The Z/Eves theorem prover is simple and user friendly when compared with related tools like ProofPower-Z [Lem03] or Isabelle-HOL [NPW03]. The learning curve to carry out quite complex proofs with high level automation is quite linear. Nevertheless, the few available tactics are strict and difficult to extend. Also, it has particular Z idioms that ought to be followed. The documentation available is good but limited. The examples coming the tool are enlightening but insufficient.

The objective of these notes is to provide additional material and insight for theory developers and users. Still, careful reading of the reference manual [MS97], and few previous experience with the prover is desirable for a greater understanding of the ideas presented here.

A historical as well as technical overview of the prover structure in comparison with related Z theorem provers is firstly presented. Next, additional material is provided on naming conventions and the possibilities offered for the different sorts of declaration usages and abilities. After that, the prover approach on goal directed proofs is mentioned. Later, further documentation of available tactics with some examples and scenarios is presented. Despite the fact Z/Eves implements the Z standard [Toy00], it is very useful indeed to know the appropriate Z “idiom” and “accent” is fundamental for proper communication with the tool. For large specifications, optimisations are worthwhile and some techniques are given. A complete example on proof planning for inductive proofs over sequences comes next. Finally, a summary of naming conventions used is shown.

1 A Quick Look Inside

Z/Eves is equipped with very powerful tactics that enables high levels of automation quite often. However, to the extent of our knowledge, these are strict and difficult to extend. Despite having powerful tactics, there are a only a small amount when compared to other similar tools such as ProofPower-Z [Art00]. For instance, Z/Eves has a handful of tactics, whereas ProofPower-Z reaches over a thousand.

Different from other theorem provers for Z, the tool is not based on high order logic and the LCF paradigm [GM93]; perhaps this is one of its sources of inflexibility. Instead, it relies on a proof engine called EVES that was inspired by the Bledsoe-Bruell prover, the Stanford Pascal Verifier, the Boyer-Moore theorem prover, and the Affirm theorem prover, and the Affirm theorem prover [Can03].

In this sense, EVES would be related to ProofPower-HOL as the back-end In other words, as ProofPower-Z is HOL extended with the standard Z toolkit [Toy00], Z/Eves is EVES extended similarly. However not built for user extension, Z/Eves provides a socket protocol API [SM05] that allows one to interact directly with prover back-end. This in theory would enable one to create a front-end with more specialised tactics.

EVES is defined with Verdi, a underlying formal notation based on untyped set theory that provides the alternative to high-order languages. In fact, Z/Eves has an intermediate layer that transforms Verdi/EVES predicates into Z and vice-versa. This is sometimes source of odd situations during proofs because not all Verdi declarations can be translated to Z! For instance, when the command help NoVerdiDeclarations is issued on the prover textual interface, the following warning message is given: “translation of some Z constructs requires that Verdi declarations be sent to EVES. This cannot be done in the middle of a proof.”

Proof obligations are generated as assertions to be proved to demonstrate desired properties. This includes that Verdi code is in consonance with the assertions introduced. This is different from the database of lemmas of ProofPower given as compiled ML code.
The deduction system of EVES is called NEVER. It is capable of performing large proof steps automatically, yet allowing fine direction by the user. This enables powerful automated support with some user control whenever necessary. That possibility make Z/Eves more related to a proof checker still being a theorem prover. The fine control of transformations enables users to closely investigate proof strategies and pinpointing reasons for failure. The prover also issues domain checks proofs for axiomatic declarations in order to guarantee the consistency of specifications. ProofPower-Z offers a similar, but not obligatory facility called consistency checks.

Regardless the tool, proving theorems is hard. Tools can be used either to explore, or to prove ideas and properties. However, having an informal sketch or plan being conducted is crucial for effective use. After all, tools do not know the questions but can only help us to work out the answers!

There are two approaches for tackling proofs: guided or directed. On one hand, guided proofs usually relies on a high level of automation leaving the internal proof steps as hidden as possible. The greater interest is the property being proved instead of the oddities of the prover. The majority of proofs belong to this category. Guided proofs are usually small and quick, provided the related theory is stable and well equipped with an appropriate database of lemmas.

On the other hand, direct proofs are usually generic, fine grained, and lengthy. Fortunately, they represent a smaller amount of necessary proofs usually. They form the basis of a stable theory that enables higher degrees of automation. Despite representing the minority of proofs in terms of number, they represent the majority of time spent on proofs.

Z/Eves is well suited and very effective for guided proofs. Some of then are even solved with the click of a button with the powerful tactics available. This is indeed amazing at times when quite complex theorems are discharged easily and nicely. However, whenever the tool gets stuck, it can be quite painful to get around the problem. Due to the small amount and low level of control on tactics, direct proofs can be very hard. In summary, few tactics have their advantages: they are easy to use and get used, to remember, and to explain. Yet, deep understanding of each tactic behaviour is fundamental for fine grained control of the prover.

2 Interacting with Z/Eves

The tool has two main interfaces: graphic and textual. The GUI [Saa99b] is written in python and it provides a rather smooth experience for theorem proving in Z enabling the user to prove quite complex theorems with few mouse clicks. This interface encodes the Z specification in a XML-like file that can be browsed and type checked easily. Edition of paragraphs and proofs are performed on separate windows where all available Z symbols and tactic commands are given as buttons and/or pop-up menus. During the performance of proofs, theorems available for automated application also also appears as pop-up menus giving some hints on possible paths to follow (or avoid) during proofs. No information about internal message issued during proofs is given tough. It is possible to import L\LaTeX documents and exporting the XML-like file to either PS or RTF. In our experience, the free version of the GUI has some shortcomings: proprietary clipboard support, lack of an L\LaTeX exporting, inability to save the work done after an error or abortion, and inability to prove all paragraphs directly.

The textual interface allows both stand alone execution, as well as integration with emacs and direct API calls via socket connections. On emacs, it provides key-maps of commands allowing to user to perform both proofs, inspection, and maintenance commands whilst editing the related documents. The input/output is L\LaTeX and the amount of information given between proof steps is verbose.

Furthermore, the textual interface also provides two modes of operation: interactive and batch. In interactive mode, one can read, type check, and prove theorems and declarations from different files. All conclusions are kept in memory and can be accessed by a series of maintenance commands [MS97, Chapter 4]. In batch mode, a script with proof and maintenance steps is loaded, while proofs are being carried out on the background. Once completed with success (i.e. all theorems and domain checks proved), the prover creates a small verifying file that ensures the theory related to the source files read is checked and
proved. Whenever the source files are modified, these verification files become outdated. Inclusion of files through z-sections [Toy00] for declarations, and z-section-proofs for proof scripts is available whenever recursive inclusion is avoided. This separation of declaration and proof sections enables not only a great degree of modularity, but also a considerable amount of control and partition of responsibilities.

3 Goal Directed Proofs

Proofs in Z/Eves are meant to be goal oriented [GM93, Chapter 4]; forward proofs might be possible tough [GM93, Section 3.1]. Goal directed proofs starts with a (possibly empty) set of assumptions implying on the goals to be proved

\[ \text{Assumptions } \Rightarrow \text{Goals} \]

Transformations are usually conducted such that the goal is always simplified (i.e. shrunk), regardless the tactic being used.

In practice this not always happen. Often, transformations on the goals are made such that instead of expanding the goal, a contradiction at the assumptions is introduced. Therefore, the real goal to prove becomes the contradictory assumption introduced. For example, in the proof of

\[ \forall x : 1 \ldots 10 \cdot \exists y : \mathbb{N} \cdot y > x \]

quantifier elimination and predicate calculus reasoning automatically transforms the goal to

\[ x \in 1 \ldots 10 \Rightarrow \exists y : \mathbb{N} \cdot x \leq y \]

Further instantiation of \( y \), say to 11, would transform the formula via the one-point rule to

\[ x \in 1 \ldots 10 \Rightarrow (11 \in \mathbb{N} \wedge 11 > x) \vee (\exists y : \mathbb{N} \cdot y > x) \]

Because this would imply in a “more complex” goal for Z/Eves, it then applies propositional calculus reasoning introducing the contradictory assumption to the formula by transforming it to

\[ x \in 1 \ldots 10 \wedge \neg(11 \in \mathbb{N} \wedge 11 > x) \Rightarrow (\exists y : \mathbb{N} \cdot y > x) \]

Now the user can choose to prove either the implied goal, or the introduced contradiction on the assumptions. Transforming the contradictory assumption to false is the usual path to finish the proof. Generally the distributive law of implication over disjunction is used to transformed such goals into the appropriate form for the tool.

4 Declarations Abilities and Usages

Definitions and theorems in Z/Eves has enabling conditions that allows the prover to perform automatic tasks without user intervention. Nevertheless, the rationale of abilities values must be carefully chosen. This is important because the prover can take the wrong turn while transforming goals; this can cause struggle with simple proofs. Abilities are used to give some control to the user on the application of automatic transformations. The rationale of abilities can represent the difference between finishing a proof, or reaching a dead end within the same theorem. Different usages plays different roles within the available tactics. There are three usages and two abilities, they are given with roman fonts.

Declarations can be either enabled or disabled, where enabled is assumed as default when no ability is given. On one hand, enabled declarations allows the prover to automatically apply the related definition or theorem whilst performing a tactic. On the other hand, disabled declarations allows the user to carefully select where and when to apply local or global transformations.
Although some guidelines do exist for the proper selection of the different usages, it is difficult to predict the appropriate ability values beforehand. Experience with the theory and the prover is the answer. When deciding the abilities, one should focus the theory users, and the sort of theorems they are likely to prove. Even so, once in a while one might come back to change particular abilities at some points. However, bare in mind that this has a big chance to interfere on direct proofs; guided proofs are less likely to be affected tough.

The available usages are called rules. There are rewriting, assumption, and forward rules. They are used for different sorts of automatic transformations. Furthermore, there are the normal theorems and axioms detailed next.

4.1 Rewriting Rules

Rewriting rules are mostly common and intuitive. They affect transformation tactics whenever there is a pattern match on the formulae. When enabled, rewriting rules are applied automatically and can either finish the goal, or generate some side condition type checks. When disabled, they can be applied point-wise by the user both locally or globally on the current theorem.

An important aspect to bare in mind while thinking about how to declare an automation rule is the way to configure the application conditions. For example, the toolkit theorem seqSize0 is given as

\begin{verbatim}
theorem rule seqSize0[X]
  \forall s : seq X • #s = 0 ⇔ s = ⟨⟩
\end{verbatim}

Whenever is is applied, the prover automatically rewrites any occurrence of

#s = 0

to the conclusion that the sequence is empty

s = ⟨⟩

Finally, even though this is an equivalence, sub-formulae matching whilst transforming the goal occurs left-to-right. For instance, in the proof of a theorem such as the size of a nonempty sequence being strictly positive,

\begin{verbatim}
\forall s : seq1 X • #s ≥ 1
\end{verbatim}

the formula is transformed (see proof script of theorem seq1SPos) after four steps to

\begin{verbatim}
(s ∈ seq X ∧ ¬s = ⟨⟩ ∧ #[Z × X]s ≥ 0) ⇒ (#[Z × X]s ≥ 1)
\end{verbatim}

Unfortunately, since the condition of seqSize0 does not match any sub-formulae. Although being an enabled rewriting rule, it cannot be applied automatically or through pop-up menus, but manually. Therefore, automation in this case got compromised by the order of declarations of theorems.

One can argue that a possible solution for a better automation in this particular case would be to redeclare the theorem as

\begin{verbatim}
theorem rule seqSize0[X]
  \forall s : seq X • s = ⟨⟩ ⇔ #s = 0
\end{verbatim}

Although this would solve our problem, it is difficult to predict the effects it might have on proofs elsewhere. Thus, the point of concern is: an wise choice of application conditions while introducing rewriting rules represents an important issue on the level of automation achieved.
Another option would be to introduce both versions. Even tough possible, careful must be taken in order to avoid unpredictable side effects like infinite loops during rewriting. In conclusion, the guideline to choose is to introduce rewriting rules where the side condition is the predicate that might appear more often. This is highly dependent on the sketch of proofs one wishes to perform. In order to keep desirable levels of automation, the bottom line of this situation can be a redesign of the proof plan that is suitable to the conditions of available rewriting rules.

The user can declare axiomatic definitions or theorems as rewriting rules by tagging then with the rule keyword. Nevertheless, some syntactic restrictions apply [MS97, Chapter 3]. The difference between a normal theorem and a rule is that the later is given as a tautology. Theorems given as tautologies is a clever representation. They explore the predicate calculus reasoning and tautology checking abilities on lower layers of the prover. For example, theorem

$$\forall x : T \land P(x)$$

is encoded as

$$x \in T \Rightarrow P(x)$$

However, if given as a rule, it automatically becomes

$$(x \in T \Rightarrow P(x)) \Leftrightarrow \text{true}$$

Declaration of rules as tautologies enables automatic application via the tautology reasoning facet of the prover.

4.2 Housekeeping Rules

Assumption and forward rules are related to “housekeeping” and are usually always enabled. Basically, they provide machinery to increase the level of automation. In contrast to the intuitive nature of rewriting rules, these rules are somewhat awkward in their format. The user needs to understand very clearly the Z idiom Z/Eves expects in order to make they play an effective role. This is usually learned through careful observation of the transformations between proof steps.

Assumption rules can only be given as theorems; they are tagged with grule. They are important for automatic discharge of side conditions and type checks generated by the application of other theorems and declarations. They are particular useful when complex data types are involved.

A simple example is given for the proof of a theorem involving sequences. On such case, one might encounter type checks declared as conditional expression, such as

$$s \in \text{seq } T \land (\text{if } Z (s \in \text{N } \Rightarrow T) \text{ then } R \text{ else } S) \Rightarrow P$$

where $R$ is regarded as a clear contradiction that would finish the proof. Therefore, one needs only to prove that

$$s \in \text{N } \Rightarrow T$$

An assumption rule suitable for this is given as

```
theorem grule gSeqIsPFun
  \forall s : \text{seq } T \land s \in \text{N } \Rightarrow T
```

This is indeed a theorem since the new type contains the original. During application of transformation tactics, Z/Eves pops out type checks related to the expression being transformed. These type transformations goes up to the last enabled rewriting rule associated to definitions of the expression being transformed. Therefore, the ability of involved theorems usually affects the “symbol” to choose (i.e. $\Rightarrow$,
In this case, the assumption rule was given according to the necessity that shown up during previous proofs involving the expression.

The proof of this assumption rule could certainly be done during the proof of the related theorem involving $R$ and $P$. However, this plays against modularity. Lets assume that this type check on sequences is a common scenario on our proofs. Giving an assumption rule provides smaller self-contained proof scripts.

For example, lets instantiate the type $T$ of the sequence above to the nonempty set of natural numbers $\mathbb{P}_1 \mathbb{N}$.

The general assumption rule to introduce is the one related to the maximal type of $s$

$$\text{theorem grule gSeqType}$$
\[ \forall s : \text{seq}(\mathbb{P}_1 \mathbb{N}) \bullet s \in \mathbb{P}(\mathbb{Z} \times \mathbb{P}\mathbb{Z}) \]

Nevertheless, under particular circumstances pointed out during proofs, it might be necessary to have a stronger (non-maximal) type on the goal. Lets say that during proofs involving such sequence, $Z/Eves$ came up with type checks related to partial functions. In this case, the appropriate assumption rule to be introduced is

$$\text{theorem grule gSeqPFunType}$$
\[ \forall s : \text{seq}(\mathbb{P}_1 \mathbb{N}) \bullet s \in \mathbb{N} \rightarrow \mathbb{P}_1 \mathbb{Z} \]

The effects are as subtle as the differences. The $\text{gSeqPFunType}$ option enables simplifications to take place automatically for a minor number of definitions since it is less generic, whereas the $\text{gSeqType}$ option enables a greater number of automatic transformations. One does not always exclude the other, however. In this particular scenario, it does not make bring much advantages because sequences are well-equipped with powerful assumption rules for simple types. They were given here for the sake of our illustration. Throughout the definitions of our theory however, such trick was used for more effective purposes.

Forward rules has a similar role as assumption rules: increase the degree of automation by smoothly discharging type checks. These rules are usually introduced to inform the prover about the types of schema components. Therefore, they are often used whenever theorems involving schemas are necessary. For instance, during refinement simulation proofs.

This is better illustrated with an example. For the given schema $Test$ below

$$Test$$
\[ x : \text{seq}(\mathbb{P}_1 \mathbb{N}) \]
\[ y : \text{seq}(\mathbb{P}_1 \mathbb{N}) \rightarrow \mathbb{P}(\text{seq}_1 \mathbb{N}) \]
\[ \text{dom } y = x \]

It might be useful to declare the forward rules about the maximal types of the schema components, as well as certain facts of its predicate part. Therefore, some possible forward rules could be

$$\text{theorem frule fTestXType}$$
\[ \forall Test \bullet x \in \mathbb{P}(\mathbb{Z} \times (\mathbb{P}\mathbb{Z})) \]

$$\text{theorem frule fTestXRelType}$$
\[ \forall Test \bullet x \in \mathbb{Z} \leftrightarrow (\mathbb{P}\mathbb{Z}) \]

$$\text{theorem frule fTestYType}$$
\[ \forall Test \bullet y \in \mathbb{P}(\mathbb{P}(\mathbb{Z} \times (\mathbb{P}\mathbb{Z})) \times \mathbb{P}(\mathbb{Z} \times \mathbb{Z})) \]

$$\text{theorem frule fTestYDomInvariant}$$
\[ \forall Test \bullet \text{dom } y = x \]

Again the same rationale used for assumption rules applies: the type on forward rules may be given according to the most common type checks returned by the prover or just the maximal type.
4.3 Theorems

Finally, there is the category of theorems for situations where either automation is not necessary/desirable, or syntactic issues restricts the declaration of rules. The syntactic restrictions are well documented and the returning error messages are quite helpful.

For instance, normal theorems not to be involved on automation schemes are often desired. The rule of thumb is: theorems that are not applied often should not be considered as rewriting rules. Bare in mind that although a high amount of rewriting rules provides higher levels of automation, the side effect is the degradation of performance. In other words, the bigger the amount of rewriting rules available, the higher the time necessary to check whether their conditional sub-formulae are satisfied.

Another reason for normal theorems arises when the tautology elimination engine automatically removes recently applied rewriting rules during transformation tactics. Therefore, the important information just introduced is removed from the assumptions before being used. Fortunately, this undesired scenario does not happen frequently.

Finally, there is a useful trick regarding rewriting rules involving generic types. There is a specific idiom to be followed in Z/Eves in order to use generic types properly (more details about this is given in Section 7.4). Basically, to allow the prover to infer generic actuals, maximal types must be used. Therefore, whenever one have a rewriting rule with generic types, the generic actuals given must be maximal in order to avoid problems.

Unfortunately, using maximal types as generic actuals is not always possible. For example, whenever the rewriting rule assumptions or conclusions mention expressions with explicit reference to non-maximal types. For instance, a theorem about the result of partial function application could be given as

\[
\text{theorem rule applyInRanPfun[A,B]} \\
\forall f : A \rightarrow B \cdot \forall a : \text{dom } f \cdot f(a) \in \text{ran } f \land f(a) \in B
\]

Usually, one have a function \( f \) on non-maximal types \( A \) and \( B \), say \( \text{seq } \mathbb{N} \) and \( \mathbb{N}_1 \) respectively. In cases where information about strictly positive numbers is needed, the generic actual for the type \( B \) explicitly mentioned on the conclusions needs to be \( \mathbb{N}_1 \) instead of \( \mathbb{Z} \). Under these circumstances the problems with generic actuals around expressions arise. Therefore, this is a definition that do not communicate with Z/Eves in the appropriate idiom.

The solution for this situation is hinted in some theorems of the toolkit. The trick is to universally quantify the generic non-maximal types of interest (in the case \( B \)) as power sets of the correspondent maximal type. Then, the actual theorem can come next with the quantified types. Since the original generic types are quantified on the corresponding maximal types no harm is introduced.

However, one shortcoming of this trick is that it usually introduces the kind of syntactic restrictions that forbids the theorem to be a rewriting rule. Hence, it must be given as a normal theorem and automation gets compromised. One might argue that it also increases the complexity of the theorem usage due to explicit necessity of instantiations. The important point is that the solution completely removes problems with generic actuals on the expression of assumptions or conclusions. For instance, the toolkit theorem \( \text{applyInRanPfun} \) is the correct version of our running example

\[
\text{theorem applyInRanPfun[X,Y]} \\
\forall A : \text{P} X; \ B : \text{P} Y \cdot \\
\forall f : A \rightarrow B \cdot \\
\forall a : \text{dom } f \cdot f(a) \in \text{ran } f \land f(a) \in B
\]

The command for appropriate use of the theorem in our given scenario is

\[
\text{use applyInRanPfun}[\text{P}(\mathbb{Z} \times \mathbb{Z})), (\mathbb{Z})] \\
[A := \text{seq } \mathbb{N}, B := \mathbb{N}_1, f := \text{myFcn}, a := \text{myVal}]\]

Note the that the generic actuals \( X \) and \( Y \) are instantiated to the maximal types, whereas the original generic actuals \( A \) and \( B \) are instantiated as desired with non-maximal types.
4.4 Axioms

An axiom is a theorem believed to be true without the need of proof. Internally, it is conceivable that \texttt{Z/Eves} transforms (unlabelled) axiomatic, generic, and schema boxes into axioms. The user can also explicitly declare an axiom through the usage axiom.

During the development of a theory, one might be interested to introduce an well-known theorem as an axiom because there is no available machinery to prove it. For example, if one wants to include theorems about Pascal’s triangle, the proofs would involve binomial coefficient and simplification of arithmetic expressions. Some of these functionalities are not available, and might be hard to implement on the prover front-end. Therefore, one might simply introduce these theorems as axioms assuming they have already been proved elsewhere.

5 Naming Conventions Preliminaries

In \texttt{Z/Eves} labels of declarations and theorems must be unique. Having some structure on the label given to paragraphs is helpful for learning and indexing. Thus, we introduce some naming conventions inspired from [Art00].

Theorems headers are declared using bold roman fonts in \texttt{Z/Eves}. Rewriting rules are prefixed with a \texttt{r}, assumption rules with a \texttt{g} (for \texttt{grule}), and forward rules with an \texttt{f}. Then it comes the theorem or declaration name capitalised on important parts. Underscores are allowed but avoided due to a bug on the \texttt{EPiX} importer of the GUI.

Assumption and forward rules are usually related to theorems for type checks or function results. Hence, we suffix them with \texttt{Type} and \texttt{Result} respectively; other suffixes are also applied as needed. Whenever specialised (non-maximal) types are necessary as rules, we give a hint for the choice just before the suffix. So, from the forward rules example, the label of the theorem with relational type for the $x$ component of schema \texttt{Test} is given as \texttt{fTestXRelType}

We follow the naming convention to append a prefix \texttt{d} for declarations (i.e. axiomatic and generic boxes), \texttt{t} for theorems, \texttt{l} for lemma, and \texttt{a} for axioms. Further naming conventions are introduced as needed.

6 Proof Commands

There are a number of commands available for both proof and management purposes. In \texttt{Z/Eves} commands are given with sans-serif font. We call proof commands tactics. For example, it is possible to \texttt{reset} the entire theory, \texttt{read} and \texttt{undo} declarations to and from the theory, \texttt{retry} or move backwards on a proof.

The theory management commands are simple and no further comment is provided. Instead, we concentrate on the commands that actually transforms the goal in some sense. Understanding the behaviour of those is crucial for an effective use of the automation power available. The tool documentation provides complementary explanations on all proof commands.

6.1 Basic Tactics

Lets start with application of definitions or theorems on the current proof. The \texttt{apply} command is available for any rewriting rule regardless the ability. The tactic is very powerful and user friendly because it makes all necessary instantiations properly (i.e. quantified variables and generic actuals with maximal types). It also works together with the tautology reasoning facility of \texttt{Z/Eves} giving rise to much simpler goals after transformations. Because it can be used either globally, or restricted to a particular expression or predicate, the tactic increases in a great extent both the degree of control sfor the user, as well as the automation level of the prover. Throughout proof scripts in our theory, unless otherwise necessary, global applications were preferred for readability.
The alternative is via the use command where instantiations must be made explicitly. It introduces on the assumptions the theorem or declaration being used, and is more likely to appear on direct proofs or particular cases of guided proofs. The tactic is preferred instead of apply whenever the rewriting rule needs to be applied with a particular instantiation not present on the sub-formulae. Since it just includes conclusions into the assumptions, it is not possible to restrict the application to a particular expression or predicate. For large proofs this can imply in performance penalties due to rewriting on useless assumptions by further tactics.

There are another three very important basic tactics: rearrangement of predicates, and equality and quantifier elimination. The rearrange command imposes an ordering on the formulae where those considered “simpler” by the prover appears first. As far as we know, the criteria for this ordering is not documented anywhere. The ordering criterion given by the complexity of expressions seems to respect the categories of: (i) simple formulae, (ii) equalities, (iii) inequalities, (iv) compound formulae, (v) quantifiers, (vi) logical implication and equivalence, and (vii) negations. Formulae of the same criterion are given in ascending alphabetic order. More precisely,

1. Undecorated type formulae (i.e. \( x \in T \)).
2. Undecorated equalities (i.e. \( x = P \)).
3. Undecorated inequalities (i.e. \( x \leq y \)).
4. Decorated versions of the three previous criteria.
5. Compound expressions follow the component binding powers.
6. Existential and universal quantifiers with their parts following these criteria.
7. Logical implication and equivalence.
8. Negations on any of the previous criteria in ascending order.

Nearly at all times, the arrangement ordering criterion is sufficient and few worries are needed. It is suggested by the documentation to rearrange formulae as often as possible; our experience shows some exceptions mainly when one needs to optimise proofs. The point is that rearranging predicates can severely affect the effectiveness of transformation tactics. It seems that these tactics executes whilst performing a single-parsing through the formulae. Therefore, whenever a simpler sub-formula comes first, the prover is usually enabled to reach more clever conclusions for latter complex sub-formulae. The rule of thumb is: unless otherwise necessary, the sooner rearrange is issue the better.

The equality substitute command substitutes any occurrence of the right-hand side of an equality whenever the left-hand side appear later on in the sub-formulae. However, careful should be taken on the formulae ordering. For example, the tactic applied to

\[
\begin{align*}
  s \in \text{seq } \mathbb{N} \land f(x) &= \{\} \land x = (((y \cup z) \cap w) \setminus s) \Rightarrow f(y \cup x) = \{\}
\end{align*}
\]

leads to

\[
\begin{align*}
  s \in \text{seq } \mathbb{N} \land f(x) &= \{\} \land x = (((y \cup z) \cap w) \setminus s) \Rightarrow \\
  f(y \cup (((y \cup z) \cap w) \setminus s)) &= \{\}
\end{align*}
\]

because the assumption about \( f \) appears before the equality.

Like the apply command, it is possible to restrict the equality locally via an expression to substitute. This also allows one to substitute the left-hand side from the right-hand side. On the given example, if the equality on \( x \) was given as

\[
(((y \cup z) \cap w) \setminus s) = x
\]

and one wanted the same results, the command

\[
\text{equality substitute } x ;
\]
would be necessary instead. Finally, global equality substitution can sometimes be used for very simple transformation with predicate calculus reasoning such as as negation of implication, equivalence, or quantifiers. For example, after applying equality substitute on formulas such as

1. \( \lnot (P \Rightarrow Q) \)
2. \( \forall x : T \cdot P(x) \cdot \text{false} \)
3. \( \lnot (\forall x : T \cdot P(x)) \)

the prover returns

1. \( P \land \lnot Q \)
2. \( \exists x : T \cdot P(x) \)
3. \( \exists x : T \cdot \lnot P(x) \)

In proofs of contradictory assumptions involving implications, or instantiation of quantifiers, this is very handy.

The final basic tactics are related to quantifiers elimination. There are two forms depending whether it appears at the goal or assumptions. If one allows us to forget about the details of quantifier manipulations of predicate calculus, the rule of thumb for quantifier elimination is:

- **prenex** removes \( \exists \) on the antecedent, and \( \forall \) on the consequent.
- **instantiate** removes \( \exists \) on the consequent, and \( \forall \) on the antecedent.

In practice, instantiation do not eliminate quantifiers, but introduces new conclusions for \( \forall \), and contradictory assumptions for \( \exists \). On one hand, if one instantiates the variable \( x \) below to \( a \)

\[
a \in \mathbb{N} \land b \in \mathbb{Z} \land (\forall x : \mathbb{N} \cdot x > b) \Rightarrow a > b
\]

the prover gives

\[
a \in \mathbb{N} \land b \in \mathbb{Z} \land (\forall x : \mathbb{N} \cdot x > b) \land (a \in \mathbb{N} \land a > b) \Rightarrow a > b
\]

On the other hand, if one instantiates the variable \( y \) below to \(-1\)

\[
c \in \mathbb{N} \Rightarrow \exists y : \mathbb{Z} \cdot y < c
\]

then the prover generates

\[
c \in \mathbb{N} \Rightarrow (\lnot 1 \in \mathbb{Z} \land -1 < c) \lor (\exists y : \mathbb{Z} \cdot y < c)
\]

automatically simplifying it to

\[
c \in \mathbb{N} \land \lnot (-1 \in \mathbb{Z} \land -1 < c) \Rightarrow (\exists y : \mathbb{Z} \cdot y < c)
\]

Instantiation of multiple variables is also possible separated with commas. In fact, it is highly recommended that all quantified variables are instantiations in order to avoid translation problems between Verdi and \( \mathbb{Z}/Eves \).

### 6.2 Transformation Tactics

There are three main transformation techniques: simplification, rewriting, and reduction. The simplifications performed by the `simplify` command are equality, integer, and predicate calculus reasoning (e.g., one-point rule), together with tautology checking. Simplification is affected by grules and frules whenever their hypothesis matches a sub-formula. The conclusion of these lemmas are then included as assumptions. Simplification offers the user the opportunity to perform direct proofs because it allows the smallest
of the transformations, hence the best performance. However, during some domain checks and complex proofs simplifications not enough and rewriting or reductions are strictly necessary.

Rewriting transformations are given by the rewrite command. It performs simplifications together with automatic application of enabled rewriting rules, that matches any sub-formula. Set containment is also transformed, such that

\[ e \in \{ x : T \mid x \subseteq f(x) \} \]

is rewritten as

\[ e \in T \land e \subseteq f(e) \]

During reduction, the invoke command is called systematically in order to expand the goal as much as possible prior to any rewriting application. For instance, invoke is necessary to expand schema declarations or data types declared as disabled, such as partial functions or injections. These expansions can be applied globally or locally to a predicate.

Reduction is the most complex transformation scheme and is given by the reduce command. It performs rewriting together with further clever, but simple deduction schemes. This leads to the biggest step on the transformation of formulae with the worst performance. In fact reduction is more than simply expansion together with rewriting. It recursively performs these activities until the formula stops changing. Also, conditional rewriting rules not applied through normal rewriting are used here if their conditions could be reduced as well.

The tool offers variations of these tactics. The trivial simplify command limits the simplification by ignoring equality and integer reasoning, not applying the one-point rule, and not using neither assumptions, nor forward rules. The trivial rewrite command is equivalent to the application of unconditional rewriting rules together with assumption and forward rules, but without simplification and some predicate calculus reasoning like one-point rule. Trivial reduction is not available.

Trivial tactics are rarely needed. Nonetheless, they allow functionalities not available elsewhere, mainly trivial rewrite. For example, in some proofs where the toolkit definition of cardinality (\# — see [Saa99a, Section 12.7]) is needed, it is only possible to proof goals by starting the first transformation with trivial rewrite. If one tries to use any other tactic, Z/Eves reaches a dead end where it is not enabled to translate Verdi declarations back to Z. Therefore, the entire formula becomes the unpleasant message “An unprintable predicate.”

Normalisation is available for all three major transformation techniques. It is useful whenever some sort of “fiddling” is needed during proofs of goals involving disjunctions, or nested conditional expressions. In Z/Eves, every logical connective is represented as conditional expressions. For example, a predicate like

\[ a \land b \]

is internally represented as the conditional expression

\[ \text{if}_{Z} a \text{ then } b \text{ else false} \]

Whenever the conditional test is itself a conditional

\[ \text{if}_{Z} (\text{if}_{Z} a \text{ then } b \text{ else } c) \text{ then } d \text{ else } E \]

Z/Eves is capable of pushing the inner conditional into the first “arm” of the outer conditional

\[ \text{if}_{Z} a \text{ then } (\text{if}_{Z} b \text{ then } d \text{ else } e) \text{ else } (\text{if}_{Z} c \text{ then } d \text{ else } e) \]

Because normalisation actually expands the goal, it works against the principle that the goal must be shrunk during transformations. Therefore, it is left disabled by default. Indeed, normalisation could be simulated with the related transformation tactic together with stacked cases analysis. Nevertheless, for relatively large goals, normalisation can cut down the proof enormously taking the most of automatic reasoning and case splitting.
6.3 Case Splitting Tactics

There are three main forms of case splitting: stacking of available goals, transformation of some logical connectives, and specialised by a side condition predicate.

The *cases* command enables one to stack compound goals and prove then separately. Recursive splitting is also possible by multiple case splitting. It is available in two situation: (i) when a conjunction is present on the goal, or (ii) when the entire formula is a conditional expression. The *next* command enables one to travel across the goal stack forwardly only. One can travel around the cases even without proving then. However, whenever last case is reached, a further *next* rebuilds the entire goal again as the conjunction of the remaining cases (i.e. cases not yet *true*). Stacked case analysis finishes when one reaches the predicate *true* after issuing *next*. Obviously, stacked case analysis can start over again. This strategy of stepping over cases is useful when one wishes to prove particular aspects of the goal first.

Like equality substitution, case splitting by logical connectives also performs some little simplifications on the formulae. It substitutes conditional expressions, logical implication, and equivalence for their corresponding versions with conjunctions or disjunctions. For example,

\[ P \Rightarrow Q \]

is simplified to

\[ (\neg P) \lor Q \]

The scope on negations on quantifications is also minimised pushing it inside. For instance,

\[ \neg (\forall D \bullet P) \]

becomes

\[ (\exists D \bullet \neg P) \]

At last, one can choose either to distribute \( \lor \) across \( \land \) with the *conjunctive* command, or \( \land \) across \( \lor \) with the *disjunctive* command. The application of any transformation tactic afterwards transforms the formula back to the default form

\[ Assumptions \Rightarrow Goals \]

Another form of case splitting is available through the *split P* command. It allows one to split on whichever side condition predicate \( P \) one find necessary. It transforms a goal

\[ G \]

into

\[ if_{Z} \ P \ then \ G \ else \ G \]

Usually, *split P* is followed by *cases*, such that two goals are stacked to be proved

1. \( P \Rightarrow G \)
2. \( \neg P \Rightarrow G \)

For example, in a theorem like

\[ \forall x : \mathbb{N} \bullet x \in 0 \ldots 10 \leftrightarrow x \in 0 \ldots 10 \]

The *split P* command can also be used to perform stacked case analysis from the assumptions when disjunctions are present. For example, a theorem such as

\[ P \land (Q \lor R) \Rightarrow S \]
can be proved by splitting on either $Q$ or $R$ followed by case analysis. Let’s say we choose split $Q$. This transforms the original formula to

$$\text{if}_{Z} Q \text{ then } (P \land (Q \lor R) \Rightarrow S) \text{ else } (P \land (Q \lor R) \Rightarrow S)$$

With further cases, this leads to the two goals being stacked

1. $Q \land P \Rightarrow S$
2. $\neg Q \land P \land R \Rightarrow S$

Finally, another clever way of using split $P$ is to conduct user controlled rearrangements. Quite few times during direct proofs one reaches situations where the available rearrangement ordering is not adequate. That happens when equalities that are considered more complex sub-formulae are rearranged towards the end needed to be substituted on the entire formula. In order to do so, it is necessary to have these equalities at the very top of the formula.

For example, on a theorem like

$$s \in \text{seq } \mathbb{N} \land f(x) = \{\} \land (((y \cup z) \cap w) \setminus s) = x \Rightarrow f(y \cup x) = \{\}$$

If one wants to substitute the value of $x$ on all applications of $f$, the following proof commands are needed

- `split (((((y \cup z) \cap w) \setminus s) = x);`
- `simplify ;`
- `equality substitute x ;`

The first step creates the case analysis by introducing the conditional expression

$$\text{if}_{Z} (((y \cup z) \cap w) \setminus s) = x \text{ then}
\begin{align*}
s & \in \text{seq } \mathbb{N} \land f(x) = \{\} \land (((y \cup z) \cap w) \setminus s) = x \Rightarrow f(y \cup x) = \{\} \\
\text{else } s & \in \text{seq } \mathbb{N} \land f(x) = \{\} \land (((y \cup z) \cap w) \setminus s) = x \Rightarrow f(y \cup x) = \{\}
\end{align*}$$

Simplification automatically eliminates the second (contradictory) branch, leading the formula to

$$\begin{align*}
\left(((y \cup z) \cap w) \setminus s) = x \land
\begin{align*}
s \in \text{seq } \mathbb{N} \land f(x) = \{\} \land (((y \cup z) \cap w) \setminus s) = x \Rightarrow f(y \cup x) = \{\}
\end{align*}
\right)
\Rightarrow (f(y \cup x) = \{\})
\end{align*}$$

At last, the proper equality substitution of $x$ can take place, and the goal is transformed to

$$\begin{align*}
\left(((y \cup z) \cap w) \setminus s) = x \land
\begin{align*}
s \in \text{seq } \mathbb{N} \land f(((((y \cup z) \cap w) \setminus s)) = \{\} \land (((y \cup z) \cap w) \setminus s) = (((((y \cup z) \cap w) \setminus s)) = \{\}
\end{align*}
\right)
\Rightarrow (f(y \cup (((y \cup z) \cap w) \setminus s)) = \{\})
\end{align*}$$

Note the necessity of the explicit expression $x$ on the local equality substitution. It is needed because $x$ appears on the right-hand side of the equality.

### 6.4 Advanced Tactics

**Z/Eves** provides tactics that can offer a great experience for theorem prover users in terms of automation, flexibility and control. Single commands as tactic combinations\(^1\) provides the highest degrees of automation. Localised exchange of global abilities allows great amount of flexibility. Finally, localised application of transformation tactics enables a fine control also amount for greater performance by avoiding useless rewriting.

\(^{1}\) In LCF paradigm these are known as tacticals.
6.4.1 Automation — Tactic Combinations

Tactic combinations are very powerful discharging even complex theorems with one step in less than a second. However, they work properly only when an underlying theory has enough housekeeping theorems, together with clever rules and appropriate abilities.

There are two commands that implicitly combine tactics: They are \texttt{prove by reduce} and \texttt{prove by rewrite}. The latter can also be written as \texttt{prove}. They repeatedly apply tactics on the formulae until no effect is observed. Based on the verbose output from the prover on the textual interface and the documentation, these tactics could be encoded with something like

\begin{verbatim}
while (changing) do{invoke ;rearrange ;rearrange ;equality substitute ;rearrange ;rewrite / reduce ;}
\end{verbatim}

Note that the rearranging of predicates is extremely important for the effectiveness of any other tactics.

However, due to the possible higher amount of unnecessary steps and transformations, these two tactics require a more time to finish. Therefore, they are not appropriate for more controlled proofs.

6.4.2 Flexibility — Abilities Exchange

There are two tactics involving global exchange of abilities. They allow the use of any transformation tactics — including the combinations — with a given set of labels where abilities have been globally exchanged to the desired value. For example, in the proof of the theorem with generic type \(X\):

\[
\forall A : P \ X \bullet \forall B : P \ A \bullet A \ \setminus \ B = \{\} \Leftrightarrow B = A
\]

we were interested on having the disabled toolkit rule enabled for automatic trivial rewriting with the command

\begin{verbatim}
with enabled (diffSuperset) trivial rewrite ;
\end{verbatim}

This increases the level of automation in the rewrite of predicates. Obviously, one could have achieved a similar effect through steps like

\begin{verbatim}
use diffSuperset[X][T := A, S := B] ;rearrange ;trivial rewrite ;
\end{verbatim}

The use of \texttt{trivial rewrite} is just to illustrate that the transformation trivial and normal variations are also possible.

6.4.3 Control — Localised Transformations

The final advanced tactics allow the application of all transformation commands, including the trivial, normalised, and combination variations, on local predicates or expressions. These are indeed very specialised allowing precise application. The price for precision and performance is paid with poor readability. They are particularly useful during proof optimisations where either global rewriting is time consuming or undesirable. Moreover, they are also useful whenever faster transformation tactics such as the ones involving \texttt{simplify} are not effective.

7 Adequate Z “Accent” for \texttt{Z/Eves}

Although \texttt{Z/Eves} has been built following the Z standard, the tool is not happy with some constructs. In other words, even tough most Z constructs are available on the tool, they ought to be given with a proper “accent”. In this section we try to point out some major aspects.

7.1 Specification and Proof Comments

Even tough not mentioned on the Z standard, \texttt{Z/Eves} allows a limited way of commenting specification and proofs through a special latex markup. This is achieved because the tool simply ignores any text inside the (user defined) \texttt{\LaTeX} command \texttt{znote}. In other words, by regarding a particular \texttt{\LaTeX} markup while parsing documents it enables the comment to be printed.

In practice one need to create the \texttt{\LaTeX} command as
Then, one could add a comment with a command like

\znote{My Comment}

provided that no other commands, environments, or new lines are included.

7.2 Universal Quantification

When declaring a paragraph, care must be taken with the variables one choose to universally quantify. Besides the intuition of quantifying all necessary values, it is sometimes useful to avoid some keeping them free when possible.

This is useful when one wants to avoid type checks to be introduced for the quantified variable or generic type. For example, if one wants to extract the pair of an known element \( e \) on a sequence \( s \), one needs the theorem

\[
\forall s : \text{seq } X \mid e : X \mid e \in s \bullet \exists x : 1..\#s; y : \text{ran } s \bullet e = (x, y)
\]

However, type checks on \( e \in X \) would represent a waste of time on the proof itself and could be avoided by leaving \( e \) free.

\[
\forall s : \text{seq } X \mid e \in s \bullet \exists x : 1..\#s; y : \text{ran } s \bullet e = (x, y)
\]

As another example, in the often used toolkit theorem \textit{tupleInCross2}, both values and generic types are left free. This simplifies the application of rewriting tactics by avoiding type checks on the free variables.

Indeed a wise use of quantified variables and generic type can make the difference on the possible levels of automation to be achieved. Throughout the toolkit this trick is used to help the prover (or the user) to choose appropriately according to the situation. For instance, the definition of set intersection is given in both flavours: with and without free variable (see theorem \textit{inCap} and [Saa99a, \textit{capDefinition} on pp.20] respectively). The quantified version is left as a disabled normal theorem, whereas the free variable version is given as an enabled rewriting rule. This decision of ability, usage, and quantification affects the capability of the prover to automatically discharge goals related to set intersection. Such care is worth taken within declarations that appear often on expressions of subsequent proofs.

7.3 Existential Quantification

Whenever possible, try to avoid existential quantification on axiomatic and generic definitions, or theorem declarations. Try to work out an equivalent version without existential quantification if feasible. This is encouraged because existential quantification is part of the syntactic restrictions on rules imposed by underlying proof engine (EVES). For example, in the theorem below

\textbf{theorem nonEmptySetHasElement[X]}

\[
\forall S : \mathbb{P}_1 (\mathbb{P} X) \bullet \exists P : \mathbb{P}_1 (\mathbb{P} X) \bullet P \subseteq S
\]

the need of existential quantification does not allow it to be a rule. Therefore, existential quantifiers on declarations can increase the complexity of using them due to the necessity of explicit instantiation of generic actuals and quantified variables.
7.4 Generic Types

The presence of generic types in Z allows higher degrees of abstraction making specifications simpler, smaller, and re usable. However, it also introduces quite awkward unexpected problems if not used carefully. The guideline is to use it only when really necessary. For instance, an easy way to find out if generic types are really needed is trying to declare them using given sets.

Another simple guideline is to avoid using definitions and theorems with ≠, ∉, and ∅. This occurs because their definition in Z/Eves is given with generic actuals being expected upon use, and the tool is not always able to infer them. Obviously these definitions appear quite often in the simplest specifications. Hence, caring generic actuals around expressions would be cumbersome and against readability. Therefore, we adapt the simple practice of using the negation prefix operator ¬, and the pair of braces { } for the empty set instead. For example, instead of declaring

\[ b \in y \land a \not\in y \Rightarrow y \neq \emptyset \]

we adopt

\[ b \in y \land \neg(a \in y) \Rightarrow \neg(y = \{\}) \]

This might also have a nice side-effect on the rearrangement of predicate sometimes.

One could argue that the rewriting rules available involving these symbols are powerful enough to solve this problem automatically. Perhaps so for simple and small formula. However, for big formulae, rewriting on unnecessary predicates becomes a waste of time.

Another nasty problem that occurs quite often is the proper instantiation of generic actuals, whilst selecting previously declared definitions. With the inappropriate instantiation, one could end up in a situation where the goal is obviously true, but the prover can offer no help whatsoever. For example, suppose one needs to instantiate the generic type X of the theorem below

**theorem inSeq[X]**

\[ \forall s : \text{seq } X \mid e \in s \cdot \exists x : 1\ldots#s ; y : \text{ran } s \cdot e = (x, y) \]

to a compound type such as

\[ (\text{seq } N \times N) \]

What would be the appropriate instantiation of X in the tool? If one tries to use the compound type directly like in

```
use inSeq[(seq N \times N)][s := mySeq, e := myElem];
```

the expressions on the theorem would need to carry the generic actuals around like in

\[ \exists x : 1\ldots#(\text{seq } N \times N)s ; y : \text{ran}[Z, (\text{seq } N \times N)]s \cdot e = (x, y) \]

which turns out to be impossible for the prover to automatically discharge. Even though usually possible to prove, discharging such conditions are difficult, time consuming, and completely outside the desired goal under concern. Therefore, this is a situation to be avoided indeed.

The appropriate answer to the question is given by the following rule of thumb: unless otherwise necessary, whenever generic actuals are needed for instantiations, the maximal type ought to be given in order to avoid side condition proof obligations. For our example, the appropriate instantiation of X is its maximal type value that is

\[ (P(Z \times Z) \times Z) \]
Instantiating the generic types with their maximal values eliminates the need to carry generic actuals around expressions because they are the biggest domain of the type. In other words, types inferred by Z/Eves are maximal and generic actuals can then be hidden. Therefore, the complex and time consuming scenario mentioned above entirely disappears and expressions become

$$\exists x : 1 \ldots \#s; \ y : \text{ran} \ s \cdot e = (x, y)$$

However, to achieve higher (or intuitively expected) levels of automation, another problem still remains. When \(X\) is instantiated with its maximal value, the prover issues a type checking proof obligation saying that the actual type of \(s\) must imply the (different) instantiated one. In practical terms, an assumption rule such as

$$\forall s : \text{seq}(\text{seq}\ N \times N) \cdot s \in \text{P}(\text{P}(\text{P}(Z \times Z) \times Z))$$

ought to be introduced.

Although not always obvious to discharge, this kind of assumption rules are true, since the maximal type always contains the original type. Assumption rules are an effective solution because they are automatically applied by every non-trivial transformation tactic. Therefore, whatever the non-trivial tactic one chooses to use, the side condition will vanish automatically in one step. Finally, with the knowledge of these assumption rules, any theorems related to such data structure can be automatically discharged by the proof engine, hence leading to more concise and modular proofs.

Under very particular (and rare) circumstances, this strategy of instantiating generic actuals with maximal types does not apply. In other words, the non-maximal type must be used as a generic actual instead of the maximal type. Because it is unusual, whenever it happens specific solutions applies.

Due to these oddities around generic types, whenever possible rewriting rules are advised. They enable the prover to properly infer the maximal generic actual for the user, thus leading to a smoother — and hence more automatic — experience with the tool. Careful and conscious reasoning might apply to the ability of such rules tough. However, regardless the decision for usage or ability, additional assumption rules are generally needed for compound types.

Generic types can even cause problems on the use of some proof tactics under particular circumstances. For example, the instantiate and equality substitute commands are affected whenever \(Z/Eves\) is not able to infer the generic types on expressions of a formula. For instance, in a theorem such as

$$\forall s, t : \text{seq}\ N \mid \#s = \#t \cdot \text{dom} \ s = \text{dom} \ t$$

\(Z/Eves\) internally infers the complete formula to be

$$\forall s, t : \text{seq}\ N \mid \#(\text{P}(Z \times \text{P}Z)) \ s = \#(\text{P}(Z \times \text{P}Z)) \ t \cdot \text{dom}[(Z, \text{P}Z)] \ s = \text{dom}[(Z, \text{P}Z)] \ t$$

Because these are the maximal generic actuals, they are hidden in order to improve readability.

Unfortunately, for the expressions where \(Z/Eves\) cannot infer the generic actuals, the user needs to make direct interventions. Not surprisingly, this situation is more common on large or direct proofs where global transformations are not appropriate. Luckily, these interventions are usually easy and not so often necessary. The most common case is related to expressions with the empty set. For example, if a proof reaches a stage with a formula such as

$$i \in N \land x \in P \ N \land i = \#{} \land x \subseteq \text{ran} \ (\{} \) \Rightarrow (\exists y : \text{seq} N \cdot y = (0, 1))$$

one cannot apply a tactic like

\text{instantiate}\ y \ ::= \ (0, 1)

because the prover issues the error message \text{ImplicitActuals}, even tough the instantiation has nothing to do with undefined generic actuals.

\(^2\) This particular error message says that “the generic actuals for a global reference were not specified, and their types could not all be inferred”. 
Under these circumstances, the expressions involving the empty set are the ones causing the trouble. On the given examples, the offending expressions are the assumptions about \( i \) and \( x \). The solution is to apply local transformations on these expressions mentioning empty sets. For the running example, one could try the proof steps

apply sizeNullSeq to expression \( \#(\{\}) \);
apply ranEmpty to expression \( \text{ran}(\{\}) \);

leading to the following transformation on the formula

\[
i \in \mathbb{N} \land x \in \mathbb{P} \mathbb{N} \land i = 0 \land x \subseteq \{\} \Rightarrow (\exists y : \mathbb{N} \cdot y = (0, 1))
\]

Now that the offending assumptions were removed, the instantiate command can be use again. Note that not all expressions related to empty sets were removed but the offending ones. The ability to identify those comes with practice. In general, these offending expressions are usually easy removed in a couple of proof steps.

7.5 Getting Around Domain Check Problems

After the declaration of paragraphs, \( \text{Z/Eves} \) automatically introduces domain checks whenever necessary in order to guarantee the consistency of specifications. The criteria for the introduction of domain checks is widely discussed on the reference manual. Although domain checks are usually obvious to proof, exceptions do occur often. Such exceptions can be rather annoying and time consuming to prove if appropriate machinery is not present.

Nevertheless annoying at times, domain checks help in a great extent to spot problems or minor mistakes on some complex definitions, as well as learning the proper Z idioms \( \text{Z/Eves} \) is happy with. This is vital information towards an effective use of the tool.

In our experience, nearly all problems with domain checks arose because of dependencies related to the totality of the functions involved on declarations. Thus, whenever possible, total functions are the option to take for declarations. Even if the function to be declared is not total, totalising it by giving special treatment for cases outside the domain might be wise. That is the case because partial functions usually involve a great amount of tricky details to be dealt with. Luckily, we were able to totalise most partial functions in our specification. Even tough functions are declared as total, domain checks are still added in order to check for the function’s totality consistency. For example, in the definitions below

\[
f : T_1 \rightarrow T_2 \\
g : T_2 \rightarrow T_3
\]

definitions of \( f \) and \( g \)

\[
h : T_1 \rightarrow T_3 \\
\forall x : T_1 \cdot h x = g(f x)
\]

\( \text{Z/Eves} \) might add a domain check requiring that

\[
x \in \text{dom } f \land f x \in \text{dom } g
\]

depending on the complexity of the definitions of \( f \) and \( g \).

Our approach is to find a pattern to follow whenever domain checks requirements arise. Following some guidelines from the toolkit documentation, we devise our own variation solution. It turned out to be quite adequate to achieve high levels of automation and performance.
In summary, for every function declared in axiomatic or generic boxes that might become involved in future domain checks, at least three housekeeping theorems are included. One assumption rule about the declared function type comes first. After that, a sufficient amount of rewriting rules about the function’s result type is included. These result theorems were given as rewriting instead of assumption rules because we wanted to have a greater degree of control during direct proofs, as well as optimise the speed of simplification tactics. Finally, a rewriting rule about the totality of the function is declared.

We added these three theorems for almost all functions defined without generic actuals. These type, result, and total theorems provided for axiomatically and generically defined functions are important for a greater level of automation in \textit{Z/Eves} also improving performance. This can indeed be checked on the verbose information about dependencies output by the prover textual interface.

The proof plan of these theorems follows a quite repetitive pattern. Therefore, despite the typing overhead and possible maintenance, they do not add much time on proof effort discovery. The declaration of these theorems and their proofs comes just after the declared function under a header named \textbf{Automation}.

For simplicity, let’s assume that for our running example, only \( f \) requires the introduction of these theorems. The order of declaration must be bottom-up because the tool makes a single-parsing through the specification file. However, that is not the case for our explanation, and we follow a top-down approach.

### 7.5.1 Totality Rule

The rewriting rule on the totality of \( f \) is given as

\[
\forall x : T1 \cdot x \in \text{dom } f
\]

We follow the naming convention to add the suffix IsTotal to the label of these theorems.

The proof of totality theorems is usually straightforward since \( f \) is indeed declared as total. The informal proof sketch is to introduce the declaration of \( f \) and expand its totality and partiality properties. Under some circumstances, it might be necessary to go down to the function’s characterising set. The point of these expansions is to introduce the necessary variables on the assumptions. After that expansions, we start changing the goal by applying theorems related to the domain of \( f \), usually toolkit theorem \textit{inDom} or the domain definition itself. These theorems transform the goal including an existential quantifier about the elements on the domain of \( f \), in this case \( T1 \). This is present on the functional property just expanded. Finally, a rearrangement of predicates, together with an appropriate instantiation of the existential quantifier with the available variables on the assumptions is usually enough to finish the proof.

### 7.5.2 Function’s Result Rule(s)

During the proofs of most totality theorems that do not involve generic actuals, other auxiliary theorems are needed. This happens due to the maximal type reduction of \textsl{Z} used in \textit{Z/Eves}. In the proof sketch just mentioned, further type checks are usually added. The first of then are related to the maximal type of the function result. For instance, for the function \( f \), the prover would add a type check like

\[
x \in T1 \land f x \in \text{max-type-of-T2}
\]

whenever type \( T2 \) is complex enough (i.e. compound types).

This asks for assumption rules about type \( T2 \) to be introduced following the rationale mentioned earlier in Section 4. Depending on the structure of type \( T2 \) and the theorems involving \( f \), it is possible that more than one result rule become necessary. In other words, the same problem on deciding whether to introduce a rule with \( \rightarrow \), \( \leftrightarrow \), or \( \mathbb{P} \) arises here. For simplicity lets assume that the only result theorem needed is related to the maximal type of \( T2 \); it is given below.

\[
\forall x : T1 \cdot f x \in \text{max-type-of-T2}
\]
As before, we follow the naming convention to add the suffix Result to the theorem label. If more than one theorem is needed, we vary it appropriately like $PFunResult$. We choose to use a rewriting rule for performance reasons of the simplify tactic, but an assumption rule might be more appropriate when this is not an issue.

The proof of result theorems is also easy usually. The informal proof sketch is to introduce the declaration of $f$ followed by the usage of toolkit theorem $applyInRanFun$ with appropriate instantiations. This theorem adds to the assumptions the needed information about the function result. Finally, a rearrangement and reduction finishes the proof.

Sometimes, even a single rewrite is enough. This happens whenever the result type is simple enough not to need toolkit theorem $applyInRanFun$. Note that this occurs because $applyInRanFun$ is not declared as an automatic rewriting rule!. This gives a nice illustration of the consequences on usage and ability choice.

### 7.5.3 Function's Type Rule
At last, in many occasions the proof of the result theorems requires yet another rule. It is related to type checks on the function’s maximal type. The assumption rule on the type of $f$ is given as

\[
\text{theorem grule gFType} \quad f \in \mathbb{P}(\text{max-type-of-T}1 \times \text{max-type-of-T}2)
\]

The proof of the type theorems follows a similar sketch as the totality ones. The function declaration is introduced and further expansions followed by reductions are enough to finish the proof usually.

### 7.6 Partial Functions

If totalisation is not an option, then some guidelines applies for declaration of partial functions. It is a good idea to have the domain of the function present on the same axiomatic box of the declaration. Depending on the generated domain check, the domain definition might go before, or after the function declaration itself.

A part from the totality theorem, the result and type theorems are also useful for partial functions and the same proof plan applies. The major problem around partial functions is the growing number of cases to be analysed during proofs they become involved. This makes the proof usually clumsy and very time consuming indeed.

### 7.7 Renamings and Instantiations

Instantiation of all quantified variables is highly recommended. This is necessary in order to avoid translation problems between Verdi and Z/Eves. Multiple instantiations are done with comma separated expressions.

The tactic use allows automatic instantiations to be inferred from the sub-formulae in the context. Although this is handy, it might be considered to be avoided as it can cause confusion and problems in proofs when names are changed.

Z/Eves do not accept renaming for actual values that are not universally quantified. Whenever one needs to rename a value that is not related to a quantified variable, a replacement is the option. Lets define the schema to show an example

\[
\begin{array}{l}
\text{TestRename} \\
\hline
x : N \\
y : \text{seq } N \\
\text{invariant predicate}
\end{array}
\]
For instance, with quantified or constant values renaming is possible
\[ \forall q : \text{seq}_1 \ N \ | \ \text{TestRename}[0/x, q/y] \cdot \cdots \]

However, the following renaming is not accepted by \textit{Z/Eves}
\[ \forall q : \text{seq}_1 \ N \ | \ \text{TestRename}[\text{head } q/x, \text{tail } q/y] \cdot \cdots \]
because expressions are not quantified. Instead, one needs to use replacements such as
\[ \forall q : \text{seq}_1 \ N \ | \ \text{TestRename}[x := \text{head } q, y := \text{tail } q] \cdot \cdots \]

In fact, replacements are just a more powerful (generic) version of renaming. Nevertheless, they are not present in the \textit{Z} standard and are a \textit{Z/Eves} technicality issue.

7.8 Nuts and Bolts

In this section we give other minor recommendations.

\textbf{Unique Names in Schema Definitions} In schema definitions were quantified variable names are repeated, it is wise to give unique (and meaningful) names instead. For example, in proofs involving a schema like

\begin{verbatim}
QtVarNamesBad
l : N
s : seq N
\forall i : 1..(#s - 1) \bullet s(i) \leq s(i + 1)
\forall i : 1..#s \bullet s(i) \leq l
\end{verbatim}

\textit{Z/Eves} will rename the second \(i\) to \(i_0\). If other schemas with a variable named \(s\) or \(i\) are involved, then this appended index increases.

Proofs involving schemas usually yields huge predicates when the schemas are expanded. Whenever one needs to instantiate a quantifier with available variables, it becomes a nightmare to see through these indexes and understand which index is appropriate. In my personal experience, such situation not just happen often but can also lead to a frustrated proof attempt.

\textbf{Tricks on the \textit{Z/Eves} GUI for Windows} For big specifications, the GUI crashes rather often in not so powerful machines. It can very between a complete crash or the lost of the socket link with the background prover. In both cases, the user is left without option but to kill the process manually. Note that whenever this happens, all unsaved work is lost because saving or exporting do not work. The advice for these situations is to understand the circumstances where the problem occur trying to avoid it in the future, or going to the nice and more reliable xemacs textual interface instead!

Another trick about the GUI is that python seems to have problems with memory management after some period of inactivity with the \textit{Z/Eves} process. In practice that means that simple operations that would take a second on a fresh GUI, can take up to a minute or even crash! The simplest (yet extreme) solution for this is to kill the GUI and open up a new one. Sometimes, saving the file before doing anything also helps even if the file has already been saved previously and no modification has taken place yet. This apparently useless save operation seems to clear internal memory cashes of python and brings the GUI back to normal operation again.
8 Proof Optimisation Techniques

A series of optimisation techniques have been applied throughout the development of our theory. To make large specifications stable and highly automated, a high amount of housekeeping theorems and rewriting rules is needed. Furthermore, the bigger the theory, the bigger the database of theorems to look for compatible rewriting rules, and the higher the amount of proofs to perform. Therefore, for large specifications, optimisation techniques ought to be applied.

There are a series of optimisations we applied gradually. Gradual application is important because of the enormous amount of side-effects a wrong optimisation can make. Furthermore, some optimisations depend on the shape of the theory, and the nature of proofs being performed. Even so, it turned out that optimisations does interfere on the original proof sketch usually. The inclusion of optimisations did not introduce problems on already finished proofs most of the time. Therefore, it does not compromise much ones schedule. Obviously they increase the amount of time spent on each proof but the results are worthwhile as long as this time is kept small. The main optimisation techniques we used are summarised below:

- Elimination of advanced tactics (i.e. prove by reduce).
- Early usage on split proofs to avoid repeated type checks.
- Local versions of tactics (i.e. apply, invoke, and equality substitute).
- Reduction of implicit expansions (i.e. reduce).
- Introduction of housekeeping and rewriting rules.
- Exchange of slower transformation tactics by alternative compound combinations.
- Localised rewriting/reductions whenever simplification is not effective.
- Simplification and/or elimination of redundant housekeeping and rewriting rules.
- Transformation of scarcely used rewriting rules in normal theorems.
- Transformation of assumption rules to rewriting rules.
- Careful rationale of abilities.
- Introduction of generic definitions whenever appropriate.
- Shape the specification to communicate with Z/Eves in the appropriate Z idiom and accent.
- Extensions on the weaker points of the Z/Eves Z toolkit.

For large specifications, it is highly recommended to follow some (or all) of these guidelines. They proved to help Z/Eves in a great extent, as large and direct proofs became easier to finish. In our theory for example, these optimisations reduced the time spent to finish proofs from couple of hours to tenths of minutes. They also help one to communicate more effectively with the tool.

In what follows, we provide some additional explanation on each of these techniques. At the time of writing, most of these guidelines were applied in more than 90% of our entire formalisation. The percentages given here are rather informal, thus not necessarily accurate for all cases.

8.1 Advanced Tactics Elimination

We substitute all occurrences of prove by reduce and prove by rewrite, with equivalent simpler combinations of tactics. By performing the proofs through fine grained steps, it is possible to figure the most efficient way to define the theory such that it will use the prover power more efficiently. Therefore, such optimisations in fact teach you to follow the yellow bricks road to wonderland!

Moreover, for simpler proofs on large formulae, these two tactics usually take a long time to complete, and sometimes the results are not as satisfactory as imagined. For instance, the application of the equality substitute or prenex during prove by reduce at the wrong place could unable one to finish a proof.
8.2 Early Usage on Split Proofs

During some proofs with case analysis, it is possible to introduce theorems or definitions that are going to be used on all cases early in the proof. This decreases the number of steps since repeated usage are avoided. However, the major advantage of this technique is that it eliminates duplicated provisos on different cases being analysed. That is, the provisos of the used paragraph can be discharged before case analysis thus leading to simpler (and hence faster) proof scripts.

8.3 Local Versions of Tactics

Three tactics were affected: application of definitions through apply, explicit expansions through invoke, and equality substitutions through equality substitute.

We followed the pattern to apply rewriting rules globally as much as possible. This makes proofs simple and easy to read but can create performance penalties on large proofs due to unnecessary transformations. Therefore, compromises were made according to the circumstances.

The most efficient optimisation making tactics local occurred with invoke. For instance, global invocation is very time consuming when schemas are present because the schema calculus is highly compact and usually full expansion is of no interest. Moreover, all equality substitutions were made local, except the ones used for trivial predicate calculus transformations.

8.4 Reduction of Implicit Expansions

Whenever possible, implicit invocations occurred through the reduce command were substituted by corresponding (couple of) proof steps using rewrite or simplify. For instance, a single reduce command was usually substituted by a local expansion followed by rewriting with

\[
\text{invoke } T_1 ; \\
\text{rewrite} ;
\]

This decreases the number of reductions in 74%. However, approximately good part of this number comes from large refinement proofs at the top-levels of the theory. Nevertheless difficult to predict accurately, these optimisations certainly reduced the time on proofs in a high degree.

8.5 Housekeeping Theorems Introduction

Plenty of assumption and forward rules were added for all compound data types and schemas. As mentioned earlier in Section 7.5 (p. 18), their proofs are straightforward most of the time. Therefore, such inclusion overhead does not sum up much on the total time spent on proofs. Furthermore, properties on the basis of our theory were declared as rules since they are highly used afterwards.

For instance, nearly 70% of all assumption and forward rules were proved with a couple of repetitive steps. In the remaining 30%, approximately 98% of proofs followed some repetitive pattern. In the end, only a handful of those rules really introduced time overheads.

8.6 Transformation Tactics Exchange

The more powerful the tactic, the higher its execution time. Whenever the database of housekeeping and rewriting rules increases, the amount of time spent in trying to pattern match available rules also grows. Since transformation tactics are the most commonly used, their optimisation are highly effective.

There two main strategies: (i) exchange of rewrite and reduce for simplify, and (ii) localised rewriting/reductions whenever simplification is not effective. Usually, simplify alone is not enough for the exchange. Therefore, a clever (nondeterministic) combination of invoke T, rearrange, equality substitute v, and
simplify applies. For large proofs, localised version of rewrite/reduce can decreases the overall execution time in around 80%.

These strategies improves performance because apart from the transformation tactics, all others are very fast. As expected, the fastest of the transformation tactics is simplify since it consider neither expansions, nor rewriting rules. For our theory, it was possible to remove combined tactics completely and to decrease the number of reductions by 76% (i.e. from 1800 to 470 +/-). From this number, 25% became were rewrite and the remaining 75% became simplify. However difficult to measure accurately, we believe this optimisation was the most effective. It certainly was the biggest contribution on decrease of time spent on proofs.

8.7 Housekeeping Theorems Elimination

The major drawbacks of having many rules are the effects on the execution time of transformation tactics. Advantages on numbers are clear but difficult to assess. The main problem here occurs because of assumption rules since they affect all transformation tactics. An increase amount of rewriting rules affects rewriting and reduction. Forward rules are not of concern since they are usually very specific to a particular schema only.

Assumption rules with non-maximal types that turned out to be scarcely used were transformed into rewriting rules or even normal theorems. Hardly used rewriting rules were into normal theorems. Finally, some rules were completely removed and their proofs copied wherever necessary.

8.8 Abilities Rationale

Based on experience within the expressions that often occurred, we reduced the number of enabled rules to 37%. However, most enabled rules are placed on the base of the theory.

A nice trick one can play for pedagogic or debugging purposes, is to have all (or part) of the specification disabled whilst trying to prove the same set of theorems. It turns out that this strategy can point out some direction towards the completion of complex proofs. It is worth trying because there are times where the informal sketch is far from the final proof script.

8.9 Generic Definitions

The top-levels of our theory relies on a basic low-level specialised automata theory. All the functions defined for this central data structure were provided with generic types. Thus, it allowed a great amount of reuse. Since we followed the generic type guidelines mentioned earlier, a couple of problems with generic types arose, but they were solved quickly. The use of generic types also lead to the expansion of the toolkit for some data structures such as sequences and injections.

8.10 Talking in Z with a Z/Eves Accent

All guidelines from Section 7 were followed. In fact, these guidelines also appear because of optimisation efforts.

8.11 Weakening Rules

In the Z toolkit for Z/Eves, there is a rather tricky section with definitions and theorems called “Weakening”. Weakening rules are those that helps the prover to chain rewriting of terms from stronger to weaker predicates, which allows greater degrees of automation.
Although this idea of weakening is very useful when introducing new theories and operators, it is not widely explained anywhere, as far as we know. Even so, some toolkit theorems take advantage of this nice trick, such as

\textbf{theorem} disabled bigcupSubsetBigcup \[X\]
\[
\forall S, T : \mathbb{P} (P \times X) \mid S \subseteq T \implies S \subseteq \bigcup T
\]

Generally, if one wants to weaken a stronger goal, say
\[
\bigcup S \subseteq \bigcup T
\]
so that more rewriting rules would be enabled, one needs to state a theorem in the following particular format:

\[\text{[rule]} \forall \text{Vars} \mid \text{Weak}(P) \bullet \text{Strong}(P)\]

That is, whenever \texttt{Z/Eves} finds predicate \texttt{Strong}(P), it will be able to automatically substitute it for the weaker predicate \texttt{Weak}(P).

### 8.12 Toolkit Extensions

The toolkit is very powerful and carefully designed to be highly automated. However, as pointed out in its texts, many extensions are needed. The main reason for these extensions is obviously increase in automation and performance.

We provide extensions in various fronts as mainly rewriting rules mainly. Normal theorems, and few assumption rules were also introduced. There are extended theorems related to properties of sets, relations, numbers, functions, finiteness, and cardinality. The last two proved to be very useful since this is one of the weak spots in the toolkit. However, the greatest improvement made by these extensions is related to sequences and injective sequences. These extensions on the toolkit are introduced later in this chapter.

### 9 Proof Sketching Examples

In this section we provide two examples used to illustrate some of the aspects presented throughout this document. Both examples tackle the problems related to proof planning. The first gives a nice example of many issues regarding available tactics and optimisations. The latter presents a full example of an inductive proof on sequences.

#### 9.1 Proof Planning

**Direct Exploratory vs. Guided Planned Proofs**

In this example we present some issues related to proof planning in general. We are trying to prove the obvious theorem

\[
\forall X : \mathbb{P} \mathbb{N} \bullet 0 < \#(\{10\} \cup X)
\]

The intention is to show that some silly examples can lie down on the category of proofs that need planning. We start by issuing the following command through the interactive textual interface via emacs

\[
\text{try} \forall X : \mathbb{P} \mathbb{N} \bullet 0 < \#(\{10\} \cup X);
\]

leading to

\[
X \in \mathbb{P} \mathbb{N} \Rightarrow 0 < \#(\{10\} \cup X)
\]

We approach this problem in two ways: (i) directed exploratory proof, and (ii) guided planned proof.
9.1.1 Direct Explanatory Proof Let's say one wants to conduct a direct proof for this theorem where no previous plan has been done yet. The idea is trying the available toolkit theorems following just our mathematical intuition and knowledge of the toolkit.

Firstly, let's apply the toolkit theorem cardCup to introduce some information about the cardinality on the goal. Because it is not a rewriting rule, direct use with appropriate instantiations was needed. In fact, not being a rule also limits the power of prove by reduce for our example.

use cardCup[Z][S := X, T := \{10\}];

Also note that we used the maximal type \(Z\) as the instantiated generic actual instead of \(N\) in order to avoid the problems mentioned in Section 7.4 (p. 16). The goal then becomes

\[
\left(\left( X \in \mathbb{FZ} \land \{10\} \in \mathbb{FZ} \Rightarrow \left( \#X + \#\{10\} = \#(X \cup \{10\}) + \#(X \cap \{10\}) \right) \right) \land X \in \mathbb{PN} \right) \Rightarrow \left( 0 < \#(\{10\} \cup X) \right)
\]

At this stage, some directions are pointed out. In order to use the conclusion of the used theorem, two type checking conditions about the finiteness of \(X\) and \(\{10\}\) must be discharged. Since the information about \(X\) appears afterwards, we tried the following

\[
\text{rearrange ; apply unitFinite ; simplify ;}
\]

leading us to

\[
\left( X \in \mathbb{PN} \land \left( X \in \mathbb{FZ} \Rightarrow \#X + \#\{10\} = \#(X \cap \{10\}) + \#(X \cup \{10\}) \right) \right) \Rightarrow \left( 0 < \#(\{10\} \cup X) \right)
\]

One might argue that simplify is not powerful enough, and something like rewrite or reduce should be used instead. In practice, the reduce command leads to an even more complex assumption with condition expressions involved and we forget about this path.

It seems that an assumption rule about the type of \(X\) must be introduced. In fact, the exploration show us an even strong problem: to use cardinality operator oblige us to have an infinite type for \(X\). Thus, we could redeclare our theorem as

\[
\forall X : \mathbb{FN} \cdot 0 < \#(\{10\} \cup X)
\]

In fact it is a good idea to leave the information about power sets as it serves as an implicit assumption rule on the type of \(X\). The theorem can then be stated as

\[
\forall X : \mathbb{PN}, X \in \mathbb{FN} \cdot 0 < \#(\{10\} \cup X)
\]

Applying the previous steps on this new version reaches the same problem, now because of the missing assumption rule below.

\[
\text{theorem grule gAFinSetType}
\]

\[
\forall A : \mathbb{FN} \cdot A \in \mathbb{FZ}
\]

From experience, it is nice to include another assumption rule linking the intermediate (non-maximal) type with the maximal type for \(A\)

\[
\text{theorem grule gASetType}
\]

\[
\forall A : \mathbb{PN} \cdot A \in \mathbb{PZ}
\]
After the inclusion of these assumption rule, the formula is then transformed to what we expect. In summary, the commands

```plaintext
retry ;
try ∀X : P N | X ∈ F N • 0 < #{10} ∪ X; use cardCup[S := X, T := {10}];
rearrange ;
apply unitFinite ;
simplify ;
```

with the appropriate theorem and added assumption rules lead the goal to

\[ (X \in P N \land X \in F N \land #X + #\{10\} = #(X \cap \{10\}) + #(X \cup \{10\})) \Rightarrow (0 < #((10) \cup X)) \]

The next step is to separate in the two cases whether \( X \) is empty or not.

```plaintext
split X = {};
simplify ;
cases ;
```

Just before the `cases` command, the goal becomes

\[
\begin{cases}
  \text{if } Z \ (X = \{\}) \text{ then } &\left(\begin{array}{l}
  \{\} \in P N \land \{\} \in F N \land \\
  #\{\} + #\{10\} = #((\{\} \cap \{10\}) + #((\{\} \cup \{10\}) \Rightarrow (0 < #((10) \cup \{\})))
\end{array}\right)\\
  \text{else } &\left(\begin{array}{l}
  X \in P N \land X \in F N \land #X + #\{10\} = #(X \cap \{10\}) + #(X \cup \{10\}) \\
  \Rightarrow (0 < #((10) \cup X))
\end{array}\right)
\end{cases}
\]

The first case is easy because the size of a singleton set is indeed greater than zero and we give the proof steps only.

```plaintext
apply cupNullLeft;
apply capNullLeft;
apply unitSubset;
simplify;
apply cupNullRight;
apply unitSubset;
simplify;
apply sizeUnit;
simplify;
next;
```

We usually indent proof steps of individual cases for the sake of readability of proof scripts. The second case where \( X \neq \emptyset \) still needs some attention. We decided to whether 10 belongs to \( X \) or not since it would simplify the expressions related to cardinality.

```plaintext
split \{10\} \cap X = \{10\};
simplify ;
back 1;
cases ;
```

The simplification after case splitting was not a good idea because it removed the needed information to early and stacked case analysis started again. The case where \{10\} belongs to \( X \) is dealt first by applying
a theorem about intersection and singleton sets

apply unitCap;
apply capUnit;
rearrange;
simplify;

leading to the goal

\[
\begin{align*}
&X \in P\mathbb{N} \land X \in P\mathbb{N} \land \\
&(10 \in X \lor \{\} = \{10\}) \land \neg X = \{\} \land \\
&\text{if } Z \ (10 \in X) \text{ then} \\
&(1 + \#X = 1 + \#(X \cup \{10\})) \\
&\text{else} \\
&(1 + \#X = \#(X \cup \{\}))
\end{align*}
\] \Rightarrow (0 < \#(X \cup \{\}))

From the disjunction on the assumptions it is possible to simplify the conditional expression. This exposes the fact that \(10 \in X\). To finish the proof we need to apply the appropriate theorem to the localised expression telling the prover that 10 indeed belongs to \(X\).

apply nullEqualUnit;
simplify;
apply cupSubsetRight to expression \(X \cup \{10\}\);
apply cupSubsetLeft to expression \(\{10\} \cup X\);
apply subsetDef;
apply unitSubset;
apply sizeUnit;
simplify;

Note that localised application was necessary due to the presence of more than one expression involving \(\cup\). However, instead of finishing the proof, the prover returns

\[
X \in P\mathbb{N} \land X \in P\mathbb{N} \land 10 \in X \land \neg X = \{\} \Rightarrow 0 < \#X
\]

This obviously true since \(X\) is not empty. Nevertheless, this information must be explicit added to the prover since the related toolkit theorems are not rewriting (automatic) rules.

use cardIsNonNegative[Z][S := X];
use card0[Z][S := X];
rearrange;
simplify;

This finishes the proof of this subcase. The final case is similar. Firstly we expose the information about singleton intersection.

next;
apply unitCap;
apply capUnit;
rearrange;
simplify;

Next, we simplify the easy assumptions

apply sizeNull;
apply sizeUnit;
simplify;
apply theIntegerElimination;
simplify;
Finally, with some more localised transformations and applications

```plaintext
with predicate (#(X ∪ {10}) ∈ ℤ) rewrite;
apply cupCommutes to expression X ∪ {10};
apply unitSubset;
simplify;
```

reaching the goal

\[
\left( X \in \mathbb{P} \land X \in \mathbb{F} \land \sim 10 \in X \land \sim \{\} = \{10\} \right) \Rightarrow (0 < #(\{10\} \cup X))
\]

After that, we use localised equality substitution in order to change the right-hand side for the left-hand side.

```plaintext
equality substitute #(\{10\} ∪ X);
```

changing the goal as expected to

\[
\left( X \in \mathbb{P} \land X \in \mathbb{F} \land \sim 10 \in X \land \sim \{\} = \{10\} \right) \Rightarrow (0 < 1 + #X)
\]

To finish the subcase and the entire proof, we just need to introduce information about cardinality as before.

```plaintext
use cardIsNonNegative[ℤ][S := X];
use card0[ℤ][S := X];
rearrange;
simplify;
next;
```

On this proof note the use of localised instantiations when needed, the trick on equality substitution and the fact we avoided more powerful transformations at each step. In fact, if one tries to use rewrite the proof becomes much smaller. Our intention in explicitly avoiding it is to show the degree of detail one might need to add when optimisations need to be taken into account.

9.1.2 Guided Planned Proof The idea here is to have an informal plan detailed enough to finish the proof by hand. Afterwards, additional details needed by Z/Eves are addressed.

For our example, one would argue as follows.

- Divide the proof in two cases: 10 ∈ X and 10 ∉ X.
- Case 1: 10 ∈ X
  - We know that \{10\} is a subset of X.
  - Use the law \( T \subseteq S \Rightarrow S \cup T = S \) with \( S = X \) and \( T = \{10\} \).
  - New goal (#X > 0) already holds because X is not empty.
Case 2: $10 \notin X$

- Use the law $\# S + \# T = \# (S \cup T) + \# (S \cap T)$ with $S = X$ and $T = \{10\}$.
- Use the law $\# x = 1$ for $x = 10$.
- We know that $\{10\}$ is not a subset of $X$.
- Therefore, we can introduce the assumption that $X \cap \{10\} = \emptyset$.
- Use the law $\# \emptyset = 0$ for $(X \cap \{10\})$.
- Thus, we can add the assumption that $\# X + 1 = \# (X \cup \{10\})$.
- Substituting the left-hand side of this equality on the original goal.
- New goal $0 < \# X + 1$ is already true via arithmetic.

This could be directly translated into the following $Z$/Eves commands

\znote{Try possible reductions first}
prove by reduce;
\znote{Split on desired condition}
split \neg 10 \in X;
cases;
  \znote{Case 1: ten belongs to X}
  apply cupSubsetRight to expression $X \cup \{10\}$;
  prove by reduce; \†
next;
  \znote{Case 2: ten does not belong to X}
  use cardCup[Z][S := X, T := \{10\}];
  prove by reduce; \‡
next;

\† — The prover cannot finish the proof yet. This happens because the law saying that the size of nonempty sets is strictly positive is not declared as a rewriting rule, thus automation is not possible. Also note that the law about $\cup$ and $\subseteq$ is a disabled rewriting rule. That is so because usually the law applies for local transformations. Although user intervention was necessary, the application is easy because instantiations are performed automatically by the prover.

\‡ — The laws on the size of singleton and empty sets are enabled rewriting rules. Therefore, the prover can use them automatically here. However, the information about cardinality being a natural number is not a rule and must be added manually to finish the proof.

Sometimes, during such proofs, $Z$/Eves comes up with necessary needed assumptions not possible to automatically infer. Whenever they are not available and general enough, it is a good idea to declare then as auxiliary lemmas, thus having a more modular and reusable proof.

### 9.1.3 Complete Script

The complete script for both direct and guided proofs are given next. An optimised version of the first script illustrating some of the techniques mentioned is also provided. We typeset the script such that it can be imported verbatim either by the GUI or the textual interface. Some examples on comments in proofs and paragraphs are also provided.

\begin{zproof}{gXFinSetType}
prove by reduce;
\end{zproof}
In this proof we can apply some of the optimisation techniques mentioned earlier. For the sake of illustration, lets assume two scenarios for the proof above: (i) expansions on reductions are time consuming
due to the presence of schemas at the assumptions, and (ii) rewriting takes a considerable time to finish and unnecessary transformation are occurring.

Firstly, let's use the two theorems about cardinality repeated among cases early in the proof. This saves us at least two reduction, one for each of the last two cases.

Next, implicit expansions and unnecessary rewriting occurred via the `reduce` command is substituted by an equivalent combination of explicit application and simplification. Perhaps lengthy sometimes, these combinations are easy to build because tprover hints the necessary information during transformations.

For example, after the first reduction in the proof above, `Z/Eves` returns the message

Which simplifies

when rewriting with `cupCommutes`, `cupNullLeft`, `unitSubset`, `inNull`, `capUnit`, `sizeUnit`, `sizeNull`, `nullFinite`, `nullSubset`

forward chaining using `KnownMember$declarationPart`, `knownMember`, {[internal items]}
with the assumptions `&cup$declaration`, `select_2_1`, `select_2_2`, `&cap$declaration`, `&cardinality$declaration`, `finset_type`, `natType`, {[internal items]} to ...

`true`

That means the simplification occurred having the mentioned enabled rewriting, forward, and assumption rules into context. With practice, one is able to quickly guess what is really necessary and what can be wasted. The optimised proof below shows the example.

\begin{theorem}{rule rXRuleDirectedExploratoryProofOptimised}
\znote{Main theorem --- optimised proof script version}
\forall X: \power\nat | X \in \finset\nat @ 0 < \#(\{10\} \cup X)
\end{theorem}

\begin{zproof}{rXRuleDirectedExploratoryProofOptimised}
use cardCup[S := X, T := \{10\}] ;
\znote{Early usage of theorems appearing in more than one case.}
use cardIsNonNegative[S := X] ;
use card0[S := X] ;
apply unitFinite ;
rearrange ;
simplify ;
split X = \{} ;
simplify ;
cases ;
\znote{Reduction substituted by combination of rules and simplification.}
\t1 apply cupNullLeft ;
\t1 apply capNullLeft ;
\t1 apply unitSubset ;
\t1 simplify ;
\t1 apply cupNullRight ;
\t1 apply unitSubset ;
\t1 simplify ;
\t1 apply sizeUnit ;
\t1 simplify ;
next ;
\t1 split \{10\} \cap X = \{10\} ;
\t1 cases ;
\t2 apply unitCap;
It is clear that optimisation can sometimes introduces more steps. Whenever simplification combinations become more difficult or not appropriate, one might try rewriting either locally, globally, or trivially instead. In our example, we applied localised rewriting when simplification became ineffective.

Finally, we want to point out some necessary paths taken during the explanatory proof such as considering $X$ empty or not. For this proof, this is irrelevant and introduces more problems than solutions. Therefore, even for detailed direct proofs, at least some partial plan is advised. Next comes the guided planned proof script without optimisation considerations.

\begin{theorem}{rule rXRuleGuidedPlannedProof}
\forall X: \power \nat \mid X \in \finset \nat \land 0 < \# (\{10\} \cup X)
\end{theorem}

\begin{zproof}[rXRuleGuidedPlannedProof]
prove by reduce;
split 10 \in X;
cases;
\t1 apply cupSubsetRight to expression X \cup \{10\};
\t1 prove by reduce;
\t1 use card0[\num ][S := X];
\t1 prove by reduce;
\znote{Inconvenient tactic because important assumptions are early removed.}
\znote{Alternative solutions must be investigated (e.g. look at prover hints).}
\t1 use card0[\num ][S := X];

9.2 Inductive Proofs

After the example on proof planning, we introduce the last example on inductive proofs for sequences.

**Inductive Proofs over Sequences**

The most difficult proofs in \( Z/Eves \) are the ones involving induction. Apart from the complexity and size of inductive proofs themselves, \( Z/Eves \) introduces a series of requirements. The tool comes with an example of inductive proofs over integers and simple structures. However, many induction proofs are related to sets or sequences. They usually have a structure not covered by the available examples. In fact, to the extent of our knowledge, there is no documentation of this elsewhere.

The example comes from Chapter 9 of Woodcock & Davies book on Z. It aims at proving that for sequences, reversing distributes over filtering\(^3\). That is, the reverse of a sequence \( s \) filtered on a set \( T \) is equivalent to filtering the reverse of \( s \) on \( T \). In mathematical terms, for a generic type \( X \) this is given as

\[
\text{theorem tRevOfFltIsFltOfRev}
\begin{align*}
\forall s: \text{seq } X;\ T: \forall X \bullet \text{rev} (s \mid T) &= (\text{rev } s) \mid T
\end{align*}
\]

To avoid problems with generic actuals, we provide \( X \) as a given set instead of a generic type.

\([X]\)

---

\(^3\) The proofs and most of the explanation in this example comes from discussions with Mark Saaltink throughout about inductive proofs on September 2003. They were formatted here with minor changes.
– **Unit Induction**
This scheme has two base cases on the empty sequence, and unit sequence. The inductive case uses concatenation to show that \( P(s \cdot t) \), assuming \( P(s) \) and \( P(t) \) as the inductive hypotheses. This scheme uses toolkit theorem \texttt{seqInduction}.

– **Left Induction**
This scheme has one base case, the empty sequence, and an inductive case to show that \( P((x)^s) \) assuming \( P(s) \). This scheme uses toolkit theorem \texttt{seqLeftInduction}.

– **Right Induction**
This scheme has one base case, the empty sequence, and an inductive case to show that \( P(s \cdot (x)) \) assuming \( P(s) \). It uses theorem \texttt{seqRightInduction} from the toolkit.

The basic recipe for any of these induction schemes is the same. To show that a property \( P \) holds for \( s \)

\[
\text{theorem } tPOnS[X] \\
\forall s : \text{seq } X \cdot P(s)
\]

one need to state the auxiliary lemma

\[
\text{theorem } lPOnSZEves[X] \\
\text{seq } X \subseteq \{s : \text{seq } X \cdot P(s)\}
\]

This lemma is necessary because it obeys the shape of the induction scheme theorems. That is, inductive proofs can only be declared in \texttt{Z/Eves} through the characterising set of the inductive property under concern. We follow the convention to append the suffix \texttt{ZEves} for these lemmas.

The next step is then to apply one of the induction theorems and prove the subgoals it introduces. Finally one is able to state the original theorem \texttt{theorem } tPOnS[X], bringing in the lemma just proved. Usually, only few simple steps are necessary to finish the proof.

Whatever the proof strategy, these schemes are the only starting point available. The choice depends upon what theorems and definitions one have. This implies that one must bare in mind the induction schemes available while providing declarations to be involved on inductive proofs.

9.2.2 Function Definitions Appropriate for Induction Schemes For our example, the definition \( \text{rev} \) is adequate but the definition of \( \downarrow \) does not match any induction scheme. Therefore, an alternative version of \( \downarrow \) must be given. In fact we choose to redefine both following the book example strictly. It follows the left inductive scheme.

The new version of \( \text{rev} \) (named \( \text{REV} \)) matching the pattern of the left inductive scheme is given below.

\[
\text{REV} : \text{seq } X \rightarrow \text{seq } X \\
\langle \text{rule dREVBasis} \rangle \\
\text{REV}(\langle \rangle) = \langle \rangle \\
\langle \text{rule dREVLInduc} \rangle \\
\forall x : X; s : \text{seq } X \cdot \text{REV}((x)^s) = (\text{REV } s)^{(x)}
\]
This declaration introduces a quite complex domain check proof

\[(\text{local } REV \in \text{seq } X \rightarrow \text{seq } X) \Rightarrow \left(\begin{array}{c}
\langle \rangle \in \text{dom local } REV \\
(\text{local } REV(\langle \rangle) = \langle \rangle) \\
\forall x : X ; s : \text{seq } X \bullet (\langle x \rangle, s) \in \text{dom}_{\downarrow}(\downarrow)
\land (x) \land s \in \text{dom local } REV \\
\land s \in \text{dom local } REV \\
\land (\text{local } REVs, (x)) \in \text{dom}_{\downarrow}(\downarrow)
\end{array}\right)\]

Now one can see the power of \textit{prove by reduce} in practice below!

\textbf{proof[REV$\$domainCheck]} \textit{prove by reduce ;}

The new version of \(\uparrow\) (named \(\text{FLT}\)) from the book is given next. For simplicity, we provide it as function instead of an infix operator.

\[
\text{FLT} : \text{seq } X \times P X \rightarrow \text{seq } X
\]

\[
\langle \text{rule dFLT Basis}\rangle
\forall T1 : P X \bullet \text{FLT}(\langle \rangle, T1) = \langle \rangle
\]

\[
\langle \text{rule dFLT InducInT}\rangle
\forall t : X ; s : \text{seq } X ; T : P X \mid t \in T \bullet \text{FLT}(\langle t \rangle \lhd s, T) = (t) \lhd \text{FLT}(s, T)
\]

\[
\langle \text{rule dFLT Induc NotInT}\rangle
\forall t2 : X ; s2 : \text{seq } X ; T2 : P X \mid t2 \notin T2 \bullet \text{FLT}(\langle t2 \rangle \lhd s2, T2) = \text{FLT}(s2, T2)
\]

However, it turns out that it does not match the pattern for left induction directly. Thus, one is encouraged to provide the less elegant (equivalent) version below

\[
\text{FLT} : \text{seq } X \times P X \rightarrow \text{seq } X
\]

\[
\langle \text{rule dFLT Basis}\rangle
\forall T1 : P X \bullet \text{FLT}(\langle \rangle, T1) = \langle \rangle
\]

\[
\langle \text{rule dFLT Induc}\rangle
\forall x : X ; s : \text{seq } X ; T : P X \bullet
\text{FLT}(\langle x \rangle \lhd s, T) = (\text{if } \exists (x \in T) \text{ then } \langle x \rangle \lhd \text{FLT}(s, T) \text{ else } \text{FLT}(s, T))
\]

\textbf{proof[FLT$\$domainCheck]} \textit{prove by reduce ;}

\[9.2.3\ \textbf{Induction Schemes Setup}\]

The property of interest is distribution of reverse over filter. Here, a second wrinkle appears: with the given declarations, there are two ways to set up the induction.

1. \(T\) varying in the induction step

\[
\forall s : \text{seq } X \bullet (\forall T : P X \bullet \text{REV}(\text{FLT}(s, T)) = \text{FLT}(\text{REV} s, T))
\]
2. Holding $T$ constant in the induction step

$$\forall T : \mathcal{P} X \bullet (\forall s : \text{seq } X \bullet \text{REV} (\text{FLT} (s, T)) = \text{FLT} (\text{REV} s, T))$$

Sometimes the generality of setup (1) is needed. In our example it is unnecessary and the extra quantifiers can be avoided. So, setup (2) is what we will use. Setup (1) would have a lemma of the form

**Theorem** tRevOfFltIsFltOfRevTryV1

\[
\text{seq } X \subseteq \{ s : \text{seq } X \mid \forall T : \mathcal{P} X \bullet \text{REV} (\text{FLT} (s, T)) = \text{FLT} (\text{REV} s, T)\}
\]

while setup (2) has the lemma

**Theorem** tRevOfFltIsFltOfRevTryV2

$$\forall T : \mathcal{P} X \bullet \text{seq } X \subseteq \{ s : \text{seq } X \mid \text{REV} (\text{FLT} (s, T)) = \text{FLT} (\text{REV} s, T)\}$$

If one tries this theorem out, the prover will struggle quickly, and the possible way out would be to try each (implicit) subgoal manually/directly.

9.2.4 **Induction Schemes Investigation** The next tread is to inspect what induction theorem is more appropriate for our proofs. We will investigate all three schemes to see which one is suitable for our theorems and which auxiliary lemmas are necessary.

**Unit Induction**

Let's start trying the unit induction theorem. It leads to two goals: one for the base case

\[ x \in X \Rightarrow \text{REV} (\text{FLT} (\langle x \rangle, T)) = \text{FLT} (\text{REV} (x), T) \]

and another for the inductive case

\[
\left( \begin{array}{c}
\forall s : \text{seq } X \wedge t : \text{seq } X \wedge T : \mathcal{P} X \\
\text{REV} (\text{FLT} (s, T)) = \text{FLT} (\text{REV} s, T) \\
\text{REV} (\text{FLT} (t, T)) = \text{FLT} (\text{REV} t, T)
\end{array} \right)
\Rightarrow
\text{REV} (\text{FLT} (s \wedge t, T)) = \text{FLT} (\text{REV} (s \wedge t), T)
\]

The base case goal hints that we should make a rule about the unit of \text{FLT}

\[ \text{FLT} (\langle x \rangle, T) \]

and another about \text{REV}

\[ \text{REV} (\langle x \rangle) \]

For the inductive case goal, we probably want to argue as follows:

\[
\text{REV} (\text{FLT} (s \wedge t, T))
\]

\[= (\text{by a lemma } r\text{FLT}\text{Cat we need to declare/prove}) \]

\[\text{REV} (\text{FLT} (s, T) \wedge \text{FLT} (t, T))\]

\[= (\text{by a lemma } r\text{REV}\text{Cat we need to declare/prove}) \]

\[\text{REV} (\text{FLT} (t, T)) \wedge \text{REV} (\text{FLT} (s, T))\]

\[= (\text{by the two hypotheses}) \]

\[\text{FLT} (\text{REV} (t, T)) \wedge \text{FLT} (\text{REV} (s, T))\]

\[= (\text{by } r\text{FLT}\text{Cat}) \]

\[\text{FLT} (\text{REV} (t) \wedge \text{REV} (s), T)\]

\[= (\text{by } r\text{REV}\text{Cat}) \]

\[\text{FLT} (\text{REV} (s \wedge t), T)\]
**Left Induction**

There are two other induction schemes to try: left and right induction. Applying the left induction theorem leads us to a goal just for the inductive case

\[
(\text{REV}(\text{FLT}(s, T)) = \text{FLT}(\text{REVs}, T)) \\
\Rightarrow \\
(\text{REV}(\text{FLT}((x) \cap s, T)) = \text{FLT}(\text{REVs} \cap (x), T))
\]

The simpler result is not at all surprising, since the functions \text{REV} and \text{FLT} have been defined according to the left induction scheme. However, considering whether or not \(x \in T\) gives two goals

\[
\text{REV}(\text{FLT}(s, T)) \cap (x) = \text{FLT}(\text{REVs} \cap (x), T)
\]

and

\[
\text{REV}(\text{FLT}(s, T)) = \text{FLT}(\text{REVs} \cap (x), T)
\]

Again this is not surprising since our inductive definition of \text{FLT} has a conditional expression. This solution hints that we need a lemma about the concatenation of \text{FLT}

\[\text{FTL}(s \cap (x), T)\]

**Right Induction**

To finish our investigation, let’s apply the right induction theorem. It leads us to a goal just for the inductive case as well

\[
(\text{REV}(\text{FLT}(s, T)) = \text{FLT}(\text{REVs}, T)) \\
\Rightarrow \\
(\text{REV}(\text{FLT}(s \cap (x), T)) = \text{FLT}(\text{REV}(s \cap (x)), T))
\]

In this case two lemmas are needed: one about the concatenation of \text{FLT}

\[\text{FTL}(s \cap (x), T)\]

and another about the concatenation of \text{REV}

\[\text{REV}(s \cap (x))\]

**Investigation Conclusions**

Not surprisingly, the simplest case is the second using left induction. However, rather than just work through that proof, it is worth noting that in our experience it usually is most effective to have lemmas for sequence functions that apply to singleton sequences and to general concatenations. These rules will be generally useful, so that other proofs have a better chance of succeeding. Therefore, we will work through the lemmas suggested by the application of the unit induction scheme. Furthermore, it also allows us to show more details about sequence induction proofs.
The proof script of our investigation is given below

\begin{verbatim}
proof[lRevOfFltIsFltOfRevTryV2]
  apply seqInduction ;
  prove ;
  retry ;
  apply seqLeftInduction ;
  prove ;
  retry ;
  apply seqRightInduction ;
  prove ;
\end{verbatim}

9.2.5 Auxiliary Lemmas

In summary, based on our choice for the unit induction scheme, we need to include four auxiliary lemmas: one for the unit and concatenation of both \textit{REV} and \textit{FLT}.

* \textit{REV Unit}

The unit rules are special cases of the defining axioms for \textit{REV} and \textit{FLT}.

\begin{verbatim}
theorem rule rREVUnit
  \forall x : X \bullet REV(x) = \langle x \rangle
\end{verbatim}

The proof of \textit{rREVUnit} needs just the right instantiation.

\begin{verbatim}
proof[rREVUnit]
  use dREVLInduc[x := x, s := \langle \rangle] ;
  rewrite ;
\end{verbatim}

* \textit{FLT Unit}

For \textit{FLT}, it is slightly more complicated because its definition is expressed with a conditional. That is, we need to cover the case of an element that is included in the result, and the case of an element that is excluded.

\begin{verbatim}
theorem rule rFLTUnit
  \forall x : X ; T ; P X \bullet FLT((x), T) = (if \exists (x \in T) then (x) else ( ))
\end{verbatim}

Again, the proof is simple and needs just the right instantiation.

\begin{verbatim}
proof[rFLTUnit]
  use dFLTLInduc[x := x, s := \langle \rangle, T := T] ;
  rewrite ;
\end{verbatim}

Note that we could have used application or tried \textit{prove}, since these functions are defined as enabled rewriting rules.

* \textit{REV Concatenation}

Next comes the concatenation rules. To show that

\[ REV(s \triangledown t) = REV(t) \triangledown REV(s) \]

we follow the induction recipe again.
As already known, we have the choice of letting $t$ be constant throughout the induction or allowing different $t$’s to be used in the induction hypothesis. We can follow the simple route and keep $t$ constant.

The induction lemma must not be a rule, otherwise the next proof fails due to tautology reasoning. This happens because rules are treated differently by the rewriting scheme.

**Theorem** lREVCatZEves

\[ \forall t : \text{seq } X \bullet \text{seq } X \subseteq \{ s : \text{seq } X | \text{REV}(s \uplus t) = \text{REV}(t) \uplus \text{REV}(s) \} \]

Investigating the induction schemes again here, we discovered that the left induction scheme matches the definition of $\text{REV}$ most closely, so that’s the best one to apply.

**Proof**

apply seqLeftInduction;
prove;

The final theorem involving concatenation of $\text{REV}$ is then introduced.

**Theorem** rule rREVCat

\[ \forall s, t : \text{seq } X \bullet \text{REV}(s \uplus t) = \text{REV}(t) \uplus \text{REV}(s) \]

The proof follows by trivial manipulations of the induction scheme (ZEves) lemma.

**Proof[rREVCat]**

use lREVCatZEves;
rewrite;
apply inPower;
instantiate e == s;
rewrite;

* FLT Concatenation

For $\text{FLT}$ the induction recipe with the considerations mentioned is applied a third time.

**Theorem** lFLTCatZEves

\[ \forall t : \text{seq } X ; T : \mathbb{P} X \bullet \text{seq } X \subseteq \{ s : \text{seq } X | \text{FLT}((s \uplus t), T) = \text{FLT}(s, T) \uplus \text{FLT}(t, T) \} \]

**Proof[lFLTCatZEves]**

apply seqLeftInduction;
prove;

Finally, we can state the theorem of interest

**Theorem** rule rFLTCat

\[ \forall s, t : \text{seq } X ; T : \mathbb{P} X \bullet \text{FLT}(s \uplus t, T) = \text{FLT}(s, T) \uplus \text{FLT}(t, T) \]

**Proof[rFLTCat]**

use lFLTCatZEves;
rewrite;
apply inPower;
instantiate e_0 == s;
rewrite;
9.2.6 Main Proof Wrap-up Now we have all the necessary lemmas and can get back to the main proof. The wrap-up is trivial. Firstly, we state the induction scheme theorem on the selected setup as usual

\textbf{theorem} \ lRevOfFltIsFltOfRevZEves

\[ \forall T : P \cdot \bullet \]

\[ \text{seq} X \subseteq \{ s : \text{seq} X \mid \text{REV} (\text{FLT} (s, T)) = \text{FLT} (\text{REV} s, T) \} \]

The proof follows directly because of the four enabled rewriting rules.

\textbf{proof}\[ lRevOfFltIsFltOfRevZEves \]

apply \textit{seqInduction};

prove;

Finally, the main theorem is given below.

\textbf{theorem} \ tRevOfFltIsFltOfRev

\[ \forall s : \text{seq} X ; T : P X \cdot \text{REV} (\text{FLT} (s, T)) = \text{FLT} (\text{REV} s, T) \]

The proof follows from manipulations with the induction scheme theorem.

\textbf{proof} \[ tRevOfFltIsFltOfRev \]

use \textit{lRevOfFltIsFltOfRevZEves} ;

\textit{rewrite} ;

\textit{apply} \textit{inPower} ;

\textit{instantiate} \ e \_ 0 == s ;

\textit{rewrite} ;

As one could see it is good practice to break a proof down into manageable parts. In fact, we did not really need to prove the singleton lemmas above. We might instead have used the appropriate instances of \textit{dREVLInduc}, \textit{or dFLTLInduc} where needed in the main induction.

However, it was worth modularising them for some reasons:

- Keeps main proof simple.
- Rewrite rules for other proofs.
- Other main proofs involving \textit{REV} and \textit{FLT} are likely to be simpler.
- These lemmas can be stated simply and directly, and their proofs are not cluttered with extraneous hypotheses.

Having an informal proof plan helps, as it can show the main steps and suggest intermediate lemmas to prove. Even if one do not have a complete proof plan, one can use the prover to help finding it. Our investigation with inductions was a good example of this approach for exploring proof plans.

As a final remark, the text of the example was typed such that it can be imported by both interfaces of Z/Eves. All proofs were completely discharged, except for the two trial theorems. For the textual interface, the following commands might be useful\(^4\)

\begin{verbatim}
read "\l:\learning\seqindproof.text.txt" ;
print history summary ;
print status ;
\end{verbatim}

This completes the entire example.

\textit{end of example} \[ \square \]

\(^4\) The full path is mandatory.
10 Summary of Naming Conventions

Theorems and lemmas containing most-significant results are framed. At the end of each chapter, a summary of declarations is provided. Throughout our use of \textit{Z/Eves} we applied the following naming conventions on the labels of paragraphs:

- **Prefixes:**
  - \texttt{g} — assumption rule.
  - \texttt{r} — rewriting rule.
  - \texttt{f} — forward rule.
  - \texttt{p} — properties (maybe rules).
  - \texttt{l} — lemma (not rule).
  - \texttt{t} — theorem (not rule).
  - \texttt{d} — axiomatic and generic boxes labels.
  - \texttt{rp} — refinement of process.
  - \texttt{ra} — refinement of action.
  - \texttt{ro} — refinement of operation.
  - \texttt{sa} — simulation applicability theorem.
  - \texttt{sc} — simulation correctness theorem.
  - \texttt{po} — refinement calculation proof obligation.

- **Naming** — capitalised meaningful names.

- **Suffixes:**
  - \texttt{Type} — maximal type rules.
  - \texttt{Result} — function maximal result rules.
  - \texttt{RelResult} — relational function result rule.
  - \texttt{PFunResult} — partial function result rule.
  - \texttt{FunResult} — total functional result rule.
  - \texttt{FinResult} — finite function result rule.
  - \texttt{WeakXXX} — for weaker (non-maximal) types.
  - \texttt{IsTotal} — totality rules for functions.
  - \texttt{Invariant} — schema invariant forward rules.
  - \texttt{PRE} — schema precondition calculation.
  - \texttt{ZEves} — \textit{Z/Eves} inductive lemma.

- \LaTeX Headers:
  - \texttt{Automation} — housekeeping theorems after axiomatic, generic, and schema boxes.
  - \texttt{Laws} — properties about a set of declarations.

- \LaTeX Labels:
  - Document and section names contraction followed relevant information.

- Files:
  - \LaTeX documents (text + declarations + proof scripts):
    - \texttt{dc.tex} — declarations and housekeeping paragraphs.
    - \texttt{pr.tex} — theorems about related declarations file.
    - \texttt{rc.tex} — refinement calculation.
    - \texttt{po.tex} — refinement proof obligations. \LaTeX
  - Script files
    - \texttt{ps.dc.tex} — proof scripts for \texttt{dc.tex} file.
    - \texttt{ps.pr.tex} — proof scripts for \texttt{pr.tex} file.
    - \texttt{zs.tex} — \textit{Z} sections for corresponding \LaTeX files.
    - \texttt{zsp.tex} — \textit{Z} section proofs for corresponding \LaTeX files.
    - \texttt{ims.tex} — interactive mode script processing.
    - \texttt{bms.tex} — batch mode script processing.
References


