A tactic language for reasoning about Z specifications

/Reference manual

This section of the CADiℤ manual is derived from a paper published at the Northern Formal Methods workshop in 1998.

ABSTRACT The syntax and semantics of a particular tactic language are defined. The language uses lazy evaluation to manage backtracking in the search space. It also uses pattern matching to associate names with formulae that will subsequently be passed as arguments to an inference rule. The result of the inference rule is accompanied by a revision of the association, so that corresponding formulae in the result may be passed as arguments to the next inference rule. The combination of this revision with lazy evaluation raises some problems for efficient implementation.

1. Introduction

This paper is about a tactic language for the CADiℤ theorem prover[?]. CADiℤ aims to provide direct support for Z notation[?] and to conform to the forthcoming Z standard[?]. Our involvement in the standardization panel has given us some confidence about what will be in the standard[?]. CADiℤ performs syntax checking, type checking, prettyprinting, browsing, and increasingly formal reasoning including proof of conjectures. The Z specification appears on the screen

in traditional mathematical notation. Well-formed formulae can be selected with the mouse. A menu of commands applicable to the selected formulae can be requested[?]. Reasoning is performed by a large number of elementary inference rules. A version of CADiℤ is freely available[?].

Finding a proof amounts to finding an appropriate composition of inference rules. The entire search space is so large that this cannot in general be done automatically. Users are able to find proofs by following particular heuristic strategies, realising for example which lemmas are relevant and where those lemmas can be used. The aim of a tactic language is to be able to express such a strategy, so that the user has merely to choose a particular tactic and then the machine can automate its application.

In many theorem provers, the language in which tactics are written is just the language in which the theorem prover itself was constructed, for example Common Lisp is used in Zola[?] and ML is used in ProofPower[?]. CADiℤ is written in C, and so this approach would not be appropriate: we want to be certain that the only way a tactic can build a proof is by the application of elementary inference rules, and in C it would be difficult to ensure this restriction. In other theorem provers, it has been found that functional languages provide a particularly appropriate basis for writing tactics. Indeed, the language ML, which has a large functional subset, evolved as the tactic language of a theorem prover[?]. More recent functional languages assume lazy evaluation[?]. The failure as an empty list of successes paradigm[?] exploits lazy evaluation: a tactic is written to search for a list of successful proofs, and lazy evaluation ensures that no more than the first is computed[?]. The backtracking that is needed in any search procedure thus happens implicitly, rather than having to be coded explicitly.

Having decided to reuse a lazy functional notation, we then had the choice of either reusing an existing implementation of a functional language, interfaced somehow to the inference rules within CADiℤ, or building an interpreter for a functional notation within the toolset. Our inference rules use an unconventional approach to the variable capture issue, which would make the exchange of goals with another tool difficult[**?**]. That, combined with the potential to exploit other aspects of CADiℤ, led to the construction of a tactic interpreter within the toolset. Similar choices have been made for other theorem provers, for example Mural[**?**].

The next section introduces some inference rules, showing how they are invoked interactively and via the tactic language. Section **??** presents an example tactic, to give a flavour of the notation. The syntax and semantics of the tactic language are then defined in subsequent sections. A further example follows, and discussion of some specific issues completes the paper.

2. Inference rules

2.1. Interactive application of rules

CADiℤ's inference rules may be categorised as single argument *in situ* replacements, multiple argument rearrangements, and rules having textual arguments. Examples are given in the following paragraphs.

The following *in situ* replacements, in which p stands for an arbitrary predicate, are called *absorptions*.

$$true \wedge p \implies p$$
$$p \vee false \implies p$$

Interactively, a user can select one or more absorbable formulae in a goal by pointing and clicking, and then the *absorption* command can be chosen from a menu of applicable commands.

Antecedent equalities between expressions, and antecedent equivalences between

predicates, can be used as a basis for rewriting other expressions and predicates respectively.

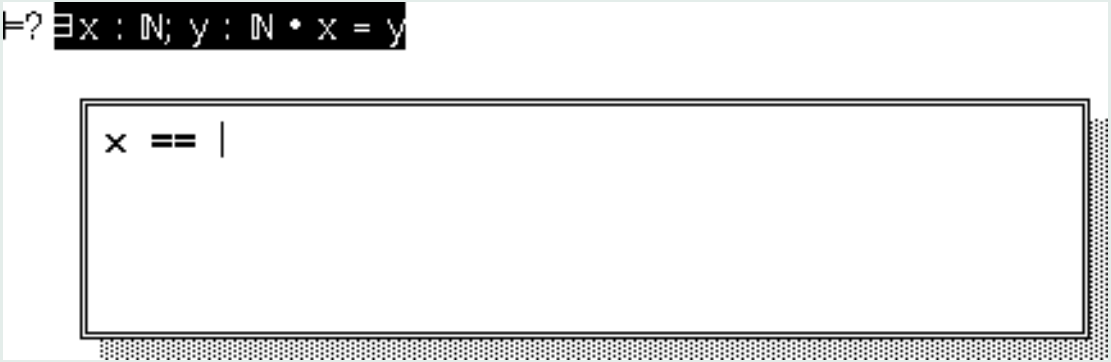
$$\begin{array}{lcl} \mid e_1 = e_2 \vdash ? \ldots e_1 \ldots & \Longrightarrow & \mid e_1 = e_2 \vdash ? \ldots e_2 \ldots \\ \mid p_1 \Leftrightarrow p_2 \vdash ? \ldots p_1 \ldots & \Longrightarrow & \mid p_1 \Leftrightarrow p_2 \vdash ? \ldots p_2 \ldots \end{array}$$

Interactively, a user selects one or more instances of expression e_1 to be rewritten and lastly selects the equality $e_1 = e_2$, then chooses the *Leibniz* command from the menu. Similarly for the predicate case.

A universal or existential quantification predicate can be instantiated with specific values for its quantified variables.

$$\forall\, i : e;\ \ldots \mid p_1 \bullet p_2 \implies (\forall\, i : e;\ \ldots \mid p_1 \bullet p_2) \land (\forall\, i : e;\ \ldots \mid i = e_1 \land \ldots \land p_1 \bullet p_2)$$
$$\exists\, i : e;\ \ldots \mid p_1 \bullet p_2 \implies (\exists\, i : e;\ \ldots \mid p_1 \bullet p_2) \lor (\exists\, i : e;\ \ldots \mid i = e_1 \land \ldots \land p_1 \bullet p_2)$$

Interactively, a user selects the quantified predicate then chooses the *quantification tac* command from the menu, then supplies the values for the quantified variables as text in response to prompts from the tool. Alternatively, existing formulae whose text would be suitable may be selected first, before the quantified predicate, instead of entering into the dialogue with the tool. The text supplied is typechecked in the environment of the quantified predicate. The treatment of formulae as text is necessary because the scopes of the declarations to which their names are bound might not extend to the quantified predicate.



2.2. Tactics that apply rules

The CADiℤ tactic language can apply the same inference rules as can be applied interactively. The syntax for applying an inference rule in the tactic language is to juxtapose the command name, written as a string literal (because some of the command names have spaces in them), with arguments appropriate for that command. Arguments that are formulae within a goal can be selected by numbers. A formula has number n if it is the $n+1$'th formula to be visited in a pre-order traversal of the abstract syntax tree. So the whole goal is number 0, and a formula has a lower number than another if it starts sooner on the page, or if they start at the same place then if it finishes later. This use of numbers suffices for recording scripts of proofs, but is inappropriate in general for tactics. This inadequacy is addressed by pattern matching facilities, as discussed in section **??**. Assume for the moment that p denotes a predicate, and $e, e1, e2$ denote expressions. Arguments that the particular inference rule interprets as text can alternatively be written as string literals, or can be omitted, in which case a dialogue with the user occurs during execution of the tactic. So the examples discussed earlier could be expressed in the tactic language as follows.

$$
\begin{array}{l}
\text{"}absorption\text{"}\ \ e\ \ p \\
\text{"}Leibniz\text{"}\ \ e1\ \ e2\ \ p \\
\text{"}quantification\ \ tac\text{"}\ \ \text{"}text\text{"}\ \ p \\
\text{"}quantification\ \ tac\text{"}\ \ p
\end{array}
$$

Note the overloading of *quantification tac*, which is resolved dynamically by inspection of the arguments to which the command is applied.

An application of an inference rule either produces a single success, or, if the rule is not applicable, fails, producing an empty list of successes. For an inference rule to be applicable, the formula and text arguments to which it is applied must be appropriate. Moreover, the rule must be applied to a single goal (as explained in section **??**, tactics are applied to lists of goals).

The syntax for applying a tactic is similar to that for applying a rule. The name of the tactic is given as a text argument to the *apply tactic* command, along with any other arguments appropriate to that tactic. Like *quantification tac*, if insufficient arguments are given, it is the earlier arguments that are presumed to be missing, and values for them are prompted for using dialogue boxes. An *apply tactic* command differs from an elementary rule in being able to return more than one success.

The application of a unary rule or unary tactic to multiple arguments (such as the absorption above) is equivalent to the sequential composition (section **??**) of that rule or tactic to individual arguments. An application to an argument that starts later is done before an application to an argument that starts earlier, so that the *in situ* replacements do not interfere with one another.

3. An example tactic

The following *flatten* tactic carries out all those *elimination* steps that generate only one sub-goal and are applicable. Subsequent sections define the notation; this example appears here to give a flavour of the notation that is to be defined.

The *flatten* tactic takes a single parameter g, which is required to be a whole goal. It is a recursive tactic: each reference to t has the effect of applying the whole tactic again. The ! notation says that there is no need to consider any success that this tactic might find beyond the first one found. There is then an alternation of four tactics, separated by | symbols.

$$
\begin{array}{l}
\mathit{TACTIC}\ \mathit{flatten}\ \mathit{goal}\ g \mid \\
\quad \mathit{rec}\ t \bullet \\
\quad\quad !(\ \text{“}\mathit{axiom}\text{”}\ g \\
\quad\quad\quad \mid (\mathit{pat}_{\mathit{goal}}\ \mathit{pred}\ p \mid \vdash ?\ p \bullet \\
\quad\quad\quad\quad \mathit{match}\ p \\
\quad\quad\quad\quad :: \mid \neg\ \_\mathit{pred} \bullet \\
\quad\quad\quad\quad\quad\quad \mid \_\mathit{pred} \lor \_\mathit{pred} \bullet \\
\quad\quad\quad\quad\quad\quad \mid \forall\ \_\mathit{stxt} \bullet \_\mathit{pred} \bullet\ \text{“}\mathit{elimination}\text{”}\ p \\
\quad\quad\quad\quad :: \ .;\ t) \\
\quad\quad\quad \mid (\mathit{pat}_{\mathit{goal}}\ \mathit{pred}\ p \mid \mid p\ \vdash ?\ \bullet \\
\quad\quad\quad\quad \mathit{match}\ p \\
\quad\quad\quad\quad :: \mid \neg\ \_\mathit{pred} \bullet \\
\quad\quad\quad\quad\quad\quad \mid \_\mathit{pred} \land \_\mathit{pred} \bullet \\
\quad\quad\quad\quad\quad\quad \mid \exists\ \_\mathit{stxt} \bullet \_\mathit{pred} \bullet\ \text{“}\mathit{elimination}\text{”}\ p \\
\quad\quad\quad\quad :: \ .\ ;\ t) \\
\quad\quad\quad \mid \mathit{skip})
\end{array}
$$

The first alternative succeeds only if the goal g is an axiom. The second alternative associates the new name p with a consequent in the goal, and matches

that consequent against three patterns: a negation, a disjunction, and a universal quantification. If it is any of those, the elimination command is applied (it will succeed), and then the tactic is applied recursively. If it is none of those, the next consequent is considered similarly. If still no match is found, this alternative fails, and the next is considered. The third alternative is similar to the second, but considers antecedents instead of consequents. The fourth alternative, *skip*, always succeeds: it ensures that the whole *flatten* tactic always succeeds, even if it could not find anything to flatten.

4. Tactic combinators

Any review of tactic languages reveals essentially the same combinators in each: sequential composition, parallel composition, alternation, skip, fail and recursion. Those combinators alone suffice as the user-accessible primitives of a tactic language. For precision, the semantics of the combinators will be defined formally. First, some data types and auxiliary functions are needed. These additional definitions are not part of the tactic language accessible to the user. The metalanguages used in these definitions are the ISO standard syntactic metalanguage, and a Haskell-like functional notation.

4.1. Data types

There is no notation for defining new data types in the tactic language. The only data types are ones corresponding to Z phrases, such as goals, and tactics themselves.

4.1.0.1. Goals A goal is a phrase conforming to the *Goal* syntax. A goal is a theorem if the conjunction of the antecedents implies the disjunction of the consequents.

$$Goal \quad = \quad GOAL, [[, Formals,]], Declarations, [|, Predicates], \vdash ? \ , Predicates, END;$$

The *GOAL* token serves to distinguish goals from other Z paragraphs. The optional *Formals* are any generic parameters of the goal. The antecedent and consequent predicates may refer to the global names of the specification, the generic parameters, and the local declarations of the goal. The generic parameters, declarations and predicates are written in the notation of draft Standard Z. The *Goal* syntax subsumes the *Conjecture* syntax of draft Standard Z.

A type *Goal* is assumed to have been defined to represent phrases of the *Goal* syntax.

4.1.0.2. Successes A success is a list of goals arising from application of an inference rule.

$type\ Success = [Goal]$

The following type declaration formally characterizes an inference rule.

$rule :: Goal \rightarrow Success$

Here, *rule* corresponds to the partial application of a command to all of the arguments it needs up to but excluding the goal.

4.1.0.3. Tactics

A tactic is formed from a composition of inference rules.

$type\ Tactic = Success \rightarrow [Success]$

A tactic is applied to either a singleton list containing a user-nominated goal or a success produced by an earlier tactic, and returns a list of all the successes that it can compute. Each success arises from the application of a single rule, perhaps via another tactic.

An application of a tactic that returns an empty list of successes is said to have failed. An application of a tactic that returns a success containing zero goals has succeeded in proving that the goal is a theorem.

Each tactic application is evaluated lazily, so that no more of its successes are computed than necessary for the initial tactic application by the user to produce a success.

4.2. Auxiliary functions

The functions ++, *concat*, *map* and *foldr*, all of which are well-known to functional programmers, are used in the definition of the tactic combinators. (++ appends two lists; *concat* concatenates a list of lists; *map* applies a function to each element in a list, forming a list of the results; and *foldr* combines the elements of a list using a binary function.)

A function, *cp*, to compute the Cartesian product of a list of lists of successes, is also needed.

$$
\begin{aligned}
&cp :: [[Success]] \to [Success] \\
&cp = foldr\ (cp2)\ [\,] \\
&\qquad\qquad where\ xs\ cp2\ ys = [x +\!\!+ y \mid x \leftarrow xs, y \leftarrow ys]
\end{aligned}
$$

Also needed is a function, *bust*, to curtail a list of successes to contain no more than the first success.

$$
\begin{aligned}
&bust :: [Success] \to [Success] \\
&bust\ [\,] = [\,] \\
&bust\ (s : ss) = [s]
\end{aligned}
$$

4.3. Combinators

4.3.0.4. Sequential composition Sequential composition combines two tactics so that, when the combination is applied to a success, the first tactic is applied to the success producing successes to which the second tactic is applied.

$(t_1;\ t_2)\ s = concat\ (map\ t_2\ (t_1\ s))$

Since the application of the first tactic produces a list of successes, the second tactic must be mapped over that list, and the resulting lists of successes are concatenated. Tactic t_1 must be able to cope with the number of goals in success s, and tactic t_2 must be able to cope with the number of goals in each success produced by the application of t_1, otherwise the application fails.

4.3.0.5. Parallel composition Parallel composition combines n tactics so that, when the combination is applied to a success comprising n goals, each tactic is applied to the corresponding goal in order.

$(t_1 \mid\mid ... \mid\mid t_n)\ [g_1, ..., g_n] = cp\ [t_1\ [g_1], ..., t_n\ [g_n]]$

The Cartesian product computes all possible combinations of successes. If any tactic application fails, the Cartesian product is empty, and so the whole parallel composition fails. If the number of goals differs from the number of tactics, the application fails.

4.3.0.6. Script composition Script composition differs from parallel composition only in the cases where it fails, when it guarantees to reduce all of the n applications of tactics to singleton goals.

$(t_1 !! ... !! t_n)\ [g_1, ..., g_n] = cp\ [t_1\ [g_1], ..., t_n\ [g_n]]$

Script composition is intended for use in recorded scripts, so that an obsolete script will replay as much as possible.

4.3.0.7. N-ary composition N-ary composition is analogous to parallel composition, but applying the same tactic to each of the given goals.

$map\ t\ [g_1, ..., g_n] = cp\ [t\ [g_1], ..., t\ [g_n]]$

The Cartesian product computes all possible combinations of successes, so if any tactic application fails, the whole n-ary composition fails. (The overloading of the name *map* to denote both an auxiliary function and a form of tactic is unfortunate for this discussion, but the user of the tactic language sees only the latter.)

4.3.0.8. Alternation Alternation combines two tactics so that, when the combination is applied to a success, the successes from applying the first tactic are appended with the successes from applying the second tactic.

$(t_1 \mid t_2)\ s = (t_1\ s)\ ++\ (t_2\ s)$

In a tactic such as $(t_a \mid t_b);\ t_c$, if t_c cannot succeed given any of the successes produced by t_a, then the successes produced by t_b will be computed and considered. The backtracking that one expects from an alternation is provided implicitly by the lazy evaluation mechanism.

4.3.0.9. Curtailment Curtailment arranges for a tactic to compute no more than a single success.

$!\ t\ s = bust\ (t\ s)$

If no successes are computed, then the whole curtailment fails similarly, other-

wise the result contains just the first success. Continuing the above example $(t_a \mid t_b);\ t_c$, if t_c can succeed from successes produced by t_a, then the deferred application of t_b might never be needed. The space cost of its representation can be reclaimed by curtailing the alternation: $!(t_a \mid t_b);\ t_c$. In essence, one does not say how to search, but one may wish to say where not to search.

4.3.0.10. Skip Skip always succeeds without doing anything.

$skip\ s = [s]$

The given success is the sole success returned. Skip provides a way of saying don't do any more.

4.3.0.11. Fail Fail always produces an empty list of successes.

$fail\ s = [\,]$

Fail provides a way of saying that the search has gone down a route that is now known to be unwanted.

4.3.0.12. Recursion Recursion allows a tactic to apply itself repeatedly.

$(rec\ j \bullet t)\ s = t[(rec\ j \bullet t)/j]\ s$

Any reference to j within t is replaced by the whole tactic $rec\ j \bullet t$. The apparent infiniteness can be avoided in the implementation either by creating a circular structure or by copying lazily.

5. Tactic definitions

A tactic definition introduces a named tactic. It associates names, called jokers, with the argument values to which the tactic is applied.

5.1. Syntax

In this syntax, the following are lexical tokens: *TACTIC*, *expr*, *pred*, *decl*, *goal*, *exprs*, *decls*, *stxt*, *name*, *names*, *type*, *decn*. Their LaTeX mark-up and troff mark-up are defined in the reference manual. These need not be reserved keywords of the Z notation, as they are used in distinct contexts. (CADiZ's lexer uses a separate table of keywords for the tactic language.)

$$NamedTacticDef \quad = \quad TACTIC, NAME, TacticDef;$$

$$TacticDef \quad = \quad TypedJoker, \{; \ , TypedJoker\}, |, Tactic;$$

$$
\begin{array}{rcl}
TypedJoker & = & expr, NAME, \{, , NAME\} \\
& | & pred, NAME, \{, , NAME\} \\
& | & decl, NAME, \{, , NAME\} \\
& | & goal, NAME, \{, , NAME\} \\
& | & exprs, NAME, \{, , NAME\} \\
& | & decls, NAME, \{, , NAME\} \\
& | & stxt, NAME, \{, , NAME\} \\
& | & name, NAME, \{, , NAME\} \\
& | & names, NAME, \{, , NAME\} \\
& | & type, NAME, \{, , NAME\} \\
& | & decn, NAME, \{, , NAME\} \\
& ; &
\end{array}
$$

The *Tactic* syntax includes all the tactic combinators discussed above, and some other notations discussed below. *NamedTacticDef*s can be written in Z specification documents. A *TacticDef* can also be written in a separate file, where the name of the file serves as the name of the tactic.

5.2. Jokers

Each joker is declared to be of a type corresponding to a particular syntactic category of the Z notation. A tactic is applicable to some arguments only if those arguments are Z formulae of types corresponding to the types of the declared jokers. The correspondence between joker types and Z notations[**?**] is as follows.

expr	*Expression*
pred	*Predicate*
decl	*Declaration or binding*
goal	*Goal*
exprs	*ExpressionList*
decls	*DeclPart or binding list*
stxt	*SchemaText*
name	*DeclName or RefName*
names	*list of names in schema hiding*
type	*Type*
decn	*Name of declaration*

5.3. A small example

TACTIC expandabsorb pred p | *“expansion” p*; *“absorption” p*

This example defines a tactic named *expandabsorb* that is applicable to a predicate p. It tries to apply the inference rules *expansion* and *absorption* in sequential composition to that predicate. Assuming that the *expansion* command is applicable to this particular predicate, the inference rule will be applied, and joker p will be rebound to the result of that inference. So the predicate that is given to the *absorption* command is the result of the *expansion* command, not the original predicate. This is not referentially transparent, though by twisting the meaning of reference, an approximation to the same can be had: the two references to p

refer to the formula at a particular position within the goal to which each inference rule or tactic is implicitly applied. Unfortunately, the description of the position isn't a constant either.

This problem of rebinding jokers whenever an inference rule is applied is particularly awkward to implement in combination with the lazy evaluation mechanism. An implementation is outlined in section **??**, after more of the tactic language has been presented.

6. Pattern matching

Many forms of pattern matching tactic are provided. Each must be applied to a success containing exactly one goal. Each takes a formula in that goal selected by an existing joker and matches it against one or more patterns. The patterns are written in the Z notation, extended to permit references to jokers. If a pattern matches, the jokers referred to by the pattern are bound to corresponding formulae within the selected formula, and a further tactic is then applied in the context of these additional jokers. Alternatively, if no pattern matches, the whole pattern match tactic fails. Jokers of type *goal* are not used in this context, as a goal is never part of a larger formula.

6.1. Syntax of patterns

Patterns are written using the Z notation itself, extended to allow use of jokers of corresponding types. In this syntax, the following are lexical tokens: *as*, *_expr*, *_pred*, *_decl*, *_exprs*, *_decls*, *_stxt*, *_name*, *_names*, *_type*. Their LaTeX mark-up and troff mark-up are defined in the reference manual. They are reserved keywords of the Z notation.

$$
\begin{array}{lcl}
Expression & = & ExprJoker \\
& | & (, ExprJoker,) \\
& | & ExprJoker, as, Expression \\
& | & \_expr \\
& | & DecnJoker \\
& | & all\ Expression\ productions\ of\ the\ standard\ syntax \\
& ; &
\end{array}
$$

$$
\begin{array}{lcl}
ExpressionList & = & ExprsJoker \\
& | & ExprsJoker, as, ExpressionList \\
& | & \_exprs \\
& | & all\ ExpressionList\ productions\ of\ the\ standard\ syntax \\
& ; &
\end{array}
$$

$$
\begin{array}{lcl}
Predicate & = & PredJoker \\
& | & (, PredJoker,) \\
& | & PredJoker, as, Predicate \\
& | & \_pred \\
& | & all\ Predicate\ productions\ of\ the\ standard\ syntax \\
& ; &
\end{array}
$$

$$
\begin{array}{lcl}
SchemaText & = & StxtJoker \\
& | & (, StxtJoker,) \\
& | & StxtJoker, as, SchemaText \\
& | & \_stxt \\
& | & all\ SchemaText\ productions\ of\ the\ standard\ syntax \\
& ; &
\end{array}
$$

$$
\begin{array}{lcl}
DeclPart & = & DeclsJoker \\
& | & (, DeclsJoker,) \\
& | & DeclsJoker, as, DeclPart \\
& | & \_decls \\
& | & all\ DeclPart\ productions\ of\ the\ standard\ syntax \\
& ; &
\end{array}
$$

$$
\begin{array}{lcl}
Declaration & = & DeclJoker \\
& | & (, DeclJoker,) \\
& | & DeclJoker, as, Declaration \\
& | & \_decl \\
& | & all\ Declaration\ productions\ of\ the\ standard\ syntax \\
& ; &
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{DeclName} & = & \mathit{NameJoker} \\
& | & \mathit{NameJoker}, \mathit{as}, \mathit{DeclName} \\
& | & \_\mathit{name} \\
& | & \mathit{all\ DeclName\ productions\ of\ the\ standard\ syntax} \\
& ; &
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{RefName} & = & \mathit{NameJoker} \\
& | & \mathit{NameJoker}, \mathit{as}, \mathit{RefName} \\
& | & \_\mathit{name} \\
& | & \mathit{all\ RefName\ productions\ of\ the\ standard\ syntax} \\
& ; &
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{NameList} & = & \mathit{NamesJoker} \\
& | & (, \mathit{NamesJoker},) \\
& | & \mathit{NamesJoker}, \mathit{as}, \mathit{NameList} \\
& | & \_\mathit{names} \\
& | & \mathit{all\ NameList\ productions\ of\ the\ standard\ syntax} \\
& ; &
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Type} & = & \mathit{TypeJoker} \\
& | & \_\mathit{type} \\
& | & \mathit{all\ Type\ productions\ of\ the\ standard\ syntax} \\
& ; &
\end{array}
$$

In the above syntax rules, parentheses around a joker request strong matching, as discussed below in section **??**. The *as* notation allows a joker to be associated with a formula as well as further jokers being associated with its components. The terminal symbols beginning with _ are wildcard patterns that always match;

they avoid having to introduce a named joker for a part of a formula that will not be referred to. For example, if p_1 and p_2 are predicate jokers, then the pattern $p_1 \mathit{as}\ \_\mathit{pred} \wedge p_2$ matches any conjunction predicate, binding p_1 to the entire conjunction and p_2 to the right conjunct.

The *DecnJoker* notation for an expression is explained below in the section on local definitions.

6.2. Matching expressions

An expression match tactic matches an expression within the current goal against one or more patterns. Patterns are preceded by declaration of new jokers, all of which must be bound by the pattern, and followed by a tactic to apply if the pattern matches. Here is an example.

$$
\begin{array}{l}
\mathit{match}\ e \\
:: \mathit{expr}\ s \mid \mathbb{P}\, s \bullet t_1 \\
:: \mathit{expr}\ f, a \mid f\ \ a \bullet t_2 \\
:: \mathit{name}\ n;\ \mathit{exprs}\ es \mid n[es] \bullet t_3 \\
:: \mid e \bullet t_4 \\
:: .
\end{array}
$$

Each case is introduced by the :: token. (The overloading in this paper of the :: symbol with the traditional functional notation for type declaration is unfortu-

nate.) The pattern starts after the | token and ends at the first • that cannot be recognised by the Z parser.

In the example, joker *e* is already bound to an expression. Because this is an expression joker, the patterns are parsed according to the *Expression* syntax. The patterns of the cases are matched in order from top to bottom, and the highest case with a matching pattern is chosen. The first case matches any powerset expression, binding joker *s* to the argument in the powerset expression. The second case matches any function application expression, binding joker *f* to the function and joker *a* to its argument. Applications of function operators are transformed within CADiℤ to this juxtaposition notation, allowing tactics to match function operator applications without being expressed in terms of particular function operators. (Z function operator applications are not curried, so an application to several arguments is actually an application to a single tuple.) The third case matches any generic instantiation expression, binding joker *n* to the name of the generic and joker *es* to the list of instantiating expressions. Instantiations of generic operators are transformed within CADiℤ to this square-bracketed notation, allowing tactics to match generic operator instantiations without being expressed in terms of particular generic operators. The fourth case is guaranteed to match—it is like a default. This is achieved by instantiating the pattern according to the existing binding of the joker before matching the pattern. Tactic t_4 is often *skip*, ensuring that the default behaviour of the *match* tactic is to do nothing. This *match* tactic will nevertheless fail if one of the earlier patterns matches, but the tactic in the matched case fails. This is because backtracking does not consider later cases in a *match*. (Perhaps it should; it can nevertheless be achieved by composing multiple matches having cases in different orders in alternation with

each other.)

6.3. Matching predicates

A predicate match tactic matches a predicate within the current goal against one or more patterns. It behaves in a similar way to an expression match tactic.

$$
\begin{array}{l}
match\ p \\
:: pred\ p \mid \neg\ p \bullet t_1 \\
:: pred\ q, r \mid q \land r \bullet t_2 \\
:: expr\ s \mid \_expr \in s \bullet t_3 \\
:: \mid p \bullet t_4 \\
:: .
\end{array}
$$

In the example, joker p is already bound to a predicate. Because this is a predicate joker, the patterns are parsed according to the *Predicate* syntax. The first case matches any logical negation predicate, binding joker p to the argument of the negation. The second case matches any logical conjunction predicate, binding joker q to the left conjunct and joker r to the right conjunct. The third case matches any relation predicate, binding joker s to the relation. Applications of relation operators are transformed within CADiℤ to membership predicates, allowing tactics to match relation operator applications without being expressed in terms of particular relation operators. It uses a wildcard to match any expression without binding a joker. The fourth case is a default case.

6.4. Matching other Z notation

Having seen examples of matching expressions and predicates, it suffices to say that expression lists, declaration lists, schema texts, declarations, name lists and types can all be matched similarly using the *match* notation: matching a joker of one of those types causes the patterns to be parsed according to the corresponding Z syntax. Matching at the goal level requires some additional tactic notation, as described next.

6.5. Matching antecedents

An antecedent pattern tactic binds a joker to the nth antecedent predicate.

$pat_{ante}\ pred\ p \mid 2 \bullet t$

In the example, joker p is bound to the second antecedent of the current goal. If the current goal has an insufficient number of antecedents, the tactic fails.

6.6. Matching consequents

A consequent pattern tactic binds a joker to the nth consequent predicate.

$pat_{cons}\ pred\ p \mid 1 \bullet t$

In the example, joker p is bound to the first consequent of the current goal. If the current goal has an insufficient number of consequents, the tactic fails.

Those proof steps that introduce new antecedents or consequents make these predicates be first in the lists, hence the number 1 is generally used in these tactics.

6.7. Matching goals

A goal pattern tactic matches the whole goal against a pattern.

$pat_{goal}\ pred\ p, q \mid \mid p \vdash ?\ q \bullet t$

In this example, the pattern ($\mid p \vdash ?\ q$) refers to two jokers. There could be additional declarations, antecedents and consequents in the goal beyond those listed in the pattern: the pattern is matched in all possible ways, and the resulting lists of successes are concatenated.

6.8. Strong versus weak matching

In all of the example patterns given above, the pattern refers only to new jokers that are to be bound to Z phrases. Patterns can also refer to jokers that were bound by earlier pattern matching and are still in the environment. Such patterns are instantiated, replacing those jokers with the Z phrases to which they are bound, before pattern matching proceeds as described above. These instantiation phrases can match either any identical Z phrase (so-called weak matching), or only the very same instance of the Z phrase to which the joker was bound (strong matching). The default behaviour is weak matching. Strong matching

is requested by enclosing the use of the joker in the pattern immediately within parentheses.

For example, if p is a predicate joker in the environment, then the following case of a match tactic

:: $pred\ q \mid (p) \Leftrightarrow q \bullet ...$

would match only those equivalences whose left operand is the particular instance of the predicate to which p is bound, whereas if the parentheses were omitted, then the pattern would match any equivalence whose left operand is a predicate that is identical to that bound to p.

The strong matching parentheses can be used with *expr*, *pred*, *stxt*, *decl*, *decls* and *names* jokers. They cannot be used with *goal* or *type* jokers, as there is no need. They cannot be used with *name* jokers, because of syntactic ambiguities. (Parentheses have the advantage of not reserving any words from use in Z.) They cannot be used with *exprs* jokers because no attempt to implement those has yet been made.

6.9. Default notation

The Z notation provides defaults for parts of some Z phrases. This allows the author to omit those parts; they are filled in by the CADiℤ parser and typechecker, allowing them to be manipulated during proofs, and are elided again if still appropriate during prettyprinting. The default notations are: the | part of a schema text, which defaults to | *true*; the • part of comprehensions, which defaults to

• characteristic tuple; and the instantiation list of a generic instantiation expression, which defaults to the carrier sets of the inferred types of the generic arguments. Patterns are themselves Z phrases and so can omit parts for which there are defaults. On the other hand, a pattern will still match if some of the defaults are written explicitly.

Parentheses cannot be matched explicitly: any parentheses serve to either override operator precedences or indicate strong matching.

7. A further example

The following tactic, *blowDecl*, simplifies a declaration, with the aid of an auxiliary *blowExpr* for simplifying expressions.

```
TACTIC blowDecl decl d |
    match d
        :: expr e | _name : e •
                "apply tactic" e "blowExpr";
                match e
                :: | {_stxt • _expr} •
                     | {_exprs} • "normalization" d
                :: | e • skip
                :: .
        :: expr e
                | _name == e •
                | e •
                     "apply tactic" e "blowExpr"
        :: .
```

The outermost *match*'s three patterns match : declarations, == declarations, and schema inclusion declarations respectively. All three involve an expression that is simplified first. The : case then checks to see if its expression has simplified to a set comprehension or a set extension. If it has, the declaration is normalized, generating a membership predicate that can be simplified later. Tactic *blowExpr* is not shown here, but it is interesting to note that it is mutually recursive with *blowDecl* via *blowStxt* and *blowDecls*, which together with *blowPred* and *blowExprs* serve to simplify core Z notation without unfolding explicitly defined notation. As an example, if *blowDecl* were applied to this declaration,

$i : \{x : \mathbb{A} \mid false \bullet x\}$

the set comprehension would simplify to an empty set, and the declaration would be normalized, resulting in the following.

$i : \mathbb{P}\,\mathbb{A} \mid i \in \{\}$

8. Other considerations

8.1. Local definitions

The let notation is provided for introducing and binding jokers to things that are not expressible using the pattern matching notation discussed above. The following example illustrates the various forms of local definition that can appear within let notation.

$$
\begin{array}{l}
\text{let } pred\ p1 == antecedent\ 1, \\
\quad pred\ p2 == consequent\ 2, \\
\quad expr\ e == parse\_expr\ \text{“}2 + 2\text{”}, \\
\quad exprs\ es == parse\_exprs\ \text{“}2, 3\text{”}, \\
\quad pred\ p3 == parse\_pred\ \text{“}2 \leq 3\text{”}, \\
\quad decl\ d == parse\_decl\ \text{“}x : \{1, 2, 3\}\text{”}, \\
\quad decls\ ds1 == parse\_decls\ \text{“}x == 3;\ y, z : \{1, 2, 3\}\text{”}, \\
\quad name\ n == parse\_name\ \text{“}x\text{”}, \\
\quad stxt\ s == parse\_stxt\ \text{“}y : \mathbb{Z} \mid y > 1\text{”}, \\
\quad decls\ ds2 == declsbefore\ d, \\
\quad decls\ ds3 == declsafter\ d \\
\quad type\ t == typeof\ e \\
\quad decl\ d == declof\ e \\
\quad decn\ d2 == decnof\ d1 \\
\qquad \bullet\ t
\end{array}
$$

antecedent binds a *pred* joker to the nth antecedent. It differs subtly from pat_{ante}, in that the joker is bound “by name” rather than “by value”—more is said about this in section **??**. Also, n can be negative, referencing the nth antecedent back from the end. Similarly, *consequent* differs subtly from pat_{cons}. *parse_expr*, *parse_exprs*, *parse_pred*, *parse_decl*, *parse_decls*, *parse_name*, and *parse_stxt* all bind a joker of corresponding type to a new formula, distinct from the current goal, by parsing a string literal. *declsbefore* binds a joker to a list of those declarations that precede a given declaration. (That list cannot be found by pattern matching because of the asymmetric representation used for lists.)

declsafter is provided for symmetry (even though the list of declarations it finds might be found by pattern matching). *typeof* binds a *type* joker to the type of the expression referred to by an *expr* joker. *declof* binds a *decl* joker to the declaration referred to by the reference expression referred to by an *expr* joker. *decnof* binds a *decn* joker to the given declaration, so that when subsequently used as an expression it is instantiated to a reference expression bound to that declaration.

8.2. Concatenations

A further notation for building new formulae is concatenation, denoted by $e_1 + +e_2$, where e_1 and e_2 are each either a string literal, a reference to a joker, the numeric address of a formula, or a nested ++ concatenation. Each part of the concatenation is converted to a string before the concatenation is performed. Such concatenations are written as arguments to inference rules that expect strings as arguments.

8.3. String jokers

In addition to the types *pred*, *expr* etc of jokers corresponding to formulae, there is a further type of joker called *string*. A string joker can be referred to whereever a string literal could be written. For example, this allows tactics to be written that are parametrised by the name of an inference rule or tactic to be applied, giving the tactic language some higher-order capability, allowing general purpose

traversal tactics to be written.

8.4. Maintaining joker bindings

Jokers are of particular types and should each be bound to a formula of that type. These formulae arise either from pattern matching against the current goal or from parsing strings. The meaning of a string depends on the environment in which it is parsed. Rather than parse strings at the time the let tactic is applied (when the environment could be only that of the whole goal), strings are instead parsed whenever the joker that is to be bound to the result is referenced. If that reference is as an argument to an inference rule, the environment is determined by that inference rule. If that reference is within a pattern, the environment is determined by that of the formula being matched against. The term "whenever" is used above to indicate call-by-name rather than call-by-need: the string is re-parsed for every reference to the corresponding joker. The same call-by-name treatment is used for all local definitions, not just the parsing of strings.

Jokers that are bound to formulae in goals have to be rebound to corresponding formulae in sub-goals on each inference. The present mechanism for maintaining joker bindings involves associating an environment of joker bindings with each goal. Before matching a pattern, any references to already bound jokers in the pattern are first instantiated according to the bindings of those jokers in the environment of the current goal. The pattern is then matched, and the bindings of new jokers created are added to the environment of the current goal.

Once the arguments to an inference rule have been obtained and found to be

satisfactory, the bindings in the environment associated with the current goal are distributed onto the abstract syntax nodes of the current goal. The inference rule is then applied, generating sub-goals with references to reused parts of the original goal. Where a formula has been replaced *in situ*, the replacement formula is given the same jokers as the original had. Each subgoal is then copied in its entirety (an aspect of CADiℤ that is superficially difficult to justify), inheriting the jokers bound to each of the subgoal's nodes. An environment is then associated with each subgoal by appending the bindings from each node, ignoring all but the first binding found for each joker (there could be several if the inference rule duplicated a formula).

By this mechanism, only those jokers that are bound to nodes present in the subgoal are retained, allowing others to be garbage collected. Bindings of jokers to text arguments are retained in a separate second environment associated with each goal. Jokers are explicitly unbound when they go out of scope, though lazy evaluation can cause this to be delayed.

8.5. Interactive selections

Whilst there are jokers for (almost) all syntactic categories of Z notation, not all syntactic categories can be selected interactively. Consider a schema text comprising a single name. That name is parsed as an *Expression*, that *Expression* as a schema inclusion *Declaration*, that *Declaration* as a singleton *DeclPart*, and that *DeclPart* as a *SchemaText*. All five of these occupy the same area on the screen. Although feedback can be given about which is currently selected,

selecting any particular one is tedious. The pain is reduced by not allowing names or *DeclParts* (or *ExpressionLists*) to be selected interactively. There are no inference rules for those, and so it is unlikely that a user would want to apply a tactic to them either. The existence of jokers for those types of notation allows manipulation of that notation by auxiliary tactics.

Interactive selections are usually contiguous pieces of text. If the text has been prettyprinted, as CADiℤ does, each selection can be approximated by a rectangle. An exception is declarations of multiple names, for example $x, y : \mathbb{N}$. This is viewed abstractly as two declarations, $x : \mathbb{N}$ and $y : \mathbb{N}$, the former of which does not appear as contiguous text in the concrete presentation. Since names cannot be selected, the selection of a declaration is indicated by highlighting just the name of the declaration, which is conveniently a contiguous rectangular piece of text sufficient to identify the declaration.

9. Further work

Types of jokers are already provided for most Z syntactic categories. Some lesser used ones remain to be done, for example renaming lists.

An *exprs* joker cannot be used with a Cartesian product expression, because of the latter's unusual syntax: some special notation will be needed.

Chained relations are similar to default notation ($a = b = c \implies a = b \land b = c$), but different in that the two copies of b must, if generic, be instantiated with the same arguments. The behaviour of the tactic language with respect to chained

relations has yet to be sorted out properly.

String jokers have provided a means of passing tactics as arguments to other tactics. It might also be useful to be able to pass patterns as arguments to tactics.

A more efficient implementation of joker rebinding would be desirable.

10. Conclusions

The inference rules in CADiℤ are a little unusual, their number and variety arising from the ability in the user interface to select any formula. The tactic combinators are effective and unsurprising. The pattern matching notation, based on the Z syntax, is quite attractive from the user perspective, so long as the abstract syntax used by the tool is close to the concrete syntax. However, its effect on the transformation advantages exhibited by Martin *et al* for Angel[**?**] have not been considered. The novel combination of lazy evaluation with environments of jokers has been quite successful, but is not entirely satisfactory. The lack of data type definition facilities is not a problem: Z notation and tactics suffice. The first-order functional language with specific combinators as primitives was a good starting point for the tactic language. String jokers suffice to provide higher-order capability, allowing reusable search strategies to be written.

11. Syntactic metalanguage

The metalanguage used in defining the syntax of tactics is the following subset of ISO/IEC 14977:1996[**?**].

Symbol	Definition
=	defines a non-terminal to be some syntax.
|	separates alternatives.
,	separates elements to be concatenated.
{ }	brackets notation to be repeated zero or more times.
[ ]	brackets optional notation.
' '	encloses terminal symbols.
;	terminates a definition.

The infix , operator binds more tightly than the infix | operator.

12. Semantic metalanguage

The metalanguage used in defining the semantics of tactics is a functional notation similar to Haskell[**?**].

The notation *name* = *type* introduces *name* as a synonym for *type*. The notation *name* :: *type* declares that the value of *name* is of the given *type*. In types, $[x]$ denotes a list whose elements are of type x, and $y \to z$ denotes a function from elements of type y to elements of type z.

Functions are defined by equations, with patterns on their left-hand sides and expressions on their right-hand sides. In both patterns and expressions, lists are either empty [ ] or non-empty $(x : xs)$, where x is the head element and xs is the tail of the list. Where a list is of known length, its elements are enumerated between square brackets, separated by commas, e.g. $[a, b, c]$. Elision ... is used where the length of the list is arbitrary.

Acknowledgements

Thanks to Sam Valentine for being the primary user of the tactic language, for providing examples, and for comments on this paper. The referees made very useful comments. The need for strong pattern matching was identified by Susan Stepney.

IT 22-Jan-2002