Lucas-Interpretation on Isabelle’s Functions

Walther Neuper
Johannes Kepler University, Linz, Austria

Abstract. 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: [2] calls “structured calculations” 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 [8], and in geometry software AR tools are used [11,3], too. But all the above mentioned systems are poor in guided interactivity.

A logic course for freshmen in AI and Computer Science\(^1\) accompanies all content with software. And respective experience shows [6] again, that much of the

\(^1\) http://fmv.jku.at/logic/
software lacks interactivity and feedback sufficient for independent learning\(^2\) in particular in application to mathematics.

In principle [21], AR provides the most powerful tools for independent learning in formal mathematics, captured by naïve 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 naïve requirements might be reconsidered, since proof assistants like Isabelle [22] hide the intricacies of AR tools and present notation to the user close to standard mathematics.

\( \mathcal{L} \mathcal{I} \) aims at meeting these requirements for more than a decade, when Peter Lucas\(^3\) shifted his interests from programming languages [16] 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 [14] could migrate to Isabelle’s function package [12,13] with surprising little effort.

Hereewith the first technically concise description of \( \mathcal{L} \mathcal{I} \) is given, after theoretical considerations [17] and application oriented discussions [18,20,19]. Since \( \mathcal{L} \mathcal{I} \) was implemented in the \textsc{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 \( \mathcal{L} \mathcal{I} \), the concept in §2.1 illustrated with examples in §2.2. Adaptation of the programs for \( \mathcal{L} \mathcal{I} \) to Isabelle’s function package is described in §3, where §3.1 describes the tactics, §3.2 the tactics and §3.3 briefly describes evaluation by rewriting. The parse-tree generated by the function package is interpreted by \( \mathcal{L} \mathcal{I} \) as shown in §4; how the parse-tree is scanned presents §4.1, how \( \mathcal{L} \mathcal{I} \) uses Isabelle’s \texttt{Proof.context} is concern of §4.2 followed by a brief description, how the prototype’s mathematics-engine embeds and guards \( \mathcal{L} \mathcal{I} \). 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 \( \mathcal{L} \mathcal{I} \) widely applied to education in mathematics.

2 Lucas–Interpretation (\( \mathcal{L} \mathcal{I} \))

The interpreter is named after the inventor of top-down-parsing in the \textsc{ALGOL} project [16], Peter Lucas. As a dedicated expert in programming languages he initially objected “yet another programming language” (although no function

\(^2\) Currently independent learning is best addressed by “flipped classes” [24]

\(^3\) https://austria-forum.org/af/AustriaWiki/Peter_Lucas_(Informatiker)

\(^4\) https://isac.miraheze.org/wiki/History
evaluation was in sight in Isabelle's development that time), but then he helped to clarify the unusual requirements for a novel programming language in the IS4C-project, which later led to the notion of LI.

LI is the most prominent component in a prototype developed in the IS4C-project, there embedded in a mathematics-engine, which interacts with a dialogue-module in a Java-based front-end managing interaction with students (out of scope of this paper).

2.1 The Concept of LI

The concept of LI is simple: LI acts as a debugger on functional programs with hard-coded breakpoints, where control is handed over to a student; a student, however, does not interact with the software presenting the program, but presenting steps of forward reasoning, where the steps are rigorously constructed by tactics acting as the break-points mentioned. In connection with LI we call programs, albeit implemented in the function-package not “function” but “program” and also shall prefer “execute” over “evaluate”.

Types occurring in the signatures of LI are as follows. Besides the program of type Program.T there is an interpreter state Istate.T, which passes data from on 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” as a sequence of steps in forward reasoning, a variant of “structured derivations” [1]. 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 [15].

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 |

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 \rightarrow Calc.T \rightarrow Istate.T \rightarrow Proof.context \\
\rightarrow Tactic.T \rightarrow input_tactic_result

datatype input_term_result = 
  Found_Step of Calc.T 
  | Not_Derivable
val locate_input_term: Calc.T \rightarrow term \rightarrow 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 Program.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 the same signature as 
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 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 ap-
plication case for Isabelle’s Sledgehammer [5], but not yet realised and prelimi-
arily 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 following program, which is shown by a screen-shot in order to demonstrate the colouring helpful for programmers (compare the old bare string version in [14, p. 92]). The program is implemented 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. The program 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 [14, p. 97]:

```
01 Problem (Biegelinie, [Biegelinien])
02 Specification:
03 Solution:
04 Problem (Biegelinie, [vonBelastungZu, Biegelinien])
05 [ V x = c + 1 \cdot q_0 \cdot x, 
06  M x = c_2 + c \cdot x + \frac{1}{2} q_0, 
07  \frac{dy}{dx} y x = c_3 + \frac{3}{2} \cdot (c_2 \cdot x + \frac{x}{2} + \frac{1}{3} \cdot q_0), 
08  y x = c_4 + c_3 \cdot x + \frac{1}{2} \cdot (\frac{c_2}{2} \cdot x^2 + \frac{c_3}{6} \cdot x^3 + \frac{1}{12} q_0 \cdot x^4) ]
09 Problem (Biegelinie, [setzeRandbedingungen, Biegelinien])
10 [ L \cdot q_0 = c, 0 = \frac{1}{2} c_2 + 2 \cdot L \cdot c + \frac{1}{2} L^2 \cdot q_0, 0 = c_4, 0 = c_3 ]
11 solveSystem (L \cdot q_0 = c, 0 = \frac{1}{2} c_2 + 2 \cdot L \cdot c + 1 \cdot L^2 \cdot q_0, 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 y x = c_4 + c_3 \cdot x + \frac{3}{2} \cdot (\frac{c_2}{2} \cdot x^2 + \frac{c_3}{6} \cdot x^3 + \frac{1}{12} q_0 \cdot x^4) 
14 Substitute [c, c_2, c_3, c_4] 
15 y x = 0 + 0 \cdot x + \frac{1}{2} \cdot (\frac{-1 \cdot L^2 \cdot q_0}{2} \cdot x^2 + \frac{L \cdot q_0}{6} \cdot x^3 + \frac{1}{24} q_0 \cdot x^4) 
16 Rewrite_Set_Inst ([(bdv, x)], make_ratpoly_in) 
```

The respective code is at [https://hg.risc.uni-linz.ac.at/wneuper/isa/file/df1b56b0d2a2/src/Tools/isac/Knowledge/Biegelinie.thy#l174](https://hg.risc.uni-linz.ac.at/wneuper/isa/file/df1b56b0d2a2/src/Tools/isac/Knowledge/Biegelinie.thy#l174).
The calculation is what a student interacts with. Above the Specification is folded in, the specification-phase and respective user interaction is skipped here. 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 students gets stuck (3) a next step (as a term or a tactic) is suggested by \( \mathcal{LI} \), which works behind the scenes; the respective functions will be introduced below (for (1) locate_input_term, (2) locate_input_tactic and for (3) find_next_step).

This example hopefully clarifies the relation between program and calculation in \( \mathcal{LI} \): 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 for Computer Algebra shall shed light on two further notions introduced later, on tacticals and on program expressions. The program in Fig. 2 simplifies rational terms (typed as “real” for convenience):

\[
6 \quad y x = \frac{\rho}{\eta} \cdot x^2 + \frac{\lambda}{\eta} \cdot x^3 + \frac{\mu}{\eta} \cdot x^4
\]

\[
18 \quad y x = \frac{\rho}{\eta} \cdot x^2 + \frac{\lambda}{\eta} \cdot x^3 + \frac{\mu}{\eta} \cdot x^4
\]

The program executes the tactic Rewrite’_Set with different canonical term rewriting systems (called “rule sets” in ZSAC) 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.

---

**Fig. 2.** Simplification implemented by the function package
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 ZSAC’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:

```plaintext
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 respective `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 ∧, ∨, but traditional notation uses sets (and these are still lists for convenience). `SubProblem` not only take arguments (`arg list` like any (sub-) program, but also three references into ZSAC’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 the operate terms, which are well fooled by Isabelle and which can contain functions evaluated as program expressions introduced in the next but one subsection.

---

6 https://hg.risc.uni-linz.ac.at/wneuper/isa/file/df1b56b0d2a2/src/Tools/isac/ProgLang/Prog_Tac.thy#l136
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 7. These tactics also cover the specification-phase (which is out of scope of the paper). And there is another `Tactic.T` defined at 8 for internal use by the mathematics-engine, which already appeared in the signature of Lucas-Interpretation on p.3.

3.2 Tacticals Guiding Flow of Execution

The example on p.6 for canonical rewriting showed, how tacticals guide the flow of execution. The complete list of tacticals is as follows.

```plaintext
cconsts
  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 ⇒ 'a` (which can be combinations of tacticals or tactics) on a bool expression; (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.6 as argument of `Take`: sometimes it is necessary to pick parts of elements of a calculation, for instance the right-hand-side of an equality or a certain element of a list. So Isabelle’s `List` is adapted for IS4C’s purposes in `ListC` 9. Such expressions are substituted from the environment in `Istate.T` and evaluated by rewriting (and for that purpose using the same rewrite-engine as for the tactics `Rewrite`).

7 https://hg.risc.uni-linz.ac.at/wneuper/isa/file/df1b56b0d2a2/src/Tools/isac/MathEngBasic/tactic-def.sml#l122
8 https://hg.risc.uni-linz.ac.at/wneuper/isa/file/df1b56b0d2a2/src/Tools/isac/MathEngBasic/tactic-def.sml#l141
9 https://hg.risc.uni-linz.ac.at/wneuper/isa/file/df1b56b0d2a2/src/Tools/isac/ProgLang/ListC.thy
4 Implementation of $\mathcal{LI}$

The implementation of the interpreter is as experimental as is the respective program 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 $\mathcal{LI}$. The evaluation part of the function package, however, implements automated evaluation in one go and automated code-generation [9] — both goals are not compatible with $\mathcal{ZSAC}$’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 $\lambda$ terms, which also encode the objects of proofs. Thus there is a remarkable collection of tools, readily available in Isabelle; but this collection seems never have encountered the requirement, to scan a term to a certain location, to remember the location and to return there later, as required by $\mathcal{LI}$. So this has been introduced

\[
\text{datatype lrd} = \text{L} \mid \text{R} \mid \text{D} \\
\text{type path} = \text{lrd list} \\
\]

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

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

\[
\text{datatype expr_val} = \\
\text{Term_Val of term} \mid \text{Reject_Tac} \\
| \text{Accept_Tac of Istate.pstate * Proof.context * Tactic.T}
\]

which models the meaning of an expression in the parse-tree: this is either a $\text{Term_Val}$ (introduced in §3.3) or a tactic (introduced in §3.1); the latter is either accepted by $\text{Accept_Tac}$ or rejected by $\text{Reject_Tac}$; an error value is still missing. The arguments of the above constant $\text{Accept_Tac}$ are the same as introduced for $\mathcal{LI}$ in §2.1. Thus $\text{expr_val}$ is the return value for functions scanning the parse-tree for an acceptable tactic:
The first argument of each function above, if of type `term`, is the whole program. It is not required by `scan_dn`, simple top-down scanning, and by `check_tac`, if a tactic has been reached. The rightmost argument, if of type `term`, serves matching as shown later. `scan_to_tactic` recognises, whether interpretation is just starting a new program (`path = []`) or interpretation resumes at a certain location:

```
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 `go_scan_up`. Both functions, scanning down and up, are shown below:

```fun scan_dn cc (ist as {act_arg, ...}) (Const ("Tactical.Try", _) $ e) =
  case scan_dn cc (ist |> path_down [R]) e of
    Reject_Tac => Term_Val act_arg
    | goback => goback
  | scan_dn cc ist (Const ("Tactical.Repeat", _) $ e) =
    scan_dn cc (ist |> path_down [R]) e
  | scan_dn cc ist (Const ("Tactical.Chain"(*2*), _) $ e1 $ e2) =
    case scan_dn cc (ist |> path_down [L, R]) e1 of
      Term_Val v => scan_dn cc (ist |> path_down [R] |> set_act v) e2
      | goback => goback
  (*|*)
  | scan_dn (cc as (_, ctxt)) (ist as {eval, ...}) t =
    if Tactical.contained_in t
    then raise TERM ("scan_dn expects Prog_Tac or Prog_Expr", [t])
    else
      case Lucin.check_leaf "next " ctxt eval (get_subst ist) t of
        (Program.Expr s, _) => Term_Val s
        | (Program.Tac prog_tac, form_arg) =>
          check_tac cc ist (form_arg, prog_tac)
```
The last case of \texttt{scan_dn} must have reached a leaf. \texttt{check_leaf} distinguishes between \texttt{Expr} and \texttt{Tac}; the former returns a \texttt{Term_Val}, the latter returns the result of \texttt{check_tac} called by \texttt{scan_dn}, where the result is either \texttt{Accept_Tac} or \texttt{Reject_Tac}. In case of \texttt{Accept_Tac} scanning is stalled (in §2.2 such a tactic has been called a break-point), the respective tactic is returned (together with respective \texttt{Istate.T} and \texttt{Proof.context} to the mathematics-engine. In case of \texttt{Reject_Tac} an explicit back tracking by \texttt{scan_up} tries other branches with \texttt{scan_dn}.

\texttt{scan_up} is as simple as \texttt{scan_dn}; an essential difference is, that the former requires the whole program for \texttt{go_scan_up} as follows:

\begin{verbatim}
( fun go_scan_up (pcc as (sc, _)) (ist as {path, act_arg, found_accept,...})= 
  if path = [R] 
    then 
      if found_accept then Term_Val act_arg else Reject_Tac 
    else 
      scan_up pcc (ist |> path_up) (go_up path sc) 
  and scan_up pcc ist (Const ("Tactical.Try"(*2*), _) $ _) = go_scan_up pcc ist 
  | scan_up pcc as (_, cc)) ist (Const ("Tactical.Repeat"(*2*), _) $ e) = 
    (case scan_dn cc (ist |> path_down [R]) e of 
      Accept_Tac ict => Accept_Tac ict 
      | Reject_Tac => go_scan_up pcc ist 
      | Term_Val v => go_scan_up pcc (ist |> set_act v |> set_found)) 
  | scan_up pcc ist (Const ("Tactical.Chain"(*3*), _) $ _) = go_scan_up pcc ist 
  | scan_up pcc as (sc, cc)) ist (Const ("Tactical.Chain"(*3*), _) $ _) = 
    let 
      val e2 = check_Seq_up ist sc 
      in 
      case scan_dn cc (ist |> path_up_down [R]) e2 of 
        Accept_Tac ict => Accept_Tac ict 
        | Reject_Tac => go_scan_up pcc (ist |> path_up) 
        | Term_Val v => go_scan_up pcc (ist |> path_up |> set_act v |> set_found) 
    end 
(*|*)
)
\end{verbatim}

\section*{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.)." [25]. The \$\$4\$-prototype introduced Isabelle's \texttt{Context} in collaboration with a student [15],
now uses them throughout construction of problem solutions and implements a specific structure \texttt{ContextC} \textsuperscript{11}.

At the beginning of the specification-phase (briefly touched by tactic \texttt{SubProblem} in §2.2 and explained in little more detail in the subsequent section) an Isabelle theory must be specified, this is followed by \texttt{Proof_Context.init_global} in the specification module. Then \texttt{Proof_Context.initialise'} takes a formalisation of the problem as strings, which are parsed by \texttt{Syntax.read_term} using the context, and finally the resulting term’s types are recorded in the context by \texttt{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 \texttt{ContextC.insert_assumptions}.

During \texttt{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 \texttt{SubProblem} applied with theorem $x \neq 0 \Rightarrow (\frac{2}{x} = b) = (a = b \cdot x)$ during equation solving.

A still not completely solved issue is switching to and from \texttt{SubProblem}s; the scope of an interpreter’s environment is different from a logical context’s scope. When calling a sub-program, \texttt{ZS4C} uses \texttt{Proof_Context.initialise}, but returning execution from a \texttt{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 \texttt{ContextC.subpbl_to_caller}\textsuperscript{12} 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

\texttt{partial_functions} as used by \texttt{LI} are alien to Isabelle/HOL for principal reasons, however, when the \texttt{ZS4C}-project started with the aim to support learning mathematics as taught at engineering faculties, it was clear, that formal specifications should guard execution of programs under \texttt{LI}. Also the example program in Fig.1 on p.5 requires preconditions in order to run reliably.

So formal specification is required for technical reasons \textit{and} for educational reasons in engineering education. The \texttt{ZS4C}-project designed a separate specification phase, where input to a problem and respective output as well as preconditions and post-condition are handled explicitly by students; example in Fig.1 shows several \texttt{SubProblem}s, which lead to mutual recursion between specification phase and phases creating a solution (supported by \texttt{LI}).

\textsuperscript{11} A closing “C” indicates an \texttt{ZS4C} extension, see https://hg.risc.uni-linz.ac.at/wneuper/isa/file/00612574cbfd/src/Tools/isac/CalcElements/contextC.sml

\textsuperscript{12} https://hg.risc.uni-linz.ac.at/wneuper/isa/file/ce071aa3eae4/src/Tools/isac/CalcElements/contextC.sml#l73
Embedding \( \mathcal{L}I \) into a dialogue-module is required in order to meet user requirements in educational settings. Looking at the calculation shown on p.5 and considering the power of the three main function (1) . . . (3) mentioned in the respective comment on p.6, it is clear, that in particular \texttt{find_next_step} needs control: The dialogue-module hands over control to a student at each step of \( \mathcal{L}I \), but students should also be bewared to use a button creating the next step too frequently. A careful reduction of help from \( \mathcal{L}I \) might make the learning system also usable for written exams.

5 Lessons Learned . . .

As already mentioned, development of Isabelle and development of \( \mathcal{L}I \) went in parallel for a long time — a great opportunity for learning in the \( \mathcal{L}S\mathcal{A}C \)-project.

5.1 . . . from the Isabelle Project

“\textit{Isabelle was not designed; it evolved. Not everyone likes this idea}” said Lawrence C. Paulson in “Isabelle: The Next 700 Theorem Provers” [23].

When the \( \mathcal{L}S\mathcal{A}C \)-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 code is tightly related to locations, where new functionality is required. In the meanwhile Isabelle evolved to a didactic model in functional programming at a large scale: polymorphic high-order functions take complex function-arguments, which allow postpone type definitions to location according to functionality; and such functionality is distilled to small abstract structures. Together with canonical argument order, function combinators and canonical iteration [25, p. 15-17] this gives elegant code with almost no glue, while the hint “never copy & paste a piece of code” disappeared 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 \( \mathcal{L}S\mathcal{A}C \)-project started in a situation, where software and user requirements might have been as unclear as with early Isabelle, albeit on another level. \( \mathcal{L}S\mathcal{A}C \)’s present code structure still looks much more like (very!) early Isabelle than present Isabelle. This paper was the occasion to make \( \mathcal{L}I \)-related components in \( \mathcal{L}S\mathcal{A}C \) 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”: \( \mathcal{L}S\mathcal{A}C \)’s code structure,
even around \( \mathcal{L} \), is still far off Isabelle’s quality. This is the same with minimal change-sets: in the present state of \( \mathcal{ISAC} \)’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 \( \mathcal{ISAC} \) started, a glance at Isabelle’s front-end convinced everyone, that educational software \textit{not} could use it. Now, a decade later, it appears clear, that \( \mathcal{ISAC} \) is best advised to re-use Isabelle/JEdit based on Isabelle/PIDE, the integrated proof development environment at 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:

- \textbf{Syntax errors} are indicated accurately at the right location; finding errors in programs represented as strings was a nightmare, if programs comprised a couple lines of code.
- \textbf{Type annotations} disappear from the program and sidestep to the heading signature; the result is much better readable.
- \textbf{Syntax highlighting} indicates how identifiers are interpreted, as constants, as free variable, as strings, etc — very instructive for working programmers.
- \textbf{Free variables} on the right-hand-side of assignments are rejected by the function package, while these were accepted by term parsing.
- \textbf{Semantic annotations} support the programmer, in particular the tooltip popups triggered by hovering and clicking with the mouse, see [26, p. 30].

These features came for free, when \( \mathcal{ISAC} \)’s programs were shifted into Isabelle/Isar’s function definition — and were immediately fruitful: Not only 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 [14, p. 92]) and triggered improved handling of program arguments in \( \mathcal{L} \).

5.3 . . . for Further \( \mathcal{ISAC} \)-Development

What \( \mathcal{ISAC} \) 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, \( \mathcal{ISAC} \) would benefit as well.

Future development in \( \mathcal{ISAC} \) will follow a list of steps as announced in [14, p. 102-103]; the work on \( \mathcal{L} \) 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 respective project became apparent. Indeed, the many challenges already identified in [10], suggest intermediate steps on the way for educational software for engineering disciplines as aimed at in the \( \mathcal{ISAC} \)-project.
A case study on GCD [18] 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 (including recursion and mutual recursion) can be executed step-wise — as an experimental approach to study algorithms in the Archive of Formal Proofs. 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 \( \text{ISAC} \)’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 readable 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 \text{Position}, re-use what Knuth has implemented in \text{LATEX} and create significant impact!

\section{Summary and Conclusions}

This paper gave the first technically concise description of Lucas-Interpretation (\( LI \)) and showed how concepts from Automated Reasoning support flexible interaction by reliable check of user input. The description focuses key points and gives many pointers into the code in a freely accessible repository \(^{13}\). 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.

\textit{Final conclusions} address education, since \( LI \) has been invented for educational purposes and the \( \text{ISAC} \)-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” [21] for education. Such mechanical models might provide experience with mathematics as the “discipline in mechanisation of thinking” (Bruno Buchberger [4]) 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 [7].

\(^{13}\) https://hg.risc.uni-linz.ac.at/wneuper/isa/
References

4. Buchberger, B.: The Role of Mathematical Thinking for 21st Century Society (March 4-6 2013), invited talk at The 2nd International Conference on Mathematics and Technology in Mathematics Education


