Proving conjectures using cadiz

/Tutorial guides

This document won't teach you how to do proofs, but if you already have some idea on that, then it will tell you how to use cadiz to help find proofs. It assumes that you know the Z notation. You should already be familiar with interacting with cadiz, but a short review is given here.

1. Contents of this page

2. The terminology of proof

A *sequent* consists of generic parameters, declarations, *antecedent* predicates, and *consequent* predicates. A sequent expresses the *conjecture* that the conjunction of the antecedents implies the disjunction of the consequents.

In the following example, there is one generic parameter (`PERSON`), three declarations (`p` and the two relations), three antecedent predicates, and one consequent predicate.

$$Topic ::= Z \mid CADiZ\_interface \mid PROOF$$

$$
\begin{array}{l}
[PERSON]\ p : PERSON;\ \_\ knows\ \_, \_\ can\_do\ \_ : PERSON \leftrightarrow Topic \\
\quad \mid p\ knows\ Z, \\
\quad p\ knows\ CADiZ\_interface, \\
\quad p\ knows\ Z \land p\ knows\ CADiZ\_interface \Rightarrow p\ can\_do\ PROOF \\
\vdash ?\ p\ can\_do\ PROOF
\end{array}
$$

The syntax of a sequent permits a name to be associated with a sequent. The symbol separating the antecedents from the consequents ($\vdash ?$) is called the *turnstile*.

A sequent is *proven* (i.e. a conjecture is shown to be a *theorem*) by the application of *proof steps*. Each proof step takes as input a sequent (the *goal*) and generates as output a collection of zero or more new sequents (the *sub-goals*). Those goals that are accepted as theorems without generating any sub-goals are the *axioms*. Viewed from the opposite perspective, conjectures, goals, sub-goals, theorems and axioms are all sequents: a conjecture is a sequent to be proven; a goal is a sequent that arises in the course of a proof; sub-goals are sequents generated by the application of a proof step to a goal; a theorem is a proven sequent; and an axiom is a sequent that is accepted as a theorem without proof.

Proof is an inductive process: a goal has been proven if the application of a proof step to it generated no sub-goals, or if all of the generated sub-goals have been proven. Applications of proof steps give rise to a tree structure of sequents: a *proof tree*.

Proof steps can be classified as follows. There is a proof step that recognises a goal as being an axiom. A proof step that replaces a formula within a goal

by an equivalent formula in the corresponding place within its single sub-goal is an *in situ replacement* step. Other elementary proof steps are called *proof rules*: each one's sub-goals have different numbers of declarations, antecedents or consequents from the original goal, or have more than one sub-goal.

Proofs can be structured using *lemmas*: instead of working directly towards axioms, the proof steps can be used to work towards lemmas, which are sequents that have been or may be proven separately.

An elementary proof step performs only a small quantity of reasoning. A larger reasoning step can be achieved using a *tactic*. The application of a tactic causes the application of a combination of other proof steps. There is nothing that can be proven by a tactic that could not be proven by a combination of other proof steps.

A *decision procedure* is a proof step that replaces a predicate by either *true* or *false* as appropriate. This cannot be done for all predicates, but it is useful to have decision procedures where possible.

3. Invoking proof steps

The proof steps can be used to attempt to prove a conjecture only if cadiz is run with the `-P` option, and only if no errors are found by parsing and typechecking of the specification. The `-P` option causes the creation of three windows. A window on the right-hand side displays the Z specification. A window on the left-hand side displays output generated during the session, such as sub-goals of proof steps.

A window at bottom-right displays a proof tree, exhibiting the structure of the current proof. This proof tree window overlaps the specification window.

Proof steps are invoked by first *selecting* their arguments, then *inspecting* those arguments to see a menu of the names of applicable proof steps, and finally *choosing* the desired proof step from the menu.

Any well-formed Z formula can be selected by pointing the mouse and clicking button 1. The selected formula is highlighted using inverse video, and its syntactic category is named in the window's title bar. If the formula thus selected is a smaller formula within the desired one, click again within that smaller formula to select the next larger formula; and repeat as necessary. To select a large formula in few clicks, click on a part of it that is involved in the fewest sub-formulae. For example, to select the whole of the predicate $\exists\, schema \bullet expression$, click on either the $\exists$ symbol or the $\bullet$ symbol, as clicking within *schema* or *expression* will select a sub-formula, leaving further clicks needed.

Selected arguments are inspected by holding down mouse button 2. While mouse button 2 is depressed, a menu appears listing just those commands (such as proof steps) that are applicable to the selected formulae. A proof step is chosen by pointing at its name in the menu and releasing button 2.

The majority of proof commands are unary, so require only one selection, but can be applied to several selections simultaneously. Such commands are offered only if they are applicable to every one of the selections.

The way to make several selections for a single command is as follows. Make one selection, then hit the `x` key. This marks the selection by drawing a cross through

it, and it ceases to be in inverse video. Repeat for the other selections, but omit crossing the last one: there has to be an inverse video selection for button 2 to pop-up the menu. This use of crosses allows all the selections to be perceived, even if they overlap. The `c` key erases the most recent cross (useful if a mistake has been made), and the `C` key clears all crosses.

Unary commands can be assumed to be applicable to multiple selections unless the documentation says otherwise. Some proof commands require more than one operand, and so interpret multiple selections differently—in particular, the order in which selections are made is significant. The necessary order is documented for each such command.

4. Using the proof tree

When a proof step is chosen from a middle button menu, the sub-goals it generates are displayed in the left-hand window, where they are appended to the end of the displayed document. This does not convey the tree structure of a proof, or even indicate which goals have yet to be proven. The tree structure of a proof is displayed in the third window. Non-leaf nodes in the proof tree correspond to goals to which proof steps have been applied; they are labelled with the names of those corresponding proof steps. Leaf nodes in the proof tree correspond to the remaining sub-goals, displayed as boxes, and to lemmas referred to by the proof, displayed as the names of those lemmas. Each node in the proof tree can be selected just like a Z formula. A message in the title bar indicates whether the corresponding goal has been proven yet. Inspecting and choosing the *context*

command causes the corresponding goal to be selected (more precisely, for a non-leaf node, the formulae within the goal, to which the proof step was applied, are selected). Equally, inspections of goals offer the *node in tree* command when appropriate.

The first attempt at a proof might not succeed. There can be several attempts to prove a goal active concurrently. The proof tree shows only one attempt, with ellipses drawn around nodes having alternative proof attempts. When a node with an allipse is inspected, the *other attempts* command is offered. This command pops up a menu naming all the other proof steps that have been applied to the inspected node's goal from which any one can be chosen. Proof attempts can be pruned from the tree when you are sure that an attempt won't succeed. The *prune* command does that, but it doesn't erase the full texts of the corresponding goals from the left window; it's merely for tidying the tree.

New sub-goals that are duplicates of known goals (in this or previous proofs in this session) are automatically identified with (replaced by) those known goals. The proof tree may then cease to be a tree but remain a directed acyclic graph, though we still call it the proof tree; steps that would create cycles are not permitted. Applying the *origin* command to a sub-goal in the left window selects its original parent goal.

As the number of nodes in the proof tree increases, cadiz scales the tree to fit within the window: the arrows between nodes become shorter, and the text labelling nodes becomes smaller. If it becomes illegible, the number of nodes in the displayed tree can be reduced using the *lift* command. This raises unproven leaves and referenced named lemmas to an inspected ancestor node, concealing

all intervening nodes. Any other descendents of the inspected node that are cross-selected are also retained. This is sufficient to cope with most trees, though the case of a single goal giving rise to dozens of immediate sub-goals remains problematic.

5. Recording and playing a proof

A script of the steps taken in proving a goal can be saved using the *record script* command. This is offered when a node in the proof tree is inspected. It prompts for a filename for the script. If the goal has a name associated with it, that name is suggested as a default filename for the script. The *record script* command saves the steps from the inspected node downwards, so is usually used on the root node. Any steps concealed by lifts are scripted, rather than the lift steps themselves. Any steps used to prove lemmas used within the recorded proof are not scripted, as it is assumed that the proofs of the lemmas are recorded separately. Incomplete proofs can be scripted.

A script can be replayed when a goal or its corresponding node in the proof tree is inspected. The *play tactic* and *apply tactic* commands are used to replay a script, as a script is recorded using a subset of the tactic language. All intermediate sub-goals are displayed by *play tactic*, whereas the *apply tactic* command displays only the residual unproven leaf goals. A script can be applied to goals other than the one from which it was recorded, but there is only a slim chance of this succeeding.

Once a collection of scripts has been accumulated, it is sometimes desirable to replay them to assess whether they are all still valid. This can be requested using the `-S` command-line option to cadiz. This option can be combined with any mode (`-v`, `-x`, `-P`, etc). It replays the scripts of conjectures in the given file and also conjectures of parent sections from other files, and produces a report to the standard error stream (the shell window from which cadiz was invoked). This option assumes that the default names for scripts have been used.

6. Printing a proof

The documents presented in the left-hand and proof tree windows can be printed, but would not be a satisfactory printed presentation of a proof because they omit the information in the links that cadiz maintains between them. A self-contained document presenting all the details of a proof can be generated by inspecting any node in the proof tree and choosing the *printed proof* command. This produces, in the proof tree window, a more traditional presentation of the proof of the goal corresponding to the inspected node. (Be sure to inspect the root of the whole tree if you want to see the whole proof.) Each goal is followed by the command name of the proof step that was applied to it. The selections to which the command was applied are marked. Where more than one sub-goal was generated, these are labelled with numbers. Each number appears again where the presentation of the proof of the goal that it labels resumes. A proof need not be complete: goals without proof are flagged as `NOT PROVEN`. Parts of the proof that are elided from the tree because of *lift* steps are not shown. To make them be shown, the *lift*

steps must be "undone" first using the *unlift* (or *other attempts*) command.

This presentation of a proof can be sent to a printer using the previewer's *printer* command (in its button 3 menu). (The print command can be edited, allowing the proof to be directed to any shell command.) Inspecting anything in this presentation of the proof offers only the *proof tree* command, which reverts this window to showing the previous proof tree. Applying a proof step has the same effect.

7. Avoiding variable capture

The bindings of names to declarations in Z are determined statically: each name is bound to the declaration with the closest surrounding scope. These bindings are inferred by cadiz's typechecker, and can be inspected using the *declaration* command.

All proof steps rearrange formulae, some of them moving expressions into different scopes, as the following example illustrates.

$$t \in \{S \bullet u\} \quad \Longrightarrow \quad \exists\, S \bullet t = u$$

This proof step says that a value is in the set denoted by a comprehension if and only if there exist values for the components of the schema for which the value is equal to the value in the set. The relevant facet of this proof step is the movement of expression t into the scope of the declarations of S. For example,

suppose we have this conjecture

$$[PERSON]\ p : PERSON$$
$$\vdash ?$$
$$p \in \{p : PERSON \bullet p\}$$

and we apply the above proof rule to the consequent.

$$[PERSON]\ p : PERSON$$
$$\vdash ?$$
$$\exists\, p : PERSON \bullet p = p$$

The result has an equality between `p` and `p`, where one `p` is intended to be bound to the declaration of the whole sequent, but instead that name is "captured" by the declaration within the existential predicate. This is the so-called "variable capture" problem.

cadiz uses an unconventional approach to the variable capture problem. Since its typechecker inferred the bindings, cadiz can remember the bindings regardless of rearrangements arising from proof steps. When the resulting sequents are presented to the user as text, cadiz automatically renames any declarations, along with names bound to them, whereever those declarations would otherwise capture references to other declarations. The new names are formed by the addition of `0` subscripts, so are easily recognised. Internally, the original names are retained.

$$[PERSON]\ p : PERSON$$
$$\vdash ?$$
$$\exists\, p_0 : PERSON \bullet p = p_0$$

If the user subsequently finds it necessary to type in one of these names (as might be required with the *cut apart* or *quantification tac* commands), it is best explicitly to rename the declaration first (using the *new name* command).

The conventional approach would have been to disable proof steps that would otherwise cause variable capture, hence requiring the user explicitly to rename declarations so as to enable use of the proof step. Another approach that might work in other contexts would be to automatically rename on doing the proof step, and to use the new names henceforth.

Schemas cause problems for all three approaches. The *new name* command has had to be restricted to avoid renaming components of schema expressions, as that would break some schema calculus operations. (Renaming components of schemas used as declarations is sound.) Also, it should not always be necessary to expand a reference to a named schema to be able to rename its components. This is solved by generating renaming notation. Once a declaration is renamed, all references to it are renamed, but then references concealed in theta expressions and schema predicates are a problem: this is solved by generating substitution notation.

In the cadiz approach, the renaming and substitution notation is generated on presenting text to the user; it is not itself inspectable, which might be surprising but for the obvious 0 subscripts.

8. The proof steps

The proof steps are invoked by commands listed in the button 2 menu. Each proof step is presented as a general template, reflecting the fact that it can be applied to many different sequents. A variable name in a template can match any formula of the appropriate syntactic class:

$p, q, ...$	*denote predicates*
$e, t, u, ...$	*denote expressions*
f	*denotes the name of a free type*
g	*denotes the name of a free type injection*
h	*denotes the name of a free type element*
x, y, z	*denote names*
d	*denotes a declaration*
$S, T, ...$	*denote schemas*
b	*denotes a binding*
ds	*denotes a list of declarations*
Q	*denotes a quantifier*

The symbol τ denotes an expression used as a type, the notation $\tau\ e$ denoting the expression that stands for the type of expression e. The formulae $sig(ds)$ and $pred(ds)$ denote the signature and property respectively of the declarations ds. The notation *chartuple* S denotes the characteristic tuple of S.

Z permits omission of the characteristic tuples of set comprehensions and mu expressions, the | parts of schema texts, and the instantiations of references to gen-

eric declarations. In each case, rules exist for determining defaults for the omitted formulae. The proof rules are all stated in a form without omitted formulae, but are nevertheless applicable where those formulae are omitted. Commands are provided for making omitted formulae visible.

Declarations, antecedent and consequent predicates that are not changed by a proof step are not mentioned in its presentation.

8.1. Proof rules

Proof rules' sub-goals have different numbers of declarations, antecedents or consequents to the original goals. A proof rule is depicted like this.

$$\frac{\textit{sub-goals}}{\textit{goal}}$$

A proof rule states that *goal* is a theorem if all *sub-goals* are theorems.

There is a complete list of proof rule commands in the reference manual.

8.2. In situ replacements

In situ replacements cause an inspected formula to be replaced by another formula implied by the original inspected formula. Several formulae can be replaced in one command: all but one must be selected and crossed (using the previewer's

`x` command), then the last should be inspected. The same command must be applicable to all the selections.

In situ replacements are depicted like this.

$$original \quad \Longrightarrow \quad replacement$$

There are a large number of *in situ* replacement steps, grouped into families according to the names of the commands that invoke them. There is a complete list of *in situ* replacement commands in the reference manual.

cadiz permits use of *in situ* replacement steps within axiomatic, schema and free type paragraphs, as well as sequents, enabling some simplification of those other paragraphs to be done.

8.3. Lemmas

From cadiz's perspective, lemmas are supported by the *lemma* and *cut conjoined* and *cut disjoined* and *cut apart* proof rules. From our perspective, however, they are a distinct proof concept and worthy of separate discussion here.

Lemmas are useful in structuring proofs into manageable parts. They often taken the form of equivalences or equalities, enabling replacement of a formula in the goal by an equivalent predicate or equal expression respectively (by subsequent use of the *Leibniz* command). This composition gives an effect similar to an *in situ* replacement, but one that can be a much larger reasoning step than any

elementary *in situ* replacement. Although the resulting proofs may appear to be much shorter, note that all of the lemmas must themselves be formally proven before cadiz will regard the original conjecture as being formally proven.

8.4. Decision procedures

From cadiz's perspective, decision procedures are just *in situ* replacements. From our perspective, however, they differ from other *in situ* replacements in performing complicated reasoning that cannot be presented as a simple rewrite rule.

The *linear decision* command solves linear arithmetic problems by the SUP-INF method. The *heuristic decision* command can solve some non-linear arithmetic problems using simulated annealing. The *model check* command solves some finite problems by model checking (using CMU's SMV). The *resolution* command solves trivial first-order predicates that can be decided regardless of any free variables.

Further commands are provided to reveal solutions found by these decision procedures. There is a list of decision procedure commands in the reference manual.

8.5. Built-in tactics

Those commands whose names end in *tac* have effects that would be achievable by combinations of more elementary proof steps, except that the tactic language cannot or could not express those combinations.

8.6. Annotations

Informal text can be attached to a goal in a proof using the *annotation* command. As there is not enough time to prove everything formally, this command is convenient for use in lieu of a formal proof.

9. Tactic language

The tactic language is intended to support the expression of search strategies to find effective compositions of inference rules. A subset of it suffices for recording scripts of specific proofs, as already mentioned above.

Tactics can appear within specifications and also in separate files. Both are applied using either the *play tactic* or *apply tactic* commands, which offer a menu of applicable tactics and the option of a dialogue box into which to type a filename. Tactics within the specification are thus easier to apply, but they cannot be revised within a cadiz session. The difference between *play tactic* and *apply tactic* is that the former reveals the proof steps taken by the tactic, whereas the latter is presented as a single step.

The toolkit defines some tactics for simplifying core Z formulae, and for unfolding applications of operators. This enables a style of proof that alternates between unfolding operations and simplifying, thus avoiding manual use of many of the elementary proof steps.

The documentation of the tactic language currently comprises a draft paper that

serves as a tutorial, and formal descriptions of its troff mark-up and LaTeX mark-up and syntax and semantics as separate entries in the reference manual.

10. Obtaining conjectures to prove

The Z notation allows the statement of conjectures, but says little about what conjectures to state. A formal method combines a formal notation with rules about how it should be used, and consequently gives rise to conjectures to verify that the rules have been obeyed. As cadiz aims to support the Z notation, independent of any particular method, it automatically generates few conjectures.

Properties that a specifier believes should be consequences of the specification may be formulated as conjectures within the specification. The laws in the toolkit are examples of such manually-formulated conjectures.

The *exists conjecture* command generates a conjecture to ensure that values can be found for the variables of a (generic) axiomatic paragraph, i.e. that there are no contradictions in the constraints between them.

The refinement commands generate verification conditions.

The *local VCs* and *all VCs* commands generate compliance verification conditions to ensure that SPARK code complies with the Z specification.

The *tame conjecture* command generates the proof obligation whose proof is required to be certain that a generic function is tame.

11. Script manager

The script manager provides a summary of the statuses of the scripts of the specification's conjectures. This display is provided in the proof tree window at the beginning of a cadiz session, and can be redisplayed subsequently using the script manager command. It assumes that the default naming convention for scripts has been used.

12. General observations

Conducting proof solely with elementary proof steps is too tedious: tools like the decision procedures and tactics are necessary if the prover is to be effective. Perhaps decision procedures can be found for proof obligations generated in very restricted contexts. However, it is too much to hope that tactics could be written that would act as decision procedures for arbitrary conjectures: proof in general involves searching in an incomputably large space, and hence is impractical without the guidance that only a human can provide.

There will never be enough time to prove all the things that could be conjectured about software. For best productivity, the theorem prover should keep itself busy sorting out the trivial mathematics, and the user busy directing an overall strategy.

IT 29-Oct-2000