Lucas-Interpretation from Users' Perspective

Walther Neuper
IICM, Institute for Computer Media, University of Technology. Graz, Austria
wneuper@ist.tugraz.at

Requirements-engineering for educational software [14] raised the question, how much efforts would be required for implementing substantial material from mechanics [16, 17] in a system based on technology from Computer Theorem Proving (TP). The question appears relevant for several kinds of "users": for decision makers, for course designers and last not least for staff from faculties of engineering, who is interested to implement their own examples and exercises in the future.

Since TP is new in the field, there are some general informations about how to implement material in a TP-based system: First there needs to be a so-called "theory" which collects or imports all definitions used in the material and which formailises respective theorems and proofs. The system under consideration is Isac [3], a prototype based on the TP Isabelle [2]. Isabelle's standard distribution contains most of the mathematics required, multivariate analysis [1] etc. Further material is in the Archive of Formal Proofs [1]. And what is not yet covered by these sources, can be defined as axioms preliminarily, see for instance[2] . Specification of problems is not much effort, see some prototype implementations[3] . The focus of the paper are the methods solving the problems.

The paper is organised as follows: §1 presents LI as a slight extension of usual interpreters for programming languages and introduces an example used throughout the paper [4], §2 discusses the present state and future development of Isac's programming language. Because the latter is purely functional, without any statement for input or output, there is the question "Where are the Interactions from?" raised in §3 before a conclusion reshapes LI's advantages for educational software.

1 A Slightly Extended Interpreter

An interpreter of a programming language works as sketched in Fig.1 on p.2: The interpreter reads a statement at a certain *location* in a *program*; the statement is *interpret*ed such that the *location* moves on to another statement to be read next; the interpreter also maintains an *environment*, which pairs identifiers encountered in statements with respective values; a step of interpretation updates the *environment* according to the interpreted statement.

A *Lucas-Interpreter (LI)* extends the above with additional elements and actions: First a step of *calculation* is constructed by each step of interpretation. Guarantee of correctness for steps in *calculation*s is the purpose of the additional elements; for logical details see [12], here follows a general explanation according to Fig.1:

A *theory* provides the language elements for certain logical expressions collected in a *context* [18]. In each step of interpretation the *context* provides the logical facts required to correctly deduce the next

---

[1] https://isabelle.in.tum.de/dist/library/HOL/HOL-Multivariate_Analysis/index.html

[2] https://intra.ist.tugraz.at/hg/isa/file/e7afa662670b/src/Tools/isac/Knowledge/Biegelinie.thy

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

[4] The running example is part of another example used with another perspective in [14].

Submitted to:
ThEdu'16

step of *calculation*; this action is called *prove* in Fig.1. A blue square in this figure indicates, that input of a *formula or tactic* to the *calculation* is *prove*d correct by automated provers using the current *context*, which is updated at each step of interpretation.

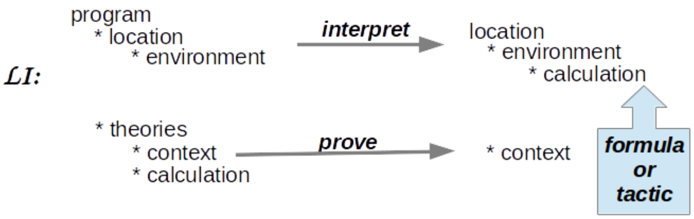

Figure 1: Survey on Lucas-Interpretation

The scope of environments with respect to programs with nested sub-programs has been clarified in the discipline of compiler-construction. However, the scope of contexts in the presence of theories and "locales" is not yet settled [18]. Isac uses the following scoping rules: a context is initialised by the pre-conditions at the start of interpretation, it is visible in sub-programs; the context of a sub-program remains local except predicates containing variables declared in the output of the respective sub-program.

The execution of the program's statements is done by rewriting, as usual with functional programs (where one speaks about "evaluation" of "functions" instead of "execution" of "programs" as we do here). In the present state of Isac's prototype there are lots of evaluators: for list-expressions in programs, for normalisation of user input (for checking correctness). for pre-conditions, etc. Compiling these evaluators is an elaborate, expensive and error-prone task. So migration of Isac's programming language to Isabelle's function package [5] shall free the programmer from these tasks. As soon as this integration is done, all the functions available for Isabelle/HOL are ready for LI, including those which implement computer algebra (see for instance [7, 13]).

A program is accompanied by a "guard"; guard and program together are called a "method" in Isac. The guard is not contained in Fig.1; below an example is copied from [14] §.1:

1 `Guard:`
11 `Given:` Masses $m = 2\,kg$, Length $l_0 = 0.3\,m$, Consts $\{c_1 = 1.1\,\frac{kg}{s^2},\ c_2 = 2.2\,\frac{kg}{s^2}\}$, Damper $d = 0.4\,\frac{Ns}{m}$
12 `Where:` $0 < m \wedge l_0 < 0$
13 `Find:` Matrixes $\{M(m), D(d), C(c_1, c_2)\}$, DiffEq $M \cdot \ddot{x} + D \cdot \dot{x} + C \cdot x = F$
14 `Relate:` $\exists x.\ \forall t.\ t > 0 \Rightarrow M \cdot \ddot{x} + D \cdot \dot{x} + C \cdot x = F$

`Given` in line `11` lists the concrete input items to the program, `Find` declares the output item(s). `Where` is the pre-condition, which shows, that the `Guard` is a conjunction of predicates restricting input (and restrictions imposed by physical contants could be added here as well). In `Relate` the post-condition relates input and output (in the sense of [4]); this particular post-condition can be proved by the theory of differential equations, but it is not immediately useful for an engineer, who wants the solution of an equation and not only some promise by $\exists$.

2 The Interpreted Language

Isac's programming language has been implemented [9] before the "function package" [5] has been introduced to Isabelle. But Isac's language has been designed such, that it anticipated the function package and now the former can be explained in terms of the latter.

Functions in Isabelle/HOL must be total in order to keep the logic consistent. Isac's programs are not total in general, their input is restricted by pre-conditions as shown in §1. Thus Isac's programs are declared as `partial_function` in Isabelle.

A major difference between Isabelle and Isac concerns the purpose of functions: while the former is built for proving properties of functions, for evaluating them and probably generating efficient code from them, the latter is built for stepwise construction of calculations solving problems in (applied) mathematics. Construction of calculations comprises interactively specifying and solving the respective problem. Below the guard from §1 above is re-used by the `Specification` (thus not shown again and folded in) for a `Problem` with a `Solution` as follows:

21 `Problem` [determine, 2-mass-oscillator, DiffEq]:

211 `Specification`:

212 `Solution`:

2121 forces of springs

2122 $[F_{c1} = c_1 x_1,\ F_{c2} = c_2(x_2 - x_1),\ F_{c3} = c_1 x_2]$

2123 forces of dampers

2124 $[F_{d1} = d\dot{x}_1,\ F_{d2} = d\dot{x}_2]$

2125 mass times acceleration equals sum of all forces

2126 $[m\ddot{x}_1 = -F_{c1} + F_{c2} - F_{d1} - F_1,\ \ m\ddot{x}_2 = -F_{c2} - F_{c3} - F_{d2} + F_2]$

2127 `Substitute` $[F_{c1}, F_{c2}, F_{c3}, F_{d1}, F_{d2}]$

2128 $[m\ddot{x}_1 = -c_1 x_1 + c_2(c_2 - x_1) - d\dot{x}_1 - F_1,\ \ m\ddot{x}_2 = -c_2(c_2 - x_1) - c_1 x_2 - d\dot{x}_2 + F_2]$

2129 `Rewrite_Set` normalise

212a $[m\ddot{x}_1 + d\dot{x}_1 + c_1 x_1 - c_2(x_2 - x_1) = F_1,\ \ m\ddot{x}_2 + d\dot{x}_2 + c_2(x_2 - x_1) + c_1 x_1 = F_2]$

212b switch to vector representation

212c $\begin{pmatrix} m & 0 \\ 0 & m \end{pmatrix} \begin{pmatrix} \ddot{x}_1 \\ \ddot{x}_2 \end{pmatrix} + \begin{pmatrix} d & 0 \\ 0 & d \end{pmatrix} \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} + \begin{pmatrix} c_1 + c_2 & -c_2 \\ -c_2 & c_1 + c_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ F_1 \end{pmatrix} + \begin{pmatrix} 0 \\ F_2 \end{pmatrix}$

22 $\begin{pmatrix} m & 0 \\ 0 & m \end{pmatrix} \ddot{x} + \begin{pmatrix} d & 0 \\ 0 & d \end{pmatrix} \dot{x} + \begin{pmatrix} c_1 + c_2 & -c_2 \\ -c_2 & c_1 + c_2 \end{pmatrix} x = \begin{pmatrix} 0 \\ F_1 \end{pmatrix} + \begin{pmatrix} 0 \\ F_2 \end{pmatrix}$

The above `Solution` is considered as close to a calculation written by hand on a blackboard as possible; on the left margin there are the formulas of the calculation (indented according to the calculation's structure), on the right margin there are the tactics and hints; the rest should be self-explanatory. The `Solution` is the result of Lucas-Interpretation of this program:

```
.     partial_function diffeq_2_mass_oscil (m, l_0, [c_1, c_2], d, springs, dampers, sums) =
1       let
11        begin_parallel
1101          springs = Take springs "forces of springs"
111         parallel
1111          dampers = Take dampers "forces of dampers"
112         parallel
1121          sums = Take sums "mass times acceleration equals sum of all forces"
12        end_parallel
13        diffeq = Take sums ""
14        diffeq = Substitute [ springs, dampers ]
15        diffeq = Rewrite_Set normalise
```

```
16      diffeq = Rewrite_Set vectorify “switch to vector representation”
2    in
21      diffeq
```

The `partial_function` *diffeq_2_mass_oscil* gets the arguments *(m, l_0, [c_1, c_2], d, springs, dampers, sums)* from the preceeding specification phase. The first lines `2122..2126` of the calculation could have been constructed in arbitrary order; this is reflected by the lines `11..12` in the above function: the statement `parallel` models parallel execution, while the remaining statements `13..17` represent a sequence.

Above there are specific statements, called “tactics” in weak analogy to TP: `Take`, `Substitute` and `Rewrite_Set` as examples for some dozen others. Tactics are handled by LI like “break points” by debuggers: interpretation halts at the tactics and passes control to the dialog component, which decides how to pass control back to LI, see the next section.

3 Where are the Interactions from?

As shown in §2 by example, a program in Isac is purely functional as is an Isabelle function, without side-effects and without input or output. However, LI creates side-effects in a specific way and cooperates with a dialogue component. The architectural design is shown in Fig.2. The *WorksheetDialog* implements the observer pattern [6] and listens to two active components: the *Worksheet* as interface to the user on the one side and to the *MathEngine* as interface to LI on the other side; the latter is active in the sense, that response may be delayed due to heavy prover activity:

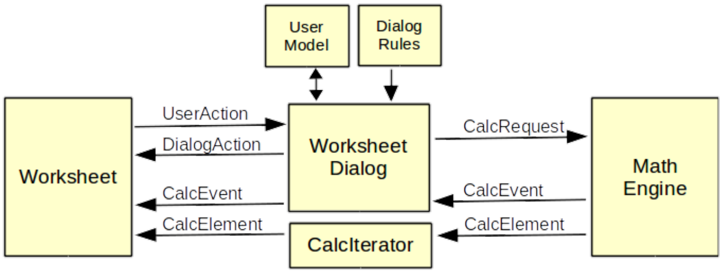

Figure 2: Isac’s dialog component

Here come some details, how automated dialog generation works, by use of the running example: LI starts after successful completion of interactive specification, reads line `11` in the function on p.3 and halts at the tactic `Take`. By delivering a *CalcEvent* the *MathEngine* passes control to the *WorksheeDialog*. Now the dialogue is free to choose various kinds of interaction from the *DialogRules* according to the *UserModel*: the user can be a novice and might prefer to watch passively, or the dialog-mode is set to “exam” and forces the student to be active, etc. So, the dialog might …

- allow the student to request the next formula (passive watching)
- suggest the kind of next step by “forces of springs”
- provide a partial next formula like $[F_{c1} = \ldots,\ F_{c2} = \ldots,\ F_{c3} = \ldots]$ to be completed as shown by [14].Fig.2.
- suggest further parts of the formula (some more activity)
- enforce input of the formula (maximal activity on the student’s side): each of the lines `2122`, `2124` or `2126` could be input due to the `parallel` statement. And the input formula can have

arbitrary format as long as it is algebraically equivalent, due to simplification to a normal form in the *MathEngine*.
- allow to review the method helping in construction of the calculation (if not in "exam" mode)
- allow to investigate Isac's mathematics knowledge base, look at other examples and continue with the calculation eventually.
- etc.

Depending on the decision of the *WorksheetDialg* after some *UserAction*s and *DialogActions* the respective *CalcRequest* is sent to the *MathEngine*. The latter responds with a *CalcEvent*, which notifies the *Worksheet* (by the way controlled by the *WorksheetDialog*) that a new *CalcElement* waits to be fetched from the *MathEngine* via a *CalcIterator*, which has read-access to the whole calculation under construction by LI.

The kind of interaction described above can be considered a dialog between partners on an equal base: both, the student and the system, are able to do steps of calculation more or less actively. The architecture reflects this balance, where LI is the source of power on the system side during interactive calulation.

In the present state of development the *DailogRules* are implemented only to an extent which allows demonstration of LI by Isac's prototype. A general machinery [8] is ready to cope with the complexity of interaction expected in the future. Respective "dialog authoring" will be an efficient investment: interactions in doing mathematics are considered independent from various areas of mathematics; also differences in behaviour of novices versus experts are considered the same in all areas.

The present state of development, the *UserModel* is still a stub. The stub is designed such, that each student is assigned an indiviual data set, which can be preset as well as updated during a session: error rates on specific knowledge items (rule, problem, method, example), preferences in interaction and current dialog mode. Three modes are envisaged at least: investigation, exercise, examination (the modes are subsets of *DailogRules*).

Relevant in the context of questions about efforts for implementation of interactive TP-based course material is: the implementation of functions is not concerned with interaction at all, a "mathematics author" can focus mathematics and nothing else.

4 Conclusion

In spite of higher complexity of TP-based systems with Lucas-Interpretation as compared with a Computer Algebra System, the implementation of course material does not require more efforts in principle, as soon as Isac's programming languages has migrated to Isabelle's function package. Purely functional programs are specifically interpreted by so-called Lucas-Interpretation (LI), which generates steps of calculation and respective dialogue guidance as side-effects of steps in interpretation.

Lucas-Interpretation maintains a context, which provide the most powerful technology available with logical facts for checking correctness of user-input. Thus there is maximal freedom for input – for algorithmic sequences as well as for formula representation: equivalence modulo a theory is checked with maximal reliability. The ability to propose a next step in calculations is the novel feature contributed by Lucas-Interpretation.

No additional efforts are required when programming methods to solve engineering problems: checking steps and proposing steps is done by Lucas-Interpretation automatically and in cooperation with a dialogue module user-guidance is generated automatically.

Finally, after fifteen years of conceptual work and of prototyping, a successful proof of concept [10, 11, 15] some time ago and after the recent requirements engineering, Isac appears ready to start development for a professional release usable at universities of applied sciences.

References

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

[2] *Generic proof assistant "'Isabelle"'*. `http://isabelle.in.tum.de/`.

[3] *Isac-project*. `http://www.ist.tugraz.at/isac/History`.

[4] Dines Bjørner (2006): *Software Engineering*. Texts in Theoretical Computer Science 1,2,3, Springer, Berlin, Heidelberg.

[5] Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkk & John Matthews (2008): *Imperative Functional Programming with Isabelle/HOL*. In Otmane Mohamed, Csar Muoz & Sofine Tahar, editors: Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science 5170, Springer Berlin / Heidelberg, pp. 134–149, doi:10.1007/978-3-540-71067-7_14. Available at `http://dx.doi.org/10.1007/978-3-540-71067-7_14`.

[6] Ralph Johnson Erich Gamma, Richard Helm & John M.Vlissides (1994): *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[7] Florian Haftmann, Andreas Lochbihler & Wolfgang Schreiner (2014): *Towards abstract and executable multivariate polynomials in Isabelle*. Isabelle Workshop 2014, `http://www.infsec.ethz.ch/people/andreloc/publications/haftmann14iw.pdf`.

[8] Markus Kienleitner (2012): *Towards "NextStep Userguidance" in a Mechanized Math Assistant*. Master's thesis, IICM, Graz University of Technology. Bakkalaureate Thesis.

[9] Walther Neuper (2001): *Reactive User-Guidance by an Autonomous Engine Doing High-School Math*. Ph.D. thesis, IICM - Inst. f. Softwaretechnology, Technical University, A-8010 Graz.
http://www.ist.tugraz.at/projects/isac/publ/wn-diss.ps.gz.

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

[11] Walther Neuper (2007): *Angewandte Mathematik und Fachtheorie*. Technical Report 683, IMST – Innovationen Machen Schulen Top!, University of Klagenfurt, Institute of Instructional and School Development (IUS), 9010 Klagenfurt, Sterneckstrasse 15.
`http://imst.uni-klu.ac.at/imst-wiki/index.php/Angewandte_Mathematik_und_Fachtheorie_2006/2007`.

[12] Walther Neuper (2012): *Automated Generation of User Guidance by Combining Computation and Deduction*. pp. 82–101, doi:10.4204/EPTCS.79.5. `http://eptcs.web.cse.unsw.edu.au/paper.cgi?THedu11.5`.

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

[14] Walther Neuper (2016): *Rigor of TP in Educational Engineering Software*. In: submitted to CICM, Bialystok, Poland. `http://www.ist.tugraz.at/projects/isac/publ/tp-engin-sw.pdf`.

[15] Walther Neuper & Johannes Reitinger (2008): *Begreifen und Mechanisieren beim Algebra Einstieg*. Technical Report 1063, IMST – Innovationen Machen Schulen Top!, University of Klagenfurt, Institute of Instructional and School Development (IUS), 9010 Klagenfurt, Sterneckstrasse 15.

`http://imst.uni-klu.ac.at/imst-wiki/index.php/Begreifen_und_Mechanisieren_beim_Algebra-Einstieg`.

[16] Wolfgang Steiner (2012): *Vorlesungsskriptum Technische Mechanik II. Sommersemester 2012*. FH OÖ Campus Wels.

[17] Wolfgang Steiner (2015): *Vorlesungsskriptum Technische Mechanik III*. FH OÖ, Fakultät für Technik und Umweltwissensschaften.

[18] Makarius Wenzel (2015): *The Isabelle/Isar Implementation*. `http://isabelle.in.tum.de/website-Isabelle2015/dist/Isabelle2015/doc/implementation.pdf`.