Towards a CalcCheck Manual

Wolfram Kahl
kahl@cas.mcmaster.ca
McMaster University, 1280 Main St. West, Hamilton, Ontario, Canada

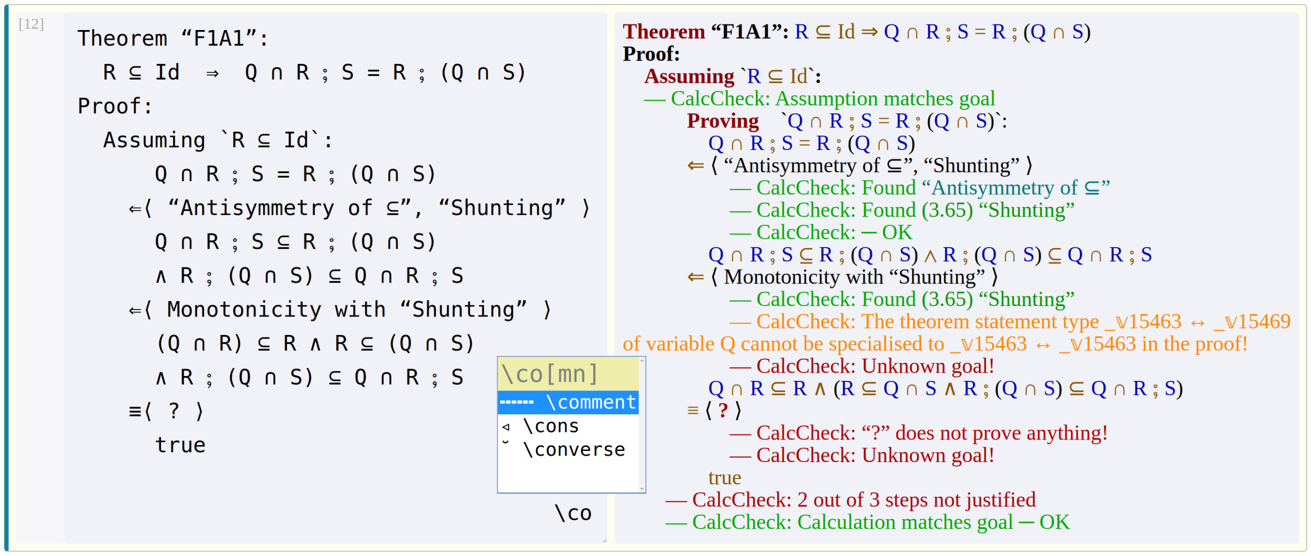
September 29, 2021

The proof checker CalcCheck has been developed for teaching calculational proofs in the style of Gries and Schneider's textbook classic "A Logical Approach to Discrete Math".

This currently very incomplete document will grow into a CalcCheck manual.

1 Introduction

One student will have seen essentially the situation depicted below in their web browser during an exam (except for the symbol input menu added for illustration, and the "\co" string associated with that).



```
[12]
Theorem “F1A1”:
  R ⊆ Id  ⇒  Q ∩ R ; S = R ; (Q ∩ S)
Proof:
  Assuming `R ⊆ Id`:
      Q ∩ R ; S = R ; (Q ∩ S)
    ⇐⟨ “Antisymmetry of ⊆”, “Shunting” ⟩
      Q ∩ R ; S ⊆ R ; (Q ∩ S)
      ∧ R ; (Q ∩ S) ⊆ Q ∩ R ; S
    ⇐⟨ Monotonicity with “Shunting” ⟩
      (Q ∩ R) ⊆ R ∧ R ⊆ (Q ∩ S)
      ∧ R ; (Q ∩ S) ⊆ Q ∩ R ; S
    ≡⟨ ? ⟩
      true
                                    \co
```

```
Theorem “F1A1”: R ⊆ Id ⇒ Q ∩ R ; S = R ; (Q ∩ S)
Proof:
  Assuming `R ⊆ Id`:
  — CalcCheck: Assumption matches goal
      Proving   `Q ∩ R ; S = R ; (Q ∩ S)`:
        Q ∩ R ; S = R ; (Q ∩ S)
      ⇐ ⟨ “Antisymmetry of ⊆”, “Shunting” ⟩
          — CalcCheck: Found “Antisymmetry of ⊆”
          — CalcCheck: Found (3.65) “Shunting”
          — CalcCheck: — OK
        Q ∩ R ; S ⊆ R ; (Q ∩ S) ∧ R ; (Q ∩ S) ⊆ Q ∩ R ; S
      ⇐ ⟨ Monotonicity with “Shunting” ⟩
          — CalcCheck: Found (3.65) “Shunting”
          — CalcCheck: The theorem statement type _𝕍15463 ↔ _𝕍15469
of variable Q cannot be specialised to _𝕍15463 ↔ _𝕍15463 in the proof!
          — CalcCheck: Unknown goal!
        Q ∩ R ⊆ R ∧ (R ⊆ Q ∩ S ∧ R ; (Q ∩ S) ⊆ Q ∩ R ; S)
      ≡ ⟨ ? ⟩
          — CalcCheck: “?” does not prove anything!
          — CalcCheck: Unknown goal!
        true
    — CalcCheck: 2 out of 3 steps not justified
    — CalcCheck: Calculation matches goal — OK
```

Via the "web application" interface CalcCheck$_{\text{Web}}$, users are presented with a page consisting of a sequence of "cells" such as the one displayed above, where the large area to the left is a text input area that supports Unicode input via a symbol completion mechanism that accepts, in most cases, common LaTeX macro names for symbols at least as one alternative. Users can request proof checking (or just syntax checking) from the server; this populates the area to the right with feedback, presented as an annotated version of the parts of the input that could be parsed (or as a parse error message if nothing could be parsed).

Some documentation and additional material, including live CalcCheck$_{\text{Web}}$ notebooks for experimentation, are available at the CalcCheck home page at http://CalcCheck.McMaster.ca/.

2 General Remarks

Throughout this document, we will present CalcCheck source in its automatically-generated LaTeX rendering — this rendering is intended to precisely represent the concrete syntax of CalcCheck source, as it can be seen on the left side of the screenshot in the introduction, while making it more readable via font choices that come closer to conventional mathematical presentation. Since proportional fonts are used, only alignments generated by indentation spacing are preserved; internal spacing is rendered as hard spaces in LaTeX, so that "spaces count", but no attempt is made to preserve alignment achieved via spacing after the initial indentation. (This LaTeX rendering in particular does not attempt to achieve traditional mathematical formula layout and spacing.)

CalcCheck syntax is layout-sensitive; larger-scale structures, including '**Theorem**', '**Proof**', '**Using**', etc., typically require all their content to be indented at least two additional spaces. Calculation step expressions also need to be indented at least two spaces more than the calculation operators, and all calculation operators of any one calculation need to be aligned in the same column. Theorem statement expressions and the expressions forming calculation steps need no delimitation beyond that provided by these layout rules; expressions in other positions are delimited by backticks (` ... `) following the conventions of Markdown [Gruber, 2004, Leonard, 2016a].

Markdown conventions also apply in the documentation sections of CalcCheck notebooks, where the Pandoc flavour [MacFarlane, 2006, Leonard, 2016b] of Markdown can be used.

While TeX math mode is designed for mathematical formulae mostly containing single-letter and single-symbol identifiers, CalcCheck is designed to work well with multi-letter identifiers, and even multi-symbol identifiers; CalcCheck therefore demands that most lexemes are separated by spaces.

The sequence "▪▪▪▪▪▪ " (obtained via the keyboard shortcut `\comment`) starts a line comment, but this is not yet accepted in all places.

3 Operator Precedences and Associating Conventions

The expression languages used in CalcCheck notbooks are mostly user-defined, with very few hard-coded primitives.

Operators are introduced by declaring a precedence for them, and optionally also an associating convention, for example:

> **Precedence** 40 **for:** $\_\wedge\_$, $\_\vee\_$
> **Associating to the right:** $\_\wedge\_$, $\_\vee\_$

The first line above declares the same precedence, namely 40, for the conjunction $\wedge$ and disjunction $\vee$ operators, following LADM. This decision, which enforces parenthesisation, is justified in LADM by the desire to avoid confusion. With the precedence declaration above in scope, CalcCheck produces, for the input '$p \ \wedge \ q \ \vee \ r$', the parse error message "Missing parentheses: $\_\wedge\_$ does not associate with $\_\vee\_$", prodding users to be explicit whether they wanted to write '$(p \ \wedge \ q) \ \vee \ r$' or '$p \ \wedge \ (q \ \vee \ r)$'.

Without the "Associating" line above, a similar message would be produced for the input '$p \ \wedge \ q \ \wedge \ r$'; due to that "Associating" line, '$p \ \wedge \ q \ \wedge \ r$' is parsed as '$p \ \wedge \ (q \ \wedge \ r)$'.

Here is the complete table of precedences for the CalcCheck library used in recent courses; this is only slightly modified from that of LADM. We omit the underscores for the argument positions of visible binary infix operators; note also that the symbol “≔” used for textual substitution is the single Unicode codepoint “COLON EQUALS”. Only textual substitution and quantification notation are part of the hardcoded CalcCheck expression syntax; everything else in this table is user-defined.

- (∞): _[_ ≔_] (textual substitution) **(highest precedence)**
- 140: unary postfix operators: _! _˘ _* _⁺ _⦇_⦈
- 130: unary prefix operators: +_ −_ ¬_ #_ ~_ ℙ_ suc_
- 120: __ (function application), @
- 115: **
- 110: · / ÷ mod gcd
- 105: ⨾ ╱ ╲
- 100: + − ∪ ∩ × ∘ ⊕ ⇨ ◁ ⩤ ▷ ⩥
- 97: ↔ (relation type)
- 95: → (function type)
- 90: ↓ ↑
- 70: #
- 60: ◃ ▹ ⌢
- 50: = ≠ < > ∈ ⊂ ⊆ ⊃ ⊇ | _⦗_⦘_ **(conjunctional)**
- 40: ∨ ∧
- 20: ⇒ ⇏ ⇐ ⇍
- 10: ≡ ≢
- 9: := (assignment command, two characters)
- 5: ; (command sequencing)
- (−∞): ⊛_|_•_ (quantification notation, for ⊛ ∈ {∀, ∃, ⋃, ⋂, ∑, ∏, …}) **(lowest precedence)**

All non-associative binary infix operators associate to the left, except **, ◃, ⇒, →, ⇨, ×, which associate to the right.

LADM does permit some combinations of same-precedence operators, in particular ‘$a + b - c$’. For dealing with this correctly, the CalcCheck setup for this row of the precedence table contains operator combinations that associate ‘with’ each other:

Precedence 100 **for:**
_+_ , _−_ , _∩_ , _∪_ , _×_ , _⊍_ , _∘_ , _➝_ , _⇨_ , _⊕_ , _⊗_ , _⊖_ , _⊛_
Associating to the right: _∩_ , _∪_ , _×_ , _⊍_ , _∘_ , _⇨_ , _➝_
Associating to the left: _+_ , _−_ , _+_ with _−_ , _−_ with _+_

The first of these specifies that ‘$a + b - c = (a + b) - c$’, and the second specifies that ‘$a - b + c = (a - b) + c$’.

For infix, prefix, and postfix operator symbols, precedences and associating conventions need to be established before the operators can be declared (given a type). This corresponds to the practice of LADM, where the precedence table is reproduced on the inside cover, and some symbols are used at different types in different chapters.

Associative operators still need an associating convention — the following makes it possible to write ‘$p \equiv q \not\equiv r$’ without parentheses although equivalence ≡ and inequivalence ≢ have the same precedence:

Precedence 10 **for:** $\_ \equiv \_$, $\_ \not\equiv \_$
Associating to the right: $\_ \equiv \_$, $\_ \not\equiv \_$, $\_ \equiv \_$ with $\_ \not\equiv \_$, $\_ \not\equiv \_$ with $\_ \equiv \_$

(In fact, it is important to remember that the associating conventions only make a difference for parsing, by allowing to omit certain parentheses, and have nothing to do with associativity properties.)

However, since CalcCheck currently does not support matching up to mutual associativity laws, for reasoning about '$p \equiv q \not\equiv r$' it is relevant to know that it denotes '$p \equiv (q \not\equiv r)$', and not '$(p \equiv q) \not\equiv r$'.

(All these notational choices could be changed by using different preloaded declarations in CalcCheck.)

LADM also supports *conjunctional* operators (see the table of precedences above); these include $\_=\_$ and $\_\subseteq\_$; this means that "$Q = R \subseteq S$" is considered as short-hand for the conjunction "$Q = R \wedge R \subseteq S$". CalcCheck allows also more-than-binary infix operators to be declared as conjunctional; we use this for "$\_ \llparenthesis \_ \rrparenthesis \_$", which is just a user-defined mixfix operator. Conjunctionality then allows us to write "$a \llparenthesis R \rrparenthesis b \llparenthesis S \rrparenthesis c$" instead of "$a \llparenthesis R \rrparenthesis b \wedge b \llparenthesis S \rrparenthesis c$".

4 Quantification in CalcCheck

For quantification, CalcCheck follows the spirit of LADM, but the (currently hard-wired) concrete syntax is closer to the Z notation [Spivey, 1989]: The general pattern of quantified expressions is:

$$\mathit{bigOp}\ \mathit{varDecls}\ \mid\ \mathit{rangePredicate}\ \bullet\ \mathit{body}$$

The shorter form "$\mathit{bigOp}\ \mathit{varDecls}\ \bullet\ \mathit{body}$" is an abbreviation where the range predicate defaults to *true*. The quantifying "big operators" are user-declared; we are working with the following set-up:

Big operator for $\_+\_$ **is** $\sum$
Big operator for $\_\cdot\_$ **is** $\prod$
Big operator for $\_\wedge\_$ **is** $\forall$
Big operator for $\_\vee\_$ **is** $\exists$
Big operator for $\_\cap\_$ **is** $\bigcap$
Big operator for $\_\cup\_$ **is** $\bigcup$

As an example for calculational quantifier reasoning, we show a proof for "Change of dummy via ~" — this allows us to "replace quantification over relations with quantification over relation complements" and is a special case of the LADM theorem "Change of dummy" of which we showed a version in Kahl [2018]. In the theorem statement expression,

$$(\forall\ x \mid R \bullet P)\ \equiv\ (\forall\ y \mid R[x := \sim y] \bullet P[x := \sim y])\ ,$$

the variables R and P are recognised as *metavariables* since they occur in scope of variable binders (here, universal quantification and substitution). In particular, all occurrences of R and P are within scope of a binder for x, so they may be instantiated with expressions containing x as free variable. However, the second occurrences of R and P are within scope of a binder for y, while their first occurrences are not within scope of a binder for y, so CalcCheck derives what LADM calls a "$\neg$*occurs* proviso"; this is by default reported in the feedback, and it is possible to let CalcCheck also report the allowed occurrences of *bound-variable-metavariables* (here, 'x') in expression metavariables:

— CalcCheck: Metavariables: P = P⟦ x ⟧ , R = R⟦ x ⟧
— CalcCheck: Proviso: ¬*occurs*(`y`, `P, R`)

Gries [1997] calls proving "theorem schemas" involving such metavariables "schematic proofs of schematic formulas", but then proposes to relax terminology and still just talk about "theorems", which is essentially the approach already adopted by Gries and Schneider [1993] in LADM. CalcCheck supports this approach to metavariables, which is quite different from the way most current proof assistants require formalisations to be done, including for example Agda [Norell, 2007], Coq [Bertot and Castéran, 2004], Isabelle/HOL [Nipkow et al., 2002], Lean [de Moura et al., 2015], Mizar [Grabowski et al., 2010], or PVS [Owre et al., 1999]: None of these include metavariables of this kind in their theory languages.

In the hints of the proof of "Change of dummy via ~", we list all theorem expressions explicitly upon first occurrence (for information only; the proof is accepted also without):

Theorem "Change of dummy via ~ ": $(\forall\ x \mid R \bullet P) \ \equiv\ (\forall\ y \mid R[x := \sim y] \bullet P[x := \sim y])$
Proof:

$(\forall\ y \mid R[x := \sim y] \bullet P[x := \sim y])$
$\equiv\langle$ "One-point rule for ∀" \`$(\forall\ x \mid x = E \bullet P) \ \equiv\ P[x := E]$\` $\rangle$
$(\forall\ y \mid R[x := \sim y] \bullet (\forall\ x \mid x = \sim y \bullet P))$
$\equiv\langle$ "Nesting for ∀" \`$(\forall\ x \mid R \bullet (\forall\ y \mid S \bullet P)) \ \equiv\ (\forall\ x, y \mid R \wedge S \bullet P)$\` $\rangle$
$(\forall\ y, x \mid R[x := \sim y] \wedge x = \sim y \bullet P)$
$\equiv\langle$ Substitution $\rangle$
$(\forall\ y, x \mid R[x := z][z := \sim y] \wedge x = \sim y \bullet P)$
$\equiv\langle$ "Replacement" (3.84a) \`$(e = f) \wedge E[z := e] \ \equiv\ (e = f) \wedge E[z := f]$\` $\rangle$
$(\forall\ y, x \mid R[x := z][z := x] \wedge x = \sim y \bullet P)$
$\equiv\langle$ Substitution $\rangle$
$(\forall\ y, x \mid R \wedge x = \sim y \bullet P)$
$\equiv\langle$ "Dummy list permutation for ∀" \`$(\forall\ x, y \mid R \bullet P) \ \equiv\ (\forall\ y, x \mid R \bullet P)$\` $\rangle$
$(\forall\ x, y \mid R \wedge x = \sim y \bullet P)$
$\equiv\langle$ "Nesting for ∀" $\rangle$
$(\forall\ x \mid R \bullet (\forall\ y \mid x = \sim y \bullet P))$
$\equiv\langle$ "Equality with ~" \`$Q = \sim R \ \equiv\ R = \sim Q$\` $\rangle$
$(\forall\ x \mid R \bullet (\forall\ y \mid y = \sim x \bullet P))$
$\equiv\langle$ "One-point rule for ∀" $\rangle$
$(\forall\ x \mid R \bullet P[y := \sim x])$
$\equiv\langle$ Substitution — since $\neg$*occurs*(\`y\`,\`P\`) $\rangle$
$(\forall\ x \mid R \bullet P)$

(The last hint here contains an "in-hint comment" introduced by '— '.)

∀-introduction can be seen as constituting one direction of Metatheorem (9.16) of LADM: "P is a theorem iff $(\forall x \bullet P)$ is a theorem", with instantiation constituting the other direction. LADM then uses Metatheorem (9.16) as a proof method corresponding to ∀-introduction; CalcCheck has a special proof construction for this: A proof for a universal quantification $(\forall\ x \bullet P)$ by ∀-introduction is presented in CalcCheck as the heading "**For any** \`$x$\`:" followed by a proof for P, or, using a different variable and writing "**For any** \`$y$\`:" followed by a proof for $P[x := y]$ (where this substitution does not need to be written explicitly). In the context of LADM this is justified by metatheorem (9.16): "P is a theorem iff $(\forall\ x \bullet P)$ is a theorem."

Consider the following theorem:

Theorem (9.21) "Distributivity of ∧ over ∃": $P \wedge (\exists\ x \mid R \bullet Q) \ \equiv\ (\exists\ x \mid R \bullet P \wedge Q)$

In LADM, this comes with the *proviso* "¬*occurs*('x', 'P')" — technically, this is therefore again a metatheorem, with metavariables P, Q, and R standing for expressions and metavariable x standing for a variable; the proviso means that theoremhood is asserted only for cases where P is instantiated with an expression in which the instantiation of x does not occur free. As mentioned before the proof of "Change of dummy via $\sim$", CalcCheck derives such provisos automatically from the theorem statement (and reports them as part of the feed-back); matching fails where such provisos are violated.

The calculation format *as such* has to be considered as conjunctional, since in particular implication chains as above are allowed by the "relaxed proof style" in LADM Chapter 4, which does not consider the implication operator $\Rightarrow$ as conjunctional. (Demanding that only conjunctional operators could be used as calculation operators would forbid such implication chains.) Indeed, where LADM introduces the calculation format in Chapter 1 on p. 15, it translates the calculation into a (meta-level) conjunction without invoking conjunctionality of the operator "$=$". In CalcCheck, we therefore just define the calculational presentation to be conjunctional *as such*, see Sect. 5.4. Therefore, in calculations where the expressions are of type $\mathbb{B}$, the two operators "$\equiv$" and "$=$" can be used interchangeably as calculation operators; we prefer to use "$\equiv$" since it documents the type $\mathbb{B}$.

The proof of the equivalence of the relation-algebraic and predicate-logical definitions of transitivity by Schmidt and Ströhlein [1993, p. 28] translates directly into CalcCheck — we only "saved" two additional (mutually inverse) applications of De Morgan laws:

Theorem "Transitivity":
$\textsf{is-transitive}\ R \quad\equiv\quad \forall\, x \bullet \forall\, y \bullet \forall\, z \bullet\ x \llparenthesis R \rrparenthesis y \llparenthesis R \rrparenthesis z \ \Rightarrow\ x \llparenthesis R \rrparenthesis z$
Proof:
$\quad \textsf{is-transitive}\ R$
$\equiv\langle$ "Definition of transitivity" $\rangle$
$\quad R \fatsemi R \subseteq R$
$\equiv\langle$ "Relation inclusion", "Relation composition" $\rangle$
$\quad \forall\, x \bullet \forall\, z \bullet (\exists\, y \bullet x \llparenthesis R \rrparenthesis y \llparenthesis R \rrparenthesis z) \Rightarrow x \llparenthesis R \rrparenthesis z$
$\equiv\langle$ "Witness" \`$(\exists\, x \bullet P) \Rightarrow Q \ \equiv\ (\forall\, x \bullet P \Rightarrow Q)$\` $\rangle$
$\quad \forall\, x \bullet \forall\, z \bullet \forall\, y \bullet x \llparenthesis R \rrparenthesis y \llparenthesis R \rrparenthesis z \Rightarrow x \llparenthesis R \rrparenthesis z$
$\equiv\langle$ "Interchange of dummies for $\forall$" $\rangle$
$\quad \forall\, x \bullet \forall\, y \bullet \forall\, z \bullet x \llparenthesis R \rrparenthesis y \llparenthesis R \rrparenthesis z \Rightarrow x \llparenthesis R \rrparenthesis z$

In the third hint above, we juxtaposed two theorem references, one by theorem name "Witness", and one by theorem expression — this pattern frequently serves documentation in LADM; in CalcCheck this is defined as referring to the intersection of the sets of theorems referred to by the individual references (using only expressions as stand-alone theorem references is by default disabled for pedagogical reasons: Some students would stop remembering theorem names, and start writing random expressions in hope to hit on a theorem).

5 Quick Systematic Overview of the CalcCheck Language

In the diagram below, the main syntactic categories of the CalcCheck language are indicated — full arrows denote the "are immediate constituents of" relation, and dotted arrows the "may be immediate constituents of" relation. A CalcCheck module currently just consists of a sequence of "top-level items" (TLIs); CalcCheck notebooks used for teaching are technically module "suffixes" that may not contain module import declarations; these and arbitrary other TLIs may be contained in the preloaded "prefix".

$$\begin{tikzcd}
 & \text{Top-level items} & \\
 & \text{Proofs} \arrow[u, dotted] \arrow[dr, dotted] & \\
\text{Calculations} \arrow[ur, dotted] & \text{Hints} \arrow[l] \arrow[u, dotted] & \text{Hint items} \arrow[l, "+"'] \\
\text{Expressions} \arrow[u] \arrow[rr] & & \text{Substitutions} \arrow[u, dotted]
\end{tikzcd}$$

5.1 CalcCheck Theory Content: Top-Level Items (TLIs)

The most prominent kind of TLI is introduced by one of the (completely interchangeable) keywords ‘**Theorem**’, ‘**Lemma**’, ‘**Proposition**’, ‘**Corollary**’, and ‘**Fact**’, with the following syntax:

Theorem $\{$*ThmName* $|$ *ThmNum*$\}^*$: *expression*
Proof: *proof*

Each theorem can have any number of theorem numbers (in parentheses) or theorem names (in *pretty double quotation marks* “. . . ”). The same theorem name or number may also be used in several theorems — this corresponds to the practice in LADM. Lack of a proof is flagged as an error.

The same syntax as for **Theorem**s is also used for **Axiom**s, except that for an **Axiom**, presence of a proof is flagged as an error.

Currently, CalcCheck (like LADM) is lacking support for formal definition principles (like inductive datatypes or definition by primitive recursion); new identifiers are introduced using **Declaration**s and may in addition be provided **Explanation**s; they are then defined using axioms, as in the following example:

Declaration: $\_\setminus\_ : A \leftrightarrow B \;\to\; A \leftrightarrow C \;\to\; B \leftrightarrow C$
Explanation: $R \setminus S$:= “The right-residual of `$S$` by `$R$` ”
Explanation: $R \setminus S$:= “Left-division of `$S$` by `$R$` ”
Explanation: $R \setminus S$:= “`$R$` under `$S$` ”

Axiom “Characterisation of right residual” “Characterisation of $\setminus$”: $X \subseteq R \setminus S \;\equiv\; R \fatsemi X \subseteq S$

Operator declarations such as the ones given above actually need to be preceded by precedence declarations (and possibly declarations specifying association to the left or right) which are then valid for all operators using the same name — this, too, corresponds to the practice of LADM, which features a single precedence table on its insider cover.

For relation composition and residuals, we fix the same precedence, but higher than that of intersection and union, and to avoid potential confusion we declare the residual operators as non-associating, so that nested applications always require parentheses:

Precedence 105 **for:** $\_\fatsemi\_$, $\_/\_$, $\_\setminus\_$
Associating to the right: $\_\fatsemi\_$
Non-associating: $\_/\_$, $\_\setminus\_$

TLIs of shape

Activate $\{$**associativity** $|$ **symmetry** $|$ **monotonicity** $|$ . . .$\}$ **property** *ThmRef* are used to enable features the soundness of which depends on the respective theorems:

- ‘**associativity**’ and ‘**symmetry**’: CalcCheck applies theorems via rewriting; matching up to associativity and/or symmetry is anabled by activating these theorems. (This is independent of parsing, which is determined by the ‘**Associating**’ det-up described above.
- ‘**transitivity**’ enables combinations of possibly different calculation operators.
- ‘**antisymmetry**’ enables ccyclic calculations for proving (conjunctions of) equalities.
- ‘**reflexivity**’ makes it possible to use equality calculations to prove, e.g., inclusions.
- ‘**converse**’ properties enaples the use of, for example, laws formulated in terms of $\Rightarrow$ to prove goals, in particular calculation steps, formulated in terms of $\Leftarrow$.
- ‘**lower-bound**’ and ‘**upper-bound**’ properties serve to recognise “trivial calculations”, in particular long calculations establishing goals of shape ‘$p \Rightarrow true$’.
- ‘**monotonicity**’ and ‘**antitonicity**’ set up properties for use with the keyword hint-items ‘Monotonicity’ and ‘Antitonicity’, see below in Sect. 5.3.

A TLI of shape

Register {**type** | **operator**} *Identifier* **for built-in** *BuiltIn*

associates user-defined operators with built-in functionality; this provides in particular an interface to integer and natural-number arithmetic.

Finally, **MarkDown** TLIs accommodate documentation using Pandoc-flavoured Markdown [MacFarlane, 2006].

5.2 Expression Syntax

The expression language of LADM relies to a very large part on mixfix operators, and uses function application only sporadically. In CalcCheck, mixfix operators can be declared with underscores indicating argument positions; mixfix operators that act as infix, prefix, or postfix operators have to be assigned a single precedence level and associating convention as explained above.

The only hard-coded special expression syntax is the quantification syntax explained in Sect. 4.

The infix function type constructor _→_ is currently hard-coded. Function application syntax by juxtaposition ‘$f\ a$’ is also hard-coded into the parser, but has to be enabled via a declaration before it can be used. Therefore, alternate function application syntax can be defined, for example via the following:

Precedence 120 **for:** _._
Associating to the left: _._
Declaration: _._ : $(A \rightarrow B) \rightarrow A \rightarrow B$

This could the either be used exclusively as syntax for function application ‘$f\ .\ a$’, as proposed by Aarts, Backhouse, et al. (1992), or besides juxtaposition, as in LADM. However, the rule of LADM that ‘.’ is used only for function application to atomic (variable or constant) arguments can currently not be enforced by CalcCheck. (The CalcCheck-based courses so far use only juxtaposition for function application, since this is the convention used by most modern theorem provers and functional programming languages.)

5.3 Hint Items

The hints in calculations are sequences of hint items separated by comma or the keyword 'and'; in nested hints, 'and' has lower precedence than 'with', and comma has higher precedence. We have already seen most kinds of hints; for a quick summary, hint items can have the following shapes:

1. $\mathit{ThmRef} ::= \{\mathit{ThmNum} \mid \mathit{ThmName} \mid `\,\mathit{expr}\,`\}^+$

   A non-empty sequence of theorem names or theorem numbers (and, but not only, backticked expressions), denoting the intersection of their theorems.

2. $`\,\mathit{var}_1, \ldots, \mathit{var}_n := \mathit{expr}_1, \ldots, \mathit{expr}_n\,`$

   Explicit substitution of theorem variables can be specified only after 'with', typically preceded by a single theorem reference:

   ThmRef with $`\,x_1, \ldots, x_n := e_1, \ldots, e_n\,`$

   Instead of the (typically single) theorem referred to by *ThmRef*, the result of applying the listed substitution to that theorem is used.

3. Assumption $`\,\mathit{expr}\,`$ or Assumption *ThmRef*

   These refer to antecedents separated from the proof by '**Assuming**', see below.

4. Local property $`\,\mathit{expr}\,`$ or Local property *ThmRef*

   These refer to properties shown in a '**Side proof**', see below.

5. Definition of $`\,\mathit{identifier}\,`$

   These refer to names defined in a '**Local definition:**', see below.

6. Substitution

   This keyword hint item performs substitutions $E[z := F]$, that is, it rewrites, for example, $(R \mathbin{⨾} S)[R := P \cap Q]$ to $((P \cap Q) \mathbin{⨾} S)$. The substitution mechanism is aware of metavariables and of $\neg \mathit{occurs}$ provisos.

7. Evaluation

   This evaluates ground (sub-)terms consisting only of number literals and operators **Register**ed for built-in operations.

8. Fact $`\,\mathit{expr}\,`$

   This justifies *expr* iff it is a ground Boolean expression that the 'Evaluation' mechanism can evaluate to *true*.

   The 'Evaluation' and 'Fact' hint items can be enabled and disabled separately for CalcCheck$_{\text{Web}}$ notebooks.

9. hi_0 with hi_1 and ... and hi_n

   The theorem(s) referred to by hi_0 can be rewritten by equations extracted from hi_1, ..., hi_n, or these can be used to discharge antecedents in an application of a hi_0 theorem via conditional rewriting. Since these two mechanisms overlap, there are cases where 'hi_1 with hi_2' works just as well as 'hi_2 with hi_1', and then it is up to the proof author to chose which way "reads" better in their opinion. But in general, neither mechanism is symmetric, in particular since only rewriting with one-directional matching is used in both mechanisms, but no narrowing, and no unification.

   If hi_1, ..., hi_n contain a substitution (as described under 2. above), then the theorem referred to by hi_0 is instantiated via that substitution first.

   When a non-empty suffix of the argument hint-item list consists only of assumptions, then instead of 'assumption \`$E_1$\` and assumption \`$E_2$\`' one may simply write 'assumptions \`$E_1$\` and \`$E_2$\`', only with 'and' (and analogously for 'local properties').

10. Monotonicity, Antitonicity

    These can only occur before 'with' and invoke non-empty sequences of activated monotonicity rules; 'Monotonicity' will also work where (in addition) even numbers of activated antitonicity rules are used, and 'Antitonicity' with odd numbers of antitonicity rules.

11. Induction hypothesis {\`*expr*\`}$^?$

    This is useful in the '**Induction step**' part of **By induction**' proofs only, and refers to the assumption available for the induction step — this is normally the goal expression of the induction proof.

    In nested induction steps, supplying the precise induction hypothesis used via \`*expr*\` is by default required, but this requirement can be disabled.

12. SubProof{ for \`*expr*\`}$^?$:
        proof

    Such subproof hint items start their own layout blocks. The goal can be omitted if CalcCheck can infer it from the proof, see Sect. 5.4.

    Such subproof hint items also occur directly as constituents in the **Using** and **With** proof constructs, see Sect. 5.4.

5.4 Proofs

While we explain the different proof structures available in CalcCheck, we also provide information about where and how proof goals may be inferred.

1. Calculations as proofs do not need to be introduced by any keyword.

   The calculational presentation *as such* is conjunctional, so that, no matter what the precedences and associating conventions for the involved operators are, a calculation of shape

$$\begin{array}{ll} & E_1 \\ \circ\!\!-\!\!\bullet & \langle\ \text{Hint}_1\ \rangle \\ & E_2 \\ = & \langle\ \text{Hint}_1\ \rangle \\ & E_3 \\ \bullet\!\!-\!\!\circ & \langle\ \text{Hint}_1\ \rangle \\ & E_4 \end{array}$$

always denotes a conjunction chain:

$$(E_1 \;\circ\!\!-\!\!\bullet\; E_2) \quad\wedge\quad (E_2 = E_3) \quad\wedge\quad (E_3 \;\bullet\!\!-\!\!\circ\; E_4)$$

To have this calculation accepted as proof for '$E_1 \;-\!\!\in\; E_4$' requires the previous activation of a transitivity law '$(x \;\circ\!\!-\!\!\bullet\; y) \Rightarrow (y \;\bullet\!\!-\!\!\circ\; z) \Rightarrow (x \;-\!\!\in\; z)$' together with registration of '=' as built-in equality (or appropriate additional transitivity laws).

If, via such transitivity, a calculation proves one of '$P \equiv \mathit{true}$', '$P \Leftarrow \mathit{true}$', '$\mathit{true} \equiv P$', or '$\mathit{true} \Rightarrow P$', then the inferred proof goal is P. Otherwise, if via transitivity the calculation proves $P \;-\!\!\in\; Q$, then that is the inferred proof goal.

Either the first or last expression of a calculation may be followed by the key phrase '— **This is**' (consisting of an em-dash and the two words "This" and "is" separated by exactly single spaces) followed by a reference to a theorem, assumption, or local property; those cases are treated the same for inferring proof goals as replacing the corresponding calculation end with *true*.

As a special case of such a reference, 'an assumption' or 'a local property' may be used; this avoids duplication, since in these cases the relevant assumption or local property has already been stated before the '— **This is**'.

2. By *Hint*

   This kind of proof discharges simple proof obligations and is typically used with a small number of small hint items.

   CalcCheck cannot infer proof goals here.

3. **For any** `` ` ``$\mathit{var}\{ : \mathit{type}\}^?$`` ` ``:

   This header introduces the bound variable *var*; if the subproof following this header proves P, then the resulting proof establishes $(\forall\ \mathit{var} : \mathit{type} \bullet P)$, so that this structure corresponds to $\forall$-introduction. The *type* is optional, and the variable can even be renamed, since CalcCheck compares expressions internally only up to α-equivalence.

   This proof shape represents use of $\forall$-introduction; if the contained proof has inferred proof goal P, then the resulting proof has inferred proof goal $(\forall\ \mathit{var} : \mathit{type} \bullet P)$.

   An additional range predicate can be provided:

   For any `` ` ``$\mathit{var}\{ : \mathit{type}\}^?$`` ` `` **satisfying** `` ` ``*expr*`` ` ``:

   With this header, if the contained proof has inferred proof goal P, then the resulting proof has inferred proof goal $(\forall\ \mathit{var} : \mathit{type} \mid \mathit{expr} \bullet P)$.

4. **Assuming** *ThmId*$^?$ \``*expression*`\` :

   While LADM uses the keyword 'Assume' for the proof methof of "assuming the antecedent" (which corresponds to implication introduction), we chose the less imperative '**Assuming**'.

   Assumptions can be referred to in hint items by the keyword 'Assumption' followed by the full assumed expression, or by any theorem identifier provided after '**Assuming**'.

   With the proof header "**Assuming** \`$P$\`:", if the contained proof has inferred proof goal Q, then the whole proof has inferred proof goal '$P \Rightarrow Q$'.

   Multiple assumptions (separated by commas) can be used for goals of shape '$P_1 \wedge \ldots \wedge P_n \Rightarrow Q$', too, but the inferred proof goal will always be '$P_1 \Rightarrow \ldots \Rightarrow P_n \Rightarrow Q$'.

5. **Assuming** *ThmName*$^?$ \``*expression*`\` **and using with** *hint* :

   This variant of '**Assuming**' has the same inferred proof goal as plain '**Assuming**'.

   The difference is that each reference to this 'Assumption' acts as if it had been modified by adding '**with** *hint*' after it.

6. **Assuming witness** \`$var\{:\ type\}^?$\` **satisfying** \``*expr*`\` :

   This header introduces the bound variable 'var', and makes *expr* available as assumption to the following subproof.

   If the contained proof has inferred proof goal P, then the resulting proof has inferred proof goal $(\exists\ var : type \bullet expr) \Rightarrow P$.

   The variant

   Assuming witness \`$var\{:\ type\}^?$\` **satisfying** \``*expr*`\` **by** *hint* :

   can be understood as providing ∃-elimination; it uses *hint* to discharge the antecedent $(\exists\ var : type \bullet expr)$ and then has inferred proof goal P.

7. **Using** $HintItem_0$:
   $subproof_1$
   ⋮
   $subproof_n$

   This proof header is followed by a sequence of subproofs hint items, as listed in Sect. 5.3.

   In most cases, this can be understood as turning a theorem referenced by $HintItem_0$ into an inference rule with n premises, and applying it to the n subproofs.

   Technically, it corresponds to a proof '**By**' the following hint item:

   $$HintItem_0 \text{ with } subproof_1 \text{ and } \ldots \text{ and } subproof_n$$

   Therefore, CALCCHECK cannot infer proof goals here.

   If only one subproof is given, and the goal of this can be inferred, then the "Subproof" header does not have to be provided.

8. **With** *Hint* :
   $subproof_1$
   ⋮
   $subproof_n$

As for 'Using', this proof header is followed by a sequence of subproofs, but the header may contain more than one hint item. If *Hint* is 'hi_1, ... , hi_m, then the resulting proof corresponds to a proof '**By**' the following hint:

$$hi_1\ \ldots\ ,\ hi_m,\ subproof_1\ \ldots\ ,\ subproof_n$$

That is, where 'Using' has a 'with', here we have another 'and'. This is essentially just syntactic sugar with better layout support than just '**By**' with subproofs.

Therefore, CalcCheck cannot infer proof goals here.

9.
```
Side proof for  ThmId? `expr`:
    ⋮
Continuing:
    proof
```

This introduces *expr* as a *local property*, for which the proof must be provided in an indented block before '**Continuing:**' with the goal of the whole proof (in another indented block). Local properties, which may refer to local variables introduced by '**For any**', and which may use assumptions in their proofs, are then referred to either in later side proofs or after '**Continuing:**' in hint items by the key phrase 'Local property" followed by their expression or name. (Before '**Continuing:**', there can be multiple side proofs at the same indentation.)

The inferred proof goal here is the same as that of the proof after '**Continuing:**'.

10.
```
Local definition:  identifier = expr
    ⋮
Continuing:
    proof
```

This introduces *identifier* as a *local constant*, the definition of which can be referred to via the hint item 'Definition of *identifier*'.

Before '**Continuing:**', there can actually be and sequence of side proofs and local definitions at the same indentation.

The inferred proof goal here is the same as that of the proof after '**Continuing:**'.

11.
```
    calculation
Proof for this:
    proof
```

This shape requires that all of *calculation* is indented at least two spaces more than '**Proof for this:**'.

Assuming that *calculation* ends in the expression *calcend*, this shape of proof is equivalent to continuing *calculation* with one more step:

```
  calculation
≡⟨ Subproof for `calcend`:
        proof
  ⟩
  true
```

The inferred proof goal for this kind of proofs is the first expression of *calculation*, which cannot have any '— **This is**'.

12. **By cases:** \`*expr*$_1$\`, …, \`*expr*$_n$\`
    Completeness: *proof*
    Case \`*expr*$_1$\`:
        proof$_1$
    ⋮
    Case \`*expr*$_n$\`:
        proof$_n$

    This introduces case analysis essentially as proposed in LADM chapter 4, except that the completeness proof for '$expr_1 \vee \cdots \vee expr_n$' is mandatory. (We do not use this proof structure in the current paper.)

    'Assumption \`*expr*$_i$\`' is available for use in the case *proof*$_i$.

    The inferred proof goal for such case analysis proofs is the inferred goal of the first case proof that has one.

13. **By induction on** \`*var* : *type*\`:
    Base case {\`*expr*\`}$^?$:
        proof$_1$
    Induction step {\`*expr*\`}$^?$:
        proof$_2$

    Although induction proofs could be presented '**Using**' induction principles, we also have this simpler structure to be able to introduce induction proofs very early, before quantification. CalcCheck currently supports '**By induction**' proofs via simple induction for natural numbers and for sequences (requiring registration of the respective constructors in either case).

    Currently CalcCheck does not attempt to infer proof goals from induction proofs.

These proof structures are actually sufficient to support and encourage fully formal proofs. (In [Kahl, 2018] we show several examples of elegant fully formal proofs where LADM resorts to prose for significant parts of the proof structure, even in cases where it would be possible to use proof syntax introduced in LADM chapter 4, such as '**Assuming**' or '**By cases**'.)

References

C. Aarts, R. C. Backhouse, P. Hoogendijk, E. Voermans, and J. van der Woude. A relational theory of datatypes. Working document, Dec. 1992. 387 pp., available at http://www.cs.nott.ac.uk/~rcb/papers/abstract.html#book.

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer, 2004. ISBN 978-3-540-20854-9.

L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean theorem prover (system description). In A. P. Felty and A. Middeldorp, editors, *Automated Deduction — CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21401-6. doi: 10.1007/978-3-319-21401-6_26.

A. Grabowski, A. Korniłowicz, and A. Naumowicz. Mizar in a nutshell. *J. Formalized Reasoning*, 3(2): 153–245, 2010.

D. Gries. Foundations for calculational logic. In M. Broy and B. Schieder, editors, *Mathematical Methods in Program Development*, pages 83–126, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-642-60858-2. doi: 10.1007/978-3-642-60858-2_16.

D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math.* Monographs in Computer Science. Springer, 1993. ISBN 3-540-94115-0. doi: 10.1007/978-1-4757-3837-7.

J. Gruber. Markdown. `https://daringfireball.net/projects/markdown/`, 2004. (last accessed 2020-03-09).

W. Kahl. CalcCheck: A proof checker for teaching the "Logical Approach to Discrete Math". In J. Avigad and A. Mahboubi, editors, *Interactive Theorem Proving*, volume 10895 of *LNCS*, pages 324–341, Cham, July 2018. Springer. ISBN 978-3-319-94820-1. doi: 10.1007/978-3-319-94821-8_19.

S. Leonard. *The text/markdown Media Type, RFC 7763.* Internet Engineering Task Force (IETF), Mar. 2016a.

S. Leonard. *Guidance on Markdown: Design Philosophies, Stability Strategies, and Select Registrations, RFC 7764.* Internet Engineering Task Force (IETF), Mar. 2016b.

J. MacFarlane. Pandoc. `http://johnmacfarlane.net/pandoc/`, 2006.

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory.* PhD thesis, Dept. Comp. Sci. and Eng., Chalmers Univ. of Technology, Sept. 2007. See also http://wiki.portal.chalmers.se/agda/pmwiki.php.

S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference, Version 2.3.* SRI International, Sept. 1999. URL `http://pvs.csl.sri.com/`.

G. Schmidt and T. Ströhlein. *Relations and Graphs, Discrete Mathematics for Computer Scientists.* EATCS-Monographs on Theoret. Comput. Sci. Springer, 1993. ISBN 3-540-56254-0, 0-387-56254-0. doi: 10.1007/978-3-642-77968-8.

J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science. Prentice Hall, 1989.