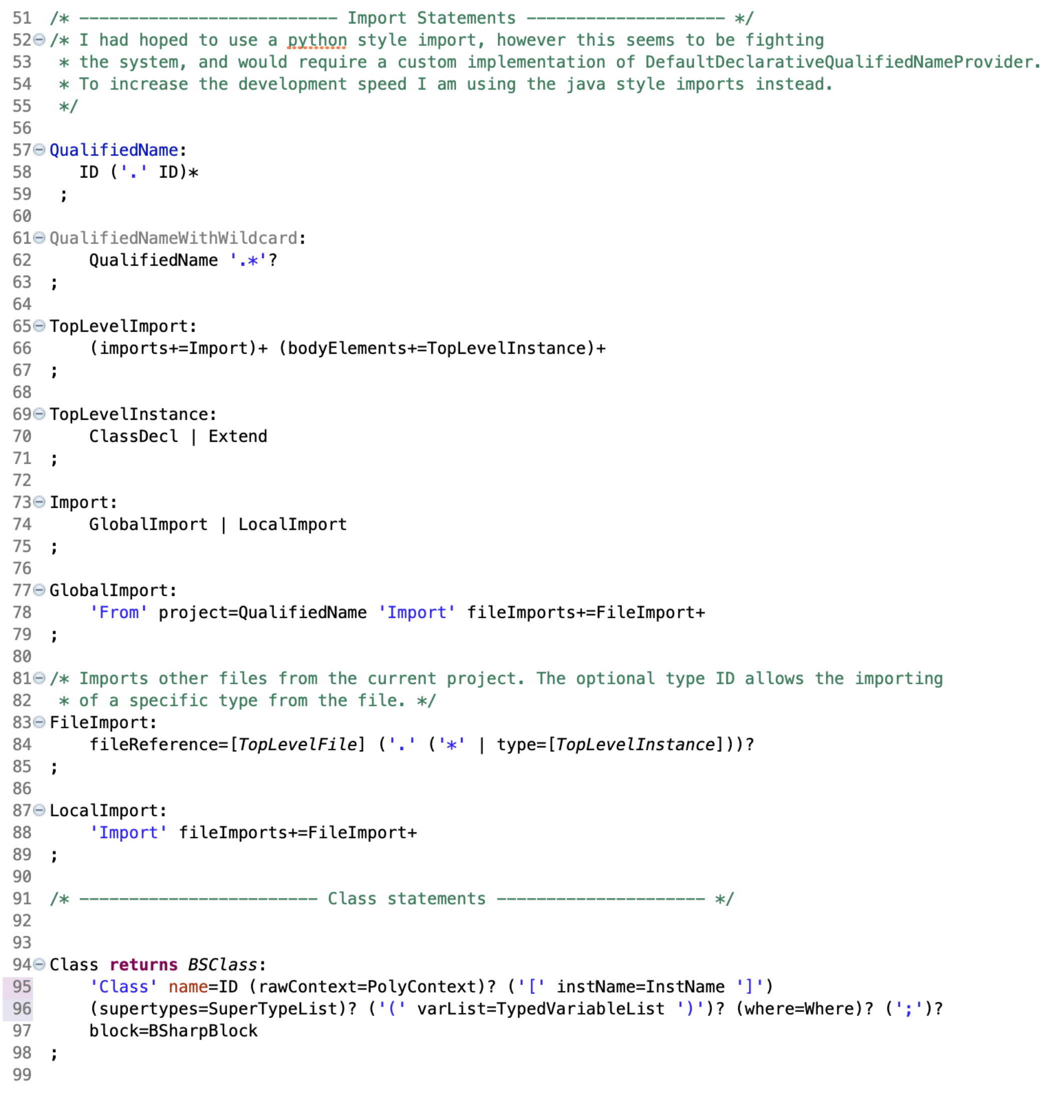
Appendix B

B$^\sharp$ Complete Syntax Definition

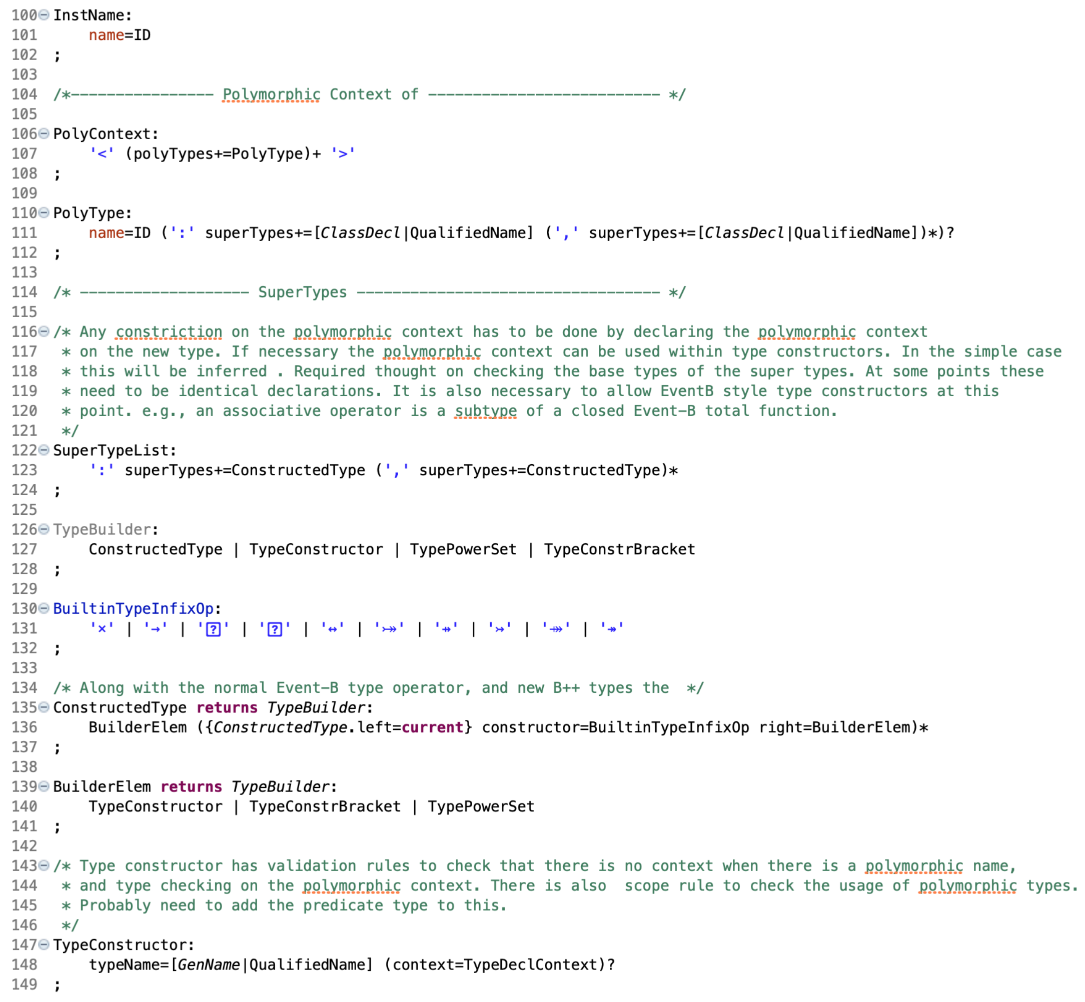
This appendex contains that Syntax definition of B$^\sharp$, which was used within Xtext to allowing the parsing of B$^\sharp$ files.



```
grammar ac.soton.bsharp.BSharp with org.eclipse.xtext.common.Terminals

import "http://www.soton.ac/bsharp/BSharp"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

TopLevel:
    'package' name=QualifiedName
    topLevelFile=TopLevelFile
;

@Override
    terminal ID: ('a'..'z'|'A'..'Z'|'_'| 'i'..'|') ('a'..'z'|'A'..'Z'|'_'| 'i'..'|'|'0'..'9')*;

// Theorem names can include white space. Work out how to change this to any string without a ':'
THM_NAME: (INT | ID | WS)* ":";

/* Currently this interface is only used for searching. I should change the interface
 * to include an elements field for the body elements, however, I'll need to fix some
 * code to do this, so this task is being deferred. It should also be combined with
 * ITheoryImportCacheProvider ('ITheoryImportCacheProvider' should be removed).
 */
IBodyElementsContainer:
    TopLevelFile | TopLevelImport
;

/* TopLevelFile has a hidden name tag which is the file name, this gets the filename into the
 * fully qualified domain names allowing multiple classes per file nicely. TopLevelImport allows
 * the easy splitting of the file based on import locations. It also makes it easy to scope imports
 * so only imports above the current location are scoped.
 */
TopLevelFile:
    {TopLevelFile}
    (noImportElements+=TopLevelInstance)* topLevelImports+=TopLevelImport*
;

ClassDecl returns ClassDecl:
    Class | Datatype
;

/* There are three Types which can be used as type variables, inbuilt types, types create
 * with ClassDecl (type classes and Datatypes) and Polymorphic types. There are different
 * occasions where each of these can be used.
 */

//Type:
//  ClassDecl
//;

GenName: PolyType | ClassDecl | InstName;

```



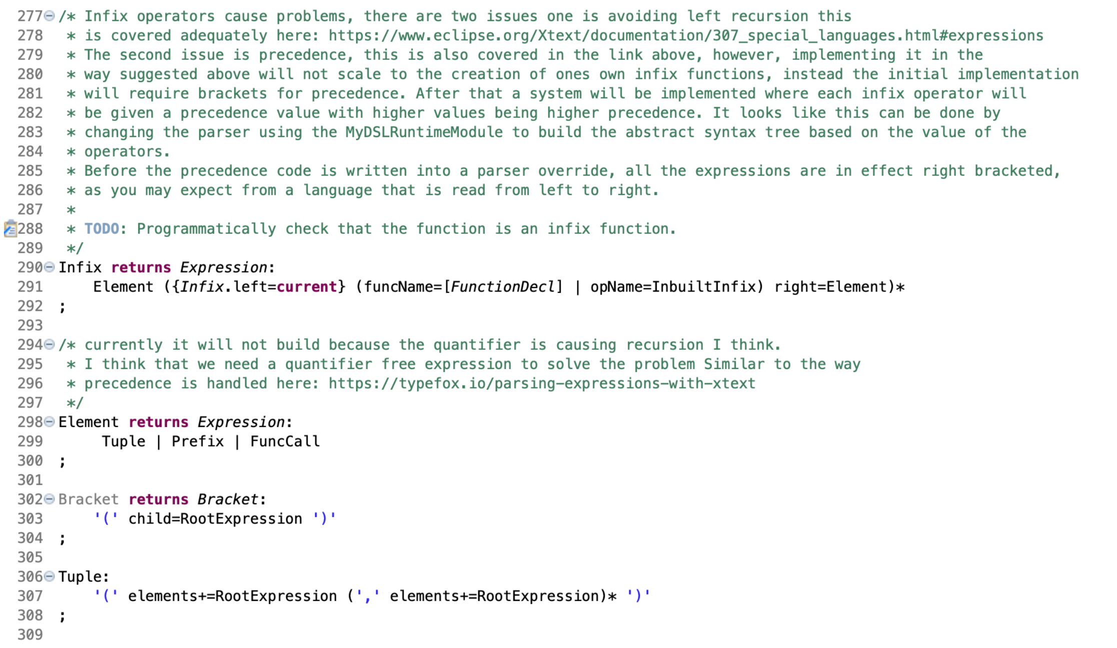
```
/* --------------------------- Import Statements --------------------- */
/* I had hoped to use a python style import, however this seems to be fighting
 * the system, and would require a custom implementation of DefaultDeclarativeQualifiedNameProvider.
 * To increase the development speed I am using the java style imports instead.
 */

QualifiedName:
   ID ('.' ID)*
 ;

QualifiedNameWithWildcard:
    QualifiedName '.*'?
;

TopLevelImport:
    (imports+=Import)+ (bodyElements+=TopLevelInstance)+
;

TopLevelInstance:
    ClassDecl | Extend
;

Import:
    GlobalImport | LocalImport
;

GlobalImport:
    'From' project=QualifiedName 'Import' fileImports+=FileImport+
;

/* Imports other files from the current project. The optional type ID allows the importing
 * of a specific type from the file. */
FileImport:
    fileReference=[TopLevelFile] ('.' ('*' | type=[TopLevelInstance]))?
;

LocalImport:
    'Import' fileImports+=FileImport+
;

/* ------------------------ Class statements --------------------- */


Class returns BSClass:
    'Class' name=ID (rawContext=PolyContext)? ('[' instName=InstName ']')
    (supertypes=SuperTypeList)? ('(' varList=TypedVariableList ')')? (where=Where)? (';')?
    block=BSharpBlock
;

```



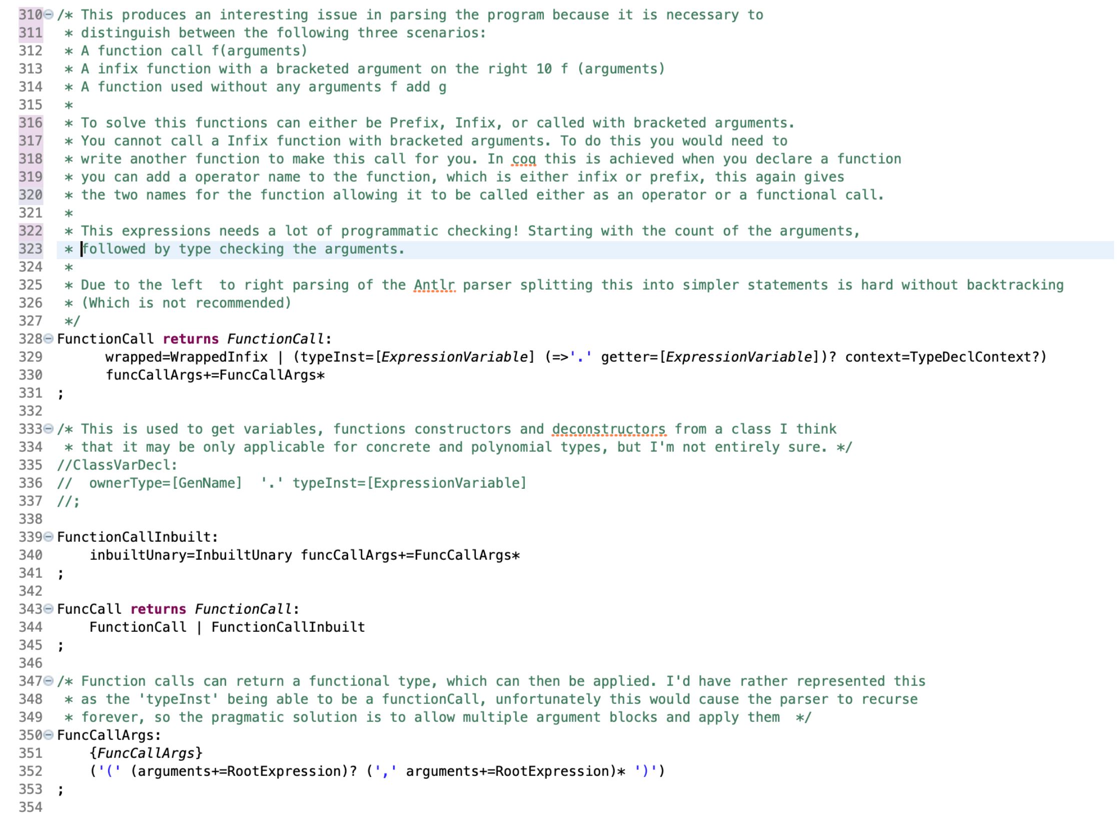
```
InstName:
    name=ID
;

/*---------------- Polymorphic Context of -------------------------- */

PolyContext:
    '<' (polyTypes+=PolyType)+ '>'
;

PolyType:
    name=ID (':' superTypes+=[ClassDecl|QualifiedName] (',' superTypes+=[ClassDecl|QualifiedName])*)?
;

/* ------------------- SuperTypes ---------------------------------- */

/* Any constriction on the polymorphic context has to be done by declaring the polymorphic context
 * on the new type. If necessary the polymorphic context can be used within type constructors. In the simple case
 * this will be inferred . Required thought on checking the base types of the super types. At some points these
 * need to be identical declarations. It is also necessary to allow EventB style type constructors at this
 * point. e.g., an associative operator is a subtype of a closed Event-B total function.
 */
SuperTypeList:
    ':' superTypes+=ConstructedType (',' superTypes+=ConstructedType)*
;

TypeBuilder:
    ConstructedType | TypeConstructor | TypePowerSet | TypeConstrBracket
;

BuiltinTypeInfixOp:
    '×' | '→' | '?' | '?' | '↔' | '⤖' | '⇸' | '↣' | '⤀' | '↠'
;

/* Along with the normal Event-B type operator, and new B++ types the  */
ConstructedType returns TypeBuilder:
    BuilderElem ({ConstructedType.left=current} constructor=BuiltinTypeInfixOp right=BuilderElem)*
;

BuilderElem returns TypeBuilder:
    TypeConstructor | TypeConstrBracket | TypePowerSet
;

/* Type constructor has validation rules to check that there is no context when there is a polymorphic name,
 * and type checking on the polymorphic context. There is also  scope rule to check the usage of polymorphic types.
 * Probably need to add the predicate type to this.
 */
TypeConstructor:
    typeName=[GenName|QualifiedName] (context=TypeDeclContext)?
;
```

```

TypePowerSet:
    'ℙ' '(' child=ConstructedType ')'
;

TypeConstrBracket:
    '(' child=ConstructedType ')'
;

/* This is used in two different situations, and may well compile the same for both so don't delete
 * unless the compilation of the two places is different (even then it is more pleasant to not have
 * two identical syntax declarations in this file).
 */
TypeDeclContext:
    '<' typeName+=ConstructedType (',' typeName+=ConstructedType)* '>'
;

/* -------------------- Where Statement -------------------------------- */

/* Type checking (which is not implemented yet) is used to type check that Expression returns a
 * predicate. Expression has not yet been written, but is far too general to be included in the where
 * statement.
 */
Where:
    'where' expressions+=RootExpression (';' expressions+=RootExpression)*
;

/* ---------------------- Datatype declarations -------------------------- */

Datatype:
    'Datatype' name=ID (rawContext=PolyContext)? ('|' constructors+=DatatypeConstructor)+ block=BSharpBlock
;

/* PolyContext is the same as PolyContext used by the class declaration above. */

DatatypeConstructor:
     name=ID ('(' decons=TypedVariableList ')')?
;

/* ------------------------ Extension statement ---------------------------- */

Extend:
    'Extend' extendedClass=[ClassDecl|QualifiedName] '(' name=ID ')' block=BSharpBlock
;

```

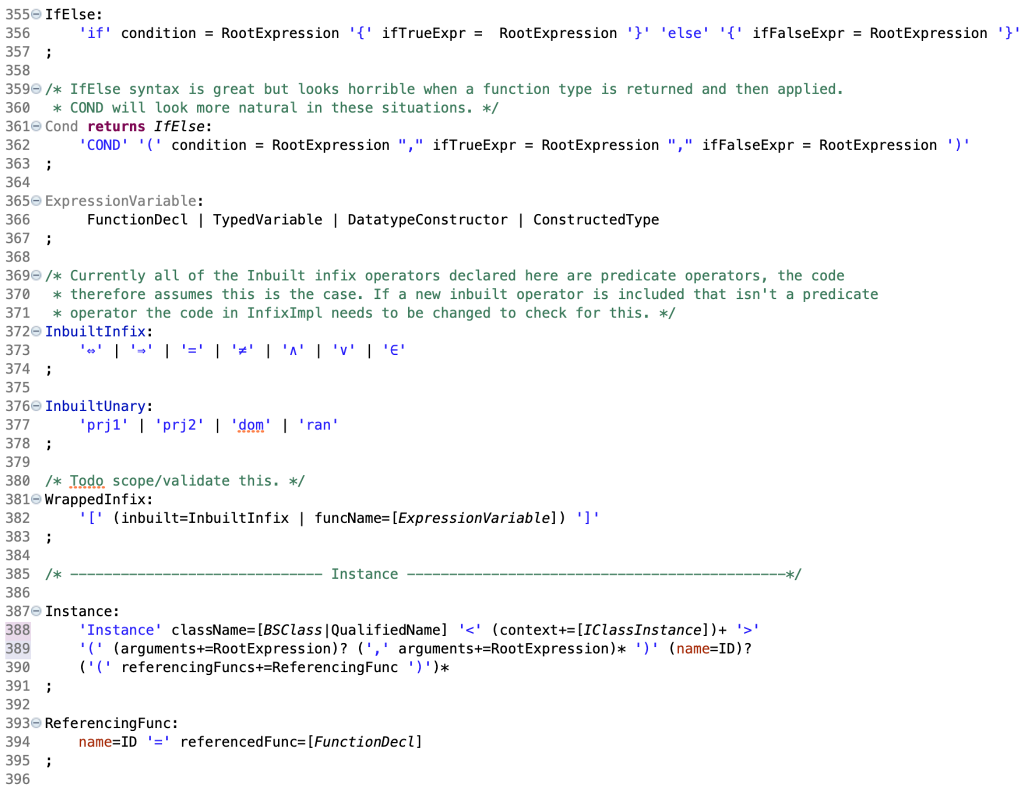
```
/* ------------------------ TypeBodyElements ------------------------------- */

/* The ordering of theorem bodies and instances is imported so they need to go into
 * a joint list to maintain the order.
 */
BSharpBlock:
    {BSharpBlock}
    '{' ((functions+=FunctionDecl) | (theorems+=TheoremBody) | theorems+=Instance)* '}'
;

/*------------------------- Functions --------------------------------- */

FunctionDecl:
    name=ID (context=PolyContext)? '(' (varList=TypedVariableList)? ')'
    ':' returnType=ConstructedType (infix='INFIX')? precedence=INT? expr=RootExpression
;

MatchStatement:
    'match' match=RootExpression '{'
    inductCase+=MatchCase (inductCase+=MatchCase)* '}'
;

MatchCase:
    '|' deconName=[DatatypeConstructor]
    ('(' variables+=TypedVariable (',' variables+=TypedVariable)* ')')? ':' expr=RootExpression
;

/* ----------------------------- Theorems -------------------------*/

TheoremBody:
    'Theorems' '{' (theoremDecl+=TheoremDecl)+ '}'
;

/* Type check that the expression is a predicate expression. From a lexing point
 * of view the semicolon is necessary because the THM_NAME rule is not good enough.
 */
TheoremDecl:
    name=THM_NAME expr=RootExpression ';'
;

TypedVariableList:
     variablesOfType+=VariableTyping (',' variablesOfType+=VariableTyping)*
;

VariableTyping:
    ((typeVar+=TypedVariable) (',' typeVar+=TypedVariable)* ':' type=ConstructedType)
;

/* I think this is here for reference purposes, e.g., otherwise it's quite difficult
 * to cross reference individual type names. */
TypedVariable:
    name=ID
;

/* -------------------------- Expressions ------------------------- */

/* -------------------------- Lambda and Quantifier --------------- */

/* The structure of Lambda and Quantifier are so similar that they're going to share a class */

Lambda returns QuantLambda:
    qType='λ' (context=PolyContext)? varList=TypedVariableList '|' expr=RootExpression
;

Quantifier returns QuantLambda:
    qType=('∀' | '∃') (context=PolyContext)? varList=TypedVariableList '·' expr=RootExpression
;

RootExpression returns Expression:
    Lambda | Quantifier | Infix | MatchStatement | IfElse
;

/* ---------------------- *Fix ------------------------------ */

Prefix returns Prefix:
    name=PrefixBuiltIn  elem=Element
;

PrefixBuiltIn:
  '¬'
;

```



```
/* Infix operators cause problems, there are two issues one is avoiding left recursion this
 * is covered adequately here: https://www.eclipse.org/Xtext/documentation/307_special_languages.html#expressions
 * The second issue is precedence, this is also covered in the link above, however, implementing it in the
 * way suggested above will not scale to the creation of ones own infix functions, instead the initial implementation
 * will require brackets for precedence. After that a system will be implemented where each infix operator will
 * be given a precedence value with higher values being higher precedence. It looks like this can be done by
 * changing the parser using the MyDSLRuntimeModule to build the abstract syntax tree based on the value of the
 * operators.
 * Before the precedence code is written into a parser override, all the expressions are in effect right bracketed,
 * as you may expect from a language that is read from left to right.
 *
 * TODO: Programmatically check that the function is an infix function.
 */
Infix returns Expression:
    Element ({Infix.left=current} (funcName=[FunctionDecl] | opName=InbuiltInfix) right=Element)*
;

/* currently it will not build because the quantifier is causing recursion I think.
 * I think that we need a quantifier free expression to solve the problem Similar to the way
 * precedence is handled here: https://typefox.io/parsing-expressions-with-xtext
 */
Element returns Expression:
     Tuple | Prefix | FuncCall
;

Bracket returns Bracket:
    '(' child=RootExpression ')'
;

Tuple:
    '(' elements+=RootExpression (',' elements+=RootExpression)* ')'
;
```



```
/* This produces an interesting issue in parsing the program because it is necessary to
 * distinguish between the following three scenarios:
 * A function call f(arguments)
 * A infix function with a bracketed argument on the right 10 f (arguments)
 * A function used without any arguments f add g
 *
 * To solve this functions can either be Prefix, Infix, or called with bracketed arguments.
 * You cannot call a Infix function with bracketed arguments. To do this you would need to
 * write another function to make this call for you. In coq this is achieved when you declare a function
 * you can add a operator name to the function, which is either infix or prefix, this again gives
 * the two names for the function allowing it to be called either as an operator or a functional call.
 *
 * This expressions needs a lot of programmatic checking! Starting with the count of the arguments,
 * followed by type checking the arguments.
 *
 * Due to the left  to right parsing of the Antlr parser splitting this into simpler statements is hard without backtracking
 * (Which is not recommended)
 */
FunctionCall returns FunctionCall:
      wrapped=WrappedInfix | (typeInst=[ExpressionVariable] (=>'.' getter=[ExpressionVariable])? context=TypeDeclContext?)
      funcCallArgs+=FuncCallArgs*
;

/* This is used to get variables, functions constructors and deconstructors from a class I think
 * that it may be only applicable for concrete and polynomial types, but I'm not entirely sure. */
//ClassVarDecl:
//  ownerType=[GenName]  '.' typeInst=[ExpressionVariable]
//;

FunctionCallInbuilt:
    inbuiltUnary=InbuiltUnary funcCallArgs+=FuncCallArgs*
;

FuncCall returns FunctionCall:
    FunctionCall | FunctionCallInbuilt
;

/* Function calls can return a functional type, which can then be applied. I'd have rather represented this
 * as the 'typeInst' being able to be a functionCall, unfortunately this would cause the parser to recurse
 * forever, so the pragmatic solution is to allow multiple argument blocks and apply them  */
FuncCallArgs:
    {FuncCallArgs}
    ('(' (arguments+=RootExpression)? (',' arguments+=RootExpression)* ')')
;
```



```
IfElse:
    'if' condition = RootExpression '{' ifTrueExpr =  RootExpression '}' 'else' '{' ifFalseExpr = RootExpression '}'
;

/* IfElse syntax is great but looks horrible when a function type is returned and then applied.
 * COND will look more natural in these situations. */
Cond returns IfElse:
    'COND' '(' condition = RootExpression "," ifTrueExpr = RootExpression "," ifFalseExpr = RootExpression ')'
;

ExpressionVariable:
     FunctionDecl | TypedVariable | DatatypeConstructor | ConstructedType
;

/* Currently all of the Inbuilt infix operators declared here are predicate operators, the code
 * therefore assumes this is the case. If a new inbuilt operator is included that isn't a predicate
 * operator the code in InfixImpl needs to be changed to check for this. */
InbuiltInfix:
    '⇔' | '⇒' | '=' | '≠' | '∧' | '∨' | '∈'
;

InbuiltUnary:
    'prj1' | 'prj2' | 'dom' | 'ran'
;

/* Todo scope/validate this. */
WrappedInfix:
    '[' (inbuilt=InbuiltInfix | funcName=[ExpressionVariable]) ']'
;

/* ------------------------------ Instance ---------------------------------------------*/

Instance:
    'Instance' className=[BSClass|QualifiedName] '<' (context+=[IClassInstance])+ '>'
    '(' (arguments+=RootExpression)? (',' arguments+=RootExpression)* ')' (name=ID)?
    ('(' referencingFuncs+=ReferencingFunc ')')*
;

ReferencingFunc:
    name=ID '=' referencedFunc=[FunctionDecl]
;
```

Appendix C

B♯ Theories

This appendix contains the B♯ code written to demonstrate the usefulness of the B♯ language and tool.

Relations:

```
package Relations

From main Import main.∗

Class ReflexRel<T>[r] : T × T → Bool where ∀ x : T · r(x, x) { }

Class SymetricRel<T>[s] : T × T → Bool where ∀ x : T, y : T · s(x, y) ⇒ s(y, x) {
  Theorems {
    sym iff:
      ∀ x, y : T · s(x, y) ⇔ s(y, x);
  }
}

Class TransRel<T>[t] : T × T → Bool where ∀ x, y, z : T · t(x, y) ∧ t(y, z) ⇒ t(x, z) {
}

Class EquivRel<T>[e] : ReflexRel<T>, SymetricRel<T>, TransRel<T> {
  Theorems {
    TransInverse:
      ∀ x, y, z : T · e(x, y) ∧ ¬ e(y, z) ⇒ ¬ e(x, z);
    TransRewrite:
      ∀ x, y, z : T · e(x, y) ⇒ e(x, z) = e(y, z);
  }
}

Class Setoid<T>[S] : ℙ(T) (equ : EquivRel<S>) { }

Class Totality<T : Setoid>[tOp] : T × T → Bool where ∀ x, y : T · ¬ T.equ(x, y) ⇒ tOp(x, y) ∨ tOp(y, x)
      {
}
```

```
Class AntiSymmetry<T : Setoid>[lt] : T × T → Bool where ∀ x, y : T · lt(x, y) ∧ lt(y, x) ⇔ T.equ(x, y) {
}

Class Orderer<S : Setoid>[lt] : AntiSymmetry<S>, TransRel<S> {
}
```

Operators:

```
package Operators

From Relations Import Relations.Setoid

 Class baseOp<T:Setoid>[bOp] : T × T → T where ∀ x, y, z : T · T.equ(x, y) ⇒ T.equ(bOp(x, z), bOp(y,
      z)) ∧ T.equ(bOp(z, x), bOp(z, y)) {
   Theorems {
     BaseOpTheorem:
       ∀ a, b, x, y : T · T.equ(a, b) ∧ T.equ(x, y) ⇒ T.equ(bOp(a, x), bOp(b, y));

     BaseOpTheoremSimple Left:
       ∀ a, x, y : T · T.equ(x, y) ⇒ T.equ(bOp(a, x), bOp(a, y));

     BaseOpTheoremSimple Right:
       ∀ x, y, a : T · T.equ(x, y) ⇒ T.equ(bOp(x, a), bOp(y, a));
   }
 }

Class AssocOp<T:Setoid>[aOp] : baseOp<T> where ∀ x, y, z : T · T.equ(aOp(x, aOp(y, z)), aOp(aOp(x, y
      ), z)) { }

Class CommOp<T:Setoid>[cOp] : baseOp<T> where ∀ x, y : T · T.equ(cOp(x, y), cOp(y, x)) { }

Class AssocCommOp<T:Setoid>[Op] : AssocOp<T>, CommOp<T> { }
```

SemiGroup:

```
package Monoids

From Relations Import Relations.∗
From Operators Import Operators.∗

Class SemiGroup[SG] : Setoid (op : AssocOp<SG>) {
   Theorems {
     2 left ident no right:
       ∀ l1, l2 : SG · (∀ x : SG · equ(op(l1, x), x)) ∧ (∀ y : SG · equ(op(l2, y), y)) ∧ ¬ equ(l1, l2)
       ⇒ ¬(∃ r : SG · ∀ z : SG · equ(op(z, r), z));

     2 right ident no left:
       ∀ r1, r2 : SG · (∀ x : SG · equ(op(x, r1), x)) ∧ (∀ y : SG · equ(op(y, r2), y)) ∧ ¬ equ(r1, r2)
       ⇒ ¬(∃ l : SG · ∀ z : SG · equ(op(l, z), z));
```

```
      left right ident is unique1:
        ∀ LRident, Rident : SG · (∀ x : SG · equ(op(LRident, x), x) ∧ equ(op(x, LRident), x)
          ∧ (∀ y : SG · equ(op(y, Rident), y))) ⇒ equ(LRident, Rident);

      left right ident is unique2:
        ∀ LRident, Lident : SG · (∀ x : SG · equ(op(LRident, x), x) ∧ equ(op(x, LRident), x)
          ∧ (∀ y : SG · equ(op(Lident, y), y))) ⇒ equ(Lident, LRident);

      left right ident is unique:
        ∀ LRident, LorRident : SG · (∀ x : SG · equ(op(LRident, x), x) ∧ equ(op(x, LRident), x)
          ∧ ((∀ y :SG · equ(op(LorRident, y), y)) ∨ (∀ z : SG · equ(op(z, LorRident), z)))) ⇒ equ(LorRident
      , LRident);
    }
}
```

Monoid:

```
package Monoids

Import SemiGroup.∗

Class Monoid[M] : SemiGroup (ident : M)
 where ∀ x : M · equ(op(x, ident), x) ∧ equ(op(ident, x), x) {
  Theorems {
    ident is ident left: //Makes this become a theorem with inheritance. TODO:Make axioms become theorems
      on inheritance.
      ∀ x : M · equ(op(x, ident), x);
    ident is ident right:
      ∀ x : M · equ(op(ident, x), x);
    ident:
      ∀ x : M · equ(op(x, ident), ident) ⇔ equ(x, ident);

    unique ident:
      ∀ oIdent : M · (∀ r : M · equ(op(oIdent, r), r)) ∨ (∀ l : M · equ(op(l, oIdent), l)) ⇔ equ(oIdent, ident)
      ;
  }
}

From pNat Import pNat.pNat

Extend Monoid (Pow) {
  raiseToL(x : M, p : pNat) : M
    match p {
      | zero : ident
      | suc(ps) : op(x, raiseToL(x, ps))
    }

  raiseToR(x : M, p : pNat) : M
    match p {
            | zero : ident
            | suc(ps) : op(raiseToR(x, ps), x)
        }

    Theorems {
        RaiseToL Equiv Preservation:
          ∀ x , y: M, p : pNat ·  equ(x, y) ⇒  equ(raiseToL(x, p), raiseToL(y, p));

        RaiseToR Equiv Preservation:
          ∀ x , y: M, p : pNat ·  equ(x, y) ⇒  equ(raiseToR(x, p), raiseToR(y, p));

        RaiseToL addRule:
          ∀ x : M, p, q :pNat ·  equ(op(raiseToL(x, p), raiseToL(x, q)), raiseToL(x, p add q));
    }
}
```

Commutative Monoid:

```
package Monoids

Import Monoid.∗

From pNat Import pNat.pNat
From Operators Import Operators.∗

Class CommMonoid[M] : Monoid where (∀ x, y : M ·  equ(op(x, y), op(y, x))) {
    Theorems {
        LeftPow equals RightPow:
            ∀ x : M, p : pNat ·   equ(raiseToL(x, p), raiseToR(x, p));
    }
}
```

Naturals (Satisfying Peano's axioms):

```
package pNat

From main Import main.∗
From Relations Import Relations.∗

Datatype pNat
| zero
| suc(prev : pNat) {
    add(x, y : pNat) : pNat INFIX 100
        match x {
            | zero : y
            | suc(xs) : suc(xs add y)
        }

    add2(x : pNat) : pNat suc(suc(x))
```

```
  apply<T>(f : T → T, x : T) : T
    f(x)

  Instance Setoid<pNat>([=])

  Theorems {
    suc out of addition:
     ∀ x, y : pNat ·  x add suc(y) = suc(x add y);
    add zero:
     ∀ x : pNat ·  x add zero = x;
    reduction:
     ∀ x, y, z : pNat ·  x add y = x add z ⇔ y = z;
    Associative:
     ∀ x, y, z : pNat ·  x add (y add z) = (x add y) add z;
    Commutative:
     ∀ x, y : pNat ·  x add y = y add x;
  }
}

From Monoids Import Monoid.*
CommMonoid.*

Extend pNat (mon_Add) {
  Instance CommMonoid<pNat>(add, zero) addMon (times = raiseToL)

  Theorems {
    Anything times zero on left:
      ∀ x : pNat ·  times(x, zero) = zero;
    Zero times anything is Zero:
      ∀ x : pNat ·  times(zero, x) = zero;
    sucX times Y:
      ∀ x, y : pNat ·  times(suc(x), y) = y add times(x, y);
    Distrib Left Times:
      ∀ x, y, z : pNat ·  times(x, y add z) = times(x, y) add times(x, z);
    Distrib Right Times:
      ∀ x, y, z : pNat ·  times(y add z, x) = times(y, x) add times(z, x);
    Times Associative:
      ∀ x, y, z : pNat ·  times(x, times(y, z)) = times(times(x, y), z);
    Times Commutative:
      ∀ x, y : pNat ·  times(x, y) = times(y, x);
  }

  Instance CommMonoid<pNat>(times, suc(zero)) timesMon (power = raiseToL)
}

Extend pNat (minus) {
    decr(x : pNat) : pNat
    match x {
      | zero : zero
      | suc(xs) : xs
    }
  minus(x, y : pNat) : pNat INFIX 99
    match y {
      | zero : x
      | suc(ys) : decr(x minus ys)
    }
}

Extend pNat (divide) {
  divMod(n, d, count : pNat) : pNat × pNat
   match n {
    | zero : (zero, zero)
    | suc(ns) :
      if suc(ns) = d {
        (suc(count), zero)
      } else {
        if suc(ns) minus d = zero {
          (count, suc(ns))
        } else {
          divMod(suc(ns) minus d, d, suc(count))
        }
      }
    }
}
```

Group:

```
package Groups

From Monoids Import Monoid.∗

Class Group[G] : Monoid where ∀ x : G · ∃ y : G · equ(op(x, y), ident) ∧ equ(op(y, x), ident) {
  Theorems {
    Indent Unique:
     ∀ x, y : G · equ(op(x, y), y) ⇔ equ(x, ident);
    Inv unique:
     ∀ a, b, c : G · equ(op(a, b), ident) ⇒ equ(op(b, a), ident);
    Op Inverse:
     ∀ a, b, x, y : G · equ(op(a, x), ident) ∧ equ(op(b, y), ident) ⇔ equ(op(op(a, b), op(y, x)), ident);
    Cancellation Law:
     ∀ a, b, c : G · equ(op(a, c), op(b, c)) ⇔ equ(a, b);
    a squared equ a implies a equ ident:
     ∀ a : G · equ(op(a, a), a) ⇒ equ(a, ident);
    Inverse Func:
     ∃ inv : G → G · (∀ x : G · equ(op(inv(x), x), ident));
  }
}
```

Abelian Group:

```
package Groups

Import Group.∗
From Operators Import Operators.∗
From Monoids Import CommMonoid

Class AbGroup[AG] : Group, CommMonoid {

}
```

Ring:

```
package Rings

From Groups Import AbGroup.∗
From Monoids Import Monoid.∗
From Relations Import Relations.∗

Class Ring[R] : Setoid (G : AbGroup<R>, M : Monoid<R>) where
  ∀ x, y, z : R ·  equ(M.op(x, G.op(y, z)), G.op(M.op(x, y), M.op(x, z)));
  ∀ x, y, z : R ·  equ(M.op(G.op(y, z), x), G.op(M.op(y, x), M.op(z, x))); {

  Theorems {
    Zero theorem:
      ∀ x : R ·  equ(M.op(x, G.ident), G.ident) ∧  equ(M.op(G.ident, x), G.ident);
  }
}
```

References

[1] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[2] Mark Adams. Introducing hol zero. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software – ICMS 2010*, pages 142–143, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[3] Stuart F Allen, Robert L Constable, Rich Eaton, Christoph Kreitz, and Lori Lorigo. The nuprl open logical environment. In *International Conference on Automated Deduction*, pages 170–176. Springer, 2000.

[4] Egidio Astesiano, Michel Bidoit, Hélene Kirchner, Bernd Krieg-Brückner, Peter D Mosses, Donald Sannella, and Andrzej Tarlecki. Casl: the common algebraic specification language. *Theoretical Computer Science*, 286(2):153–196, 2002.

[5] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 171–177, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.

[7] Howard Barringer, Jen H Cheng, and Cliff B Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21(3):251–269, 1984.

[8] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[9] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[10] 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.

[11] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda – a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 73–78, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[12] Rod M Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.

[13] M. Butler, D. Dghaym, T. Fischer, T. S. Hoang, K. Reichl, C. Snook, and P. Tummeltshammer. Formal modelling techniques for efficient development of railway control products. In Alessandro Fantechi, Thierry Lecomte, and Alexander Romanovsky, editors, *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification*, pages 71–86, Cham, 2017. Springer International Publishing.

[14] Michael J. Butler, Jean-Raymond Abrial, and Richard Banach. Modelling and refining hybrid systems in Event-B and Rodin. In Luigia Petre and Emil Sekerinski, editors, *From Action Systems to Distributed Systems - The Refinement Approach.*, pages 29–42. Chapman and Hall/CRC, 2016.

[15] 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.

[16] Pierre Chartier. Formalisation of B in Isabelle/HOL. In *International Conference of B Users*, pages 66–82. Springer, 1998.

[17] Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

[18] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The maude 2.0 system. In *International Conference on Rewriting Techniques and Applications*, pages 76–87. Springer, 2003.

[19] Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-corn, the constructive Coq repository at nijmegen. In *MKM*, volume 3119 of *Lecture Notes in Computer Science*, pages 88–103. Springer, 2004.

[20] Haskell Brooks Curry, Robert Feys, William Craig, and William Craig. *Combinatory logic, vol. 1.* North-Holland Publ., 1958.

[21] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[22] Dana Dghaym, Matheus Garay Trindade, Michael Butler, and Asieh Salehi Fathabadi. A graphical tool for event refinement structures in event-b. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z: Proceedings of the 5th International Conference, ABZ 2016, Linz, Austria, May 23-27, 2016 (23/02/16)*, pages 269–274, May 2016.

[23] Razvan Diaconescu and Kokichi Futatsugi. *CafeOBJ report: The language, proof techniques, and methodologies for object-oriented algebraic specification*, volume 6. World Scientific, 1998.

[24] Antoni Diller. *Z: An introduction to formal methods*. John Wiley & Sons, Inc., 1990.

[25] Guillaume Dupont, Y Aït-Ameur, Marc Pantel, and Neeraj Kumar Singh. Handling refinement of continuous behaviors: A proof based approach with event-b. In *2019 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 9–16. IEEE, 2019.

[26] John Fitzgerald, Juan Bicarregui, Peter Gorm Larsen, and Jim Woodcock. Industrial deployment of formal methods: Trends and challenges. In *Industrial Deployment of System Engineering Methods*, pages 123–143. Springer, 2013.

[27] Gerhard Gentzen. Untersuchungen über das logische schließen. i. *Mathematische zeitschrift*, 39(1):176–210, 1935.

[28] Kurt Godel and George William Brown. *The consistency of the axiom of choice and of the generalized continuum-hypothesis with the axioms of set theory*. Princeton University Press Princeton, NJ, 1940.

[29] Joseph Goguen. *Theorem proving and algebra*. MIT, 1996.

[30] Joseph Goguen, Claude Kirchner, Hélène Kirchner, Aristide Mégrelis, José Meseguer, and Timothy Winkler. An introduction to obj 3. In S. Kaplan and J. P. Jouannaud, editors, *Conditional Term Rewriting Systems*, pages 258–263, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[31] Joseph Goguen, Kai Lin, and C Rosu. Circular coinductive rewriting. In *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, pages 123–131. IEEE, 2000.

[32] Michael J. C. Gordon, Robin Milner, L. Morris, Malcolm C. Newey, and Christopher P. Wadsworth. A metalanguage for interactive proof in LCF. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 119–130. ACM Press, 1978.

[33] Michael JC Gordon and Tom F Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[34] James Gosling, Bill Joy, Guy L Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. Addison-Wesley Professional, 2014.

[35] John V. Guttag and James J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1):27–52, 1978.

[36] Florian Haftmann. Haskell-style type classes with isabelle. Technical report, Isar. Technical report, Technische Universität München, 2014. http://www. cl ..., 2007.

[37] T Hardin, P Francois, W Pierre, and D Damien. Focalize: tutorial and reference manual, version 0.9. 1. *CNAM/INRIA/LIP6*, 2016.

[38] John Harrison. HOL light: A tutorial introduction. In Mandayam K. Srivas and Albert John Camilleri, editors, *Formal Methods in Computer-Aided Design, First International Conference, FMCAD '96, Palo Alto, California, USA, November 6-8, 1996, Proceedings*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer, 1996.

[39] John Harrison. A machine-checked theory of floating point arithmetic. In Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin, editors, *Theorem Proving in Higher Order Logics*, pages 113–130, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[40] John Harrison. Formal proof—theory and practice. *Notices of the AMS*, 55(11):1395–1406, 2008.

[41] John Harrison. Hol light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 60–66, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[42] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[43] Charles Antony Richard Hoare. Recursive data structures. *International Journal of Computer & Information Sciences*, 4(2):105–132, 1975.

[44] Warren A Hunt Jr, Matt Kaufmann, J Strother Moore, and Anna Slobodova. Industrial hardware and software verification with acl2. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104):20150399, 2017.

[45] Apple Inc. *The Swift Programming Language*. Apple Inc., 2019.

[46] Bart Jacobs and Tom Melham. Translating dependent type theory into higher order logic. In *International Conference on Typed Lambda Calculi and Applications*, pages 209–229. Springer, 1993.

[47] Michael Jastram and Prof Michael Butler. *Rodin User's Handbook: Covers Rodin v. 2.8*. CreateSpace Independent Publishing Platform, 2014.

[48] Cliff B Jones. *Systematic software development using VDM*, volume 2. Citeseer, 1990.

[49] Clifford B. Jones. *Systematic software development using VDM*. Prentice Hall International Series in Computer Science. Prentice Hall, 1986.

[50] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[51] Matt Kaufmann and J Strother Moore. *Design goals for ACL2*. Computational Logic, Incorporated, 1994.

[52] Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of z in isabelle/hol. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics*, pages 283–298, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[53] Alexander Krauss. Defining recursive functions in Isabelle/HOL. In *Proceedings of the Isabelle Workshop*, 2008.

[54] Peter Gorm Larsen, John Fitzgerald, and Tom Brookes. Applying formal specification in industry. *IEEE software*, 13(3):48–56, 1996.

[55] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[56] José Meseguer, Joseph A. Goguen, and Gert Smolka. Order-sorted unification. *Journal of Symbolic Computation*, 8(4):383–413, 1989.

[57] Christophe Métayer and Laurent Voisin. The event-b mathematical language. *Systerel, March*, 2009.

[58] Robin Milner. Logic for computable functions description of a machine implementation. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1972.

[59] Robin Milner. Models of lcf. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1973.

[60] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[61] Tobias Nipkow. Order-sorted polymorphism in isabelle. *Logical environments*, pages 164–188, 1993.

[62] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[63] Tobias Nipkow and Gregor Snelting. Type classes and overloading resolution via order-sorted unification. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, volume 523 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 1991.

[64] Sam Owre, John M Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.

[65] Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In Alan Bundy, editor, *Automated Deduction - CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26 - July 1, 1994, Proceedings*, volume 814 of *Lecture Notes in Computer Science*, pages 148–161. Springer, 1994.

[66] Lawrence C Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

[67] Benjamin C Pierce. *Advanced topics in types and programming languages*. MIT press, 2005.

[68] Dag Prawitz. *Natural deduction: A proof-theoretical study*. Courier Dover Publications, 2006.

[69] Ken Robinson. A concise summary of the event b mathematical toolkit. *Version April*, 21:1996–2009, 2009.

[70] John Rushby, Sam Owre, and Natarajan Shankar. Subtypes for specifications: Predicate subtyping in pvs. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.

[71] Matthias Schmalz. Term rewriting in logics of partial functions. In Shengchao Qin and Zongyan Qiu, editors, *Formal Methods and Software Engineering - 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings*, volume 6991 of *Lecture Notes in Computer Science*, pages 633–650. Springer, 2011.

[72] Matthias Schmalz. *Formalizing the logic of Event-B: Partial functions, definitional extensions, and automated theorem proving*. PhD thesis, ETH, 2012.

[73] Steve Schneider. *The B-method: An introduction*. Palgrave, 2001.

[74] Dana S Scott. A type-theoretical alternative to iswim, cuch, owhy. *Theoretical Computer Science*, 121(1-2):411–440, 1993. Circulated privately in 1969.

[75] Thoralf Skolem. Einige bemerkungen zur axiomatischen begründung der mengenlehre. In *̧Iteskolem:Swl*, pages 137–52. 1922.

[76] Konrad Slind and Michael Norrish. A brief overview of hol4. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 28–32, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[77] Colin Snook and Michael Butler. Uml-b: Formal modeling and design aided by uml. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):92–122, 2006.

[78] James Snook, Michael Butler, and Thai Son Hoang. Developing a new language to construct algebraic hierarchies for event-b. In *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, pages 135–141. Springer, 2018.

[79] Matthieu Sozeau and Nicolas Oury. First-class type classes. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2008.

[80] P. Stankaitis, G. Dupont, N. K. Singh, Y. Ait-Ameur, A. Iliasov, and A. Romanovsky. Modelling hybrid train speed controller using proof and refinement. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 107–113, 2019.

[81] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[82] Robert Roth Stoll. *Set theory and logic*. Courier Corporation, 1979.

[83] Bjarne Stroustrup. *The C++ programming language*. Pearson Education India, 2000.

[84] James W. Thatcher, Eric G. Wagner, and Jesse B. Wright. Data type specification: Parameterization and the power of specification techniques. In Richard J. Lipton, Walter A. Burkhard, Walter J. Savitch, Emily P. Friedman, and Alfred V. Aho, editors, *Proceedings of the 10th Annual ACM Symposium on Theory of Computing, May 1-3, 1978, San Diego, California, USA*, STOC '78, pages 119–132. ACM, 1978.

[85] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM, 1989.

[86] Markus Wenzel. Type classes and overloading in higher-order logic. In *International Conference on Theorem Proving in Higher Order Logics*, pages 307–322. Springer, 1997.

[87] F Wiedijk. The seventeen provers of the world. lncs (lnai), vol. 3600, 2006.

[88] Freek Wiedijk. The qed manifesto revisited. *Studies in Logic, Grammar and Rhetoric*, 10(23):121–133, 2007.

[89] J Wiegand et al. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.

[90] Toby Wilkinson, Michael Butler, Martin Paxton, and Xanthippe Waldron. A formal approach to multi-UAV route validation. In *Fourth International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS 2015)*, page 143.

[91] Ernst Zermelo. Untersuchungen über die grundlagen der mengenlehre. i. *Mathematische Annalen*, 65(2):261–281, 1908.