Automated Generation of User Guidance by Combining Computation and Deduction

Walther Neuper
Institute for Software Technology
University of Technology
Graz, Austria
neuper@ist.tugraz.at

Herewith, a fairly old concept is published for the first time and named "Lucas Interpretation". This has been implemented in a prototype, which has been proved useful in educational practice and has gained academic relevance with an emerging generation of educational mathematics assistants (EMA) based on Computer Theorem Proving (CTP).

Automated Theorem Proving (ATP), i.e. deduction, is the most reliable technology used to check user input. However ATP is inherently weak in automatically generating solutions for arbitrary problems in applied mathematics. This weakness is crucial for EMAs: when ATP checks user input as incorrect and the learner gets stuck then the system should be able to suggest possible next steps.

The key idea of Lucas Interpretation is to compute the steps of a calculation following a program written in a novel CTP-based programming language, i.e. computation provides the next steps. User guidance is generated by combining deduction and computation: the latter is performed by a specific language interpreter, which works like a debugger and hands over control to the learner at breakpoints, i.e. tactics generating the steps of calculation. The interpreter also builds up logical contexts providing ATP with the data required for checking user input, thus combining computation and deduction.

The paper describes the concepts underlying Lucas Interpretation so that open questions can adequately be addressed, and prerequisites for further work are provided.

1 Introduction

Motivated by planning for a large project on building educational math assistants (EMAs) at the state-of-the-art of computer mathematics[1], Peter Lucas[2] supervised the essential design decisions for a "system that explains itself", which covers a major part of mathematics as taught to a major target group: solving problems in engineering and in applied sciences in academic courses and at high-schools.

Two requirements were identified that are essential but conflicting: (1) design interaction and notation as close as possible to what is written into textbooks during interactive tutoring and on blackboards during lectures and (2) implement software mechanisms as general and (logically) reliable as possible. The design tackled the conflict in several ways. They key idea is introduced as "Lucas-Interpretation" in this paper. Lucas-Interpretation operates on a novel kind of programming language based on Computer Theorem Proving (CTP). A program written in this language determines the next steps while the interpreter builds up logical context providing automated theorem provers (ATP) with facts required to prove user input derivable or not.

[1] MacSchubert http://www.iist.unu.edu/www/docs/annualreports/1993/main_7.html under founding director Dines Bjørner

[2] Peter Lucas was one of the pioneers in compiler construction and in formal methods [16, 17, 18]. http://www.austria-lexikon.at/af/Wissenssammlungen/Biographien/Lucas_Peter

P. Quaresma and R.-J. Back (Eds.); THedu'11
EPTCS 79, 2012, pp. 82–101, doi:10.4204/EPTCS.79.5

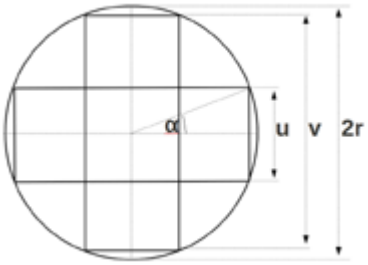


Figure 1: A coil with a cross-shaped kernel

The design has been implemented in a prototype called ISAC[3], first the mathematics engine based on Isabelle [26], then a multi-user front-end based on Java-Swing. ISAC has been used successfully in high-schools [22, 23, 25]. Discussion of underlying concepts has started with focusing education at CADGME [24] and continued with focusing technology at THedu'11. The latter workshop confirmed the relevance and (still!) novelty of ISAC's conception. So, after ten years of implementation in ISAC, the underlying concepts herewith are published concisely first time. All concepts described in the paper appear appropriate in ISAC's implementation; *not* (yet) implemented features are explicitly designated as such.

The paper is structured as follows: §2 gives a detailed example which shall motivate the subsequent definitions. §3 presents how ISAC uses deduction for checking user input, §4 describes how computation is used to construct solutions of problems and §5 introduces Lucas-Interpretation combining deduction and computation for automatic generation of user guidance. Open questions and related work are given ample space in §6, and finally §7 gives a summary of Lucas-Interpretation and a conclusion for pedagogy.

2 Running Example

In order to illustrate the design principles, *one* example will be used throughout the paper. This example is from a problem-class with strong traditions in German speaking countries called 'Extremwert Aufgaben'. The example's complexity is at an intermediate level between high school and university, thus indicating the concepts' usability from early introduction of variables up to mathematics applied in studies of science and of engineering.

Example problem: *Given a circle with radius r, where two rectangles with length u and width v each are inscribed as shown in Fig.1. This figure shows the section through a coil; the induction current is proportional to the area of the cross-shaped kernel of the coil. Determine u and v such that the kernel's area A is a maximum.*

Traditionally the problem-class 'Extremwert Aufgaben' is taught in calculus courses, thus specification takes geometric considerations 'as obvious' and immediately comes to algebraic formulas as used in the problem's formal specification below. The running example has at least two different specifications:

Specification no.1:
input : $\{ r \}$
precond : $0 < r$

[3]ISAC stands for *ISA*belle for *C*alculations in Applied Mathematics, http://www.ist.tugraz.at/projects/isac

output : $\{ max\,A,\ u, v \}$
postcond : $A = 2uv - u^2 \ \wedge\ (\frac{u}{2})^2 + (\frac{v}{2})^2 = r^2 \ \wedge$
$\forall\, A'\, u'\, v'.\ (A' = 2u'v' - (u')^2 \wedge (\frac{u'}{2})^2 + (\frac{v'}{2})^2 = r^2) \Rightarrow A' \leq A$
props : $\{ A = 2uv - u^2,\ (\frac{u}{2})^2 + (\frac{v}{2})^2 = r^2 \}$

Field tests showed, that learners understand what to input in the fields 'input', 'output' and 'props'; the proper postcondition in field 'postcond', however, is out of scope for most students of the target group. The first two fields, 'input' and 'output', are standard, the last one is additional: the field 'props' contains elements of 'postcond', and although the "logical part" $(\forall, \exists.\ \Rightarrow, \wedge)$ is dropped, 'props' distinguishes problems nicely; in fact, the second specification of the running example differs characteristically in 'props', not only in 'postcond':

Specification no.2:
input : $\{ r \}$
precond : $0 < r$
output : $\{ max\,A,\ u, v \}$
postcond : $A = 2uv - u^2 \ \wedge\ \exists\alpha.\ (\frac{u}{2} = r\sin\alpha \ \wedge\ \frac{v}{2} = r\cos\alpha) \wedge$
$\forall A'\, u'\, v'.\, (A' = 2u'v' - (u')^2 \wedge \exists\alpha'.\ (\frac{u'}{2} = r\sin\alpha' \wedge \frac{v'}{2} = r\cos\alpha')) \Rightarrow A' \leq A$
props : $\{ A = 2uv - u^2,\ \frac{u}{2} = r\sin\alpha,\ \frac{v}{2} = r\cos\alpha \}$

In these two examples the fields 'postcond' have the same structure; without having this implemented in ISAC yet, we expect that postcondition can be generated automatically from the equations in 'props' and a pattern of the postcondition's structure for the respective problem class. The following definition is (almost) standard.

Definition 1 (Specification) *A quintuple $(I, p(I), O, q(I,O), r(I,O,A))$ is called a specification where I is a set of variables (the **input variables**), $p(I)$ is a formula in which only the variables from I occur freely (the **precondition**), O is a set of variables (the **output variables**) such that $I \cap O = \{\}$, $q(I,O)$ is a formula in which only the variables from $I \cup O$ occur freely (the **postcondition**) and $r(I,O,A)$ is a set of elements of $q(I,O)$ without quantifiers such that*

$$\forall I, O.\ p(I) \Rightarrow (q(I,O) \Rightarrow \exists A.r(I,O,A))$$

*is called **properties**. A vector of terms $S(I)$ in which only I occurs freely (the **solutions**) solves the specified problem, if the following holds:*

$$\forall I.\ p(I) \Rightarrow \forall O.\ O = S(I) \Rightarrow q(I,O)$$

The specification's elements $I, p(I), O, q(I,O), r(I,O,A)$ are labeled by 'input', 'precond', 'output', 'props' respectively in the two above example specifications. A in 'props' contains the variables which were existentially bound in $q(I,O)$.

Given a specification as an *implicit* definition of output values, a **calculation** is a transformation to an *explicit* definition by stepwise construction of output values. For instance, give specification no.2 from above, a calculation might be given by the steps in lines #01..#20 below. The calculation as shown below is the result of various experiments trying to accomplish the two conflicting requirements mentioned in the introduction, (1) traditional notation and (2) reliable mechanized treatment.

The details of the calculation below created by ISAC will be explained later together with the mechanisms which create the steps and which check the steps if input by the learner. Reactions of students and teachers to this format suggest to just explain what lists like *[maximum_by, calculus]* or *[make, diffable, function]* mean: In ISAC the question for the lists is answered interactively by following a link exactly to the respective specification. Here on paper a link to all specifications in ISAC is given[4].

[4] http://www.ist.tugraz.at/projects/isac/www/kbase/pbl/index_pbl.html

01 • `Problem` *[maximum_by, calculus]* - - -
02 $\vdash$ $A = 2 \cdot u \cdot v - u^2$ - - -
03 • `Problem` *[make, diffable, function]* - - -
04 ... $\widetilde{A}(\alpha) = 8 \cdot r^2 \cdot \sin\alpha \cdot \cos\alpha - 4 \cdot r^2 \cdot (\sin\alpha)^2$ - - -
05 • `Problem` *[on_interval, max, argument]* - - -
06 $\vdash$ $\widetilde{A}(\alpha) = 8 \cdot r^2 \cdot \sin\alpha \cdot \cos\alpha - 4 \cdot r^2 \cdot (\sin\alpha)^2$
`Subproblem` *[differentiate, function]*
07 • `Problem` *[differentiate, function]*
`Apply_Method` *Differentiate*
`Take` $\frac{d}{d\alpha}(8 \cdot r^2 \cdot \sin\alpha \cdot \cos\alpha - 4 \cdot r^2 \cdot (\sin\alpha)^2)$
08 $\vdash$ $\frac{d}{d\alpha}(8 \cdot r^2 \cdot \sin\alpha \cdot \cos\alpha - 4 \cdot r^2 \cdot (\sin\alpha)^2)$
`Rewrite` $\frac{d}{dx}(a \cdot u - b \cdot v) = a \cdot \frac{d}{dx}u - b \cdot \frac{d}{dx}v$
09 $\equiv$ $8 \cdot r^2 \cdot \frac{d}{d\alpha}(\sin\alpha \cdot \cos\alpha) - 4 \cdot r^2 \cdot \frac{d}{d\alpha}(\sin\alpha)^2$
`Rewrite` $\frac{d}{dx}(u \cdot v) = u \cdot \frac{d}{dx}v + \frac{d}{dx}u \cdot v$
10 $\equiv$ $8 \cdot r^2 \cdot (\sin\alpha \cdot \frac{d}{d\alpha}\cos\alpha + (\frac{d}{d\alpha}\sin\alpha) \cdot \cos\alpha) - 4 \cdot r^2 \cdot \frac{d}{d\alpha}(\sin\alpha)^2$
`Rewrite` $\frac{d}{dx}(u^n) = n \cdot u^{n-1}\frac{d}{dx}u$
11 $\equiv$ $8 \cdot r^2 \cdot (\sin\alpha \cdot \frac{d}{d\alpha}\cos\alpha + (\frac{d}{d\alpha}\sin\alpha) \cdot \cos\alpha) - 4 \cdot r^2 \cdot 2 \cdot (\sin\alpha)^{2-1}\frac{d}{d\alpha}\sin\alpha$
12 $\equiv$ $\vdots$
13 $\equiv$ $8 \cdot r^2 \cdot (-(\sin\alpha)^2 + (\cos\alpha)^2 - 2 \cdot \sin\alpha \cdot \cos\alpha)$
`Check_Postcond` *[differentiate, function]*
14 ... $\widetilde{A}'(\alpha) = 8 \cdot r^2 \cdot (-(\sin\alpha)^2) + (\cos\alpha)^2 - 2 \cdot \sin\alpha \cdot \cos\alpha)$ - - -
15 • `Problem` *[on_interval, goniometric, equation]* - - -
16 ... $\widehat{\alpha} = \tan^{-1}(-1 + \sqrt{2})$
`Check_Postcond` *[on_interval, max, argument]*
17 ... $\widehat{\alpha} = \tan^{-1}(-1 + \sqrt{2})$ - - -
18 • `Problem` *[find_values, tool]* - - -
19 ... *[* $u = 0.23 \cdot r$, $v = 0.76 \cdot r$ *]* - - -
20 ... *[* $u = 0.23 \cdot r$, $v = 0.76 \cdot r$ *]*

The numbers on the left margin above do not belong to the calculation, they are for reference in this paper. The strengths of the above format come to bear on interactive worksheets, for instance, the • paired with ... (on the same level of indentation) indicate places where intermediate steps are collapsed (as in lines #03..#04) or can be unfolded (as in lines #07..#14) — collapsed or unfolded on request by the learner, whether requiring a survey (collapsing and getting the big picture) or whether requiring details (unfolding and pushing parts out of the screen which are not relevant at the moment).

Calculations show two kinds of elements, formulas and tactics. **Formulas** are shifted to the left margin, which are indented according to a tree structure. **Tactics** are shifted to the right margin. In the above calculation the tactics are `Subproblem`, `Apply_Method`, `Take`, `Rewrite` and `Check_Postcond`; the others are not shown (- - -), according to the tradition to omit on blackboards whatever is not relevant at the moment.

So, the above calculation for the running example provides a first impression, which will motivate rigorous treatment and formal definitions in the sequel.

3 Deduction for Checking User Input

The present prototype checks formulas input by proving equivalence modulo a specified algebraic theory, that means by rewriting to normal forms and checking respective equality. For instance, acceptable inputs at line #12 in the above calculation would be those at #12 *(a...z)* below:

$$
\begin{array}{lcl}
11 & & 8\cdot r^2\cdot(\sin\alpha\cdot\frac{d}{d\alpha}\cos\alpha+(\frac{d}{d\alpha}\sin\alpha)\cdot\cos\alpha)-4\cdot r^2\cdot 2\cdot(\sin\alpha)^{2-1}\cdot\frac{d}{d\alpha}\sin\alpha \\
12\ (a) & \equiv & 8\cdot r^2\cdot(\sin\alpha\cdot(-\sin\alpha)+\cos\alpha\cdot\cos\alpha)-4\cdot r^2\cdot 2\cdot(\sin\alpha)^{2-1}\cdot\frac{d}{d\alpha}\sin\alpha \\
or\ (b) & \equiv & 8\cdot r^2\cdot(\sin\alpha\cdot\frac{d}{d\alpha}\cos\alpha+(\frac{d}{d\alpha}\sin\alpha)\cdot\cos\alpha)-4\cdot r^2\cdot 2\cdot(\sin\alpha)^{1}\cdot\cos\alpha \\
or\ (..) & \equiv & \ldots \\
or\ (z) & \equiv & 8\cdot r^2\cdot(-(\sin\alpha)^2)+(\cos\alpha)^2-2\cdot\sin\alpha\cdot\cos\alpha)
\end{array}
$$

Each of the inputs *(a...z)* has *(z)* as a normal form[5], so each of the inputs is equivalent modulo the simplifier. Although relying on Isabelle [26] as much as possible, ISAC initially implemented a rewrite engine of it's own in order to meet the requirement of transparent single-stepping discussed in §3.1. Since there are plans to provide Isabelle's simplifier with appropriate features of transparency[6], ISAC has implemented Isabelle's proof contexts [30] recently in order to re-use Isabelle's simplifiers.

In Isabelle a proof context is created from an Isabelle theory, i.e. initially it contains all the facts from that theory, and then it is extended dynamicall with facts encountered during interactive proof construction. Isabelle's proof language Isar [21] is block-structured and proof contexts follow this structure. So proof contexts contain all *locally generated and modified* facts for calling Isabelle's ATP tools, one of which is the simplifier.

ISAC will re-use the mechanism of Isabelle's proof contexts in order to not only use Isabelle's simplifier, but also Isabelle's other ATP tools. So we claim for a judgment like

$$F \vdash_x f \tag{1}$$

where F is the sequence of formulas already present in a calculation, f is a newly input formula and x is a context providing all facts required to derive f via an ATP tool ($\vdash$). F and f are the formulas visible in the calculation, x are the invisible facts additionally required for mechanical proof. §5.2 shows, how x is generated step by step for the calculation of the running example on p.85.

3.1 Steps in Derivations

Experience from educational practice indicates a specific requirement on judgments from ATP: *judgments shall be presented with comprehensible intermediate steps*. This requirement is hard for ATP, often relying on tools like tableau provers, SMT solvers, on sequent calculi etc., which employ abstract representations hardly understood by novices. However, in case ATP involves simplification (which is frequently the case) this requirement can be successfully accomplished by grouping the rules and showing the rewrites of the groups (and postpone further details to further inquiry). The issue has been acknowledged by CTP development[7], and it seems worth the effort for educational reasons:

When novices are in the phase of acquiring confidence in an EMA, then their curiosity should be decisively supported and questions answered; otherwise EMAs reinforce the wide spread impression of mathematics as magic which cannot be understood. So EMAs should be "transparent" to learners' inquiries any time and respond with comprehensible intermediate steps.

The requirement for steps coincides with the practical requirement to have some means to stepwise construct a calculation; these means are tactics as shown (partially) in the example calculation on p.85.

[5] Algebraic equivalence modulo *thy* of these variants of f is not decidable in general, since there is no normal form for terms containing trigonometric functions [4]; however, in the above case and in most other cases the class of terms is restricted such that ATP reliably can reject or accept an input.

[6] http://isabelle.in.tum.de/gsoc-ideas.html#simptrace

[7] Every few weeks the Isabelle mailing list receives a request by a novice for intermediate steps. A "visual tracing facility for the rewriting engine" is already addressed at http://isabelle.in.tum.de/gsoc-ideas.html. A related feature exists already in "proof reconstruction": data returned from ATP are inserted into Isar proofs in a human readable manner.

Since formulas in a calculation do not contain all information required for mechanical treatment, and this information is contained in contexts, tactics need to operate on contexts. So we have two kinds of tactics, (1) those introduced in the example calculation on p.85 and (2) those explicitly operating on contexts; there is a one-to-one correspondence between (1) and (2), so we need not distinguish between them, except in rare cases: then (1) will be called **external tactics** and (2) will be called **internal tactics**.

Contexts contain the preconditions $p(I)$ of a specification, the type-constraints given by the input variables I and output variables O, assumptions generated and used by conditional rewriting etc. All these facts decide whether a tactic is **applicable** or not.

Applicability need to be defined for each tactic separately. With respect to the running example on p.85, tactic `Rewrite` applied at #08 requires the fact, that $\widetilde{A}(\alpha)$ is differentiable, and also the left-hand side of the theorem $\frac{d}{dx}(a \cdot u - b \cdot v) = a \cdot \frac{d}{dx}u - b \cdot \frac{d}{dx}v$ must match a redex in the respective formula. Tactic `Subproblem` *[tool, find_values]* is applicable, if the input variables of the respective specification have been generated in the calculation already, and if the precondition holds. There are also more complicated tactics, for instance handling case-distinctions, which do not show up in the running example. The following definition seems straight forward.

Definition 2 (Step of calculation) *Given a specification $s = (I, p(I), O, q(I,O), r(I,O,A))$, a sequence F of formulas, a formula f', contexts x and x' and a tactic t, then a step is described by the rule*

$$\frac{t \text{ is\_applicable\_in } (x,F) \qquad x \subseteq x' \qquad F \vdash_x x' \qquad F \vdash_{x'} f'}{x, F \rightarrow^t x', F \oplus f'}$$

where $F \oplus f'$ means insertion of f' into F at a position according to t. The rule is applied to the initial configuration x_0, F_0 with tactic t_o, were $x_0 = p(I) \cup x_{types(I,O)} \cup x_{thy}$, i.e. x_0 consists of the precondition, the type constraints $x_{types(I,O)}$ and the context x_{thy} initialized by theory thy, $F_0 = \emptyset$ and t_0 is either `Take` *or* `Subproblem`.

Tactics are designed functional such that for all t, x, F there exists exactly one x', F' such that $x, F \rightarrow^t x', F'$. So we can say $\mathbf{t}$ ***applied_to*** $(\mathbf{x}, \mathbf{F}) = (\mathbf{x}', \mathbf{F}')$

This definition does not tell whether a step is triggered by input of formula f, by input of a tactic t or something else. Although steps are functional, calculations generated by these steps are non-deterministic, because there may be many tactics applicable at one and the same configuration; this variability of calculations is the challenge for learners when deciding on the next step of calculation. §5 will tell details about input of formulas and/or tactics after some further prerequisites have been introduced.

First follows a definition of calculation. The above approach taken via steps suggests in operational view according to [28]:

Definition 3 (Calculation) *Given a specification $s = (I, p(I), O, q(I,O), r(I,O,A))$, then a calculation is a labeled terminal transition system $c = \langle X, \mathscr{P}(F), T \rightarrow, S \rangle$, where the set of configurations is $X \times \mathscr{P}(F)$, X a set of* ***contexts*** *and $\mathscr{P}(F)$ a set of sequences F of* ***formulas****, the actions $t \in T$ are called* ***tactics****, the transitions in $\rightarrow^t \subseteq (X \times \mathscr{P}(F)) \times T \times (X \times \mathscr{P}(F))$ are the steps of calculation introduced in Def.2.*

The terminal configurations in set $S \subset X \times \mathscr{P}(F)$ are called ***solutions*** *of c.*

Further details on *solutions* are given in §3.2.

Summarizing, *a calculation consists of steps each of which is derived from the steps previously constructed*; derivation of the steps relies on logical facts (in contexts x) not shown to the learner without request. The internal machinery handling the contexts x will be introduced in the subsequent sections.

3.2 Checking Solutions of Problems

This subsection presents the only part of the introduced concepts, which is not yet implemented in ISAC. This part raises open questions, but is essential for "a system which explains itself": such a system should not only construct a calculation stepwise as described in the previous section, such a system also should show that the result of the calculation is a solution of the specified problem. This means (in analogy to judgment (1) on p.86) we want to have a judgment

$$F \vdash_x q(I,O) \tag{2}$$

where F are the formulas in the (completed) calculation, x is the final context and $q(I,O)$ is the postcondition on the input I and the output O in the respective specification.

A look at the running example sheds light on the challenges for designing the details for an implementation of (2). In the example $p(I,O)$ is

$$\begin{aligned}&\text{postcond} : A = 2uv - u^2 \;\wedge\; \exists\alpha.\ (\tfrac{u}{2} = r\sin\alpha \;\wedge\; \tfrac{v}{2} = r\cos\alpha) \;\wedge\\ &\qquad \forall A'\,u'\,v'.\ (A' = 2u'v' - (u')^2 \wedge \exists\alpha'.\ (\tfrac{u'}{2} = r\sin\alpha' \wedge \tfrac{v'}{2} = r\cos\alpha')) \Rightarrow A' \leq A\end{aligned}$$

The postcondition's $\exists$ is accomplished by constructing α in the calculation's lines #15–#16. A complete proof would have to re-curse to the postcondition of (Sub-)`Problem` *[on_interval, max, argument]* at line #05 which might look like

$$\forall\alpha.\ \alpha \in]0, \frac{\pi}{2}[\Rightarrow \widetilde{A}(\alpha) \leq \widetilde{A}(\tan^{-1}(-1+\sqrt{2}))$$

A closer look suggests to have not only this postcondition in the context x, but also the function $\widetilde{A}(\alpha)$, the facts $A = \widetilde{A}(\tan^{-1}(-1+\sqrt{2}))$, $u = 0.23$, $v = 0.76$ etc. — so (automatically!) proving the final postcondition needs to rely on a well-stocked context x; building x is the main concern of §4.

Since we have introduced a calculation as "consisting of steps each of which is derived from the previous steps" the steps need to be considered. The final step in the example calculation is in line #19, obtaining the values *[$u = 0.23 \cdot r$, $v = 0.76 \cdot r$]* as results of (Sub-)`Problem` *[tool, find_values]*. These values are approximations of exact terms which are too large to be carried through a calculation by hand; due to the design decision for notation "as close to what is written to traditional blackboards as possible" the running example also uses approximate values — however, approximate values cannot prove the postcondition given above; the postcondition would require re-formulation by using the notion of "approximation errors", which would make the postcondition and the proof even more complicated.

An implementation for checking solutions shall be analogous to Def.2 on stepwise construction of calculations, i.e. there should be a final step which completes the calculation such that the postcondition holds:

Definition 4 (Result of calculation) *Given a specification $s = (I, p(I), O, q(I,O), r(I,O,A))$ and a calculation $c = \langle X, \mathscr{P}(F), T, \rightarrow, S\rangle$ as a labeled terminal transition system with transitions according to Def.2 for steps of calculation with $x_0, \emptyset \rightarrow^{t_0} \ldots \rightarrow^{t_n} x_n, F_n \rightarrow^{\texttt{Check\_Postcond}} x_n, F_n$ such that*

$$F_n \vdash_{x_n} q(I,O)$$

and for every variable $o \in O$ there is an equation $o = r$ in F_n such that all the equations can be ordered in a sequence

$$o_1 = r_1, o_2 = r_2, \ldots, o_n = r_n$$

*where every o_i does not occur in $r_1, \ldots, r_i$. Then $r_1, \ldots, r_n$ is called a **result** of c and x_n, F_n is called a **solution** constructed by c. We also say that calculation '$\mathbf{c}$ **is_completed_with** $(\mathbf{x_n}, \mathbf{F_n})$'.*

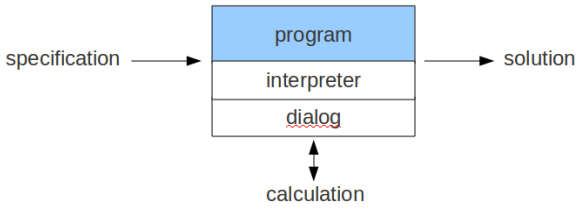


Figure 2: A calculation is a side-effect of a functional program

4 Computation for Constructing Solutions

Since computers exist, they are used to compute some output from some input, where the resulting output fulfills certain expectations — i.e. they are used just for the purpose formalized in the above Def.4 for solutions of computations. Symbolic computation allows to handle formal objects of arbitrary level of abstraction, even real numbers like $\tan^{-1}(-1+\sqrt{2})$ in *finite* computers ($\sqrt{2}$ etc. not as a floating point number, of course).

Computation envisaged in this section is used to create steps of calculations. ISAC's design provides a functional programming language for doing that computation; given the characteristic of functional programs to produce a result (without intermediate states), the question arises, how intermediate steps can be created such that interaction on these steps is possible (since this clearly requires states related to these steps). Fig.2 on p.89 gives a preliminary answer to this question: the calculation and respective steps are created as a side-effect of interpreting the program; the states and interaction on the steps of calculation are handled by the interpreter of the program. The role of a dialog in such an architecture seems natural, but details are out of scope in this paper.

4.1 CAS-like Functionality in Programming Languages

For constructing calculations we need functionality well-known from Computer Algebra Systems (CAS): for instance, differentiation and equation solving in the running example on p.85. Indeed, CAS-based programming languages [1, 20] are most successful in software production for engineering and science. However, CAS-based languages do not commend themselves for educational math assistants — users of CAS are responsible for when to apply CAS functions and for how to use their results; so learners might be over-challenged by such responsibility: equation solvers might have lost some solutions, simplification might imply partiality conditions without mentioning them (e.g. $\frac{x^2-1}{x-1} = x+1$ without mentioning $x \neq 1$) etc. For education we want to have systems "which never make mistakes" [12].

Deduction is already in the game since §3, so we advocate rigor corresponding to CTP: we claim any CAS-function to be accompanied by an explicit formal specification. Coming back to the running example, differentiation as shown for the example calculation in lines #07..#13 on p.85 is specified by

Specification of `Problem` *[differentiate, function]*:
input : $\{ f, v \}$
precond : $\lambda v.\ f$ *is_differentiable*
output : $\{ f', range \}$
postcond : $\forall v.\ v \in range \Rightarrow (\lambda v.\ f')(v) = \lim_{h \to 0} \frac{(\lambda v.\ f)(v+h) - (\lambda v.\ f)(v)}{h}$

and this implicit specification is transformed to an explicit specification, i.e. a calculation, by the tactic `Subproblem` *[differentiate, function]*: in line #07 this tactic starts the Sub-`Problem` *[differentiate, func-*

tion] and solves the problem by tactic `Apply_Method` *Differentiate*. The respective program is shown on p.92 below.

The above requirements advocate a novel kind of programming called "CTP-based". Considerable work is already done preparing such languages [12, 13, 14]; this is discussed under related work in §6.3.

4.2 A CTP-based Prototype Language

ISAC implements a programming language, which is appropriate for describing mathematics found in problems of engineering and of science. The following program generates the calculation on p.85 for the running example; at a first glance the reader sees a purely functional program with the usual keywords `LET..IN`, `IF..THEN..ELSE`; at this point the only difference are the function calls: identified by `Subproblem`, these calls are associated with a theory (e.g. *Reals*), a reference to a specification (e.g. *[make, diffable, function]* and a reference to the function itself (e.g. *make_fun*).

In the same way the `Subproblem`s are accompanied by theory, specification and program, the example program below is accompanied by a theory and a specification already introduced in two variants (for the same program!) no.1 or no.2 on p.84. That means, the arguments of `Program` *Max* are checked by the respective precondition and the return-value *finds* is related to the input by the respective postcondition on termination of the program:

```
01 Program Max (givens::real list) (max::real) (finds::real list) (rels::bool list)
02                 (var::real) (interval::real set) (errbound::real) =
03     LET
04         maxequ = Take ((HD o (FILTER (contains max ))) rels) ;
05         funterm = (IF 1 < LEN rels
06             THEN (Subproblem ( Reals, [make, diffable, function], make_fun)
07                         [ max::real, var::real, rels::bool list, interval::real set ] )
08             ELSE (HD rels )) ;
09         max = Subproblem ( Real_Algebra, [on_interval, max, argument],
10                                                 maximum_on_interval )
11                         [ funterm::real, var::real, interval::real set ] ;
12         find_rels = FILTER_OUT (ident maxequ) rels ;
13         finds = Subproblem ( Reals, [tool, find_values], find_values)
14             [ max::real, (RHS funterm)::real, var::real, max::real,
15               find_rels::bool list, errbound::real ]
16     IN  finds
```

In ISAC this program is parsed as an Isabelle term with `LET..IN`, `IF..THEN..ELSE` from Isabelle/HOL extended by definitions for `Program`, for "tactics" like `Take`, `Subproblem` and for some functions `LEN`gth (of a list), `RHS` (right-hand side of an equality), `FILTER_OUT`; some of these functions are already contained in the recent Isabelle versions. The reader may note that the notion of "tactic" is being extended: Tactics have been introduced in Def.2 as parts of steps in calculations, in the above program certain *statements* are called "tactic" — those statements which generate the steps of the respective calculation. §5.1 will unify the notions of tactics formally.

The function's arguments are worth one more remark: For instance, the `Subproblem` at lines #09..#11 above takes two arguments, a triple and a list; the triple addresses three kinds of knowledge as mentioned above:

1. deductive knowledge: *Real_Algebra* is an **Isabelle theory** comprising all the axioms, definitions and theorems required for the formulas of the generated calculation and for proving the postcondition of the calculation's specification. This is where the differentiation rules come from in the second example program on p.92.

2. application-oriented knowledge: *[on_interval, max, argument]* is a path into a hierarchy of **specification**s: the element retrieved from the hierarchy specifies the calculation to be generated.

   The specification is expected to be proved by ATP during interpretation: before the call the precondition is proved for the function's arguments, on return the postcondition is proved for arguments and return value.

3. algorithmic knowledge: *maximum_on_interval* is a reference to the **program** describing the algorithm which creates the calculation.

The second argument of the `Subproblem` contains some formal arguments *funterm, var, interval* as known from function definitions:

[funterm::real, var::real, interval::real set]

The list of arguments might contain more items than the respective specification — this becomes clear when considering the example program above which has these formal arguments:

```
01 Program Max (givens::real list) (max::real) (finds::real list) (rels::bool list)
02                 (var::real) (interval::real set) (errbound::real) = ...
```

The number of arguments is seven, although both respective specifications no.1 and no.2 on p.84 only have one input, { *r* } for the first argument *givens* — the other arguments provide the input for all the `Subproblem` of the program!

This program is the first of two examples for a "CTP-based programming language" (where the second example program is on p.92 below). A thorough introduction of such a language is out of scope in this paper, [8] gives further details and §6.3 discusses open questions.

The reader may note, that the above programs are purely functional, thus do *neither* they contain input statements *nor* output statements, and thus cannot be concerned with interaction[8]. Interaction, however, is the key point of this paper and will be tackled subsequently.

5 Deduction *and* Computation for User Guidance

Now all prerequisites are prepared for introducing "Lucas-Interpretation": specifications of problems, calculations stepwise creating an explicit specification by applying tactics, calculations augmented by logical contexts, contexts containing facts required to check user input and to check the postcondition by ATP, a CTP-based programming language involving theories, specifications and programs, CAS-functionality included in this language.

Now the idea of Lucas-Interpretation is simple:

- The interpreter works like a debugger jumping from break-point to break-point in a program.
- The breakpoints are the tactics in the program; each tactics generates exactly one step in the calculation (a step which might comprise an arbitrary number of sub-steps, which are not shown to the user — rather, the system is ready to show them on request of the user).
- Each step also generates specific logical facts; the interpreter builds up contexts with these facts according to certain rules.

[8] The programmer of a CTP-based program is much concerned with theories, specifications and algorithms; so excluding concerns of interaction, of user guidance, etc. is a helpful separation of concerns.

- All steps are presented to the user on a "worksheet" (if not a dialog component decides for some other kind of interaction, for instance in a written exam).
- After a step has been presented to the user, control is handed over to the user (to the dialog component, respectively).
- Since the user is in control of the system, she or he has a variety of choices collected under four categories:
  - Inspect the calculation generated so far: ask for theorems applied at certain steps, ask for intermediate steps for instance in `Rewrite_Set` *simplifier*, etc.
  - Investigate the underlying knowledge in theory, specification or program: inspection starts at the particular knowledge item used in the current step.
  - Input an own step independently, a formula or a tactic (or something else according to the dialog component, for instance, a partial formula presented for filling in just some (crucial) part).
  - If got stuck in the calculation ask the system to do the next step. This feature enables a dialog component to provide **user guidance**; dialog details are out of scope of this paper; see [15].
- User input is checked by ATP, which is called by the interpreter using facts collected in the context. Feedback from ATP goes to the user (probably filtered by the dialog component).
- After input the interpreter resumes execution, probably at another location in the program due to free user input.

The subsequent sections describe first how the interpreter jumps from break-point to break-point, then how logical context are built up during interpretation and finally how user guidance is created automatically.

5.1 Steps of Interpretation Create Steps of Calculation

Steps of calculation (Def.2) comprise tactics as parts which give a denotative name to the step; within programs those statements are called (program) tactics which create a step of the respective calculation. Programs comprise further statements, which do *not* create steps; these are called **non-generating statements**.

Since `Program` *Max* contains only one non-generating statement in line #12, we give another example: a program calculating the derivative of a function and making the specification on p.89 explicit:

```
01 Program Differentiate (interval::real set) (f::real) (v::real) =
02     LET f' = Take ( d/d v f)
03     IN (REPEAT
04           ( Rewrite_Inst [(bdv, v)] diff_sum ) OR
05           ( Rewrite_Inst [(bdv, v)] diff_product ) OR
06           ( Rewrite_Inst [(bdv, v)] diff_sin ) OR
07           ( Rewrite_Inst [(bdv, v)] diff_cos ) OR
08           ( Rewrite_Inst [(bdv, v)] ..... ) OR
09           ( Rewrite_Inst [(bdv, v)] diff_fraction )) @@
10        ( TRY (Rewrite_Set simplifier )) f'
```

Again, the syntax of this kind of program is discussed in detail in [8], here are the hints for reading the program text: After constructing the formula from the program's arguments in line #02 the body of `LET..IN` forward chaines (`@@`) two functions, `REPEAT`(...) and `TRY` *(*`Rewrite_Set` *simplifier)*. The

former models the non-deterministic behavior of term rewriting systems: the rules *diff_sin* `OR` *diff_cos* `OR` ...[9] or other rules can apply or not, and as soon none applies `REPEAT` terminates. The latter function `TRY`s to simplify the resulting formula.

The lines #08–#13 in the example calculation on p.85 could have been generated by this program. Let us assume this calculation has been continued as follows:

$$\begin{aligned} 11 \;\equiv\; & 8\cdot r^2\cdot(\sin\alpha\cdot\tfrac{d}{d\alpha}\cos\alpha+(\tfrac{d}{d\alpha}\sin\alpha)\cdot\cos\alpha)-4\cdot r^2\cdot 2\cdot\sin\alpha\cdot\tfrac{d}{d\alpha}\sin\alpha \\ & \qquad\qquad \texttt{Rewrite}\ \tfrac{d}{d\alpha}(\sin\alpha)=\cos\alpha \quad \#06 \\ \equiv\; & 8\cdot r^2\cdot(\sin\alpha\cdot\tfrac{d}{d\alpha}\cos\alpha+\cos\alpha\cdot\cos\alpha)-4\cdot r^2\cdot 2\cdot\sin\alpha\cdot\tfrac{d}{d\alpha}\sin\alpha \\ & \qquad\qquad \texttt{Rewrite}\ \tfrac{d}{d\alpha}(\cos\alpha)=-\sin\alpha \quad \#07 \\ \equiv\; & 8\cdot r^2\cdot(\sin\alpha\cdot(-\sin\alpha)+\cos\alpha\cdot\cos\alpha)-4\cdot r^2\cdot 2\cdot\sin\alpha\cdot\tfrac{d}{d\alpha}\sin\alpha \\ & \qquad\qquad \texttt{Rewrite}\ \tfrac{d}{d\alpha}(\sin\alpha)=\cos\alpha \quad ??? \end{aligned}$$

First line #06 of the program has been interpreted (indicated by the line number at the right margin above), then the subsequent line #07. Finally line #06 with the rule *diff_sin* should be interpreted again in order to derive $\frac{d}{d\alpha}\sin\alpha$ — how does the interpreter come back to the respective location in the program?

Starting from location #07, this is accomplished by the sequence of statements
#07 $\rightarrow^{\texttt{OR}}$ ... $\rightarrow^{\texttt{Rewrite}\ diff\_fraction}$ #09 $\rightarrow^{\texttt{REPEAT}}$ #03 $\rightarrow^{\texttt{Rewrite}\ diff\_sum}$ #04
#04 $\rightarrow^{\texttt{OR}}$ #05 $\rightarrow^{\texttt{Rewrite}\ diff\_product}$ #05
#05 $\rightarrow^{\texttt{OR}}$ #06 $\rightarrow^{\texttt{Rewrite}\ diff\_sin}$

The statements `OR`, `REPEAT` (and also `LET..IN`, `@@`, `Try`) are concerned with controlling the sequence of interpretation and are called **tacticals**; they do *not* change the calculation or the respective context. ISAC's interpreter controls such sequences by locations in the program and by environments as usual.

The above example leads to the following definition in a straight forward manner, just adding a program-state to Def.2 for steps of calculation:

Definition 5 (Step of interpretation) *Given a specification $s=(I,p(I),O,\_,\_)$, a program m and a respective program-state γ, given non-generating statements s_{ng} and a (program)tactic t_m both in m, given an (internal)tactic t_c according to Def.2, a context x and a sequence of formulas F, then a* ***step*** *is an application of the rule*

$$\frac{\gamma\rightarrow^{s_{ng}}\ldots\rightarrow^{s'_{ng}}\gamma'\rightarrow^{t_m}\gamma'' \qquad t_m\approx t \qquad t\ applied\_to\ (x,F)=(x',F')}{\gamma,x,F\rightarrow^{t}\gamma'',x',F'}$$

where $t_m\approx t_c$ denotes equivalency induced by a bijective mapping between program tactics and tactics in calculations – due to this equivalency the distinction is omitted in the rule's conclusion and the tactic named t.

The initial configuration for applying the rule is γ_0,x_0,F_0 with $\gamma_0=(\emptyset,\{(i_1,v_1),\ldots,(i_n,v_n)\})$ where $\emptyset$ is the location at the root of program m and (i_i,v_i) are the identifier-value pairs of the arguments of m, and x_0,F_0 are as defined in Def.2. We say 'the steps are ***initialized_by*** *$\mathbf{s},\mathbf{m}$'.*

Interpretation stops at the first applicable tactic found after passing non-generating statements s_{ng}, which are either tacticals or expressions containing no tactic. A subsequent step is triggered from outside without caring about program tactics, so we introduce a judgement **do_next** generalizing over all tactics t

$$\frac{\gamma,x,F\rightarrow^{t}\gamma',x',F'}{\gamma,x,F\rightarrow^{do\_next}\gamma',x',F'} \qquad do\_next$$

A detailed description of the tacticals' semantics is out of scope of this paper. We just note, that the example program on p.92 would raise an error in line #10 if the tactical `TRY` would be omitted and the tactic `Rewrite_Set` *simplifier* would be *not* applicable.

[9] In order to come traditional notation as close as possible, the type of the function term is *real* and not *real* $\Rightarrow$ *real*; so we need to instantiate *bdv* by the actual value of *v* in the rules before rewriting.

5.2 Building up Logical Context during Interpretation

Contexts are required to check user input (§3) by ATP during interpretation, to check preconditions and post-conditions of `Subproblem`s and finally check the solution of a problem (§3.2) by ATP as well. How contexts are used to accomplish these tasks is introduced in §5.3.

How logical contexts x_i are built up during execution can be observed in the running example as follows (the numbers on the left refer to those in program on p.90):

$$
\begin{array}{lll}
01..02 & & x_0 = \{\ 0 < r\ \} \cup x_{types(r,A,u,v)} \cup x_{thy} \\
03..04 & & x_1 = x_0 \cup \{\ A = 2uv - u^2\ \} \\
05..08 & & x_2 = x_1 \cup \{\ \widetilde{A}(\alpha) = 8 \cdot r^2 \cdot \sin\alpha \cdot \cos\alpha - 4 \cdot r^2 \cdot (\sin\alpha)^2), \\
 & & \quad \widetilde{A}(\alpha)\ is\_differentiable\_on\ ]0, \tfrac{\pi}{2}[\ \} \\
09..12 & a & x_3 = x_2 \cup \{\ \widetilde{A}'(\alpha) = 8 \cdot r^2 \cdot (-(\sin\alpha)^2) + (\cos\alpha)^2 - 2 \cdot \sin\alpha \cdot \cos\alpha), \\
 & b & \quad \alpha = \tan^{-1}(-1+\sqrt{2}),\ \widetilde{A}'(\alpha) = 0, \\
 & c & \quad \forall \alpha'.\ \alpha' \in]0, \tfrac{\pi}{2}[\wedge \widetilde{A}'(\alpha') = 0 \Rightarrow\ \alpha' = (\tan^{-1}(-1+\sqrt{2})) \\
 & d & \quad \forall \alpha'.\ \alpha' \in]0, \tfrac{\pi}{2}[\Rightarrow \widetilde{A}(\alpha') \leq \widetilde{A}(\alpha)\ \} \\
13..15 & & x_4 = x_3 \cup \{u = 2 \cdot r \cdot \sin\alpha,\ u \approx 0.23 \cdot r,\ v = 2 \cdot r \cdot \cos\alpha,\ u \approx 0.76 \cdot r\ \}
\end{array}
$$

The initial values in x_0 comprise the precondition and $0 < r$, the type constraints for the input variables I and the output variables O of the specification as well as all required knowledge collected in theory *thy*.

The facts added in lines #09..#12 to context x_2 deserves explanation: the first line (a) is the result of Sub-`Problem`*[differentiate, function]* (see calculation on p.85, specification on p.89 and program on p.92). The second and third line (b,c) come from `Problem` *[on_interval, goniometric, equation]*: the latter is the postcondition stating that the equation has exactly one solution within the interval given. A single solution is required for this type of problem at this place, and it is the responsibility of the math author to prepare a specification with an appropriate interval. Line (d) is the postcondition of `Subproblem` *[on_interval, goniometric, equation]* on p.85.

The reader may note that contexts do not follow the scoping rules for program variables (which enclose variables within subprograms): `Problem` *(Reals, [on_interval, max, argument]* exports the values and the post-conditions of the respective sub-problems (lines (b,c)), `Problem`*[differentiate, function]* and `Problem` *[on_interval, goniometric, equation]*. The scoping rules for contexts are a consequence of the common practice to have unique variable names within a calculation.

Although there is (still) no implementation of proofs for postconditions in ISAC, there is the conviction that the final context x_4 contains sufficient facts for such proofs.

5.3 Lucas-Interpretation Combines Computation and Deduction

A Lucas-Interpreter stepwise executes a CTP-based program (§4.2) and creates calculations (Def.3) step by step (Def.2). In order to integrate these concepts, the following definition is close to Def.3, Def.2 and Def.4; actually the only additions are a program and a program state:

Definition 6 (Lucas-Interpreter) *Given a specification s, a calculation $c = \langle X, \mathscr{P}(F), T, \rightarrow, S\rangle$ and a program m, then a labeled terminal transition system $\mathscr{L} = \langle \Xi, T_m, \rightarrow_m, S_m\rangle$ is called a Lucas-Interpreter iff*

(i) the configuration $\Xi = (\Gamma \times X \times \mathscr{P}(F))$ contains Γ the program states (Def.5) and $X \times \mathscr{P}(F)$ the configurations in c (Def.3)

(ii) the actions T_m contain (program) tactics in m and T_m is bijectively mapped with T

(iii) the transition relations $\rightarrow^{t_m}$ are steps of interpretation according to Def.5 with $\gamma,x,F \rightarrow^{t_m} \gamma',x',F'$ and are itinitialized_by s,m.

(vi) the terminal configuarions $S_m \subset (\Gamma_m \times X \times \mathscr{P}(F))$ contain Γ_m the terminating states of m.

We say **c *is_generated_by*** $(\mathbf{s},\mathbf{m},\mathscr{L})$.

The bijective mapping between T and T_m continues the bijective mapping between *external* tactics and *internal* tactics from p.87; again further distinction seems not necessary and we write t instead t_m in the sequel.

The steps of interpretation in the above definition are triggered by the user requesting *do_next* according to Def.5. Instead of *do_next* the user also could have input a step of his or her own choice; after such input *do_next* raises the usual issue for debuggers: in which cases can execution (in our case: interpretation) resume, and in which cases it cannot resume due to too invasive operations at the breakpoint. A step can be given by two classes of input, which might be modified and combined by a dialog component, see [15]. The first class is characterized by input of an (external) tactic, the second by input of a formula:

Input of a tactic is done by the learner at steps where the theorem to be applied is the point (e.g. a rewrite rule applied to a huge term too laborious for input), where some sub-term needs to be picked out for the next step, where a sub problem has to be started, or other cases, where the learner prefers to input a tactic instead of a formula (or the dialog component decides according to some rule in the dialog mode). The following definition describes how Lucas-Interpretation handles the input (external) tactic t_I:

Definition 7 (Locatable tactics) *Given a specification s, a program m, a Lucas-Interpreter $\mathscr{L} = \langle \Xi, T_m, \rightarrow_m, S_m \rangle$ at configuration $\chi = (\gamma,x,F) \in \Xi$, a calculation c with c is_generated_by $(s,m,\mathscr{L})$ at configuration x,F and finally given an (external) tactic t_I input by the learner, then we have the rule*

$$\frac{t_I \textit{ applied\_to } (x,F) = (x',F') \qquad \gamma,x,F \rightarrow^* \gamma',x,F \rightarrow^{t_m} \gamma'',x',F' \qquad t_I \approx t_m}{\gamma,x,F \rightarrow^{t_m} \gamma'',x',F'}$$

*If the rule is applicable (in particular if for t_I an equivalent t_m in the program is found, $t_I \approx t_m$), we say '***t *is_locatable_at*** χ*' iff applicable; otherwise we say $\mathscr{L}$* ***is_helpless_at*** χ.

The definition's rule has t_I *applied_to* (x,F) as premise: if this premise is given, we get a step of *calculation* leading to x',F', but if no $t_I \approx t_m$ is found during interpretation, we don't get a γ'', a program state to resume interpretation form — $\mathscr{L}$ is helpless.

Note that the transitions $\rightarrow^* \ldots \rightarrow^t$ searching for $t_I \approx t_m$ only change the program state γ and not the configuration x,F of the calculation; so x',F' resulting from t_I is presented to the user, before control is handed over. This behavior is consistent with transitions being functions and not relations in both definitions, in calculations (Def.3) as well as in Lucas-Interpretation (Def.6). And the strategy implemented by Def.7 works particularly well with programs like the example on p.92.

If a t_m with $t_I \approx t_m$ can*not* be found, the step of calculation can still be done (according to t_I *applied_to* (x,F)) — but no appropriate tactic could be found in the program, thus the Lucas-Interpreter cannot determine a next step and "is helpless". Since this case easily can happen (for instance, if a "creative" rewrite rule is applied which cannot be found in the program), there is a stronger feature of Lucas-Interpretation for input of formulas.

Input of a formula concerns the second class of input for a learner constructing a calculation. Since user interfaces are expected to support copy-and-paste, input need not be laborious even for huge formulas. The following definition describes how Lucas-Interpretation handles an input formula f_I:

Definition 8 (Derivable formula) *Given a specification s, a program m, a Lucas-Interpreter* $\mathscr{L} = \langle \Xi, T_m, \rightarrow_m, S_m \rangle$*, a calculation c with c is_generated_by* $(s, m, \mathscr{L})$ *at configuration* x, F *and finally given a formula* f_I *input by the learner, then we have the rule*

$$\frac{\gamma, x, F \rightarrow^* \gamma', x', F' \qquad F \vdash_{x'} f_I}{\gamma, x, F \rightarrow^{t^*} \gamma', x', F_I}$$

If the rule is applicable we say '$\mathbf{f_I}$ **is_derived_from** χ*'; otherwise we say '*$\mathbf{f_I}$ **is_not_derivable_from** χ*'.*

In this case the interpreter executes the next tactics found in program m until a context x' is generated, which can justify the input f_I, i.e. $c \vdash_{x'} f_I$. This justification might involve algebraic simplification as mentioned in §3. If successful, the step is presented to the user, before control is handed over; t^* is a tactic called "ad-hoc derivation"; it usually comprises a sequence of tactics.

If not successful, i.e. the interpreter executes program m until termination and no step is found with $c \vdash_{x'} f_I$, then f_I is "not derivable". This judgment can be costly in resources, since it relies on an (internal) computation of the program, including all the subprograms, until termination.

6 Related Work and Open Questions

6.1 Proofs, Structured Derivations and Calculations

As mentioned above, educational systems shall be "systems which do not make mistakes", so well-designed logical foundations are indispensable.

Proof languages like Isar [21] are natural to compare with the approach taken in this paper, since the underlying prototype is based on the CTP Isabelle [26]. Unlike the proof languages of Coq [10], Ltac [10] and SSReflect [6], Isar is self-contained and does not refer to a proof state explicitly.

The requirement to be "as close to what is written to traditional blackboards as possible" inhibited the above calculations to follow the principle of being self-contained. Rather, calculations rely on *internal* contexts employed in all definitions. This is the price for high usability for the target group. The price is considered not too high, since contexts consist of formulas in predicate logic and thus can easily be presented to the user, as soon as the user requests for them (and then they can be filtered by a dialog component).

Isar specifically supports calculational proof [31] and integrates it well with other elements of the proof language. Nevertheless, the formalisms were considered too far from "what is written on traditional blackboards" for our target group.

Structured derivations (SD) [2] continue work on calculational proof [5, 7], provide foundations on natural deduction, are self-contained such that they need not refer to a state separate from the SD and also are "close to what is written on blackboards".

So, SD meet essential requirements for mechanized calculation and call for justification of any other choice of format. Major parts of the example calculation on p.85 exactly model SD, for instance lines

[10] http://coq.inria.fr/stdlib/Coq.Program.Tactics.html

#08..#13. The sub-`Problems` strengthen the rigor by requiring explicit formal specification (which might be hidden from the learner and done automatically behind the scenes). And the rigor of SD is weakened by allowing to omit the tactics (at the right margin on p.85). The weakened rigor allows to address a wider range of users, who need no prior introduction to the system — "next step guidance" provides perfect ad-hoc introduction. Another point is, that SD concerning sub-terms of formulas involve "window inferencing" [19], which is already more complicated.

For these reasons a format more general than SD has been implemented in ISAC.

6.2 "Correctness" in Lucas-Interpretation

This subsection doesn't address related work, only open questions on "correctness". Correctness of calculations is indispensable for an educational "system which never makes mistakes" [12]. Final correctness of a calculation is given by Def.4: the solution meets the postcondition. The steps (tactic/formula) of calculation constructing such a solution are either input by the user or determined by a program under Lucas-Interpretation. An input tactic is checked if it *is_applicable_in* the calculation (Def.2) – this implies "correctness". An input formula is proven derivable (Def.8) or not, thus it is "correct" or not. Both notions of "correctness", however, cannot exclude steps which are "misleading" to somewhere and not to a solution. But if the user relies on the system and requests *do_next* (Def.5), the system shall guarantee to come to a solution finally:

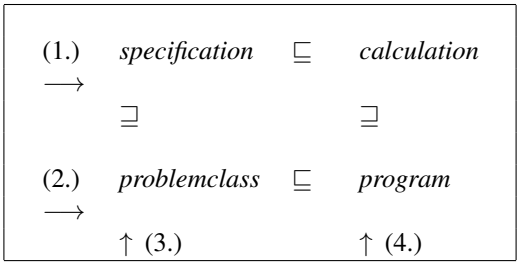
Definition 9 (Correctness under Lucas-Interpretation) *Given a specification s, a program m, a Lucas-Interpreter* $\mathscr{L} = \langle \Xi, T_m, \rightarrow_m, S_m \rangle$*, a calculation c with c is_generated_by* $(s, m, \mathscr{L})$ *and a sequence of steps of interpretation triggered by do_next* $\chi_0 \rightarrow^{do\_next} \ldots \rightarrow^{do\_next} \chi_n = (\gamma_n, x_n, F_n)$*. Then c is correct under* $s, m, \mathscr{L}$ *iff*

$$\chi_n \in S_m \Rightarrow c \; is\_completed\_with \; (x_n, F_n)$$

This definition says: if Lucas-Interpretation terminates, the generated calculation provides a solution (x_n, F_n). So we only have a definition of correctness, but we want to have a proof! The definition describes the general proof obligation for the programmer of *m* and *s*. In order to actually verify *m* with respect to *s* a rigorous formal semantics for each tactic *t* will be required: such a semantics just would have to add specific properties of *t* to Def.2 (which then would carry over to Def.5). This theoretical work seems most promising if combined with RTD of a development environment which integrates specific verification tools: a CTP-based IDE for a CTP-based programming language.

ISAC already shows the advantageous consequences of the above definition: the learner, if got stuck after some trials, always can backtrack to a step where reaching a solution is guaranteed — this feature allows interactive learning known from chess programs: when reaching a disadvantageous chess configuration just backtrack to another configuration and start a new trial (a novel kind of interaction for EMAs). Experience with the prototype shows, that learners get used to find out the position where to back-track very quickly.

The open questions above on correctness have already raised vague ideas, that the components involved in Lucas-Interpretation are related by refinements [3]. The components are specification, calculation, problem-class and program (where problem-class has not yet been discussed):



A look at the above refinement relations ($\sqsubseteq$) gives the following big picture:

1. *specification* $\sqsubseteq$ *calculation* seems clear to a mathematician; however, the underlying definitions (Def.2, Def.3 and Def.4) might be not strong enough for describing this refinement.
2. *problemclass* $\sqsubseteq$ *program* involves a notion not introduced here because not yet clarified: "problemclass" would be some "specification-pattern" which allows to mechanically generate a specification of the program from the input 'props': see example specifications no.1 and no.2 and imagine all the other examples belonging to the problemclass "Extremwert Aufgaben" (which is claimed to be solved by one and the same program, the example program) — how would a specification (-pattern), as general as the program, would look like?
3. *problemclass* $\sqsubseteq$ *specification* is expected to help clarifying the question from the previous Pt.2.
4. *program* $\sqsubseteq$ *calculation* is the crucial point addressed by Def.9.

6.3 On CTP-based Programming Languages

Lucas-Interpretation takes a certain kind of "CTP-based programming language" as a prerequisite, which only exists in ISAC under consideration. However, [8] argue for a value of their own for such languages and identify interest for CTP-based languages: For CAS-based programming languages great demand appears in successful application to software for engineering and science, most of which use CAS-based languages like [1, 20] as an efficient base for development — while lacking reliability of CAS is widely recognized, see for instance [9]; so one may expect developments meeting the demand indicated by the success of CAS-based languages with more reliable CTP-based languages.

Significant preparatory work [13, 14] for CAS-like functionality in a CTP framework has already been done. In the course of eventual development of CTP-based languages several further issues left open by CAS might be tackled: Reliable handling of partiality conditions has been mentioned in §4.1. Multivaluedness is another issue [11]. Very appealing for educational purposes would be to calculate exact approximation given by floating-point numbers (see [27] for Coq) throughout all algebraic operations involved in a calculation; already [9] has laid the grounds for tackling this issue.

Using CTP-based programs not only for implementation of applications like the running example (p.90), but also for the CAS-functions themselves will lead to "systems that explain themselves" at an extreme level: For instance, cancellation of multivariate polynomials like

$$\frac{x^2-y^2}{x^2-x\cdot y}=\frac{(x+y)\cdot(x-y)}{x\cdot(x-y)}=\frac{x+y}{x}\quad \textit{assuming } x\neq 0\wedge x-y\neq 0$$

is a frequent task already in early years of math education; usually it is explained "from right to left". But a smart student might ask, how a machine accomplishes the task from left to right (by factorization, by Hensel lifting or the like). If factorization or Hensel lifting is programmed in a CTP-based

language and interpreted by a Lucas-Interpreter, then such a student gets stepwise user guidance through the comprehensive algorithms on request:

With a CTP-based language and Lucas-Interpretation "implementation of tutoring becomes a side-effect of programming", see Fig.2 on p.89.

With that in mind interesting questions arise when re-implementing CAS-functions: CAS-algorithms at the state-of-the-art are high-brow in general (not only in the above case, also in integration, solving equational systems etc.). In order to foster students' curiosity and do not bother them upon questions of "how is that done", the possibility to select between high-brow algorithms and elementary algorithms (the latter probably not able to solve a certain problem) is already implemented in ISAC.

7 Summary and Conclusions

This paper presented an innovation in base technology for educational math assistants which is mainly motivated by pedagogical requirements. So we finish with a summary of technology and a conclusion concerning pedagogy.

A Summary of Lucas-Interpretation states a novel base technology for automatic generation of user guidance in stepwise solving problems in engineering and in applied science: Given a program written in a novel kind of CTP-based programming language which solves a problem, Lucas-Interpretation generates steps (Def.2, Def.5) of a calculation (Def.3) as a side-effect (Fig.2 on p.89) of the (functional) program. During stepwise interpretation a logical context is built up in order to provide ATP tools with the facts required for proving derivability of user input. At each tactic in the program control is handed over to the user (Def.6). The user is free to investigate the underlying knowledge, to freely input a step independently (checked by ATP immediately) or to ask the system to propose a next step (Def.7, Def.8)

According to the novelty of the design essential questions have been left open and are addressed in a separate section: In particular, the main theorems of Lucas-Interpretation are not proved due to the lack of a formalized operational semantics: The solution of a calculation is correct if the user does not interfere by other input than requesting the next step (Def.9) and interpretation can guarantee to resume after user input provided certain constraints on the input.

Although still a prototype, the Lucas-Interpreter has already established a stable interface to a dialog component, ready for experts in learning theory to start with detailed design of interaction in stepwise problem solving — this task is rigorously separated from extending the math topics of the problems: each problem just requires to program the algorithm solving the problem, interaction is handled by Lucas-Interpretation.

Conclusions for pedagogy expect impact on math education: CTP-technology allows to relate formal facts and activities in stepwise problem solving *mechanically within* the system. This is a considerable advantage over CAS or DGS (Dynamic Geometry Systems), where specific steps (integration, equation solving, etc.) are accomplished by the systems, well, but the whole process of stepwise problem solving, the relation between input (precondition), output (postcondition) and intermediate steps are left to the programmer media designer, who is overwhelmed by hard-coding feedback for the variants of steps.

On the other hand, CTP-based software models all mechanical parts of problem solving and thus allows to mechanically generate feedback to the learner: ATP ensures both, the most reliable *and* the most general checks of user input. This extends the scope of EMAs far beyond CAS or DGS. *So upcoming*

CTP-based educational math assistants will be novel and powerful tools for patient and reliable feedback in exercising and assessing stepwise problem solving.

If, in addition, educational math assistants want to go in line with renewed science education [29] and decisively support inquiry-based learning and independent learning, then *"next step guidance"* is required: students can explore *new* topics, tackle motivating problems and interactively try out solutions passing new stuff by help of the system, students can try variants in calculations and ask for the next step if got stuck — the latter is *the unique feature Lucas-Interpretation contributes as a base technology for educational math assistants.*

Acknowledgments

The author thanks Peter Lucas for granting the approval to connect his name with the ideas presented in this paper, ideas which have not yet gained success comparable with his other research and developments.

Thanks also to Franz Wotawa for his long-lasting support in developing ISAC until usability in educational practice and academic discussion of the underlying ideas have been established.

This paper could not have been written without the help of two colleagues: Bernhard Aichernig, Peter Lucas' former student and his assistant, gave the hints which formalisms to use; last but not least Wolfgang Schreiner, RISC Linz, greatly helped to appropriately apply these formalisms.

References

[1] Victor Aladjav & Marijonas Bogdevicius (2006): *Maple: Programming, Physical and Engineering Problems*. Fultus Corporation.

[2] Ralph-Johan Back (2009): *Structured Derivations as a Unified Proof Style for Teaching Mathematics*. TUCS Technical Report 949, TUCS - Turku Centre for Computer Science, Turku, Finland.

[3] Ralph-Johan Back & Joakim von Wright (1998): *Refinement Calculus: A Systematic Introduction*. Springer-Verlag. Graduate Texts in Computer Science.

[4] Bruno Buchberger & R. Loos (1982): *Computer Algebra - Symbolic and Algebraic Computation*, chapter Algebraic Simplification, pp. 11–44. Springer, Vienna, New York.

[5] Edsger W. Dijkstra & Carel S. Scholten (1990): *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science, Springer verlag, New York.

[6] Georges Gonthier (2007): *The Four Colour Theorem: Engineering of a Formal Proof*. In Deepak Kapur, editor: *ASCM*, *Lecture Notes in Computer Science* 5081, Springer, p. 333, doi:10.1007/978-3-540-87827-8_28.

[7] D. Gries & F. Schneider (1995): *Teaching math more effectively through calculational proofs*. *Am. Math. Monthly* , pp. 691–697doi:10.2307/2974638.

[8] 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.

[9] John R. Harrison (1998): *Theorem proving with the real numbers*. Distinguished Dissertations, Springer, doi:10.1007/978-1-4471-1591-5.

[10] G. Huet, G. Kahn & C. Paulin-Mohring (1994): *The Coq Proof Assistant*. CNRS-ENS Lyon.

[11] D.J. Jeffrey & A.C. Norman (2004): *Not seeing the roots for the branches: multivalued functions in computer algebra*. *SIGSAM Bull.* 38(3), pp. 57–66, doi:10.1145/1040034.1040036.

[12] Cezary Kalisyk (2009): *Correctness and Availability. Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Ipa dissertation series 2009-18, Radboud University Nijmegen. Promotor Herman Geuvers.

[13] Cezary Kaliszyk (2008): *Automating Side Conditions in Formalized Partial Functions*. In Serge Autexier, John Campbell, Julio Rubio, Volker Sorge, Masakazu Suzuki & Freek Wiedijk, editors: *AISC/MKM/Calculemus*, *LNCS* 5144, Springer, pp. 300–314, doi:10.1007/978-3-540-85110-3_26.

[14] Cezary Kaliszyk & Freek Wiedijk (2007): *Certified Computer Algebra on Top of an Interactive Theorem Prover*. In Manuel Kauers, Manfred Kerber, Robert Miner & Wolfgang Windsteiger, editors: *Calculemus*, *LNCS* 4573, Springer, pp. 94–105, doi:10.1007/978-3-540-73086-6_8.

[15] Alan Krempler & Walther Neuper (2008): *Formative Assessment for User Guidance in Single Stepping Systems*. In Michael E. Aucher, editor: *Interactive Computer Aided Learning, Proceedings of ICL08*, Villach, Austria.

[16] 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.

[17] Peter Lucas (1981): *Formal Semantics of Programming Languages: VDL*. *IBM Journal of Devt. and Res.* 25(5).

[18] Peter Lucas & Kurt Walk (1970): *On the Formal Description of PL/I*. *Annual Review in Automatic Programming* 6, Pergamon Press, Oxford.

[19] Christoph Lüth & Burkhart Wolff (2000): *TAS — A Generic Window Inference System*. In J. Harrison & M. Aagaard, editors: *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, *Lecture Notes in Computer Science* 1869, Springer Verlag, pp. 405–422, doi:10.1007/3-540-44659-1_25.

[20] Roman E. Maeder (1997): *Programming in Mathematica*. Addison-Wesley, Reading, Mass.

[21] Wenzel Makarius (2007): *Isabelle/Isar — a generic framework for human-readable proof documents*. In R. Matuszewski & A. Zalewska, editors: *From Insight to Proof — Festschrift in Honour of Andrzej Trybulec*, *Studies in Logic, Grammar, and Rhetoric* 10, University of Bialystok.

[22] Walther Neuper (2006): *Angewandte Mathematik und Fachtheorie*. Technical Report 357, Institute of Instructional and School Development (IUS), University of Klagenfurt, Austria.

[23] Walther Neuper (2007): *Angewandte Mathematik und Fachtheorie*. Technical Report 683, Institute of Instructional and School Development (IUS), University of Klagenfurt, Austria.

[24] Walther Neuper (2010): *Common grounds for modelling mathematics in educational software*. *Int. Journal for Technology in Mathematics Education* 17(3).

[25] Walther Neuper & Johannes Reitinger (2008): *Begreifen und Mechanisieren beim Algebra Einstieg*. Technical Report 1063, Institute of Instructional and School Development (IUS), University of Klagenfurt, Austria.

[26] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. *LNCS* 2283, Springer, doi:10.1007/3-540-08766-4_8.

[27] Russell O'Connor (2009): *Incompleteness and Completeness.* Ph.D. thesis, Radboud University Nijmegen.

[28] Gordon D. Plotkin (1981): *A structural approach to operational semantics*. Technical Report DAIMI FN-19, CS, Aarhus University.

[29] Michel Rocard & al. (2007): *Science Education NOW: A Renewed Pedagogy for the Future of Europe*. Technical Report, European Communities, Directorate-General for Research.

[30] Makarius Wenzel (2011): *The Isabelle/Isar Implementation*. (included in the Isabelle distribution). With contributions by Florian Haftmann and Larry Paulson.

[31] Markus Wenzel & Gertrud Bauer (2001): *Calculational reasoning revisited - an Isabelle/Isar experience*. In R. J. Boulton & P. B. Jackson, editors: *Theorem Proving in Higher Order Logics*, *LNCS* 2152, TPHOLs 2001, Springer, doi:10.1007/3-540-44755-5_7.