Lucas-Interpretation on Isabelle's Functions

Walther Neuper

Johannes Kepler University, Linz, Austria

`walther.neuper@jku.at`

Software tools of Automated Reasoning are too sophisticated for general use in mathematics education and respective reasoning, while Lucas-Interpretation provides a general concept for integrating such tools into educational software with the purpose to reliably and flexibly check formal input of students.

This paper gives the first technically concise description of Lucas-Interpretation at the occasion of migrating a prototype implementation to the function package of the proof assistant Isabelle. The description shows straightforward adaptations of Isabelle's programming language and shows, how simple migration of the interpreter was, since the design (before the function package has been introduced to Isabelle) recognised appropriateness of Isabelle's terms as middle end.

The paper gives links into the code in an open repository as invitation to readers for re-using the prototyped code or adopt the general concept. And since the prototype has been designed before the function package was implemented, the paper is an opportunity for recording lessons learned from Isabelle's development of code structure.

1 Introduction

This paper concerns application of Automated Reasoning (AR) to education, the issue to narrow the gap between powerful, but highly sophisticated technologies of AR on the one side and requirements of education in mathematics as taught in engineering studies. Lucas-Interpretation (in the sequel abbreviated by *LI*) provides a general concept for integrating AR tools into educational software with the purpose to reliably and flexibly check formal input of students.

A prerequisite for *LI* is a logical framework, which models rigorous formal derivation, in particular forward reasoning. There are several educational software products of this kind: [4] calls "structured-derivation" what here is called "calculation" and uses the PVS prover in the background, IMPS is an Interactive Mathematical Proof System intended to provide organizational and computational support for the traditional techniques of mathematical reasoning [11], and in geometry software AR tools are used [5, 15], too. But all the above mentioned systems are poor in guided interaction.

A logic course for freshmen in AI and Computer Science[1] accompanies all content with software. And corresponding experience shows [8] again, that much of the software lacks interactivity and feedback sufficient for independent learning[2] in particular in application to mathematics.

In principle [32], AR provides the most powerful tools for independent learning in formal mathematics, captured by naive requirements like these: given a formal problem specification, AR checks steps of forward reasoning towards a problem solution, a step determined by an input term or by an input theorem to be applied to the current state of solution; and the system can provide a next step, if a student gets stuck. Nowadays such naive requirements might be reconsidered, since proof assistants like Isabelle [37] hide the intricacies of AR tools and present notation to the user close to standard mathematics.

---

[1] `http://fmv.jku.at/logic/`

[2] Currently independent learning is best addressed by "flipped classes" [40]

LI aims at meeting these requirements for more than a decade, when Peter Lucas[3] shifted his interests from programming languages [24] to education. His specific contribution has been named after him, and now reveals the ingenuity of the original design, when the prototype has been migrated: a proprietary version [22] was migrated to Isabelle's function package [20, 21] with surprising little effort.

Herewith the first technically concise description of *LI* is given, after theoretical considerations [27] and application oriented discussions [28, 29, 30]. Since *LI* was implemented in the *ISAC*-project[4] several years before the function package was introduced to the proof assistant Isabelle, we take the paper as an opportunity for recording lessons learned from Isabelle's development.

The paper is structured as follows: §2 introduces *LI*, the concept in §2.1 illustrated with examples in §2.2. §2.3 notes *LI*'s relevance for educational mathematics software. Adaptation of the programs for *LI* to Isabelle's function package is described in §3, where §3.1 describes the tactics, §3.2 the tacticals and §3.3 briefly describes evaluation by rewriting. The parse-tree generated by the function package is interpreted by *LI* as shown in §4; §4.1 gives a description of how the parse-tree is scanned, §4.2 of how *LI* uses Isabelle's *Proof.context* and §4.3 of how the prototype's mathematics-engine embeds and guards *LI*. Lessons have been learned from Isabelle's code structure and development process §5.1, from specific Isabelle features (§5.2) and now direct future prototyping (§5.3). In §6 the summary concludes with expectations on *LI* widely applied to education in mathematics.

2   Lucas–Interpretation (*LI*)

The interpreter is named after the inventor of top-down-parsing in the ALGOL project [24], Peter Lucas. As a dedicated expert in programming languages he initially objected "yet another programming language" (in analogy to "Yacc" [16]), but then he helped to clarify the unusual requirements for a novel programming language in the *ISAC*-project, which later led to the notion of *LI*.

LI is the most prominent component in a prototype developed in the *ISAC*-project, there embedded in a mathematics-engine, which interacts with a dialogue-module in a Java-based front-end managing interaction with students (briefly touched below in §2.3).

2.1   The Concept of *LI*

The concept of *LI* is simple: *LI acts as a debugger on functional programs with hard-coded break-points, where control is handed over to a student; a student, however, does* not *interact with the software presenting the program, but with a software presenting steps of forward reasoning, where the steps are rigorously constructed by tactics acting as the break-points mentioned.* Following the *LI* terminology, we will call "programs" the functions defined with the function package and refer to their evaluation as "execution".

Types occurring in the signatures   of *LI* are as follows. Besides the program of type `Program.T` there is an interpreter state `Istate.T`, a record type passing data from one step of execution to the next step, in particular a location in the program, where the next step will be read off, and an environment for evaluating the step. As invisible in the program language as the interpreter state is `Calc.T`, a "calculation"

---

[3]`https://austria-forum.org/af/AustriaWiki/Peter_Lucas_(Informatiker)`
[4]`https://isac.miraheze.org/wiki/History`

as a sequence of steps in forward reasoning, a variant of "structured derivations" [3]. Visible in the language and in the signature, however, are the tactics `Tactic.T`, which create steps in a calculation.

Novel in connection with calculations is the idea to maintain a logical context in order to provide automated provers with data required for checking input by the student. Isabelle's `Proof.context`, as is, turned out perfect for this task [23].

The signatures of the main functions, the functions `find_next_step`, `locate_input_tactic` and `locate_input_term`, are as follows:

```
signature LUCAS_INTERPRETER =
sig
  datatype next_step_result =
      Next_Step of Istate.T * Proof.context * Tactic.T
    | Helpless | End_Program of Istate.T * Tactic.T
  val find_next_step: Program.T -> Calc.T -> Istate.T -> Proof.context
    -> next_step_result

  datatype input_tactic_result =
      Safe_Step of Istate.T * Proof.context * Tactic.T
    | Unsafe_Step of Istate.T * Proof.context * Tactic.T
    | Not_Locatable of message
  val locate_input_tactic: Program.T -> Calc.T -> Istate.T -> Proof.context
      -> Tactic.T -> input_tactic_result

  datatype input_term_result =
    Found_Step of Calc.T | Not_Derivable
  val locate_input_term: Calc.T -> term -> input_term_result
end
```

`find_next_step` accomplishes what usually is expected from a `Program.T`: find a `Next_Step` to be executed, in the case of *LI* to be inserted into a `Calc.T` under construction. This step can be a `Tactic.T`, directly found in `next_step_result`, or a `term` produced by applying the tactic. If such a step cannot be found (due to previous student interaction), *LI* is `Helpless`.

`locate_input_tactic` gets a `Tactic.T` as argument (which has been input and checked applicable in `Calc.T`) and tries to locate it in the `Prog.T` such that a `Next_Step` can be generated from the subsequent location in the `Program.T`. A step can be an `Unsafe_Step`, if the input `Tactic.T` cannot safely be associated with any tactic in the `Program.T`. This function has a signature similar to `find_next_step` (here the respective input `Tactic.T` and the one in the result are the same) in order to unify internal technicalities of *LI*.

`locate_input_term` tries to find a derivation from the current `Proof.context` for the term input to `Calc.T` and to locate a `Tactic.T` in the `Program.T`, which as `Found_Step` allows to find a next step later. Such a `Found_Step` can be located in another program (a sub-program or a calling program); thus `Program.T`, `Istate.T` and `Proof.context` are packed into `Calc.T` at appropriate positions and do not show up in the signature.

AR is concern of the third function `locate_input_term`, actually a typical application case for Isabelle's Sledgehammer but not yet realised and preliminarily substituted by *ISAC*'s simplifier. `locate_input_term` is the function used predominantly in interaction with students: these input term by term into

the calculation under construction (as familiar from paper&pencil work) and *LI* checks for correctness automatically.

2.2   Examples for *LI*

LI's novel concept relating program execution with construction of calculations is best demonstrated by examples. The first one is from structural engineering given by the following problem statement:

> *Determine the bending line of a beam of length L, which consists of homogeneous material, which is clamped on one side and which is under constant line load q_0; see the Figure right.*
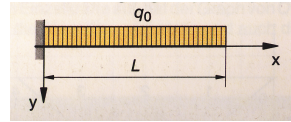
This problem is solved by the above program, which is shown by a screen-shot[5] This problem is solved by the above program, which is shown by a screen-shot[6] in order to demonstrate the colouring helpful for programmers (compare the old bare string version in [22, p. 92]). The program is implemented

```
partial_function (tailrec) biegelinie ::
  "real ⇒ real ⇒ real ⇒ (real ⇒ real) ⇒ bool list ⇒ real list ⇒ (real ⇒ real) ⇒ bool"
  where
"biegelinie l q v b s vs id_abl = (
  let
    funs = SubProblem (''Biegelinie'', [''vonBelastungZu'', ''Biegelinien''],
      [''Biegelinien'', ''ausBelastung'']) [REAL q, REAL v, REAL_REAL b, REAL_REAL id_abl];
    equs = SubProblem (''Biegelinie'', [''setzeRandbedingungen'', ''Biegelinien''],
      [''Biegelinien'', ''setzeRandbedingungenEin'']) [BOOL_LIST funs, BOOL_LIST s];
    cons = SubProblem (''Biegelinie'', [''LINEAR'', ''system''],
      [''no_met'']) [BOOL_LIST equs, REAL_LIST vs];
    B = Take (lastI funs);
    B = ((Substitute cons) #> (Rewrite_Set_Inst [(''bdv'', v)] ''make_ratpoly_in'')) B
  in B)"
```

Figure 1: The program implemented by the function package

as a `partial_function`: it calculates results for a whole class of problems and termination cannot be proven in such generality, although preconditions guard execution. The program reflects a "divide & conquer" strategy by calling three `SubProblems`. It was implemented for field-tests in German-speaking countries, so arguments of the `SubProblems` are in German, because they appear in the calculation as well. The third `SubProblem` adopts the format from Computer Algebra. The calculation, if finished successfully, looks as follows (a screen-shot of intermediate steps on the prototype's front-end is shown in [22, p. 97]):

```
01  Problem (Biegelinie, [Biegelinien])
02      Specification:
03      Solution:
04          Problem (Biegelinie, [vonBelastungZu, Biegelinien])
```

---

[5]The corresponding code is at `https://hg.risc.uni-linz.ac.at/wneuper/isa/file/df1b56b0d2a2/src/Tools/isac/Knowledge/Biegelinie.thy#l174`.

[6]The corresponding code is at `https://hg.risc.uni-linz.ac.at/wneuper/isa/file/df1b56b0d2a2/src/Tools/isac/Knowledge/Biegelinie.thy#l174`.

05     $[\ \ V\,x = c + -1 \cdot q_0 \cdot x,$

06       $M_b\,x = c_2 + c \cdot x + \frac{-1 \cdot q_0}{2 \cdot x^2},$

07       $\frac{d}{dx}y\,x = c_3 + \frac{-1}{EI} \cdot (c_2 \cdot x + \frac{c}{2 \cdot x^2} + \frac{-1 \cdot q_0}{6 \cdot x^3},$

08       $y\,x = c_4 + c_3 \cdot x + \frac{-1}{EI} \cdot (\frac{c_2}{2} \cdot x^2 + \frac{c}{6} \cdot x^3 + \frac{-1 \cdot q_0}{24} \cdot x^4)\ ]$

09   Problem (Biegelinie, [setzeRandbedingungen, Biegelinien])

10     $[\ \ L \cdot q_0 = c,\ 0 = \frac{2 \cdot c_2 + 2 \cdot L \cdot c + -1 L^2 \cdot q_0}{2},\ 0 = c_4,\ 0 = c_3\ ]$

11   solveSystem $(L \cdot q_0 = c,\ 0 = \frac{2 \cdot c_2 + 2 \cdot L \cdot c + -1 \cdot L^2 \cdot q_0}{2},\ 0 = c_4,\ 0 = c_3\ ],\ [c, c_2, c_3, c_4])$

12     $[\ \ c = L \cdot q_0,\ c_2 = \frac{-1 \cdot L^2 \cdot q_0}{2},\ c_3 = 0,\ c_4 = 0]$

13   Take $y\,x = c_4 + c_3 \cdot x + \frac{-1}{EI} \cdot (\frac{c_2}{2} \cdot x^2 + \frac{c}{6} \cdot x^3 + \frac{-1 \cdot q_0}{24} \cdot x^4)$

14     $y\,x = c_4 + c_3 \cdot x + \frac{-1}{EI} \cdot (\frac{c_2}{2} \cdot x^2 + \frac{c}{6} \cdot x^3 + \frac{-1 \cdot q_0}{24} \cdot x^4)$

15   Substitute $[c, c_2, c_3, c_4]$

16     $y\,x = 0 + 0 \cdot x + \frac{-1}{EI} \cdot (\frac{\frac{-1 \cdot L^2 \cdot q_0}{2}}{2} \cdot x^2 + \frac{L \cdot q_0}{6} \cdot x^3 + \frac{-1 \cdot q_0}{24} \cdot x^4)$

17   Rewrite_Set_Inst $([[(bdv, x)], make\_ratpoly\_in)$

18     $y\,x = \frac{q_0 \cdot L^2}{4 \cdot EI} \cdot x^2 + \frac{L \cdot q_0}{6 \cdot EI} \cdot x^3 + \frac{q_0}{24 \cdot EI} \cdot x^4$

19 $y\,x = \frac{q_0 \cdot L^2}{4 \cdot EI} \cdot x^2 + \frac{L \cdot q_0}{6 \cdot EI} \cdot x^3 + \frac{q_0}{24 \cdot EI} \cdot x^4$

The calculation is what a student interacts with. Above the `Specification` is folded in, the specification phase and respective user interaction is skipped here, because user-interaction in this phase is not guided by *LI* (while the result of the phase, a concise formal specification of the problem is a necessary pre-condition for *LI*). The `Solution` must be constructed step-wise line by line (while the line numbers do not belong to the calculation) in forward reasoning. A student inputs either (1) a term (displayed on the left above with indentations) or (2) a tactic (shifted to the right margin). If the student gets stuck (3) a next step (as a term or a tactic) is suggested by *LI*, which works behind the scenes; the corresponding functions have been introduced in §2.1 (for (1) `locate_input_term`, (2) `locate_input_tactic` and for (3) `find_next_step`). A student can combine actions (1)..(3) arbitrarily.

This example hopefully clarifies the relation between program and calculation in *LI*: execution of the program stops at tactics (above called break-points), displays the tactic or the result produced by the tactic (as decided by the dialogue-module) in the calculation under construction, hands over control to the student working on the calculation and resumes checking user input.

A typical example from Computer Algebra shall shed light on a notions introduced later, on tacticals. The program in Fig.2 simplifies rational terms (typed as "real" due to limited development resources):

```
partial_function (tailrec) simplify :: "real ⇒ real"
  where
"simplify term = (
  (Try (Rewrite_Set ''discard_minus'') #> Try (Rewrite_Set ''rat_mult_poly'') #>
   Try (Rewrite_Set ''make_rat_poly_with_parentheses'') #>
   Try (Rewrite_Set ''cancel_p_rls'') #>
   (Repeat (
     Try (Rewrite_Set ''add_fractions_p_rls'') #>
     Try (Rewrite_Set ''rat_mult_div_pow'') #>
     Try (Rewrite_Set ''make_rat_poly_with_parentheses'') #>
     Try (Rewrite_Set ''cancel_p_rls'') #> Try (Rewrite_Set ''rat_reduce_1''))) #>
   Try (Rewrite_Set ''discard_parentheses1''))
  term)"
```

Figure 2: Simplification implemented by the function package

The program executes the tactic `Rewrite'_Set`[7] with different canonical term rewriting systems (called "rule sets" in *ISAC*) guided by tacticals. Chained functions (by `#>`) are applied curried to `term`. The initial part of the chain, from `''discard_minus''` to `''cancel''` does preprocessing, for instance replaces $a-b$ by $a+(-b)$ in `''discard_minus''` for reducing the number of theorems to be used in simplification. The second chain of `Rewrite_Set` is `Repeated` until no further rewrites are possible; this is necessary in case of nested fractions.

In cases like the one above, a confluent and terminating term rewrite system, not only *all* correct input terms are accepted, but also *all* incorrect input is rejected as `Not_Derivable` by the function `locate_-input_term`.

2.3   *LI*'s Relevance for Educational Mathematics Software

Main concern of this paper is describing technical aspects of *LI*. But here is a brief summary of promises for education, which should make the paper selfcontained for educators — promises derived from *LI*'s three functions introduced on p.81 and interpreted with respect to students' interaction on p.83.

***LI* is generally usable**   by all software, which constructs a solution for a given problem — this, in principle, includes all programs, for instance, written in Mathematica [8] and millions of educational software products: just (re-)write the programs calculating a solution as shown in Fig.1. Doing so would cover most of mathematics as taught at engineering faculties as well as taught at high-schools.

Of course, that is an endless deal of work: but it can be done by lecturers and teachers, who like writing software for the problems they teach. And, of course, in order to make their development efficient, comprehensive libraries need to be available free of charge (like some of the libraries available for Mathematica). Such libraries would be concern of publicly funded academic development projects.

Such projects can build upon mechanised mathematics knowledge, which goes far beyond the contents of undergraduate studies (for instance in Isabelle's archive of formal proofs [1]) in mathematics and in engineering studies. From a certain point, development would become efficient and sustainable, because based on software with a huge user community extending the communities working on mechanisation of mathematics knowledge.

***LI* features "models of mathematics",**   transparent, complete and interactive models, as already described in [32] and briefly repeated here. Mathematical problem solutions (in the sequel called *calculations*) can be "transparent" by mouse-click on an element of a calculations, which immediately leads to the respective definition [9] and related properties together with (more or less) readable proofs. Thus mathematical calculations can be "complete" with respect to deduction from first principles (where students can follow these deductions at their own pace, or even not). And mathematical calculations can be "interactive" relying on *LI* as described in this paper.

What makes such calculations to "models of mathematics" is the fact, that it covers all of mathematics which can be mechanised, while most computer mathematicians agree that (almost all of) mathematics can be mechanised (as soon as it is fully understood). And each item of knowledge and each step in a calculations is proven correct, unambiguously and based on logic — which fullfills the main property

---

[7]Isabelle's document preparation system preserves the apostroph from the definition, while Isabelle's pretty printer drops it in the presentation of the program as above. The apostroph excapes the underscore as required by Isabelle's lexer.

[8]`https://www.wolfram.com/mathematica/online/`

[9]This feature is known from software development environments (IDEs). Since Isabelle/jEdit realises this feature on terms [43], it appears an indispensable requirement for mathematics software at the state of the art.

of mathematics, distinguishing it from other sciences. As, in history of science, rigorous logic was important to overcome belief and magic, comprehending this main property is an important goal for education in mathematics and in science. However, educational practice shows, that this goal is hard to achieve.

The educational gain from mechanised "models of mathematics" can be seen as follows: Many students *prefer explanations in human language over looking at the structure of formulas* — where the latter is indispensable for developing confidence in mathematics and for overcoming the impression of authority or magic eventually. Software works mechanically on formulas (and bypasses natural language). Thus it continuously offers direct experience with the mechanical nature of mathematics (the science of mechanisation of thinking [6]) — at any time a student is ready to recognise it, a steady mechanical offer to mature the understanding of formal language. Such understanding wouldmake academic lectures on formal logic, on meta-mathematics, on theory of science and the like better comprehensible. And, probably, freshmen would start with more confidence (and in larger numbers) into science studies.

***LI* features "systems that explain themselves"** as already addressed in [31, 32] and increases the range of learning scenarious to be supported by software: from high-school up to elderly, who want to understand "what mathematics really is", from organised learning at various institutions up to individual study in private settings, from independent learning via learning by trial and error via "flipped classroom instruction" [9], private teaching and exercising difficult problem solutions up to written exams with automated evaluation.

The crucial feature for systems that explain themselves in mathematics is `find_next_step` as introduced on p.81: a student always can request a hint from the system for a next step towards a solution — and this feature also makes clear, that flexible limitation is needed to adapt to realistic learning scenarios. Such adpation is concern of a dialogue-module: by use of pattern-matching on the next step a hint can be given in full or partially (from a formula or a tactic, see the example calculation on p.82), can be a list of options or can be idle (in case of a written exam).

Such a dialogue-module is implemented as a stub in *ISAC*'s prototype [22], but several field tests [35, 33, 34] already clarified a lot of requirements for such a module; realisation shall be in collaboration with experts in educational psychology — which exhibits a great advantage of *LI* over present principles of educational software: design and development in computer mathematics (writing *LI*-programs, mechanising deductive knowledge) are strictly separated from design and development of dialogues (extending prototypes like [18] and adopting educational theories).

3   Adaptation of Isabelle's Functions

Isabelle's function package presents functions in "inner syntax" to users, i.e. as terms in Isabelle/HOL. The *LI* design recognised these terms suitable for parse trees of programs, Isabelle realised the same idea in a more general way with the function package a few years later — so migration from *ISAC*'s programs to Isabelle's functions was surprisingly easy. The main features required were tactics, tacticals and program expressions as described below.

3.1 Tactics Creating Steps in Calculations

The examples in the previous section showed how tactics in programs create steps in calculations. Tactics in programs are defined as follows[10] :

```
consts
  Calculate          :: "[char list, 'a] => 'a"
  Rewrite            :: "[char list, 'a] => 'a"
  Rewrite'_Inst      :: "[(char list * 'a) list, char list, 'a] => 'a"
  Rewrite'_Set       :: "[char list, 'a] => 'a"
  Rewrite'_Set'_Inst :: "[((char list) * 'a) list, char list, 'a] => 'a"
  Or'_to'_List       :: "bool => 'a list"
  SubProblem         ::
    "[char list * char list list * char list list, arg list] => 'a"
  Substitute         :: "[bool list, 'a] => 'a"
  Take               :: "'a => 'a"
```

These tactics transform a term of some type `'a` to a resulting term of the same type. `Calculate` applies operators like $+, -, *$ as `char list` (i.e. a string in inner syntax) to numerals of type `'a`. The tactics beginning with `Rewrite` do exactly what they indicate by applying a theorem (in the program given by type `char list`) or a list of theorems, called "rule-set". The corresponding `_Inst` variants instantiate bound variables with respective constants before rewriting (this is due to the user requirement, that terms in calculations are close to traditional notation, which excludes λ-terms), for instance modelling bound variables in equation solving. `Or_to_List` is due to a similar requirement: logically appropriate for describing solution sets in equation solving are equalities connected by $\wedge, \vee$, but traditional notation uses sets (and these are still lists for convenience). `SubProblems` not only take arguments (`arg_list` like any (sub-)program, but also three references into *ISAC*'s knowledge base (theory, formal specification, method) for guided interaction in the specification phase.

Tactics appear simple: they operate on terms adhering to one type — different types are handled by different tactics and (sub-) programs; and they cover only basic functionality — but they operate on terms, which are well tooled by Isabelle and which can contain functions evaluated as program expressions (which will be introduced in §3.3 below).

Section §2.2 showed how tactics can be input by students. So tactics in programs have analogies for user input with type `Tactic.input` defined at[11]. These tactics also cover the specification phase (wich is out of scope of the paper). And there is another `Tactic.T` defined at[12] for internal use by the mathematics-engine, which already appeared in the signature of Lucas-Interpretation on p.81.

3.2 Tacticals Guiding Flow of Execution

The example on p.83 for canonical rewriting showed, how tacticals guide the flow of execution. The complete list of tacticals is as follows.

---

[10]https://hg.risc.uni-linz.ac.at/wneuper/isa/file/df1b56b0d2a2/src/Tools/isac/ProgLang/Prog_
Tac.thy#l36
[11]https://hg.risc.uni-linz.ac.at/wneuper/isa/file/df1b56b0d2a2/src/Tools/isac/MathEngBasic/
tactic-def.sml#l122
[12]https://hg.risc.uni-linz.ac.at/wneuper/isa/file/df1b56b0d2a2/src/Tools/isac/MathEngBasic/
tactic-def.sml#l241

```
consts
  Chain    :: "['a => 'a, 'a => 'a, 'a] => 'a" (infixr "#>" 10)
  If       :: "[bool, 'a => 'a, 'a => 'a, 'a] => 'a"
  Or       :: "['a => 'a, 'a => 'a, 'a] => 'a" (infixr "Or" 10)
  Repeat   :: "['a => 'a, 'a] => 'a"
  Try      :: "['a => 'a, 'a] => 'a"
  While    :: "[bool, 'a => 'a, 'a] => 'a" ("((While (_) Do)//(_))" 9)
```

`Chain` is forward application of functions and made an infix operator #> ; `If` decides by `boolean` expression for execution of one of two arguments of type `'a` *Rightarrow* `'a` (which can be combinations of tacticals or tactics); `Or` decides on execution of one of the arguments depending of applicability of tactics; `Repeat` is a one way loop which terminates, if the argument is not applicable (e.g. applicability of a theorem for rewriting on a term) any more; `Try` skips the argument if not applicable and `While` is a zero way loop as usual.

3.3   Program Expressions to be Evaluated

Some tacticals, `If` and `While`, involve boolean expressions, which need to be evaluated: such expressions denote another element of programs. This kind of element has been shown in the example on p.82 as argument of `Take`: sometimes it is necessary to pick parts of elements of a calculation, for instance the last element from a list. So Isabelle's `List` is adapted for *ISAC*'s purposes in `ListC`[13]. Such expressions are substituted from the environment in `Istate.T`, evaluated to terms by rewriting (and for that purpose using the same rewrite engine as for the tactics `Rewrite*`) and marked by the constructor `Term_Val` which is introduced below.

4   Implementation of *LI*

The implementation of the interpreter is as experimental as is the respective programming language introduced in the previous section. So below there will be particularly such implementation details, which are required for discussing open design & implementation issues.

All of the function package's syntactic part plus semantic markup is perfect for *LI*. The evaluation part of the function package, however, implements automated evaluation in one go and automated code-generation [12] — both goals are not compatible with *ISAC*'s goal to feature step-wise construction of calculations, so this part had to be done from scratch.

4.1   Scanning the Parse Tree

Isabelle's function package parses the program body from function definitions to terms, the data structure of simply typed λ terms, which also encode the objects of proofs. Thus there is a remarkable collection of tools, readily available in Isabelle; but this collection does not accommodate the requirement of scanning a term to a certain location, remembering the location and returning there later, as required by *LI*. So this has been introduced

```
datatype lrd = L | R | D
type path = lrd list
```

---

[13]https://hg.risc.uni-linz.ac.at/wneuper/isa/file/df1b56b0d2a2/src/Tools/isac/ProgLang/ListC.thy

```
fun at_location [] t = t
  | at_location (D :: p) (Abs(_, _, body)) = at_location p body
  | at_location (L :: p) (t1 $ _) = at_location p t1
  | at_location (R :: p) (_ $ t2) = at_location p t2
  | at_location l t =
    raise TERM ("at_location: no " ^ string_of_path l ^ " for ", [t]);
```

with a `path` to a location according to the term constructors `$` and `Abs`. This is an implementation detail; an abstract, denotational view starts with this datatype

```
datatype expr_val =
    Term_Val of term   | Reject_Tac
  | Accept_Tac of Istate.pstate * Proof.context * Tactic.T
```

which models the meaning of an expression in the parse-tree: this is either a `Term_Value` (introduced in §3.3) or a tactic (introduced in §3.1); the latter is either accepted by `Accept_Tac` or rejected by `Reject_-Tac`; an error value is still missing. The arguments of the above constant `Accept_Tac` are the same as introduced for *LI* in §2.1. Thus `expr_val` is the return value for functions scanning the parse-tree for an acceptable tactic:

```
scan_to_tactic: term * (Calc.T * Proof.context) -> Istate.T -> expr_val;
scan_dn: Calc.T * Proof.context -> Istate.pstate -> term -> expr_val;
go_scan_up: term * (Calc.T * Proof.context) -> Istate.pstate -> expr_val;
scan_up: term * (Calc.T * Proof.context) -> Istate.pstate -> term -> expr_val;
```

The first argument of each function above, if of type `term`, is the whole program required for scanning given a particular `path` using `at_location`. This is not required by `scan_dn`, which is simple top-down scanning. The rightmost argument, if of type `term`, serves matching in `scan_dn` and in `scan_up` [14]

The central function is `scan_to_tactic` which accomplishes what the identifier indicates: if a student creates a next step at a current position in the `Solution` of a problem, then this position is associated with a certain `path` in the parse-tree, see example on p.82. *LI* supports creating steps by one of the three actions described on p.83. According to the action chosen *LI* scans from the certain `path` to the next tactic found in the parse-tree in order to either check the input formula (1) or tactic (2), or to propose a next step (3). Function `scan_to_tactic` calls either `scan_dn`, if interpretation has just started a new program (`path = []`) or `scan_dn` via `go_scan_up`:

```
fun scan_to_tactic (prog, cc) (Pstate (ist as {path, ...})) =
    if path = []
    then scan_dn cc (ist |> set_path [R]) (Program.body_of prog)
    else go_scan_up (prog, cc) ist
  | scan_to_tactic _ _ =
    raise ERROR "scan_to_tactic1: uncovered pattern in fun.def"
```

`scan_dn` sets the path in the interpreter state `ist` to `R`, the program body, while `go_scan_up` goes one level up the path in order to call `scan_up`. The latter uses the `path` found in `ist`, the interpreter state from the previous step.

---

[14]The code for these functions is found at `https://hg.risc.uni-linz.ac.at/wneuper/isa/file/113997e55e71/src/Tools/isac/Interpret/lucas-interpreter.sml#l125` and at `https://hg.risc.uni-linz.ac.at/wneuper/isa/file/113997e55e71/src/Tools/isac/Interpret/lucas-interpreter.sml#l192` respectively.

It might be helpful to state, that the functions `scan_dn` and `scan_up` need not be mutually recursive: as soon as these functions have found their goal, an appropriate tactic, *LI* terminates, stores the current state and hands over control to the student. Afterwards, depending on the `path`, either `scan_dn` or `scan_up` are called, so they are independent from each other.

4.2   Use of Isabelle's Contexts

Isabelle's *"logical context represents the background that is required for formulating statements and composing proofs. It acts as a medium to produce formal content, depending on earlier material (declarations, results etc.)."* [41]. The *ISAC*-prototype introduced Isabelle's `Context` in collaboration with a student [23], now uses them throughout construction of problem solutions and implements a specific structure `ContextC`[15].

At the beginning of the specification phase (briefly touched by tactic `SubProblem` in §2.2 and explained in little more detail in the subsequent section) an Isabelle theory must be specified, this is followed by `Proof_Context.init_global` in the specification module. Then `Proof_Context.initialise'` takes a formalisation of the problem as strings, which are parsed by `Syntax.read_term` using the context, and finally the resulting term's types are recorded in the context by `Variable.declare_constraints`. The latter relieves students from type constraints for input terms. Finally, as soon as a problem type is specified, the respective preconditions are stored by `ContextC.insert_assumptions`.

During *LI* the context is updated with assumptions generated by conditional rewriting and by switching to and from sub-programs. An example for the former is tactic `SubProblem` applied with theorem $x \neq 0 \Rightarrow (\frac{a}{x} = b) = (a = b \cdot x)$ during equation solving.

A still not completely solved issue is switching to and from `SubProblem`s; the scope of an interpreter's environment is different from a logical context's scope. When calling a sub-program, *ISAC* uses `Proof_Context.initialise`, but returning execution from a `SubProblem` is not so clear. For instance, if such a sub-program determined the solutions $[x = 0, \ x = \sqrt{2}]$, while the calling program maintains the assumption $x \neq 0$ from above, then the solution $x = 0$ must be dropped, this is clear. But how determine in full generality, which context data to consider when returning execution to a calling program? Presently the decision in `ContextC.subpbl_to_caller`[16] is, to transfer all content data which contain at least one variable of the calling program and drop them on contradiction.

4.3   Guarding and Embedding Execution

`partial_functions` as used by *LI* are alien to HOL for fundamental reasons. And when the *ISAC*-project started with the aim to support learning mathematics as taught at engineering faculties (in that generality, see §2.3), it was clear, that formal specifications should guard execution of programs under *LI* (i.e. execution only starts, when the specification's preconditions evaluate to true).

So formal specification is required for technical reasons *and* for educational reasons in engineering education. The *ISAC*-project designed a separate specification phase, where input to a problem and corresponding output as well as preconditions and post-condition are handled explicitly by students; example in Fig.1 shows several `SubProblem`s, which lead to mutual recursion between specification phase

---

[15]A closing "C" indicates an *ISAC* extension, see `https://hg.risc.uni-linz.ac.at/wneuper/isa/file/00612574cbfd/src/Tools/isac/CalcElements/contextC.sml`

[16]`https://hg.risc.uni-linz.ac.at/wneuper/isa/file/ce071aa3eae4/src/Tools/isac/CalcElements/contextC.sml#l73`

and phases creating a solution (the latter supported by *LI*). Embedding *LI* into a dialogue-module is required in order to meet user requirements as already discussed in §2.3.

5   Lessons Learned ...

As already mentioned, development of Isabelle and development of *LI* went in parallel for a long time — a great opportunity for learning in the *ISAC*-project.

5.1   ... from the Isabelle Project

"*Isabelle was not designed; it evolved. Not everyone likes this idea*" said Lawrence C. Paulson in "Isabelle: The Next 700 Theorem Provers" [38].

When the *ISAC*-project started about the year 2000, Isabelle's code structure still reflected the enormous efforts of Paulson to make a great idea a usable product. Such a situation naturally leads to code, where placement of code is determined by the context of new feature requests. In the meanwhile Isabelle evolved to a didactic model in functional programming at a large scale: polymorphic higher order functions take complex function-arguments, which allow to postpone type definitions to locations according to functionality; and such functionality is distilled to small abstract structures. Together with canonical argument order, function combinators and canonical iteration [41, p. 15-17] this gives elegant code with almost no glue, so the hint "never copy & paste a piece of code" could disappear from the implementation manual some time ago. And now a layered structure becomes apparent, best reflected by exception hierarchies. So Isabelle is in a state, where "everything can be changed anywhere in the code" in research&development.

Also Isabelle's development process is a didactic model in efficient collaborative development distributed all around the world and in minimisation of administrative efforts. Visible outcomes from this process are formally checked documentation and a code repository with minimal change-sets, which denote essential feature changes with a minimum of updated code.

The *ISAC*-project started in a situation, where software and user requirements might have been as unclear as with early Isabelle, albeit on another level. *ISAC*'s present code structure still looks much more like (very!) early Isabelle than present Isabelle. This paper was the occasion to make *LI*-related components in *ISAC* as close to Isabelle's style — and it is a hard experience, that this cannot be done in one go and will require many "rounds of reform": *ISAC*'s code structure, even around *LI*, is still far off Isabelle's quality. This is the same with minimal change-sets: in the present state of *ISAC*'s code so much improvements occur along current work, that it is just inefficient to separate respective change-sets in many cases.

5.2   ... from Isabelle's Function Package

When the development of *ISAC* started, a glance at Isabelle's front-end convinced everyone, that educational software could *not* use it. Now, two decades later, it appears clear, that *ISAC* is best advised to re-use Isabelle/JEdit based on Isabelle/PIDE, the integrated proof development environment as the state of the art. This is particularly evident from the work presented in this paper, shifting Lucas-Interpretation into Isabelle/Isar's function definition, which makes all the advanced features of Isabelle/JEdit available for the working programmer:

- **Syntax errors** are indicated accurately at the right location; finding errors in programs represented as strings was a nightmare, if programs comprised more than a couple lines of code.

- **Type annotations** disappear from the program and sidestep to the heading signature; the result is much better readable.

- **Syntax highlighting** indicates how identifiers are interpreted, as constants, as free variable, as strings, etc — very instructive for working programmers.

- **Free variables** on the right-hand-side of assignments are rejected by the function package, while these were accepted by term parsing.

- **Semantic annotations** support the programmer, in particular the tooltip popups triggered by hovering and clicking with the mouse, see [42, p. 30].

These features came for free, when *ISAC*'s programs were shifted into Isabelle/Isar's function definition — and were immediately fruitful: Not only the implementation of programs is much more efficient, also errors have been revealed. The features helped to detect additional free variables on the right-hand side in programs (see for instance this kind of errors in [22, p. 92]) and triggered improved handling of program arguments in *LI*.

5.3   ... for Further *ISAC*-Development

What *ISAC* can learn from Isabelle's code structure and from Isabelle's development process has been described in §5.1, while Isabelle's packaging and deployment has not yet been mentioned, *ISAC* would benefit as well.

Future development in *ISAC* will follow a list of steps as announced in [22, p. 102-103]; the work on *LI* presented in this paper is the first respective step and confirms the relevance of the other steps. However, in the meanwhile difficulties in funding a corresponding project became apparent. Indeed, the many challenges already identified in [13], suggest intermediate steps on the way to educational software for engineering disciplines as aimed at in the *ISAC*-project.

A case study on GCD [28] already investigated possibilities for such intermediate steps: the Euclidean Algorithm creates a "calculation" as presented above, if the respective invariant (or fixpoint) is output at each recursive call of the algorithm — in this example *LI* could immediately be used to explain and to exercise calculation of the greatest common divisor of integers and polynomials. So the next challenge is to adapt *LI* such that arbitrary functions can be executed step-wise — as an experimental approach to study algorithms in the Archive of Formal Proofs [17]. Algorithms in formal logic like Resolution, Binary Constraint Propagation or the DPLL algorithms can be implemented for step-wise interpretation by *LI* already at the present state.

Such intermediate steps postpone the requirement of 2-dimensional term representation on screen — the requirement identified as most urgently missing in *ISAC*'s field tests, and also not provided by Isabelle/jEdit: Isabelle's line-oriented presentation of terms perfectly accomplishes user requirements for proof assistants but *not* for mathematics in general, where fractions are read as $\frac{a}{b}$ and not as a/b. However, Isabelle's semantic markup in the front-end with links to types, definitions etc is indispensable in software at the actual state of computer mathematics. Presently there is no formula editor available, which features both, 2-dimensional representation *and* semantic markup — a separate challenge for academic open source development: just add two integers to Isabelle's `Position`, re-use what Knuth has implemented in LaTeX for an editor (probably embedded into Isabelle/jEdit) — and create significant impact on mathematics software of various kinds (including Isabelle itself)!

---

[17]https://www.isa-afp.org/

6   Summary and Conclusions

This paper gave the first technically concise description of Lucas-Interpretation (*LI*) and showed how concepts from Automated Reasoning support flexible student interaction by reliable check of student input. The description focuses key points and gives many pointers into the code in a freely accessible repository[18]. This is to invite readers to re-use prototyped code and/or the concept of *LI*; the previous section gave some hints for re-use and for future development, more will be thinkable after further experiences.

The example from structural engineering on p.82 makes two practical consequences of *LI* clear. Firstly, *all* kinds of mathematical problems, which can be described by an algorithm, can be associated with automatically generated user-guidance using *LI* – this covers most of mathematics as taught at arbitrary engineering faculties. Secondly implementation of respective algorithms is simple, see Fig.1; and programming becomes more efficient with each `SubProblem` implemented earlier: re-using these makes development sustainable.

Final conclusions   address education, since *LI* has been invented for educational purposes and the *ISAC*-project tries to adopt and adapt technologies with pedagogical concepts in mind.

The migration of *LI* to the function package appears to illustrate the flexible conception of the Isabelle framework, and the integrative nature of this conception confirms the hope to realise "complete, transparent & interactive models of mathematics" [32] for education. Such mechanical models might provide experience with mathematics as the "discipline in mechanisation of thinking" (Bruno Buchberger [6]) as an indispensable complement to teaching mathematics with human intuition — an experience required to understand not only the strengths, but also the limitations of mathematical thinking technology [9].

References

[1] *Archive of Formal Proofs.* `http://afp.sourceforge.net`.

[2] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. Murray, G. Klein & G. Heiser (2016): *Cogent: Verifying High-Assurance File System Implementations.* In: *International Conference on Architectural Support for Programming Languages and Operating Systems*, Springer Berlin / Heidelberg, Atlanta, GA, USA, pp. 175–188, doi:10.1145/2872362.2872404.

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

[4] Ralph-Johan Back, Victor Bos & Johannes Eriksson (2007): *MathEdit: Tool Support for Structured Calculational Proofs.* Technical Report 854, TUCS. Available at `http://tucs.fi/publications/view/?pub_id=tBaBoEr07a`.

[5] F. Botana, M. Hohenwarter, P. Janičić, Z. Kovács, I. Petrović, T. Recio & S. Weitzhofer (2015): *Automated Theorem Proving in GeoGebra: Current Achievements.* Journal of Automated Reasoning 55(1), pp. 39–59, doi:10.1007/s10817-015-9326-4.

[6] B. Buchberger (2013): *The Role of Mathematical Thinking for 21st Century Society.* Invited talk at The 2nd International Conference on Mathematics and Technology in Mathematics Education.

[7] Jasmin C.Blanchette: *Hammering Away. A User's Guide to Sledgehammer for Isabelle/HOL.* contained in the Isabelle distribution. Available at `http://isabelle.in.tum.de/doc/sledgehammer.pdf`.

---

[18]`https://hg.risc.uni-linz.ac.at/wneuper/isa/`

[8]   David M. Cerna (2019): *Evaluation of the VL Logic (342.208-9) 2018W End of Semester Questionnaire*. RISC Report Series, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Schloss Hagenberg, 4232 Hagenberg, Austria.

[9]   Maurice Chiodo & Toby Clifton (2019): *The Importance of Ethics in Mathematics*. Newsletter of the European Mathematical Society 114, pp. 34–37, doi:10.4171/NEWS/114/9.

[10] Gabriella Daróczy & Walther Neuper (2013): *Error-Patterns within "Next-Step-Guidance" in TP-based Educational Systems*. In: eJMT, the Electronic Journal of Mathematics & Technology, 7, pp. 175–194. Available at `https://php.radford.edu/~ejmt/ContentIndex.php#v7n2`. Special Issue "TP-based Systems and Education".

[11] W. D. Farmer, J. D. Guttman & F. J. Thayer (1993): *IMPS: An Interactive Mathematical Proof System*. Journal of Automated Reasoning 11(2), pp. 213–248, doi:10.1007/BF00881906.

[12] Florian Haftmann: *Code generation from Isabelle/HOL theories*. Contained in the Isabelle distribution. Available at `http://isabelle.in.tum.de/doc/prog-prove.pdf`.

[13] Florian Haftmann, Cezary Kaliszyk & Walther Neuper (2010): *CTP-based programming languages ? Considerations about an experimental design*. ACM Communications in Computer Algebra 44(1/2), pp. 27–41, doi:10.1145/1838599.1838621.

[14] Florian Haftmann & Tobias Nipkow (2010): *Code Generation via Higher-Order Rewrite Systems*. In Matthias Blume, Naoki Kobayashi & Germán Vidal, editors: Functional and Logic Programming, Lecture Notes in Computer Science 6009, Springer Berlin / Heidelberg, pp. 103–117, doi:10.1007/978-3-642-12251-4_9.

[15] Predrag Janičić (2006): *GCLC — a tool for constructive euclidean geometry and more than that*. In: Mathematical Software – ICMS 2006, 4151, pp. 58–73, doi:10.1007/11812289_22.

[16] Stephen C. Johnson (1975): *Yacc: Yet Another Compiler-Compiler*. Technical Report 32, AT&T Bell Laboratories, Murray Hill, New Jersey. Retrieved 31 January 2020.

[17] Stefan Karnel (2002): *Größte gemeinsame Teiler in Polynomringen und Implementierung im ISAC-Projekt*. Master's thesis, University of Technology, Institute of Mathematics, Graz, Austria. Available at `https://static.miraheze.org/isacwiki/3/3e/GGTs-von-Polynomen.pdf`.

[18] Markus Kienleitner (2012): *Towards "NextStep Userguidance" in a Mechanized Math Assistant*. Master's thesis, IICM, Graz University of Technology. Available at `https://static.miraheze.org/isacwiki/0/0d/Mkienl_bakk.pdf`.

[19] Boris Koichu & Alon Pinto (2019): *The Seoncdary-Tertiary Transition in Mathematics. What are our current challenges and what can we do about them?* EMS Newsletter, pp. 34–35, doi:10.4171/NEWS.

[20] Alexander Krauss: *Defining Recursive Functions in Isabelle/HOL*. Munich. Available at `http://isabelle.in.tum.de/doc/functions.pdf`. Part of the Isabelle distribution.

[21] Alexander Krauss (2006): *Partial Recursive Functions in Higher-Order Logic*. In Ulrich Furbach & Natarajan Shankar, editors: Automated Reasoning, Third International Joint Conference, IJCAR 2006, Lecture Notes in Computer Science 4130, Springer, pp. 589–603, doi:10.1007/11814771_48.

[22] Alan Krempler & Walther Neuper (2018): *Prototyping "Systems that Explain Themselves" for Education*. In Pedro Quaresma & Walther Neuper, editors: Proceedings 6th International Workshop on Theorem proving components for Educational software, Gothenburg, Sweden, 6 Aug 2017, Electronic Proceedings in Theoretical Computer Science 267, Open Publishing Association, pp. 89–107, doi:10.4204/EPTCS.267.6.

[23] Mathias Lehnfeld (2011): *Verbindung von 'Computation' und 'Deduction' im ISAC-System*. Master's thesis, Institut für Computersprachen, Technische Universität Wien. Bakkalaureate project.

[24] Peter Lucas (1978): *On the Formalization of Programming Languages: Early History and Main Approaches*. In D. Bjørner & C. B. Jones, editors: The Vienna Development Method: The Meta-Language, LNCS 16, Springer, doi:10.1007/3-540-08766-4_8.

[25] Roman E. Maeder (2012): *Programming in Mathematica*, 3rd edition. Addison-Wesley, Reading, Mass.

[26] Walther Neuper (2001): *Reactive User-Guidance by an Autonomous Engine Doing High-School Math*. Ph.D. thesis, IICM - Inst. f. Softwaretechnology, Technical University, A-8010 Graz. Available at `https://static.miraheze.org/isacwiki/8/8a/Wn-diss.pdf`.

[27] Walther Neuper (2012): *Automated Generation of User Guidance by Combining Computation and Deduction*. In Pedro Quaresma & Ralph-Johan Back, editors: *Electronic Proceedings in Theoretical Computer Science*, 79, Open Publishing Association, pp. 82–101, doi:10.4204/EPTCS.79.5.

[28] Walther Neuper (2014): *GCD — A Case Study on Lucas-Interpretation*. In: *Joint Proceedings of the MathUI, OpenMath and ThEdu Workshops and Work in Progress track at CICM*, Coimbra, Portugal. Available at `http://ceur-ws.org/Vol-1186/paper-17.pdf`. Urn:nbn:de:0074-1186-1.

[29] Walther Neuper (2016): *Lucas-Interpretation from Users' Perspective*. In: *Joint Proceedings of the FM4M, MathUI, and ThEdu Workshops, Doctoral Program, and Work in Progress at the Conference on Intelligent Computer Mathematics*, Bialystok, Poland, pp. 83–89. Available at `http://cicm-conference.org/2016/ceur-ws/CICM2016-WIP.pdf`. Urn:nbn:de:0074-1785-8.

[30] Walther Neuper (2018): *Lucas-Interpretation from Programmers' Perspective*. Available at `https://static.miraheze.org/isacwiki/8/83/Lucin-prog-view.pdf`. Abstract for ThEdu'18.

[31] Walther Neuper (2018): *Mechanical Explanation in "Systems that explain themselves"*. In Osman Hasan et al., editors: *Workshop Papers at 11th Conference on Intelligent Computer Mathematics CICM 2018*, Conference on Intelligent Computer Mathematics CICM, Hagenberg, Austria. Available at `https://www.cicm-conference.org/2018/infproc/paper1.pdf`. Urn:nbn:de:0074-2307-7.

[32] Walther Neuper (2019): *Technologies for "Complete, Transparent & Interactive Models of Math" in Education*. In Pedro Quaresma & Walther Neuper, editors: Proceedings 7th International Workshop on *Theorem proving components for Educational software*, Oxford, United Kingdom, 18 july 2018, *Electronic Proceedings in Theoretical Computer Science* 290, Open Publishing Association, pp. 76–95, doi:10.4204/EPTCS.290.6.

[33] Walther Neuper & Christian Dürnsteiner (2007): *Angewandte Mathematik und Fachtheorie mithilfe adaptierter Basis-Software*. Technical Report 683, IMST – Innovationen Machen Schulen Top!, University of Klagenfurt, Institute of Instructional and School Development (IUS), 9010 Klagenfurt, Sterneckstrasse 15. Available at `https://www.imst.ac.at/imst-wiki/images/f/f9/683_Kurzfassung_Neuper.pdf`.

[34] Walther Neuper, Johannes Reitinger & Angelika Gründlinger (2008): *Begreifen und Mechanisieren beim Algebra Einstieg*. Technical Report 1063, IMST – Innovationen Machen Schulen Top!, University of Klagenfurt, Institute of Instructional and School Development (IUS), 9010 Klagenfurt, Sterneckstrasse 15. Available at `https://www.imst.ac.at/imst-wiki/images/9/9d/1063_Langfassung_Reitinger.pdf`.

[35] Walther Neuper et al. (2006): *Angewandte Mathematik und Fachtheorie*. Technical Report 357, IMST – Innovationen Machen Schulen Top!, University of Klagenfurt, Institute of Instructional and School Development (IUS), 9010 Klagenfurt, Sterneckstrasse 15. Available at `http://imst.uni-klu.ac.at/imst-wiki/index.php/Angewandte_Mathematik_und_Fachtheorie`.

[36] Tobias Nipkow: *Programming and Proving in Isabelle/HOL*. contained in the Isabelle distribution. Available at `http://isabelle.in.tum.de/doc/prog-prove.pdf`.

[37] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2008): *Isabelle/HOL, a proof assistant for higher-order logic*. Springer Verlag.

[38] Lawrence C. Paulson (1990): *Isabelle: The Next 700 Theorem Provers*. In P. Odifreddi, editor: *Logic and Computer Science*, Academic Press, pp. 361–386. Available at `https://arxiv.org/abs/cs/9301106`.

[39] Pedro Quaresma & Ralph-Johan Back, editors (2012): Proceedings First Workshop on *CTP Components for Educational Software (THedu'11)*. 79, Open Publishing Association, doi:10.4204/EPTCS.79.

[40] Y. Shi, Y. Ma, J. MacLeod et al. (2019): *College students' cognitive learning outcomes in flipped classroom instruction: a meta-analysis of the empirical literature*. Journal of Computers in Education, pp. 1–25, doi:10.1007/s40692-019-00142-8.

[41] Makarius Wenzel: *The Isabelle/Isar Implementation*. Contained in the Isabelle distribution. Available at `http://isabelle.in.tum.de/doc/implementation.pdf`.

[42] Makarius Wenzel: *Isabelle/jEdit*. Munich. Available at `http://isabelle.in.tum.de/doc/jedit.pdf`. Part of the Isabelle distribution.

[43] Makarius Wenzel (2014): *System description: Isabelle/jEdit in 2014*. In: *Proceedings Eleventh Workshop on User Interfaces for Theorem Provers, UITP 2014, Vienna, Austria, 17th July 2014.*, pp. 84–94, doi:10.4204/EPTCS.167.10.