# CalcCheck: A Proof Checker for Teaching the "Logical Approach to Discrete Math"

Wolfram Kahl

McMaster University, Hamilton, Ontario, Canada, kahl@cas.mcmaster.ca

**Abstract.** For calculational proofs as they are propagated by Gries and Schneider's textbook classic "A Logical Approach to Discrete Math" (LADM), automated proof checking is feasible, and can provide useful feedback to students acquiring and practicing basic proof skills. We report on the CalcCheck system which implements a proof checker for a mathematical language that resembles the rigorous but informal mathematical style of LADM so closely that students very quickly recognise the system, which provides them immediate feed-back, as not an obstacle, but as an aid, and realise that the problem is finding proofs.
Students interact with this proof checker trough the "web application" front-end $\text{CalcCheck}_{\text{Web}}$ which provides some assistance for proof entry, but intentionally no assistance for proof finding. Upon request, the system displays, side-by-side with the student input, a version of that input annotated with the results of checking each step for correctness.
$\text{CalcCheck}_{\text{Web}}$ has now been used twice for teaching an LADM-based second-year discrete mathematics course, and students have been solving exercises and submitting assignments, midterms, and final exams on the system — for examinations, there is the option to disable proof checking and leave only syntax checking enabled. CalcCheck also performed the grading, with very limited human overriding necessary.

## 1 Introduction

The textbook "A Logical Approach to Discrete Math" (referred to as "LADM") by Gries and Schneider (1993) is a classic introduction to reasoning in the *calculational style*, which allows for rigorous-yet-readable proofs. Gries and Schneider (1995) establish a precise logical foundation for such calculations in propositional logic, and Gries (1997) expands this also to predicate logic, so that we do not need to dwell on these aspects in the current paper.

We present a mechanised theory language that has been designed to be as close to the "informal" but rigorous language of LADM, and the proof checker CalcCheck designed for supporting teaching based on LADM. A predecessor system (Kahl, 2011) using LaTeX-based interaction in the style of *f*uzz (Spivey, 2008) only supported checking isolated calculations in a hard-coded LADM-like expression language, and recognised only hard-coded theorem numbers in unstructured hints; the current version of CalcCheck admits user-defined operators, and has a completely new language for theories, structured proofs, and structured calculation hints.

Since students still need to learn "what a proof is" and "how different proofs can be", we consciously do not offer any assistance in proof finding, but we turned the proof checker into a web application, so that students can obtain instant feedback for their proof attempts, all while writing proofs that are recognisably in the style of the textbook.

For example, on p. 55 of LADM we find the following calculation (with relatively detailed hints), reproduced here almost exactly (only with slightly different spacing):

As an example, we prove theorem (3.44a): $p \wedge (\neg p \vee q) \equiv p \wedge q$:

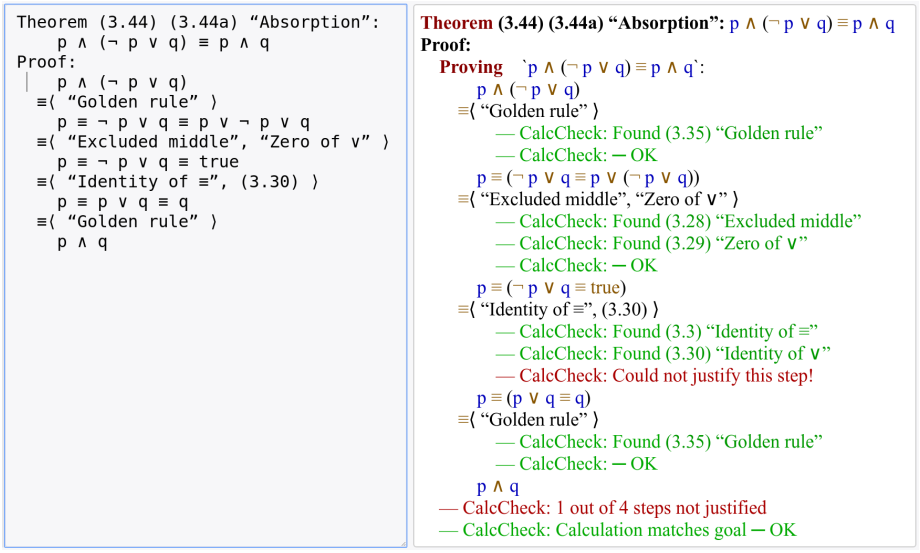
$$
\begin{array}{ll}
 & p \wedge (\neg p \vee q) \\
= & \quad \langle \text{ Golden rule (3.35), with } q := \neg p \vee q \ \rangle \\
 & p \equiv \neg p \vee q \equiv p \vee \neg p \vee q \\
= & \quad \langle \text{ Excluded middle (3.28) } \rangle \\
 & p \equiv \neg p \vee q \equiv \mathit{true} \vee q \\
= & \quad \langle \text{ (3.29), } \mathit{true} \vee p \equiv \mathit{true} \ \rangle \\
 & p \equiv \neg p \vee q \equiv \mathit{true} \\
= & \quad \langle \text{ Identity of } \equiv \quad (3.3) \ \rangle \\
 & p \equiv \neg p \vee q \\
= & \quad \langle \text{ (3.32), } p \vee \neg q \equiv p \vee q \equiv p, \\
 & \qquad \text{with } p, q := q, p \text{ — to eliminate operator } \neg \ \rangle \\
 & p \equiv p \vee q \equiv q \\
= & \quad \langle \text{ Golden rule (3.35) } \rangle \\
 & p \wedge q
\end{array}
$$

In CalcCheck, this theorem together with this proof can be entered as follows in plain Unicode text:

```
Theorem (3.44) (3.44a) "Absorption": p ∧ (¬ p ∨ q) ≡ p ∧ q
Proof:
    p ∧ (¬ p ∨ q)
  ≡⟨ "Golden rule" (3.35) with `q ≔ ¬ p ∨ q` ⟩
    p ≡ ¬ p ∨ q ≡ p ∨ ¬ p ∨ q
  ≡⟨ "Excluded middle" (3.28) ⟩
    p ≡ ¬ p ∨ q ≡ true ∨ q
  ≡⟨ (3.29) `true ∨ p ≡ true` ⟩
    p ≡ ¬ p ∨ q ≡ true
  ≡⟨ "Identity of ≡" (3.3) with `q ≔ p ≡ ¬ p ∨ q` ⟩
    p ≡ ¬ p ∨ q
  ≡⟨ (3.32) `p ∨ q ≡ p ∨ ¬ q ≡ p`
      with `p, q ≔ q, p` — to eliminate operator ¬ ⟩
    p ≡ p ∨ q ≡ q
  ≡⟨ "Golden rule" (3.35) ⟩
    p ∧ q
```

Except for the comment "— to eliminate operator ¬", everything here is formal content, and checked by the system. We will explain some of the details in Sect. 2. It should however be obvious that the correspondence is very close, with the small differences mostly due to either the fact that we are using a plain text format, or to the requirement that the language needs to be unambiguously parse-able for automatic checking to become feasible.

A student encountering only the theorem statement of this in their homework might, if allowed to use two theorem references per hint, write the variant shown below to the left in the CalcCheck$_{\text{Web}}$ interface in their web browser:



```
Theorem (3.44) (3.44a) "Absorption":
    p ∧ (¬ p ∨ q) ≡ p ∧ q
Proof:
  p ∧ (¬ p ∨ q)
≡⟨ "Golden rule" ⟩
  p ≡ ¬ p ∨ q ≡ p ∨ ¬ p ∨ q
≡⟨ "Excluded middle", "Zero of ∨" ⟩
  p ≡ ¬ p ∨ q ≡ true
≡⟨ "Identity of ≡", (3.30) ⟩
  p ≡ p ∨ q ≡ q
≡⟨ "Golden rule" ⟩
  p ∧ q
```

**Theorem (3.44) (3.44a) "Absorption":** p ∧ (¬ p ∨ q) ≡ p ∧ q
**Proof:**
**Proving** `p ∧ (¬ p ∨ q) ≡ p ∧ q`:
p ∧ (¬ p ∨ q)
≡⟨ "Golden rule" ⟩
— CalcCheck: Found (3.35) "Golden rule"
— CalcCheck: ─ OK
p ≡ (¬ p ∨ q ≡ p ∨ (¬ p ∨ q))
≡⟨ "Excluded middle", "Zero of ∨" ⟩
— CalcCheck: Found (3.28) "Excluded middle"
— CalcCheck: Found (3.29) "Zero of ∨"
— CalcCheck: ─ OK
p ≡ (¬ p ∨ q ≡ true)
≡⟨ "Identity of ≡", (3.30) ⟩
— CalcCheck: Found (3.3) "Identity of ≡"
— CalcCheck: Found (3.30) "Identity of ∨"
— CalcCheck: Could not justify this step!
p ≡ (p ∨ q ≡ q)
≡⟨ "Golden rule" ⟩
— CalcCheck: Found (3.35) "Golden rule"
— CalcCheck: ─ OK
p ∧ q
— CalcCheck: 1 out of 4 steps not justified
— CalcCheck: Calculation matches goal ─ OK

After sending this to the server for checking, the box to the right will be filled in by the system as shown in the screen-shot above, and the student will likely notice that they mis-typed the number of theorem (3.32), one of the few nameless theorems in LADM that are emphasised as worth remembering the number of.

We proceed with explaining the basics of the CalcCheck theory language in Sect. 2. In Sect. 3 we strive to give an idea of how interaction with such theories works in practice, before proceeding to more advanced language features: In Sect. 4 we present the main hard-coded proof structuring principles, and in Sect. 5 we discuss our treatment of quantification, substitution, and metavariables. More complicated hints are covered in Sect. 6, and mechanisms for selectively making reasoning features available in Sect. 7. Finally, we highlight some aspects of the implementation in Sect. 8 and discuss some related work in Sect. 9. Some additional documentation is available at the CalcCheck home page at http://CalcCheck.McMaster.ca/.

## 2 The Basic CalcCheck Language

A CalcCheck module consists of a sequence of *top-level items* (TLIs), which include declarations, axioms, theorems, and several kinds of administrative items, as for example precedence declarations.

```
Precedence 40 for: _∧_
Associating to the right: _∧_
Declaration: _∧_ : 𝔹 → 𝔹 → 𝔹
Axiom (3.35) "Golden rule": p ∧ q ≡ p ≡ q ≡ p ∨ q
```

The language is layout-sensitive: Everything after the first line in a top-level item, or inside other high-level syntactic constructs, needs to be indented at least two spaces farther than the first line. The only exception to this is the "Proof:" for a theorem, which starts in the same column as the theorem.

Instead of the word Theorem, one may alternatively use Lemma, Corollary, Proposition, or Fact without any actual differences. (Technically, Axioms are theorems that are just not allowed to have proof.) A theorem may have any number of *theorem numbers* (always in parentheses and without spaces, such as (3.35) above and (3.44a) in Sect. 1) and *theorem names* (always in pretty double quotes — as opposed to the plain double quotation mark character "" — such as ""Absorption"" in Sect. 1). The same names and numbers may be given to several theorems, which implements the way LADM uses "Absorption (3.44)" to refer to uses of either (3.44a) or (3.44b) or both.

A *calculation*, such as the proof body in Sect. 1, consists of a sequence of *expressions* interleaved with *calculation operators* (in Sect. 1 only "≡") attached to a pair of *hint brackets* "⟨ ... ⟩" enclosing a *hint*. A hint is a sequence of *hint items* separated by commas or "and" or both; so far, we only have seen *theorem references* as hint items. (Comments, such as "— to eliminate operator ¬" in Sect. 1, are currently only supported inside hints.)

A theorem reference can be either a theorem name in pretty double quotes, or a theorem number in parentheses, or an expression in back-ticks (‘...‘), or several theorem references separated by white-space. In Sect. 1 we have seen a few examples of the latter in the LADM calculation; they refer to the intersection of the sets of theorems referred to by the constituent atomic theorem references. It is configurable whether theorem references in the shape of expressions can be used alone; this is forbidden by default: Learning the theorem names is, for the most part, learning the vocabulary of the language of discrete math, and therefore part of the learning objectives. (With this setting, a theorem with no names nor numbers cannot be referred to, but may still be useful for documentation, for example as a Fact.)

For the expression syntax, almost arbitrary sequences of printable Unicode characters are legal identifiers, as in Agda (Norell, 2007), so that almost all lexemes need to be separated by spaces. (Parentheses need no spaces.)

CalcCheck follows LADM in supporting conjunctional operators: The expression $1 < 2 \in S \subseteq T$ is considered shorthand for $(1 < 2) \wedge (2 \in S) \wedge (S \subseteq T)$. The set of conjunctional operators and their precedence is not hard-coded; for emulating LADM we write in our "prelude":

```
Precedence 50 for:
   _=_ , _≠_ , _<_ , _>_ , _≤_ , _≥_ , _≮_ , _≯_ , _≰_ , _≱_ , _∣_ ,
   _∈_ , _∉_ , _∋_ , _∌_ , _⊂_ , _⊆_ , _⊄_ , _⊈_ , _⊃_ , _⊅_ , _⊇_ , _⊉_
Conjunctional:
   _=_ , _≠_ , _<_ , _>_ , _≤_ , _≥_ , _≮_ , _≯_ , _≰_ , _≱_ , _∣_ ,
   _∈_ , _∉_ , _∋_ , _∌_ , _⊂_ , _⊆_ , _⊄_ , _⊈_ , _⊃_ , _⊅_ , _⊇_ , _⊉_
```

As could be seen already in the example at the beginning of this section, such declarations of operator precedence and associating behaviour come before the actual declarations of the operator: Like LADM, CalcCheck supports operator overloading. Since operator precedence and associating behaviour have to be

declared before the actual Declarations, it is easy to enforce coherent precedences also in larger developments. (In LADM, this declaration-independent precedence table can be found on the inside cover.)

Underscores denote argument positions of mixfix operators. Arbitrary binary infix operators can be used as calculation operators, that is, preceding hint brackets ⟨ ... ⟩. The calculation notation as such is considered conjunctional, which enables us to use the non-conjunctional associative operator ≡ as calculation operator in the examples in Sect. 1, or, later, also implication.

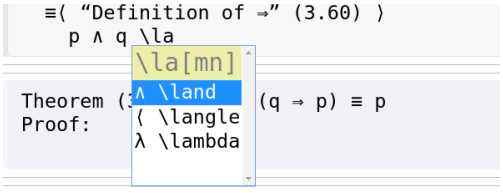
Two of the steps in the first calculation specify substitutions to variables in the referenced theorem with expressions (containing variables of the currently-proven theorem), for example "with $p, q := q, p$" in the second-last step. This is also allowed in CalcCheck, except that the substitution is delimited by back-ticks. We choose back-ticks because they are also used in MarkDown for embedding code in prose — CalcCheck allows MarkDown blocks as top-level items for "literate theories" documentation. We us backticks for embedding expressions, and other expression-level material such as substitutions, inside "higher-level structures" in many places. In theorems and proofs, essentially the only places where backticks are not used are after the colon in the theorem statement, and outside the hints in calculations.

## 3 The CalcCheck$_{\text{Web}}$ Front-End

Since CalcCheck source files are just plain Unicode text files, editing them using any editor is certainly possible. However, the preferred way to edit CalcCheck source, and the only way currently offered to students, is via the "web application" CalcCheck$_{\text{Web}}$, which can be accessed via websocket-capable web browsers.

A "notebook" style view is presented with a vertical sequence of "cells". MarkDown TLIs are shown in cells containing a single box, and "code cells" with a horizontal split into two boxes (as already shown in the screenshot at the end of Sect. 1). The left box is for code entry, and the right box is populated with feedback from the server, which performs, upon request, syntax checking, or syntax and proof checking combined. (For exams, proof checking can be disabled.)

For text and code entry, CalcCheck$_{\text{Web}}$ provides symbol input via mostly LaTeX-like escape sequences; typing a backslash triggers a pop-up displaying the escape sequence typed so far, and the possible next characters. In experienced use, this pop-up is irrelevant, and disappears when characters beyond the completion of the escape sequence are entered. Alternatively, upon a TAB key press, this pop-up also displays a menu, as in the following screenshot:



Similar completion is provided for theorem names, after typing a prefix (of at least length three) of a theorem name preceded by either pretty opening double quotes '“' or the simple double-quote character '"', hitting the TAB key brings up a theorem name completion menu containing only theorem names currently in scope, but intentionally not filtered in any other way.

Support for indentation currently provided includes toggling display of initial spaces as "visible space" "␣", and key bindings for increasing or decreasing the indentation of whole blocks.

## 4 Structured Proofs

Calculations, as shown in Sect. 1, are just one kind of proof supported by CalcCheck. LADM emphasises the use of axioms (and theorems) in calculations over other inference rules, so not many other proof structures are needed. Besides calculations, the other options for proof in CalcCheck (explained in more detail below) are:

- "By *hint*" for discharging simple proof obligations,
- "Assuming '*expression*':" corresponding to implication introduction,
- "By cases: '$expression_1$','...,'$expression_n$'" for proofs by case analysis,
- "By induction on '*var* : *type*':" for proofs by induction,
- "For any '*var* : *type*':" corresponding to ∀-introduction,
- "Using *hint*:" for turning theorems into inference rules, see Sect. 6.3.

With these (nestable) proof structures, we essentially formalise the slightly more informal practices of LADM, which, in Chapt. 4, introduces what appears to be formal syntax for proofs by cases, and for proofs of implications by assuming the antecedents. However, in actual later LADM proofs, this syntax is typically not used. For example, on p. 305 we find some cases listed in a way that does not easily correspond to the pattern in LADM Chapt. 4, and the assumption of the antecedent is almost hidden in the surrounding prose that replaces the explicit proof structure. We can emulate the calculation there very closely again, and we embed it into a fully formal proof that is, in our opinion, at least as clear and readable as the arrangement in LADM:

```
Theorem (15.34) “Positivity of squares”: b ≠ 0 ⇒ pos (b · b)
Proof:
  Assuming `b ≠ 0`:
    By cases: `pos b`, `¬ pos b`
      Completeness: By “Excluded middle”
      Case `pos b`:
        By “Positivity under ·” with assumption `pos b`
      Case `¬ pos b`:
          pos (b · b)
        ≡⟨ (15.23) `- a · - b = a · b` ⟩
          pos ((- b) · (- b))
        ⇐⟨ “Positivity under ·” (15.31) ⟩
          pos (- b) ∧ pos (- b)
        ≡⟨ “Idempotency of ∧”, “Double negation” ⟩
          ¬ ¬ pos (- b)
        ≡⟨ “Positivity under unary minus” (15.33) with assumption `b ≠ 0` ⟩
          ¬ pos b      — This is Assumption `¬ pos b`
```

Our syntax for assuming the antecedent should be self-explaining — the keyword assumption for producing hint items referring to an assumption (which may also be given a local theorem name in double quotes) may also be written Assumption. The assumed expression is again delimited by backticks.

For proof by cases, we follow the pattern proposed in LADM Chapt. 4, except that we insist on a proof of Completeness of the list of patterns to be explicitly supplied. In the case above, we discharge this proof obligation via By "Excluded middle" — this is another variant of proofs, where just a hint (that is, a sequence of hint items) is provided after the keyword By. The expression of the current Case is available in the proof via the Assumption keyword.

At the end of the calculation above, we have "— This is ..."; this is used in LADM without the words "This is" as a "formal comment" indicating that the last expression in the calculation is the indicated assumption, or, more frequently, an instance of the indicated theorem. Later, Gries and Schneider (1995) explain this via the inference rule "Equanimity". For CalcCheck, such "— This is ..." clauses are not considered comments at all, but are part of the calculation syntax, and require exactly this phrasing. As in LADM, this can be used at either end of a calculation. Several further details of the above proof of "Positivity of squares" will be explained below in sections 6 and 7.

The first proof structure beyond calculations that is introduced in the course is actually successor-based natural induction, where natural numbers have been introduced inductively from zero "0" and the successor operator "S_", and the inductive definitions for operations have been provided as sequences of axioms, as the following for subtraction:

```
Declaration: _-_ : ℕ → ℕ → ℕ
Axiom "Subtraction from zero":                      0 - n     = 0
Axiom "Subtraction of zero from successor":     (S m) - 0     = S m
Axiom "Subtraction of successor from successor": (S m) - (S n) = m - n
```

With this, even nested induction proofs such as the following become easy to produce for the students:

```
Theorem "Subtraction after addition": (m + n) - n = m
Proof:
  By induction on `m : ℕ`:
    Base case:
        (0 + n) - n
      =⟨ "Identity of +" ⟩
        n - n
      =⟨ "Self-cancellation of subtraction" ⟩
        0
    Induction step `(S m + n) - n = S m`:
      By induction on `n : ℕ`:
        Base case:
            (S m + 0) - 0
          =⟨ "Identity of +" ⟩
            S m - 0
          =⟨ "Subtraction of zero from successor" ⟩
            S m
```

```
Induction step:
    (S m + S n) - S n
  =⟨ "Definition of +" ⟩
    S (m + S n) - S n
  =⟨ "Subtraction of successor from successor" ⟩
    (m + S n) - n
  =⟨ "Adding the successor", "Definition of +" ⟩
    (S m + n) - n
  =⟨ Induction hypothesis `(S m + n) - n = S m` ⟩
    S m
```

The proof goals for base case and induction step may optionally be made explicit — we show this here only for the outer induction step. In nested induction steps where several induction hypotheses are available, the system currently requires the keyword phrase **Induction hypothesis** to be accompanied by the chosen induction hypothesis, but only for pedagogical reasons.

Currently, besides natural induction, also induction on sequences is supported by this hard-coded By induction on proof format; the '`m : ℕ`' after this keyword phrase above indicates the induction variable and its type, which selects the induction principle, if one is implemented and activated for that type.

## 5 Quantification, Substitution, Metavariables

For quantification, CalcCheck follows the spirit of LADM, but in the concrete syntax is closer to the Z notation (Spivey, 1989): The general pattern of quantified expressions is "*bigOp varDecls* ❙ *rangePredicate* • *body*", and we have, for example:

$$(\sum\ i \mid 0 \le i < 5 \bullet i!) \quad = \quad 0! + 1! + 2! + 3! + 4!$$
$$(\forall\ k, n : \mathbb{N} \mid k < n < 3 \bullet k \cdot n < 5) \quad \equiv \quad 0 \cdot 1 < 5 \wedge 0 \cdot 2 < 5 \wedge 1 \cdot 2 < 5$$

The range predicate, when omitted together with the " ❙ ", defaults to *true*. As in Z, parentheses around quantifications can be omitted, and the scope of the variable binding then extends "as far as syntactically possible". (This a a conscious notational departure from LADM, where parentheses around quantifications are compulsory, and ":" is used instead of "•".) In another notational departure, we denote function application by (typically space-separated) juxtaposition, "$f\ x$", instead of "$f.x$" for atomic arguments in LADM.

The following proof is for a stronger variant of the LADM theorem (8.22) "Change of dummy", which both LADM and Gries (1997) show without the range predicate R in the assumption (but when LADM refers to (8.22) later, in chapters 12 and 17, it actually always would have to use our variant). Here, as in LADM, "⋆" is used as a metavariable for a quantification operator, that is, a symmetric and associative binary operator (usually equipped with an identity).

```
Theorem (8.22a) "Change of restricted dummy":
  (∀ x ❙ R • (∀ y • x = f y  ≡  y = g x))
  ⇒ (⋆ x ❙ R • P) = (⋆ y ❙ R[x ≔ f y] • P[x ≔ f y])
```

```
Proof:
  Assuming "Inverse" `∀ x ❙ R • ∀ y • x = f y  ≡  y = g x`:
      (⋆ y ❙ R[x ≔ f y] • P[x ≔ f y])
    =⟨ "One-point rule for ⋆" ⟩
      (⋆ y ❙ R[x ≔ f y] • (⋆ x ❙ x = f y • P))
    =⟨ "Nesting for ⋆" ⟩
      (⋆ y, x ❙ R[x ≔ f y] ∧ x = f y • P)
    =⟨ Substitution ⟩
      (⋆ y, x ❙ R[x ≔ z][z ≔ f y] ∧ x = f y • P)
    =⟨ "Replacement" ⟩
      (⋆ y, x ❙ R[x ≔ z][z ≔ x] ∧ x = f y • P)
    =⟨ Substitution ⟩
      (⋆ y, x ❙ R ∧ x = f y • P)
    =⟨ "Dummy list permutation for ⋆" ⟩
      (⋆ x, y ❙ R ∧ x = f y • P)
    =⟨ "Nesting for ⋆" ⟩
      (⋆ x ❙ R • (⋆ y ❙ x = f y • P))
    =⟨ "Range replacement in nested ⋆" with assumption "Inverse" ⟩
      (⋆ x ❙ R • (⋆ y ❙ y = g x • P))
    =⟨ "One-point rule for ⋆" ⟩
      (⋆ x ❙ R • P[y ≔ g x])
    =⟨ Substitution ⟩
      (⋆ x ❙ R • P)
```

LADM and Gries (1997) both refrain from formalising the assumption "$f$ has an inverse" as part of the theorem statement, since they present *all* general quantification theorems before introducing universal quantification. With a different theory organisation, we introduce universal quantification as instance of a restricted theory of general quantification, and then use universal quantification to state and prove theorems like this about general quantification which mention universal quantification.

This "Change of restricted dummy" theorem is really a metatheorem: Its statement contains metavariables $x$ and $y$ for *different* variables, and $P$ and $R$ for expressions that may have free occurrences of $x$, and it also contains explicit substitutions. Gries (1997) calls such proofs of metatheorems using metatheorems "schematic proofs".[1] The fact that $P$ and $R$ must not have free occurrences of $y$ is expressed by Gries and Schneider as the *proviso* "$\neg occurs(\text{`}y\text{'}, \text{`}P, R\text{'})$" in the metalanguage.

CalcCheck takes a slightly different approach to metavariables: For consistency with LADM, we keep the inference rule substitution, and use only the substitution notation $E[v := G]$. Once quantification is introduced, we emphasise that substitution binds variables, too (where only occurrences of $v$ in $E$ are bound in $E[v := G]$), and application of substitution may need to rename bound

[1] Gries (1997) restricts metavariables to be named by single upper-case letters, (non-meta-)variables by single lower-case letters. Gries (1997) then distinguishes between "uniform substitution" written $E[V := G]$ for metavariables $V$, and "textual substitution" written $E^v_G$ for variables $v$, where only the latter renames variable binders to avoid capture of free variables of $G$. However, the use of "$R[x := f\ y]$" in the statement and proof of (8.22) there is then unclear — it will have to be understood as "textual substitution" since otherwise $y$ might be captured by binders in $R$.

variables (in $E$) to avoid capture of free (in $G$) variables. Expression equality in CalcCheck is only up to renaming of bound variables; students are encouraged to use "Reflexivity of =" calculation steps to document such renaming.

When introducing quantification and variable binding, we (re-)explain axiom schemas, and emphasise that metavariables are *instantiated* (and not substituted), but do not provide notation for that. (Instantiation of metavariables does not rename binders and therefore can capture variables that are free in the instantiating expression. Such capture is the point of metavariables — in (8.22a) above, R is meant to be instantiated with expression containing free occurences of x.)

In a theorem statement, metavariables for expressions are defined (and recognised by CalcCheck) as looking like free variables in the scope of a variable binder. Metavariables with occurrences in scope of different sets of variable binders may only be instantiated with expressions in which only the intersection of all these binders occurs free. Bound variables that are allowed to occur in metavariables for expressions have to be considered metavariables for variables, and matched consistently. Thus, the "¬*occurs*" provisos can be derived from the theorem statement; for the theorem above, if metavariable reporting (by default disabled) and proviso reporting are both enabled, CalcCheck generates the following output:

> **Theorem (8.22a) "Change of restricted dummy":** (∀ x ❙ R • (∀ y • x = f y ≡ y = g x ) ) ⇒ (★ x ❙ R • P ) = (★ y ❙ R[x ≔ f y] • P[x ≔ fy] )
> — CalcCheck: Metavariables: P = P⟦ x ⟧ , R = R⟦ x ⟧ , f = f⟦ y ⟧ , g = g⟦ x, y ⟧
> — CalcCheck: Proviso: ¬*occurs*(`x`, `f`), ¬*occurs*(`y`, `P, R`)

The proof above contains three steps where the hint is the keyword Substitution; this hint item is used for performing substitutions. For both Substitution steps here, it is necessary that ¬*occurs*('$z$', '$R$'); for such new variable binders, this is handled automatically by remembering also which variables *are* allowed to occur in $R$, as shown in the "**Metavariables**" information report above.

Above we used "Replacement" (3.84a): $e = f \ \wedge P[z := e] \ \equiv \ e = f \ \wedge P[z := f]$ (called "Substitution" in LADM). This is another example for a metatheorem; its statement involves substitution, and the metavariables $z$ for variables and $P$ for expressions that may have free occurrences of $z$. Since CalcCheck currently does not use second-order matching, the reverse Substitution step preceding the application of "Replacement" (3.84a) is necessary for establishing the matching of the metavariables $z$ and $P$, here to the variable $z$ and the expression $R[x := z]$ respectively. The second Substitution could be merged with the "Replacement" (3.84a) step, but has been left separate here for readability.

Note that "Dummy list permutation" is a quantification axiom missing from LADM and also not mentioned by Gries (1997), but used implicitly in the proof of (8.22) in both places.

The proof above is almost identical to the proof for (8.22) of LADM, except for the step using the assumption "Inverse", where the proof for (8.22) only has to invoke that assumption. In the proof above, we use the following lemma:

```
Theorem “Range replacement in nested ⋆”:
    (∀ x ❙ R • (∀ y • Q₁ ≡ Q₂))
    ⇒ (⋆ x ❙ R • (⋆ y ❙ Q₁ • P)) = (⋆ x ❙ R • (⋆ y ❙ Q₂ • P))
Proof:
      ∀ x ❙ R • (∀ y • Q₁ ≡ Q₂)
    ≡⟨ “Nesting for ∀” ⟩
      ∀ x • ∀ y ❙ R • Q₁ ≡ Q₂
    ≡⟨ “Trading for ∀” ⟩
      ∀ x • ∀ y • R ⇒ (Q₁ ≡ Q₂)
    ≡⟨ (3.62) ⟩
      ∀ x • ∀ y • R ∧ Q₁ ≡ R ∧ Q₂
    ⇒⟨ Subproof:
          Assuming “A” `∀ x • ∀ y • R ∧ Q₁ ≡ R ∧ Q₂`:
              (⋆ x ❙ R • (⋆ y ❙ Q₁ • P))
            =⟨ “Nesting for ⋆” ⟩
              (⋆ x, y ❙ R ∧ Q₁ • P)
            =⟨ Assumption “A” ⟩
              (⋆ x, y ❙ R ∧ Q₂ • P)
            =⟨ “Nesting for ⋆” ⟩
              (⋆ x ❙ R • (⋆ y ❙ Q₂ • P))
      ⟩
      (⋆ x ❙ R • (⋆ y ❙ Q₁ • P)) = (⋆ x ❙ R • (⋆ y ❙ Q₂ • P))
```

In the first step here, two different rules that are both called “Nesting for ∀” are applied in sequence, and in opposite directions. The last hint here contains a single Subproof hint item; inside such a subproof, any kind of proof can be written.

The necessity to distinguish metavariables for variables becomes most obvious from considering theorem (11.7) of LADM (Gries (1997) does not cover set comprehension):

**Theorem (11.7) (11.7x) “Simple Membership”:** x ∈ { x ❙ P } ≡ P
— CalcCheck: Metavariables: P = P⟦ x ⟧

If one were to consider the left-most x here as a normal free variable, then the rule for deriving provisos given above would imply that x must not occur free in P, since the right-most P does not occur in the scope of a binder for x.

It is useful to consider (11.7) in the context of metatheorem (9.16) of LADM and Gries (1997): “$P$ is a theorem iff $(\forall x \bullet P)$ is a theorem.” In the universally quantified version, both occurrences of P are within scope of a binder for x, so no proviso is derived:

**Theorem (11.7) (11.7∀) “Simple Membership”:** (∀ x • x∈ { x ❙ P } ≡ P )
— CalcCheck: Metavariables: P = P⟦ x ⟧

This is really a theorem — in our development, we actually prove this version first, and then obtain (11.7x) via instantiation.

By classifying x in (11.7x) as a metavariable for variables, we identify the “free-looking” occurrence of x as a binder in the scope of which the right-most P occurs. The effect of this approach is to let CalcCheck derive the same metavariable occurrence and ¬*occurs* provisos for (11.7x) as for (11.7∀), compatible with (9.16).

Note, however, that (9.16) talks about *theorems*, not metatheorems (or theorem schemas). A version that would make sense for metatheorems would need

to add the meta-proviso that the same provisos are derived. As a case in point, consider the one-point rule:

**Axiom (8.14) "One-point rule" "One-point rule for ∀":** (∀ x ❙ x = E • P ) ≡ P[x ≔ E]
— CalcCheck: Metavariables: E = E⟦ ⟧ , P = P⟦ x ⟧
— CalcCheck: Proviso: ¬*occurs*(`x`, `E`)

Naïvely applying (9.16) to that would yield the following, where E always occurs in scope of a binder for x:

**Axiom "Spuriously-quantified one-point rule for ∀":** (∀ x • (∀ x ❙ x =E • P ) ≡ P[x ≔ E] )
— CalcCheck: Metavariables: E = E⟦ x ⟧ , P = P⟦ x ⟧

This "axiom"-schema however is invalid for instantiations where x occurs free in E — just try to instantiate P with $(x < 4)$ and E with $(5 \cdot x)$.

# 6 Combined Hint Items

While in Sect. 1, the keyword "with" appeared followed by substitutions, in "Positivity of squares" in Sect. 4 as well as in "Change of restricted dummy" in Sect. 5 there are occurrences of the shape "$hi_1$ with $hi_2$" for two hint items $hi_1$ and $hi_2$. This is the simplest case of the following pattern:

$$hi_1 \text{ with } hi_2 \text{ and } \ldots \text{ and } hi_n$$

In CalcCheck, this pattern has the two formal interpretations explained in sections 6.1 and 6.2, together covering probably most of the informal uses of the word "with" in LADM.

## 6.1 Conditional Rewriting

If among the theorems, assumptions, and induction hypotheses referred to by $hi_1$ there is one that can be seen as an implication with an equality (or equivalence) as consequent,

$$A_1 \Rightarrow \cdots \Rightarrow A_m \Rightarrow L = R,$$

then this is used as a conditional rewrite rule: If rewriting using $L \longrightarrow R$ succeeds with substitution $\sigma$, then CalcCheck attempts to prove the antecedents $A_1\sigma, \ldots, A_m\sigma$ using the hint items $hi_2, \ldots, hi_n$.

The with uses in sections 4 and 5 all are of this kind.

## 6.2 Rule Transformation

A different way the hint item construct above can be used is by extracting rewriting rules from $hi_2$ to $hi_n$ and using these to rewrite the theorems referenced by $hi_1$. The results of that rewriting are then used to prove the goal of the hint. The following proof contains two such cases:

```
Theorem "Positivity": pos a ≡ a ≠ 0 ∧ ¬ pos (- a)
Proof:
    a ≠ 0 ∧ ¬ pos (- a)
  ≡⟨ "Positivity under unary minus" (15.33) with (3.62) ⟩
    a ≠ 0 ∧ pos a
  ≡⟨ "Positive implies non-zero" with (3.60) ⟩
    pos a
```

The two instances of $hi_1$ here are:

```
Axiom (15.33) "Positivity under unary minus":
                        b ≠ 0  ⇒  (pos b ≡ ¬ pos (- b))
Theorem "Positive implies non-zero": pos a ⇒ a ≠ 0
```

These are rewritten using:

```
Theorem (3.60) "Definition of ⇒":  p ⇒ q ≡ p ∧ q ≡ p
Theorem (3.62):  p ⇒ (q ≡ r) ≡ p ∧ q ≡ p ∧ r
```

In both cases, this rewriting produces precisely what is needed for the respective calculation step.

### 6.3 Theorems as Proof Methods — "Using"

LADM contains, on p. 80, an example for the "proof method" *proof by contrapositive*, almost completely in prose, with only a two-step calculation corresponding to the third and fourth steps in the calculation part of our fully formal proof:

```
Theorem "Example for use of Contrapositive":
    x + y ≥ 2  ⇒  x ≥ 1 ∨ y ≥ 1
Proof:
  Using "Contrapositive":
    Subproof for `¬ (x ≥ 1 ∨ y ≥ 1)  ⇒  ¬ (x + y ≥ 2)`:
        ¬ (x ≥ 1 ∨ y ≥ 1)
      ≡⟨ "De Morgan" ⟩
        ¬ (x ≥ 1) ∧ ¬ (y ≥ 1)
      ≡⟨ "Complement of <" with (3.14) ⟩
        x < 1 ∧ y < 1
      ⇒⟨ "<-Monotonicity of +" ⟩
        x + y < 1 + 1
      ≡⟨ Evaluation ⟩
        x + y < 2
      ≡⟨ "Complement of <" with (3.14) ⟩
        ¬ (x + y ≥ 2)
```

The general pattern for keyword Using is with a hint item and followed by an indented sequence of subproofs:

Using $hi_1$:
  $sp_1$
  ⋮
  $sp_n$

Technically, this is considered as syntactic sugar for a single-hint-item proof using a combined hint item in the pattern explained above:

By $hi_1$ with $sp_1$ and ... and $sp_n$

However, using the By shape would be quite awkward to write for larger subproofs.

Pragmatically, one rather tends to consider "Using" as a *proof method generator* — mutual implication, antisymmetry laws, set extensionality, indirect equality, etc. all are frequently used to produce readable proofs in this way. Since $hi_1$ can again be a combined hint item, the "Using" proof pattern introduces considerable flexibility.

"Using" also liberates the user from the restriction to the induction principles hard-coded for "By induction on": Given, for example the induction principle for sequences with empty sequence $\epsilon$ and list "cons" operator $\_ \triangleleft \_$ (as in LADM):

```
Axiom "Induction over sequences":
    P[xs ≔ ϵ]
    ⇒ (∀ xs : Seq A ❙ P • (∀ x : A • P[xs ≔ x ◁ xs]))
    ⇒ (∀ xs : Seq A • P)
```

The example proof below Using this induction principle also is the first proof we show containing our construct for $\forall$-introduction: "For any 'vs': *proof-for-P*" proves $\forall$ vs $\bullet$ $P$, and "For any 'vs' satisfying '$R$': *proof-for-P*" proves $\forall$ vs $\mid$ $R \bullet P$ while *proof-for-P* may use assumption $R$.

```
Theorem (13.7) "Tail is different":
                  ∀ xs : Seq A • ∀ x : A • x ◁ xs ≠ xs
Proof:
  Using "Induction over sequences":
    Subproof for `∀ x : A • x ◁ ϵ ≠ ϵ`:
      For any `x : A`: By "Cons is not empty"
    Subproof for `∀ xs : Seq A ❙ (∀ x : A • x ◁ xs ≠ xs)
                  • (∀ z : A • (∀ x : A • x ◁ z ◁ xs ≠ z ◁ xs))`:
      For any `xs : Seq A`
           satisfying "Ind. Hyp." `(∀ x : A • x ◁ xs ≠ xs)`:
        For any `z : A`, `x : A`:
            x ◁ z ◁ xs ≠ z ◁ xs
          ≡⟨ "Definition of ≠", "Injectivity of ◁" ⟩
            ¬ (x = z ∧ z ◁ xs = xs)
          ⇐⟨ "De Morgan", "Weakening", "Definition of ≠" ⟩
            z ◁ xs ≠ xs   — This is Assumption "Ind. Hyp."
```

## 7 Activation of Features

The CalcCheck language has actually no hard-coded operators — everything can be introduced by the user via "Declaration" TLIs.

To make available functionality of the proof checker that depends on certain language elements, it is necessary to "Register" operators for built-in operators, and to "Activate" theorems on which built-in functionality relies. For example:

- Equality _=_ and equivalence _≡_ need to be registered to become available for extraction of equations for rewriting.
- *true* needs to be registered in particular for making it possible to omit "— This is (3.4)" at the end of an equivalence calculation ending in true.
- Activation of associativity and symmetry (commutativity) properties is necessary for using the internal AC representation and AC matching for the respective operators, which enables the reasoning up to symmetry and associativity that LADM also adapts throughout.

These first three items are already required for LADM Chapter 3, but only these — to force students to produce proofs conforming to the setting of Chapter 3, the remaining features need to be turned off.

LADM Chapter 4 "Relaxing the Proof Style" introduces the structured proof mechanisms described in Sect. 4 together with a number of other relaxations, that are all justified in terms of Chapter-3-style proofs. Correspondingly, CalcCheck needs to be made aware of these justifications:

- Implication needs to be registered for Assuming and conditional rewriting (Sect. 6.1) to become available.
- Registration of conjunction is required in particular for implicit use of "Shunting" in conditional rewriting, and, as the operator underlying universal quantification, also for implicit use of "Instantiation" (i.e., $\forall$-elimination) in rule extraction from hint items.
- Transitivity of equality and equivalence is built-in, and also transitivity of equality with other operators, as an instance of Leibniz. For two or more non-equality operators to be accepted as calculation operators in the same calculation, the corresponding transitivity law needs to be activated.
- For equality (or equivalence) calculations to be accepted for example when proving an implication, the relevant reflexivity law needs to be activated.
- Activation of converse laws, such as (3.58) "Consequence": $p \Leftarrow q \;\equiv\; p \Rightarrow q$, makes mentioning their use superfluous.
- Activation of monotonicity and antitonicity laws makes it possible to use a style similar to that explained by (Gries, 1997, Sect. 4.1), but not restricted to formulae: Writing "Monotonicity with ..." respectively "Antitonicity with ..." then replaces the deeply-nested with-cascades of monotonicity laws that otherwise are frequently necessary.

Beyond LADM Chapter 4, some further features also depend on declared correspondence of user-defined operators with built-in constructors:

- Disjunction is required for representing set enumerations $\{1, 2\}$ as set comprehensions $\{x \mid x = 1 \vee x = 2\}$.
- Arithmetic operators like $\_+\_$, $\_-\_$, $\_\cdot\_$ and Boolean operators including also $\neg\_$ need to be registered for the keyword hint item Evaluation, seen in the first proof in Sect. 6.3, to be able to evaluate ground expressions.
- The built-in induction mechanisms also require registration of the respective operators.

## 8 Implementation Aspects

CalcCheck is implemented in Haskell, with $\text{CalcCheck}_{\text{Web}}$ using Haste by Ekblad (2016) to compile the client part from Haskell to JavaScript running in the user's web browser, and to generate the client server communication.

The core of proof checking in CalcCheck consists in translating hints into rewrite rules, and attempting to confirm the correctness of individual proof steps by rewriting. For a calculation step "$e_1\ op\langle\ hint\ \rangle\ e_2$", the system will use the rewriting rules derived from *hint* to search for a common reduct of $e_1$ and $e_2$ if *op* is an equality operator, and otherwise (respectively alternatively) attempt

to rewrite "$e_1$ *op* $e_2$" to *true*. Rewriting is mainly performed in depth-limited breadth-first search.

Since the previous, LaTeX-based version of CalcCheck (Kahl, 2011), the term datastructure used in the AC-enabled rewriting engine has seen the addition of binding structures essentially along the lines of Talcott (1993), and also a separate representation of metavariables. As mentioned before, both syntax checking and proof checking run on the server; each time a user triggers checking from a cell, all preceding cells are sent along, since they might contain changes that affect even parsing, and also changes in the theorem names they provide. For each code cell, the theorem names it provides are sent back to the client in addition to the visible feedback, and used for theorem name completion.

For typical use, in particular in the teaching context, CalcCheck "notebooks" consist of two parts: A "prefix" that is preloaded once by the server process, and contains all the theory imports, declarations, local theorems, activations, etc., that should be available everywhere in the user view, and a "suffix" that is displayed in the user's browser as described in Sect. 3. In suffixes, import declarations and certain other features (configurable) are not available, so that the only interaction with the server file system is saving the user state of the suffix into files with server-generated names; saving is restricted to users registered via the local learning management system.

Each attempt to use a hint for justifying some goal (in particular calculation steps) is guarded by a time-out, and for grading, longer time-outs are used. During the recent final exam written on 12 $\text{CalcCheck}_{\text{Web}}$ notebook server processes by 199 students, the 6-core machine acting as server has been observed to occasionally reach loads beyond the equivalent of one core being 100% busy, peaking at 1.4 cores.

## 9 Discussion of Related Work

A system with apparently quite similar goals is Lurch (Carter and Monks, 2017), which lets users use conventional mathematical prose for the top-level structure of proofs, with embedded mathematical formulae marked up (unobtrusively for the prose reader) with their rôles in the mathematical development. Although this may in a certain sense be perceived to be "nicer", it is mainly nicer in the sense of supporting students of mathematics who will be expected to confidently write mathematical prose that will not normally be expected to be subjected to mechanised checking. The goal of CalcCheck however is different: It is targeting future computer scientists and software professionals, who will need to be ready to productively use formal specification languages and automated proof systems of many different kinds, whether these are full-fledged proof assistants like Coq or Isabelle, or model checkers or automated provers like Spin or Prover9, or "modelling languages" like JML. For use of all these systems, precise understanding of issues of scope and variable binding is needed; this is frequently "hand-waived" in conventional mathematical prose. By offering a precise concept of what a proof is, and by being able to force students to produce proofs with varying

levels of detail, CalcCheck also strives to equip students with a mindset from which understanding the limitations of other verification systems will be easier, so that they will be better positioned to use them productively.

A flavour of calculational proof presentation that is slightly different from LADM are the "structured derivations" of Back (2010). These share with CalcCheck the goal of readable fully formal, mechanically checkable proofs; MathEdit by Back et al. (2007) appears to have been a first attempt to provide tool support for this.

## 10 Conclusion

The proofs we arrive at are perhaps not always the ultimate in the elegance the calculational style is famous for, but they are coming close, and by virtue of providing formal syntax for useful kinds of structured proofs, frequently it is actually easier to achieve elegance in CalcCheck than in the calculational style embedded in conventional mathematical prose for larger-scale proof structure. Many students showed significant skills in finding quite elegant and widely different proofs even in exam settings, and student feedback about CalcCheck has been almost unanimously positive.

## References


R.-J. Back. Structured derivations: A unified proof style for teaching mathematics. *Formal Aspects of Computing*, 22(5):629–661, 2010. doi: 10.1007/s00165-009-0136-5.

R.-J. Back, V. Bos, J. Eriksson. MathEdit: Tool support for structured calculational proofs. TUCS Tech. Rep. 854, Turku Centre for Comp. Sci., 2007.

N. C. Carter, K. G. Monks. A web-based toolkit for mathematical word processing applications with semantics. In H. Geuvers et al. (eds.), *Intelligent Computer Mathematics, CICM 2017*, pp. 272–291, Cham, 2017. Springer Intl. doi: 10.1007/978-3-319-62075-6_19.

A. Ekblad. High-performance client-side web applications through Haskell EDSLs. In G. Mainland (ed.), *Proc. 9th Intl. Symp. on Haskell, Haskell 2016*, pp. 62–73. ACM, 2016. doi: 10.1145/2976002.2976015.

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

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

D. Gries, F. B. Schneider. Equational propositional logic. *Inform. Process. Lett.*, 53:145–152, 1995. doi: 10.1016/0020-0190(94)00198-8.

W. Kahl. The teaching tool CalcCheck: A proof-checker for Gries and Schneider's "Logical Approach to Discrete Math". In J.-P. Jouannaud, Z. Shao (eds.), *Certified Programs and Proofs, CPP 2011*, LNCS, vol. 7086, pp. 216–230. Springer, 2011. doi: 10.1007/978-3-642-25379-9_17.

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

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

M. Spivey. *The fuzz type-checker for Z, Version 3.4.1,* and *The fuzz Manual, Second Edition*, 2008. Available from http://spivey.oriel.ox.ac.uk/corner/Fuzz (last accessed 2018-04-15).

C. L. Talcott. A theory of binding structures and applications to rewriting. *Theoret. Comput. Sci.*, 112:68–81, 1993. doi: 10.1016/0304-3975(93)90240-T.