Lucas Interpretation from Programmers' Perspective

Walther A. Neuper

University of Technology, Graz, Austria
wneuper@ist.tugraz.at

1 Introduction

This extended abstract describes the initial, but already effective step of a major development task in the ISAC-project[1]

A recent survey on ISAC's [6, p. 102] development listed the task described here at the first place: Shift ISAC's programming language towards Isabelle's function package (FP) [4]. The original definition [9] is updated according to what has stabilised during prototyping. In the meanwhile also logical foundations of Lucas Interpretation have been clarified [10]. Since a short paper on the users' perspective [12] and a case study [11], this paper will focus the programmers' view and take the opportunity to point at advantages introduced by Isabelle/jEdit [16] for programming, also for ISAC's programms.

The paper is structured as follows: §2 presents the current state of the programming language, §3 sketches what a programmer can expect from combining Lucas Interpretation and Isabelle's FP, §4 gives a detailed account of tasks to accomplish in further integration of Lucas Interpretation into the FP and §5 are the final conclusions.

2 The Program Language

The original definition [9, p. 86] has been adapted due to experiences during prototyping; presently the definition in Backus-Naur form (BNF) is as follows (terminal symbols are written **bold** face, the numbers on the left serve referencing and do not belong to the BNF):

```
01  definition      ::=  partial_function (tailrec) fun-id :: signature where program
02    program       ::=  " fun-id arg+ = ( body ) "
03      fun-id      ::=  identifier
04      arg         ::=  identifier
05      body        ::=  if bool-expr then expr else expr
06                   |   let assigns in expr
07                   |   expr
08        assigns   ::=  (assign ;)* assign
09        assign    ::=  identifier = body
10        expr      ::=  ( tac-expr | no-tac-expr )
```

The elements `signature`, `identifier` and `bool-expr` are as given by the FP, as well as `if`, `let` and `no-tac-expr`, the kind of expressions native to the FP. The specific element is `tac-expr`:

```
21  tac-expr        ::=  tacs  no-tac-expr
22                   |   SubProblem ( identifier , key , key) probl-args
23    tacs          ::=  tactical-1  tacs
24                   |   tactical-2  tacs  tacs
```

[1] http://www.ist.tugraz.at/isac/History

```
25                  |   tactic
26    key           ::= [ ( ID ,)* ID ]
27      ID          ::= identifier                    (* declared as constant *)
28    probl-args    ::= [ ( probl-arg ,)* probl-arg ]
29      probl-arg   ::= type-con no-tac-expr
30        type-con  ::= REAL | REAL_LIST | REAL_SET | BOOL | BOOL_LIST
```

Before the `tactics` and `tacticals` are discussed in detail, a hint on respective semantics is given and an issue with the FP is recognised:

The distinctive semantics of tactics is, that they are recognised as breakpoints by Lucas Interpretation. The breakpoints hand over control to a student (or a dialogue module [6, p. 97ff]), while the programmer can "forget user interaction" (see the respective paragraph in §3). The tactic `SubProblem` specifically involves "work on libraries of theories, specifications and methods" (see the respective paragraph in §3, too).

The issue with tactics in conjunction with the function pacakge is introduced by a valuable feature of the latter: it rejects free variables on the right-hand side of equalities (`assign`ments, line `09` above). While this is helpful in programming generally, it requires the arguments of tactics to be constants (in Isabelle's term language). Similarly with `SubProblem`: for the sake of generality the formal arguments are collected in a list, where the type is unified by `type-con`straints like `REAL, REAL_LIST,` etc.

Finally, here comes the list of tactics and of tacticals, both notions borrowed from computer theorem proving:

```
41    tactic      ::= Take
42                |   Rewrite ID
43                |   Rewrite_Inst subst ID
44                |   Calculate op
45                |   Rewrite_Set ID
46                |   Rewrite_Inst subst ID
47                |   Substitute
48      subst     ::= [ ( (identifier, no-tac-expr) ,)* (identifier, no-tac-expr) ]
49      op        ::= PLUS | MINUS | TIMES | DIVIDE | POWER
```

Tactics contribute to visible steps in calculations in the following ways:

Take assembles a term as described by a `no-tac-expr` and displays it on a worksheet as shown, for instance at [6, p. 97].

Rewrite takes a term, rewrites it according to a theorem (given by an identifier of type `ID`) and displays the result in a worksheet. In case the theorem does not match, the program terminates with an exception[2].

Rewrite_Inst works like `Rewrite`, but instantiates the theorem by use of a `subst`itution. This allows to disburden students from λ-notation) in equation solving and with functions as first-order terms: bound variable(s) are substituted by the respective identifier encoded as a constant.

Calculate takes a term and calculates two adjacent numerals according to `op`. Adjacent numerals are recognised with respect to associativity, e.g. $(a + 1) + 2$ would be simplified by `PLUS` as well as $a + (1 + 2)$, but not $(1 + a) + 2$.

[2]Exception handling is not yet implemented, see Pt.5a on p.6

Rewrite_Set works like `Rewrite`, but rewrites with a normalising term rewrite system (in ISAC called "rule-set").

Rewrite_Set_Inst instantiates all theorems in the rule-set and the rewrites with this rule-set.

Substitute assembles a term as described by a `no-tac-expr`, `substitutes` and displays it on a worksheet[3].

Tacticals combine tactics; they take one or two arguments, the latter kind is declared as infix as shown in the following BNF:

```
61  tactical-1  ::=  Try
62              |    Repeat
63              |    While bool-expr
64  tactical-2  ::=  Or                    (* infix *)
65              |    @@                    (* infix *)
```

Try takes a `tactic` (or nested tacticals) to be interpreted. If there are no applicable tactics, then an idle step is performed (without displaying anything on the worksheet).

Repeat works the same way as `Try`, but requires at least one applicable tactic and proceeds until none of the tactics are applicable.

While works similar to `Repeat` but terminates according to `bool-expr`.

Or takes two arguments (two `tactics` or two nested tacticals), checks the first one and in case some tactic is applicable, interpretation of this argument is done; otherwise the second argument is interpreted. In case none of the arguments contains an applicable tactic, an exception is thrown.

@@ takes two arguments (two `tactics` or two nested tacticals) and interpretes them in sequence; this is forward chaining of functions. Each argument must contain an applicable tactic.

Examples of programs are given in [6, 10, 12] and one in the sequel.

3 The Programmers' Perspective

For widespread usage of the ISAC tutoring system not only usability for students will be essential, but also convenient programming: Course designers, lecturers and teachers are supposed to program examples of engineering mathematics they want their students to study and to exercise. This section focuses the features considered appealing to programmers, issues of developing the software machinery behind the scenes will be considered in the subsequent secition §4.

[3] There are design considerations to determine the substitution just by a list of respective identifiers, which are substituted from the current environment

Focus algorithms and forget user interaction is the first advantage of Lucas Interpretation, already described in [12] and discussed from a technical point of view [6, p. 97]. User interaction on the new kind of powerful mathematics engine has been expected highly complex from the very beginning [5]; recently this complexity has been extended to the specification phase [14]. However, the expectation, that this complexity can be mastered by rule-based systems [3] is still speculative. Respective contacts to experts in didactics, human computer interaction, cognitive science and the like show, that it will be hard to recruit expertise required; one has to hope for fruitful on-the-job learning.

Take advantage from Isabelle's function package (FP), which has been experienced in transferring the example [6, p. 92] from plain parsing of strings as terms [4] to the FP: syntax errors are indicated accurately at the right location by Isabelle/jEdit [16], type annotations for the function's arguments shift into the initial signature, less type annotations are required within the code, syntax highlighting indicates how identifiers are interpreted (as constants, as free variable, etc), free variables on the right-hand-side of equalities (assignments in line 09 on p.1) are rejected, etc. The result is Fig.1.

```
partial_function (tailrec) biegelinie ::
  "real ⇒ real ⇒ real ⇒ (real ⇒ real) ⇒ bool list ⇒ bool"
where
  "biegelinie l q v b s =
    (let
      funs = (SubProblem (Biegelinie',
        [vonBelastungZu, Biegelinien], [Biegelinien, ausBelastung])
        [REAL q, REAL v]);
      equs = (SubProblem (Biegelinie',
        [setzeRandbedingungen, Biegelinien], [Biegelinien, setzeRandbedingungenEin])
        [BOOL_LIST funs, BOOL_LIST s]);
      cons = (SubProblem (Biegelinie', [LINEAR, system], [no_met])
        [BOOL_LIST equs, REAL_LIST [c, c_2, c_3, c_4]]);
      B = Take (lastI funs);
      B = ((Substitute cons) @@
           (Rewrite_Set_Inst [(bdv, v)] make_ratpoly_in False)) B
    in B)                                                          "
```

Figure 1: New appearance of the program introduced in [6, p. 92].

In comparison to [6, p. 92] the format of the program in Fig.1 adopts Isabelle's coding standards. There are already a considerable number of programs developed during prototying (see below) which need to be reformatted as well. As a preview on the efficiency of programming in ISAC: the one program in Fig.1 makes a whole section of a textbook [15] interactive, see[5] and the subsequent section.

Work on libraries of theories, specifications and methods. Work in programming languages integrated into Computer Algebra Systems [1, 8] is efficient, because it can resort to powerful libraries of methods. A same kind of library is expected for ISAC, already prototyping

[4] The respective code was at https://intra.ist.tugraz.at/hg/isa/file/c0fe04973189/src/Tools/isac/Knowledge/Biegelinie.thy#l317.

[5] http://www.ist.tugraz.at/projects/isac/www/kbase/exp/exp_Statics_Biegel_Timischl.html

resulted in several dozens of programs[6]. But there is an essential difference between Computer Algebra and ISAC: the former are generally under-specified (if explicitly specified at all), while ISAC is subject to Isabelle's formal rigor. Thus each method in ISAC combines a program with a guard, i.e. a formal specification. There is a preliminary collection from prototyping[7].

The collection of specifications is structured as a tree, which allows for automated problem refinement in special cases, for instance in equation solving. This feature is not yet well documented (see [2, 7]), but Fig.1 may serve as an example: given an equation (or in the case of Fig.1 an equational system in the variable c, c_1, c_2, c_3), one has to normalise the equation (-ional system) such that the type can be determined — that is done by the pre-conditions of the respective specifications, here starting a breadth-first seach from the node `[system]` in the tree of specifications down the branches, until the pre-conditions match. The matching node contains also a method (or several ones for interactive choice by the student[8]), so there is `no_method` assigned in Fig.1. This program also determines the system as `LINEAR`, because other types of systems cannot arise in this calculation.

There is a third kind of libraries, native to theorem proving, libraries of theories. Not only theories contained in the Isabelle distribution[9] but also the Archive of Formal Proofs[10]. For instance, importin theories like [13] would provide interactively playing with messages in ISAC — this is modelled by rewriting und thus interaction would be there for free.

4 Integration into the Function Package

In order to enjoy the advantages of Isabelle's FP and to make programming in ISAC as convenient as described in the previous section, major development efforts are required. This paper is a snapshot of work under construction.

The first step done only touched the surface, but was already nicely effective as shown in Fig.1. In technical terms these steps were:

1. Replace plain parsing of strings as terms and use parsing by the FP's instead.
2. Define constants, which replace free variables formerly accepted by term parsing. This preliminary workaround calls for two further design decisions:
   (a) Make handling of identifiers for theorems, rule-sets and keys convenient.
   (b) Review early design decisions for fixing certain notions as constants, e.g. `bdv` for variables bound by a (univariate) function, which is represented as a term (in order to disburden students from λ-notation). And probably find more elegant ways to determine variables to solve equational systems in, e.g. c, c_1, c_2 and c_3 in the program example.

The above steps clarified which further steps are required to reach the goals.

[6] This http://www.ist.tugraz.at/projects/isac/www/kbase/met/index_met.html is a specific view on ISAC's programs.

[7] http://www.ist.tugraz.at/projects/isac/www/kbase/met/index_met.html

[8] Like all other tactics `SubProblem` is recognised as a breakpoint by Lucas Interpretation; this breakpoint requests specific user interaction, which is guided by a given theory `identifier` and a reference `key` into a tree of specifications; the interaction succeeds with selecting a method – i.e. a function call for solving the specified problem.

[9] https://isabelle.in.tum.de/dist/library/HOL/index.html

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

The most important step for reaching the goal of convenient programming is to compile all rule-sets automatically required by a method[11]. Evaluation by Lucas Interpretation uses ISAC's simplifier (which is different from Isabelle's simplifier [6, p. 94]) and for that purpose the programmer has to compile the following five rule-sets by hand:

- crls: evaluates the post-condition of a specification
- erls: evaluates assumptions of theorems applied by `Rewrite` or `Rewrite_Inst` (if any)
- nrls: canonical simplifier for checking formulas input by a student at breakpoints during interpretation of the program
- prls: evaluates predicates in pre-conditions of specifications
- srls: evaluates `no-tac-expr` as mentioned in line `10` on p.1

The reason for the many rule-sets is, that they should be minimal such, that a student has a chance to review them and to understand how they work. The rule-sets must be confluent and terminating term rewriting systems, so compiling these is a much more tedious taks than writing a program.

For accomplishing the task of automated generation of rule-sets Isabelle's machinery behind `value` shall be studied and adopted as much as possible.

A comprehensive list of steps to do is as follows:

1. Find a convenient way handling for identifiers, i.e. the issue listed as Pt.2 on p.5. There are several possibilities, for instance: localize the constant definitions, replace the identifiers of type `ID` (as found as arguments of `SubProblem`, `Rewrite`, ect) in a pre-processing phase of the FP, etc.

2. Generate rule-sets for evaluation of programs automatically (see previous paragraph)

3. Adapt Isabelle's `value` such that it delivers results from Lucas Interpretation *with bypassing user interaction.*

4. Improve the program language in several points

   (a) Review priorities in constant definitions of tactics and tacticals in order to reduce the number of parentheses in programs

   (b) Remove the boolean argument from the `Rewrite*` tactis; this is left over from early prototyping

5. Improve the Lucas Interpreter in several points

   (a) introduce proper exception handling

   (b) support interactive debugging as a specialisation of the debugger of Isabelle/PolyML; presently there is only a tracing facility triggered by `trace_script`

6. Consider to replace the Isar command `partial_function` by another one depending on how invasive the above adaptions of the FP to the needs of Lucas Interpretation are.

[11] https://intra.ist.tugraz.at/hg/isa/file/c0fe04973189/src/Tools/isac/calcelems.sml#l600

5 Summary and Conclusion

Within the process of pushing the ISAC prototype towards maturity for service in engineering courses one of the most crucial points is to make programming convenient in ISAC. Programming means to describe mathematical algorithms (a functional program without input/output, so no concern with didactics and dialogues, which are handled by a separate component not discussed here) of engineering problems by use of libraries of methods and specifications as well as of respective theories. ISAC builds upon the theorem prover Isabelle, which offers a convenient programming environment and a specific function package. This is considered an appropriate means to improve programming in ISAC as required. The first steps of improvement are described in this paper and the other steps are listed in detail.

The first step, adoption of parsing from Isabelle's function package, was already nicely effective as described in this paper. Further steps improving ISAC's programming environment will require studying the huge code base of Isabelle in order to exploit respective mechanisms optimally. We hope for support from the Isabelle developer team at Munich.

References

[1] Victor Aladjav and Marijonas Bogdevicius. *Maple: Programming, Physical and Engineering Problems*. Fultus Corporation, February 27 2006.

[2] Matthias Goldgruber and Richard Lang. Eine explizite Hierarchie von Typen elementarer Gleichungen. In Josef Böhm and Bernhard Kutzler, editors, *Integrating Technology into Mathematics Education*. ACDCA, July 10-13 2002. http://www.ist.tugraz.at/projects/isac/publ/visitme02-M023.pdf.

[3] Markus Kienleitner. Towards "nextstep userguidance" in a mechanized math assistant. Master's thesis, IICM, Graz University of Technology, 2012. http://www.ist.tugraz.at/projects/isac/publ/mkienl_bakk.pdf.

[4] Alexander Krauss. *Defining Recursive Functions in Isabelle/HOL*. Munich, 2017. Part of the Isabelle distribution.

[5] Alan Krempler and Walther Neuper. Formative assessment for user guidance in single stepping systems. In Michael E. Aucher, editor, *Interactive Computer Aided Learning, Proceedings of ICL08*, Villach, Austria, 2008. http://www.ist.tugraz.at/projects/isac/publ/icl08.pdf.

[6] Alan Krempler and Walther Neuper. Prototyping "systems that explain themselves" for education. In Pedro Quaresma and Walther Neuper, editors, Proceedings 6th International Workshop on *Theorem proving components for Educational software,* Gothenburg, Sweden, 6 Aug 2017, volume 267 of *Electronic Proceedings in Theoretical Computer Science*, pages 89–107. Open Publishing Association, 2018. https://arxiv.org/pdf/1803.01470v1.pdf.

[7] Richard Lang. Elementare Gleichungen der Mittelschulmathematik in der ISACWissensbasis. Master's thesis, University of Technology, Institute of Software Technology, Graz, Austria, March 2003. http://www.ist.tugraz.at/projects/isac/publ/da-rlang.ps.gz.

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

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

[10] Walther Neuper. Automated generation of user guidance by combining computation and deduction. In Pedro Quaresma and Ralph-Johan Back, editors, *Electronic Proceedings in Theoretical Computer Science*, volume 79, pages 82–101. Open Publishing Association, 2012. http://eptcs.web.cse.unsw.edu.au/paper.cgi?THedu11.5.

[11] Walther Neuper. 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, July 7-11 2014. http://ceur-ws.org/Vol-1186/paper-17.pdf.

[12] Walther Neuper. Lucas-interpretation from users' perspective. In *Joint Proceedings of the FM4M, MathUI, and ThEdu Workshops, Doctoral Program, and Work in Progress at the Conference on Intelligent Com-*

puter Mathematics, pages 83–89, Bialystok, Poland, July 25-29 2016. http://cicm-conference.org/2016/ceur-ws/CICM2016-WIP.pdf.

[13] Christoph Sprenger and Ivano Somaini. Developing security protocols by refinement. *Archive of Formal Proofs*, May 2017. http://isa-afp.org/entries/Security_Protocol_Refinement.html, Formal proof development.

[14] FH-Design Team. *ISAC*-project: User stories, user requirements document, use cases document. http://www.ist.tugraz.at/projects/isac/publ/isac-doc2.pdf, 2017.

[15] Wolfgang Timischl and Gerald Kaiser. *Ingenieuer-Mathematik*, volume 3. E. Dorner, Wien, 1999.

[16] Makarius Wenzel. *Isabelle/jEdit*. Munich, 2018. Part of the Isabelle distribution.