Trials with TP-based Programming for Interactive Course Material

Jan Ročnik
jan.rocnik@student.tugraz.at
IST, SPSC
Graz University of Technology
Austria

Abstract

Traditional course material in engineering disciplines lacks an important component, interactive support for step-wise problem solving. Theorem-Proving (TP) technology is appropriate for one part of such support, in checking user-input. For the other part of such support, guiding the learner towards a solution, another kind of technology is required.

Both kinds of support can be achieved by so-called Lucas-Interpretation which combines deduction and computation and, for the latter, uses a novel kind of programming language. This language is based on (Computer) Theorem Proving (TP), thus called a "TP-based programming language".

This paper is the experience report of the first "application programmer" using this language for creating exercises in step-wise problem solving for an advanced lab in Signal Processing. The tasks involved in TP-based programming are described together with the experience gained from a prototype of the programming language and of it's interpreter.

The report concludes with a positive proof of concept, states insufficiency usability of the prototype and captures the requirements for further development of both, the programming language and the interpreter.

1 Introduction

Traditional course material in engineering disciplines lacks an important component, interactive support for step-wise problem solving. The lack becomes evident by comparing existing course material with the sheets collected from written exams (in case solving engineering problems is *not* deteriorated to multiple choice tests) on the topics addressed by the materials. Theorem-Proving (TP) technology can provide such support by specific services. An important part of such services is called "next-step-guidance", generated by a specific kind of "TP-based programming language". In the $\mathcal{ISAC}$-project [1] such a language is prototyped in line with [7] and built upon the theorem prover Isabelle [14] [2]. The

[1] http://www.ist.tugraz.at/projects/isac/
[2] http://isabelle.in.tum.de/

TP services are coordinated by a specific interpreter for the programming language, called Lucas-Interpreter [12]. The language will be briefly re-introduced in order to make the paper self-contained.

The main part of the paper is an account of first experiences with programming in this TP-based language. The experience was gained in a case study by the author. The author was considered an ideal candidate for this study for the following reasons: as a student in Telematics (computer science with focus on Signal Processing) he had general knowledge in programming as well as specific domain knowledge in Signal Processing; and he was *not* involved in the development of $\mathcal{ISAC}$'s programming language and interpreter, thus being a novice to the language.

The goals of the case study were: (1) to identify some TP-based programs for interactive course material for a specific "Advanced Signal Processing Lab" in a higher semester, (2) respective program development with as little advice as possible from the $\mathcal{ISAC}$-team and (3) to document records and comments for the main steps of development in an Isabelle theory; this theory should provide guidelines for future programmers. An excerpt from this theory is the main part of this paper.

The major example resulting from the case study will be used as running example throughout this paper. This example requires a program resembling the size of real-world applications in engineering; such a size was considered essential for the case study, since there are many small programs for a long time (mainly concerned with elementary Computer Algebra like simplification, equation solving, calculus, etc. [3])

The mathematical background of the running example is the following: In Signal Processing, "the $\mathcal{Z}$-transform for discrete-time signals is the counterpart of the Laplace transform for continuous-time signals, and they each have a similar relationship to the corresponding Fourier transform. One motivation for introducing this generalization is that the Fourier transform does not converge for all sequences, and it is useful to have a generalization of the Fourier transform that encompasses a broader class of signals. A second advantage is that in analytic problems, the $\mathcal{Z}$-transform notation is often more convenient than the Fourier transform notation." [15, p. 128]. The $\mathcal{Z}$-transform is defined as

$$X(z)=\sum_{n=-\infty}^{\infty}x[n]z^{-n}$$

where a discrete time sequence $x[n]$ is transformed into the function $X(z)$ where z is a continuous complex variable. The inverse function is addressed in the running example and can be determined by the integral

$$x[n]=\frac{1}{2\pi j}\oint_C X(z)\cdot z^{n-1}dz$$

where the letter C represents a contour within the range of convergence of the $\mathcal{Z}$-transform. The unit circle can be a special case of this contour. Remember that j is the complex number in the domain of engineering. As this transform requires high effort to be solved, tables of commonly used transform pairs are used in education as well as in engineering practice; such tables can be found at [20] or [15, Table 3.1] as well. A completely solved and more detailed example can be found at [15, p. 149f].

Following conventions in engineering education and in practice, the running example solves the problem by use of a table.

[3]The programs existing in the $\mathcal{ISAC}$ prototype are found at http://www.ist.tugraz.at/projects/isac/www/kbase/met/index_met.html

Support for interactive stepwise problem solving in the ISAC prototype is shown in Fig.1 [4]: A student inputs formulas line by line on the *"Worksheet"*, and each step (i.e. each formula on completion) is immediately checked by the system, such that at most *one inconsistent* formula can reside on the Worksheet (on the input line, marked by the red ⊗). If the student gets stuck and does not

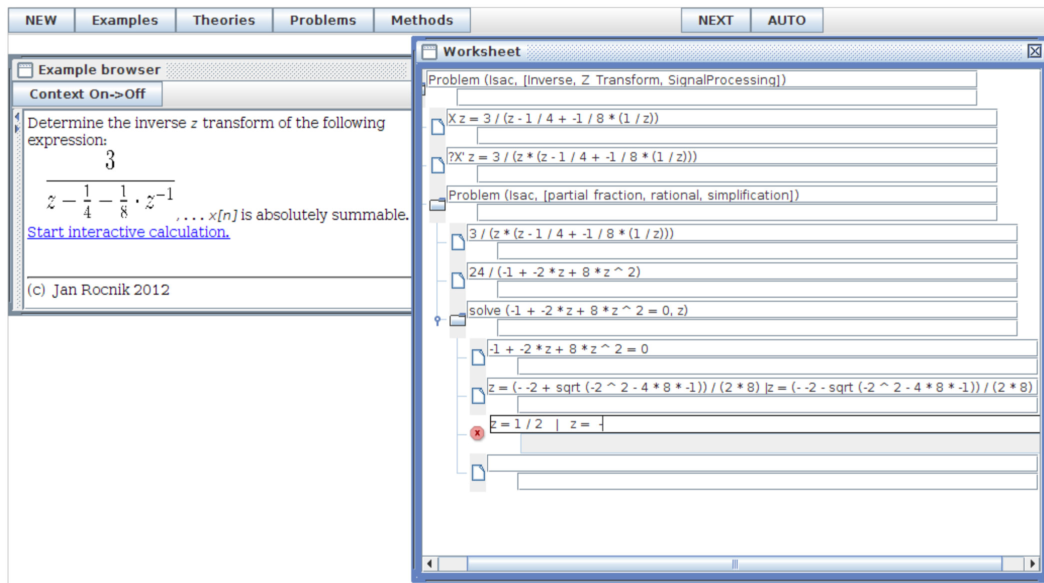

Figure 1: Step-wise problem solving guided by the TP-based program

know the formula to proceed with, there is the button NEXT presenting the next formula on the Worksheet; this feature is called "next-step-guidance" [12]. The button AUTO immediately delivers the final result in case the student is not interested in intermediate steps.

Adaptive dialogue guidance is already under construction [4] and the two buttons will disappear, since their presence is not wanted in many learning scenarios (in particular, *not* in written exams).

The buttons Theories, Problems and Methods are the entry points for interactive lookup of the underlying knowledge. For instance, pushing Theories in the configuration shown in Fig.1, pops up a "Theory browser" displaying the theorem(s) justifying the current step. The browser allows to lookup all other theories, thus supporting indepentend investigation of underlying definitions, theorems, proofs — where the HTML representation of the browsers is ready for arbitrary multimedia add-ons. Likewise, the browsers for Problems and Methods support context sensitive as well as interactive access to specifications and programs respectively.

There is also a simple web-based representation of knowledge items; the items under consideration in this paper can be looked up as well [5] [6] [7].

[4] Fig.1 also shows the prototype status of ISAC; for instance, the lack of 2-dimensional presentation and input of formulas is the major obstacle for field-tests in standard classes.

[5] http://www.ist.tugraz.at/projects/isac/www/kbase/thy/browser_info/HOL/HOL-Real/Isac/**Inverse_Z_Transform.thy**

[6] http://www.ist.tugraz.at/projects/isac/www/kbase/thy/browser_info/HOL/HOL-Real/Isac/**Partial_Fractions.thy**

[7] http://www.ist.tugraz.at/projects/isac/www/kbase/thy/browser_info/HOL/HOL-Real/Isac/**Build_Inverse_Z_Transform.thy**

The paper is structured as follows: The introduction §1 is followed by a brief re-introduction of the TP-based programming language in §2, which extends the executable fragment of Isabelle's language (§2.1) by tactics which play a specific role in Lucas-Interpretation and in providing the TP services (§2.2). The main part §3 describes the main steps in developing the program for the running example: prepare domain knowledge, implement the formal specification of the problem, prepare the environment for the interpreter, implement the program in §3.1 to §3.6 respectively. The work-flow of programming, debugging and testing is described in §4. The conclusion §5 will give directions identified for future development.

2 ISACS's Prototype for a Programming Language

The prototype of the language and of the Lucas-Interpreter is briefly described from the point of view of a programmer. The language extends the executable fragment of Higher-Order Logic (HOL) in the theorem prover Isabelle [14][8].

2.1 The Executable Fragment of Isabelle's Language

The executable fragment consists of data-type and function definitions. It's usability even suggests that fragment for introductory courses [13]. HOL is a typed logic whose type system resembles that of functional programming languages. Thus there are

base types, in particular *bool*, the type of truth values, *nat*, *int*, *complex*, and the types of natural, integer and complex numbers respectively in mathematics.

type constructors allow to define arbitrary types, from *set*, *list* to advanced data-structures like *trees*, red-black-trees etc.

function types, denoted by $\Rightarrow$.

type variables, denoted by $'a, 'b$ etc, provide type polymorphism. Isabelle automatically computes the type of each variable in a term by use of Hindley-Milner type inference [9, 10].

Terms are formed as in functional programming by applying functions to arguments. If f is a function of type $\tau_1 \Rightarrow \tau_2$ and t is a term of type τ_1 then $f\ t$ is a term of type τ_2. $t\ ::\ \tau$ means that term t has type τ. There are many predefined infix symbols like $+$ and $\leq$ most of which are overloaded for various types.

HOL also supports some basic constructs from functional programming:

01 (`if` b `then` t_1 `else` t_2)
02 (`let` $x = t$ `in` u)
03 (`case` t `of` $pat_1 \Rightarrow t_1 \mid \ldots \mid pat_n \Rightarrow t_n$)

The running example's program uses some of these elements (marked by `tt-font` on p.101): for instance `let...in` in lines 02 ...13. In fact, the whole program is an Isabelle term with specific

[8] http://isabelle.in.tum.de/

function constants like `program`, `Take`, `Rewrite`, `Subproblem` and `Rewrite_Set` in lines 01, 03. 04, 07, 10 and 11, 12 respectively.

Formulae are terms of type *bool*. There are the basic constants *True* and *False* and the usual logical connectives (in decreasing order of precedence): $\neg, \wedge, \vee, \rightarrow$.

Equality is available in the form of the infix function $=$ of type $a \Rightarrow a \Rightarrow bool$. It also works for formulas, where it means "if and only if".

Quantifiers are written $\forall x.\ P$ and $\exists x.\ P$. Quantifiers lead to non-executable functions, so functions do not always correspond to programs, for instance, if comprising
$(\ if\ \exists x.\ P\ then\ e_1\ else\ e_2\ )$.

2.2 ISAC's Tactics for Lucas-Interpretation

The prototype extends Isabelle's language by specific statements called tactics [9]. For the programmer these statements are functions with the following signatures:

Rewrite: $theorem \Rightarrow term \Rightarrow term * term\ list$: this tactic applies *theorem* to a *term* yielding a *term* and a *term list*, the list are assumptions generated by conditional rewriting. For instance, the *theorem* $b \neq 0 \wedge c \neq 0 \Rightarrow \frac{a \cdot c}{b \cdot c} = \frac{a}{b}$ applied to the *term* $\frac{2 \cdot x}{3 \cdot x}$ yields $(\frac{2}{3}, [x \neq 0])$.

Rewrite_Set: $ruleset \Rightarrow term \Rightarrow term * term\ list$: this tactic applies *ruleset* to a *term*; *ruleset* is a confluent and terminating term rewrite system, in general. If none of the rules (*theorem*s) is applicable on interpretation of this tactic, an exception is thrown.

Substitute: $substitution \Rightarrow term \Rightarrow term$: allows to access sub-terms.

Take: $term \Rightarrow term$: this tactic has no effect in the program; but it creates a side-effect by Lucas-Interpretation (see below) and writes *term* to the Worksheet.

Subproblem: $theory * specification * method \Rightarrow argument\ list \Rightarrow term$: this tactic is a generalisation of a function call: it takes an *argument list* as usual, and additionally a triple consisting of an Isabelle *theory*, an implicit *specification* of the program and a *method* containing data for Lucas-Interpretation, last not least a program (as an explicit specification) [10].

The tactics play a specific role in Lucas-Interpretation [12]: they are treated as break-points where, as a side-effect, a line is added to a calculation as a protocol for proceeding towards a solution in step-wise problem solving. At the same points Lucas-Interpretation serves interactive tutoring and hands over control to the user. The user is free to investigate underlying knowledge, applicable theorems, etc. And the user can proceed constructing a solution by input of a tactic to be applied or by input of a formula; in the latter case the Lucas-Interpreter has built up a logical context (initialised with the precondition of the formal specification) such that Isabelle can derive the formula from this context — or give feedback, that no derivation can be found.

[9] $\mathcal{ISAC}$'s. These tactics are different from Isabelle's tactics: the former concern steps in a calculation, the latter concern proofs.

[10] In interactive tutoring these three items can be determined explicitly by the user.

2.3 Tactics as Control Flow Statements

The flow of control in a program can be determined by `if then else` and `case of` as mentioned on p.94 and also by additional tactics:

Repeat: $tactic \Rightarrow term \Rightarrow term$: iterates over tactics which take a *term* as argument as long as a tactic is applicable (for instance, `Rewrite_Set` might not be applicable).

Try: $tactic \Rightarrow term \Rightarrow term$: if *tactic* is applicable, then it is applied to *term*, otherwise *term* is passed on without changes.

Or: $tactic \Rightarrow tactic \Rightarrow term \Rightarrow term$: If the first *tactic* is applicable, it is applied to the first *term* yielding another *term*, otherwise the second *tactic* is applied; if none is applicable an exception is raised.

@@: $tactic \Rightarrow tactic \Rightarrow term \Rightarrow term$: applies the first *tactic* to the first *term* yielding an intermediate term (not appearing in the signature) to which the second *tactic* is applied.

While: $term :: bool \Rightarrow tactic \Rightarrow term \Rightarrow term$: if the first *term* is true, then the *tactic* is applied to the first *term* yielding an intermediate term (not appearing in the signature); the intermediate term is added to the environment the first *term* is evaluated in etc. as long as the first *term* is true.

The tactics are not treated as break-points by Lucas-Interpretation and thus do neither contribute to the calculation nor to interaction.

3 Concepts and Tasks in TP-based Programming

This section presents all the concepts involved in TP-based programming and all the tasks to be accomplished by programmers. The presentation uses the running example from Fig.1 on p.93.

3.1 Mechanization of Math — Domain Engineering

The running example requires to determine the inverse $\mathcal{Z}$-transform for a class of functions. The domain of Signal Processing is accustomed to specific notation for the resulting functions, which are absolutely capable of being totalled and are called step-response: $u[n]$, where u is the function, n is the argument and the brackets indicate that the arguments are discrete. Surprisingly, Isabelle accepts the rules for z^{-1} in this traditional notation [11]:

01 axiomatization where
02 rule1: "$z^{-1}\ 1 = \delta[n]$" and
03 rule2: "$||z|| > 1 \Rightarrow z^{-1}\ z/(z-1) = u[n]$" and
04 rule3: "$||z|| < 1 \Rightarrow z/(z-1) = -u[-n-1]$" and
05 rule4: "$||z|| > ||\ \alpha\ || \Rightarrow z/(z-\alpha) = \alpha^n \cdot u[n]$" and
06 rule5: "$||z|| < ||\alpha|| \Rightarrow z/(z-\alpha) = -(\alpha^n) \cdot u[-n-1]$" and
07 rule6: "$||z|| > 1 \Rightarrow z/(z-1)^2 = n \cdot u[n]$"

[11] Isabelle experts might be particularly surprised, that the brackets do not cause errors in typing (as lists).

These 6 rules can be used as conditional rewrite rules, depending on the respective convergence radius. Satisfaction from accordance with traditional notation contrasts with the above word *axiomatization*: As TP-based, the programming language expects these rules as *proved* theorems, and not as axioms implemented in the above brute force manner; otherwise all the verification efforts envisaged (like proof of the post-condition, see below) would be meaningless.

Isabelle provides a large body of knowledge, rigorously proved from the basic axioms of mathematics [12]. In the case of the $\mathcal{Z}$-transform the most advanced knowledge can be found in the theories on Multivariate Analysis [13]. However, building up knowledge such that a proof for the above rules would be reasonably short and easily comprehensible, still requires lots of work (and is definitely out of scope of our case study).

3.2 Preparation of Simplifiers for the Program

All evaluation in the prototype's Lucas-Interpreter is done by term rewriting on Isabelle's terms, see §3.5 below; in this section some of respective preparations are described. In order to work reliably with term rewriting, the respective rule-sets must be confluent and terminating [1], then they are called (canonical) simplifiers. These properties do not go without saying, their establishment is a difficult task for the programmer; this task is not yet supported in the prototype.

The prototype rewrites using theorems only. Axioms which are theorems as well have been already shown in §3.1 on p.96 , we assemble them in a rule-set and apply them in ML as follows:

```
01  val inverse_z = Rls
02      {id       = "inverse_z",
03       rew_ord  = dummy_ord,
04       erls     = Erls,
05       rules    = [Thm ("rule1", @{thm rule1}), Thm ("rule2", @{thm rule1}),
06                   Thm ("rule3", @{thm rule3}), Thm ("rule4", @{thm rule4}),
07                   Thm ("rule5", @{thm rule5}), Thm ("rule6", @{thm rule6})],
08       errpatts = [],
09       scr      = ""}
```

The items, line by line, in the above record have the following purpose:

01..02 the ML-value *inverse_z* stores it's identifier as a string for "reflection" when switching between the language layers of Isabelle/ML (like in the Lucas-Interpreter) and Isabelle/Isar (like in the example program on p.101 on line 12).

03..04 both, (a) the rewrite-order [1] *rew_ord* and (b) the rule-set *erls* are trivial here: (a) the *rules* in 07..12 don't need ordered rewriting and (b) the assumptions of the *rules* need not be evaluated (they just go into the context during rewriting).

05..07 the *rules* are the axioms from p.96; also ML-functions (§3.3) can come into this list as shown in §4.1; so they are distinguished by type-constructors *Thm* and *Calc* respectively; for the purpose of reflection both contain their identifiers.

[12] This way of rigorously deriving all knowledge from first principles is called the LCF-paradigm in TP.
[13] http://isabelle.in.tum.de/dist/library/HOL/HOL-Multivariate_Analysis

08..09 are error-patterns not discussed here and *scr* is prepared to get a program, automatically generated by *ISAC* for producing intermediate rewrites when requested by the user.

3.3 Preparation of ML-Functions

Some functionality required in programming, cannot be accomplished by rewriting. So the prototype has a mechanism to call functions within the rewrite-engine: certain redexes in Isabelle terms call these functions written in SML [11], the implementation *and* meta-language of Isabelle. The programmer has to use this mechanism.

In the running example's program on p.101 the lines 05 and 06 contain such functions; we go into the details with *argument_in X_z;*. This function fetches the argument from a function application: Line 03 in the example calculation on p.104 is created by line 06 of the example program on p.101 where the program's environment assigns the value *X z* to the variable *X_z*; so the function shall extract the argument *z*.

In order to be recognised as a function constant in the program source the constant needs to be declared in a theory, here in *Build_Inverse_Z_Transform.thy*; then it can be parsed in the context *ctxt* of that theory:

```
01   consts
02     argument'_in :: "real => real" ("argument'_in _" 10)
```

The function body below is implemented directly in SML, i.e in an `ML {* *}` block; the function definition provides a unique prefix `eval_` to the function name:

```
01   ML {*
02     fun eval_argument_in _
03       "Build_Inverse_Z_Transform.argument'_in"
04       (t as (Const ("Build_Inverse_Z_Transform.argument'_in", _) $(f $arg))) _ =
05         if is_Free arg (*could be something to be simplified before*)
06         then SOME (term2str t ^"="^ term2str arg, Trueprop $(mk_equality (t, arg)))
07         else NONE
08     | eval_argument_in _ _ _ _ = NONE;
09   *}
```

The function body creates either `NONE` telling the rewrite-engine to search for the next redex, or creates an ad-hoc theorem for rewriting, thus the programmer needs to adopt many technicalities of Isabelle, for instance, the *Trueprop* constant.

This sub-task particularly sheds light on basic issues in the design of a programming language, the integration of differential language layers, the layer of Isabelle/Isar and Isabelle/ML.

Another point of improvement for the prototype is the rewrite-engine: The program on p.101 would not allow to contract the two lines 05 and 06 to

05/06 *(z::real) = argument_in (lhs X_eq) ;*

because nested function calls would require creating redexes inside-out; however, the prototype's rewrite-engine only works top down from the root of a term down to the leaves.

How all these technicalities are to be checked in the prototype is shown in §4.1 below.

3.4 Specification of the Problem

Mechanical treatment requires to translate a textual problem description like in Fig.1 on p.93 into a *formal* specification. The formal specification of the running example could look like is this [14]:

Specification:
input : $\mathit{filterExpression}\ \ X\ z = \frac{3}{z-\frac{1}{4}+-\frac{1}{8}*\frac{1}{z}},\ \mathit{domain}\ \mathbb{R}-\{\frac{1}{2},\frac{-1}{4}\}$
precond : $\frac{3}{z-\frac{1}{4}+-\frac{1}{8}*\frac{1}{z}}\ \ \mathit{continuous_on}\ \mathbb{R}-\{\frac{1}{2},\frac{-1}{4}\}$
output : stepResponse $x[n]$
postcond : TODO

The implementation of the formal specification in the present prototype, still bar-bones without support for authoring, is done like that:

```
00 ML {*
01  store_specification
02    (prepare_specification
03      "pbl_SP_Ztrans_inv"
04      ["Jan Rocnik"]
05      thy
06      ( ["Inverse", "Z_Transform", "SignalProcessing"],
07        [ ("#Given", ["filterExpression X_eq", "domain D"]),
08          ("#Pre"  , ["(rhs X_eq) is_continuous_in D"]),
09          ("#Find" , ["stepResponse n_eq"]),
10          ("#Post" , [" TODO "])])
11        prls
12        NONE
13        [["SignalProcessing","Z_Transform","Inverse"]]);
14 *}
```

Although the above details are partly very technical, we explain them in order to document some intricacies of TP-based programming in the present state of the $\mathcal{ISAC}$ prototype:

01..02 *store_specification:* stores the result of the function *prep_specification* in a global reference *Unsynchronized.ref*, which causes principal conflicts with Isabelle's asynchronous document model [18] and parallel execution [17] and is under reconstruction already.

prep_specification: translates the specification to an internal format which allows efficient processing; see for instance line 07 below.

03..04 are a unique identifier for the specification within $\mathcal{ISAC}$ and the "mathematics author" holding the copy-rights.

05 is the Isabelle *theory* required to parse the specification in lines 07..10.

06 is a key into the tree of all specifications as presented to the user (where some branches might be hidden by the dialogue component).

[14]The "TODO" in the postcondition indicates, that postconditions are not yet handled in the prototype; in particular, the postcondition, i.e. the correctness of the result is not yet automatically proved.

07..10 are the specification with input, pre-condition, output and post-condition respectively; note that the specification contains variables to be instantiated with concrete values for a concrete problem — thus the specification actually captures a class of problems. The post-condition is not handled in the prototype presently.

11 is a rule-set (defined elsewhere) for evaluation of the pre-condition: *(rhs X_eq) is_continuous_in D*, instantiated with the values of a concrete problem, evaluates to true or false — and all evaluation is done by rewriting determined by rule-sets.

12 *NONE:* could be *SOME "solve ..."* for a problem associated to a function from Computer Algebra (like an equation solver) which is not the case here.

13 is a list of methods solving the specified problem (here only one list item) represented analogously to 06.

3.5 Implementation of the Method

A method collects all data required to interpret a certain program by Lucas-Interpretation. The `program` from p.101 of the running example is embedded on the last line in the following method:

```
00 ML {*
01  store_method
02    (prep_method
03      "SP_InverseZTransformation_classic"
04      ["Jan Rocnik"]
05      thy
06      ( ["SignalProcessing", "Z_Transform", "Inverse"],
07        [ ("#Given", ["filterExpression X_eq", "domain D"]),
08          ("#Pre"  , ["(rhs X_eq) is_continuous_in D"]),
09          ("#Find" , ["stepResponse n_eq"]),
10        rew_ord  erls
11        srls  prls  nrls
12        errpats
13        program);
14 *}
```

The above code stores the whole structure analogously to a specification as described above:

01..06 are identical to those for the example specification on p.99.

07..09 show something looking like the specification; this is a *guard*: as long as not all *Given* items are present and the *Pre*-conditions is not true, interpretation of the program is not started.

10..11 all concern rewriting (the respective data are defined elsewhere): *rew_ord* is the rewrite order [1] in case *program* contains a *Rewrite* tactic; and in case the respective rule is a conditional rewrite-rule, *erls* features evaluating the conditions. The rule-sets *srls, prls, nrls* feature evaluating (a) the ML-functions in the program (e.g. *lhs, argument_in, rhs* in the program on p.101, (b) the pre-condition analogous to the specification in line 11 on p.99 and (c) is required for the derivation-machinery checking user-input formulas.

12..13 *errpats* are error-patterns [4] for this method and *program* is the variable holding the example from p.101.

The many rule-sets above cause considerable efforts for the programmers, in particular, because there are no tools for checking essential features of rule-sets.

3.6 Implementation of the TP-based Program

So finally all the prerequisites are described and the final task can be addressed. The program below comes back to the running example: it computes a solution for the problem from Fig.1 on p.93. The reader is reminded of §2.1, the introduction of the programming language:

```
00 ML {*
00 val program =
01 "Program InverseZTransform (X_eq::bool) =
02   let
03     X_eq = Take X_eq ;
04     X_eq = Rewrite prep_for_part_frac X_eq ;
05     (X_z::real) = lhs X_eq ;
06     (z::real) = argument_in X_z;
07     (part_frac::real) = SubProblem
08                    ( Isac, [partial_fraction, rational, simplification], [] )
09                    [ (rhs X_eq)::real, z::real ];
10     (X'_eq::bool) = Take ((X'::real => bool) z = ZZ_1 part_frac) ;
11     X'_eq = ((Rewrite_Set prep_for_inverse_z) @@
12            (Rewrite_Set inverse_z)) X'_eq
13   in
14     X'_eq"
15 *}
```

The program is represented as a string and part of the method in §3.5. As mentioned in §2 the program is purely functional and lacks any input statements and output statements. So the steps of calculation towards a solution (and interactive tutoring in step-wise problem solving) are created as a side-effect by Lucas-Interpretation. The side-effects are triggered by the tactics `Take`, `Rewrite`, `SubProblem` and `Rewrite_Set` in the above lines 03, 04, 07, 10, 11 and 12 respectively. These tactics produce the respective lines in the calculation on p.103.

The above lines 05, 06 do not contain a tactics, so they do not immediately contribute to the calculation on p.103; rather, they compute actual arguments for the `SubProblem` in line 09 [15]. Line 11 contains tactical @@.

The above program also indicates the dominant role of interactive selection of knowledge in the three-dimensional universe of mathematics. The `SubProblem` in the above lines 07..09 is more than a function call with the actual arguments *[(rhs X_eq)::real, z::real]*. The programmer has to determine three items:

1. the theory, in the example *Isac* because different methods can be selected in Pt.3 below, which are defined in different theories with *Isac* collecting them.

[15] The tactics also are break-points for the interpreter, where control is handed over to the user in interactive tutoring.

2. the specification identified by *[partial_fraction, rational, simplification]* in the tree of specifications; this specification is analogous to the specification of the main program described in §3.4; the problem is to find a "partial fraction decomposition" for a univariate rational polynomial.

3. the method in the above example is *[ ]*, i.e. empty, which supposes the interpreter to select one of the methods predefined in the specification, for instance in line 13 in the running example's specification on p.99 [16].

The program code, above presented as a string, is parsed by Isabelle's parser — the program is an Isabelle term. This fact is expected to simplify verification tasks in the future; on the other hand, this fact causes troubles in error detection which are discussed as part of the work-flow in the subsequent section.

4 Work-flow of Programming in the Prototype

The new prover IDE Isabelle/jEdit [19] is a great step forward for interactive theory and proof development. The $\mathcal{ISAC}$-prototype re-uses this IDE as a programming environment. The experiences from this re-use show, that the essential components are available from Isabelle/jEdit. However, additional tools and features are required to achieve acceptable usability.

So notable experiences are reported here, also as a requirement capture for further development of TP-based languages and respective IDEs.

4.1 Preparations and Trials

The many sub-tasks to be accomplished *before* the first line of program code can be written and tested suggest an approach which step-wise establishes the prerequisites. The case study underlying this paper [16] documents the approach in a separate Isabelle theory, *Build_Inverse_Z_Transform.thy* [17]. Part II in the study comprises this theory, LaTeXed from the theory by use of Isabelle's document preparation system. This paper resembles the approach in §3.1 to §3.5, which in actual implementation work involves several iterations.

For instance, only the last step, implementing the program described in §3.5, reveals details required. Let us assume, this is the ML-function *argument_in* required in line 06 of the example program on p.101; how this function needs to be implemented in the prototype has been discussed in §3.3 already.

Now let us assume, that calling this function from the program code does not work; so testing this function is required in order to find out the reason: type errors, a missing entry of the function somewhere or even more nasty technicalities ...

```
01   ML {*
02     val SOME t = parseNEW ctxt "argument_in (X (z::real))";
03     val SOME (str, t') = eval_argument_in ""
```

[16]The freedom (or obligation) for selection carries over to the student in interactive tutoring.

[17]http://www.ist.tugraz.at/projects/isac/publ/Build_Inverse_Z_Transform.thy

```
04        "Build_Inverse_Z_Transform.argument'_in" t 0;
05      term2str t';
06    *}
07    val it = "(argument_in X z) = z": string
```

So, this works: we get an ad-hoc theorem, which used in rewriting would reduce `argument_in X z` to `z`. Now we check this reduction and create a rule-set `rls` for that purpose:

```
01    ML {*
02      val rls = append_rls "test" e_rls
03        [Calc ("Build_Inverse_Z_Transform.argument'_in", eval_argument_in "")]
04      val SOME (t', asm) = rewrite_set_ @{theory} rls t;
05    *}
06    val t' = Free ("z", "RealDef.real"): term
07    val asm = []: term list
```

The resulting term `t'` is `Free ("z", "RealDef.real")`, i.e the variable `z`, so all is perfect. Probably we have forgotten to store this function correctly ? We review the respective `calclist` (again an *Unsynchronized.ref* to be removed in order to adjust to Isabelle/Isar's asynchronous document model):

```
01    calclist:= overwritel (! calclist,
02     [("argument_in",
03      ("Build_Inverse_Z_Transform.argument'_in", eval_argument_in "")),
04        ...
05     ]);
```

The entry is perfect. So what is the reason ? Ah, probably there is something messed up with the many rule-sets in the method, see §3.5 — right, the function `argument_in` is not contained in the respective rule-set *srls* ... this just as an example of the intricacies in debugging a program in the present state of the prototype.

4.2 Implementation in Isabelle/ISAC

Given all the prerequisites from §3.1 to §3.5, usually developed within several iterations, the program can be assembled; on p.101 there is the complete program of the running example.

The completion of this program required efforts for several weeks (after some months of familiarisation with $\mathcal{ISAC}$), caused by the abundance of intricacies indicated above. Also writing the program is not pleasant, given Isabelle/Isar/ without add-ons for programming. Already writing and parsing a few lines of program code is a challenge: the program is an Isabelle term; Isabelle's parser, however, is not meant for huge terms like the program of the running example. So reading out the specific error (usually type errors) from Isabelle's message is difficult.

Testing the evaluation of the program has to rely on very simple tools. Step-wise execution is modeled by a function `me`, short for mathematics-engine [18]:

```
01    ML {* me; *}
02    val it = tac -> ctree * pos -> mout * tac * ctree * pos
```

[18]The interface used by the front-end which created the calculation on p.93 is different from this function

This function takes as arguments a tactic `tac` which determines the next step, the step applied to the interpreter-state `ctree * pos` as last argument taken. The interpreter-state is a pair of a tree `ctree` representing the calculation created (see the example below) and a position `pos` in the calculation. The function delivers a quadruple, beginning with the new formula `mout` and the next tactic followed by the new interpreter-state.

This function allows to stepwise check the program:

```
01   ML {*
02     val fmz =
03       ["filterExpression (X z = 3 / ((z::real) + 1/10 - 1/50*(1/z)))",
04        "stepResponse (x[n::real]::bool)"];
05     val (dI,pI,mI) =
06       ("Isac",
07        ["Inverse", "Z_Transform", "SignalProcessing"],
08        ["SignalProcessing","Z_Transform","Inverse"]);
09     val (mout, tac, ctree, pos)  = CalcTreeTEST [(fmz, (dI, pI, mI))];
10     val (mout, tac, ctree, pos)  = me tac (ctree, pos);
11     val (mout, tac, ctree, pos)  = me tac (ctree, pos);
12     val (mout, tac, ctree, pos)  = me tac (ctree, pos);
13     ...
```

Several dozens of calls for `me` are required to create the lines in the calculation below (including the sub-problems not shown). When an error occurs, the reason might be located many steps before: if evaluation by rewriting, as done by the prototype, fails, then first nothing happens — the effects come later and cause unpleasant checks.

The checks comprise watching the rewrite-engine for many different kinds of rule-sets (see §3.5), the interpreter-state, in particular the environment and the context at the states position — all checks have to rely on simple functions accessing the `ctree`. So getting the calculation below (which resembles the calculation in Fig.1 on p.93) is the result of several weeks of development:

01	• `Problem` *(Inverse_Z_Transform, [Inverse, Z_Transform, SignalProcessing])*	
02	$\vdash\ X z = \frac{3}{z-\frac{1}{4}-\frac{1}{8}\cdot z^{-1}}$	`Take` *X_eq*
03	$X z = \frac{3}{z+\frac{-1}{4}+\frac{-1}{8}\cdot\frac{1}{z}}$	`Rewrite` *prep_for_part_frac X_eq*
04	• `Problem` *[partial_fraction,rational,simplification]*	`SubProblem...`
05	$\vdash\ \frac{3}{z+\frac{-1}{4}+\frac{-1}{8}\cdot\frac{1}{z}} =$	---
06	$\frac{24}{-1+-2\cdot z+8\cdot z^2}$	---
07	• *solve* $(-1+-2\cdot z+8\cdot z^2,\ z\ )$	---
08	$\vdash\ \frac{3}{z+\frac{-1}{4}+\frac{-1}{8}\cdot\frac{1}{z}} = 0$	---
09	$z = \frac{2+\sqrt{-4+8}}{16} \vee z = \frac{2-\sqrt{-4+8}}{16}$	---
10	$z = \frac{1}{2} \vee z =$ ___	---
	___	---
11	$\ldots\ \frac{4}{z-\frac{1}{2}} + \frac{-4}{z-\frac{-1}{4}}$	
12	$X'z = \ddagger^{-1}(\frac{4}{z-\frac{1}{2}} + \frac{-4}{z-\frac{-1}{4}})$	`Take` *((X'::real => bool) z = ZZ_1 part_frac)*
13	$X'z = \ddagger^{-1}(4\cdot\frac{z}{z-\frac{1}{2}} + -4\cdot\frac{z}{z-\frac{-1}{4}})$	`Rewrite_Set` *prep_for_inverse_z X'_eq*
14	$X'z = 4\cdot(\frac{1}{2})^n\cdot u[n] + -4\cdot(\frac{-1}{4})^n\cdot u[n]$	`Rewrite_Set` *inverse_z X'_eq*
15	$\ldots\ X'z = 4\cdot(\frac{1}{2})^n\cdot u[n] + -4\cdot(\frac{-1}{4})^n\cdot u[n]$	`Check_Postcond`

The tactics on the right margin of the above calculation are those in the program on p.101 which create the respective formulas on the left.

4.3 Transfer into the Isabelle/ISAC Knowledge

Finally *Build_Inverse_Z_Transform.thy* has got the job done and the knowledge accumulated in it can be distributed to appropriate theories: the program to *Inverse_Z_Transform.thy*, the sub-problem accomplishing the partial fraction decomposition to *Partial_Fractions.thy*. Since there are hacks into Isabelle's internals, this kind of distribution is not trivial. For instance, the function `argument_in` in §3.3 explicitly contains a string with the theory it has been defined in, so this string needs to be updated from `Build_Inverse_Z_Transform` to `Atools` if that function is transferred to theory *Atools.thy*.

In order to obtain the functionality presented in Fig.1 on p.93 data must be exported from SML-structures to XML. This process is also rather bare-bones without authoring tools and is described in detail in the ISAC wiki [19].

5 Summary and Conclusions

A brief re-introduction of the novel kind of programming language by example of the ISAC-prototype makes the paper self-contained. The main section describes all the main concepts involved in TP-based programming and all the sub-tasks concerning respective implementation in the ISAC prototype: mechanisation of mathematics and domain modeling, implementation of term rewriting systems for the rewriting-engine, formal (implicit) specification of the problem to be (explicitly) described by the program, implementation of the many components required for Lucas-Interpretation and finally implementation of the program itself.

The many concepts and sub-tasks involved in programming require a comprehensive work-flow; first experiences with the work-flow as supported by the present prototype are described as well: Isabelle + Isar + jEdit provide appropriate components for establishing an efficient development environment integrating computation and deduction. However, the present state of the prototype is far off a state appropriate for wide-spread use: the prototype of the program language lacks expressiveness and elegance, the prototype of the development environment is hardly usable: error messages still address the developer of the prototype's interpreter rather than the application programmer, implementation of the many settings for the Lucas-Interpreter is cumbersome.

5.1 Conclusions for Future Development

From the above mentioned experiences a successful proof of concept can be concluded: programming arbitrary problems from engineering sciences is possible, in principle even in the prototype. Furthermore the experiences allow to conclude detailed requirements for further development:

1. Clarify underlying logics such that programming is smoothly integrated with verification of the program; the post-condition should be proved more or less automatically, otherwise working engineers would not encounter such programming.

[19] http://www.ist.tugraz.at/isac/index.php/Generate_representations_for_ISAC_Knowledge

2. Combine the prototype's programming language with Isabelle's powerful function package and probably with more of SML's pattern-matching features; include parallel execution on multi-core machines into the language design.

3. Extend the prototype's Lucas-Interpreter such that it also handles functions defined by use of Isabelle's functions package; and generalize Isabelle's code generator such that efficient code for the whole definition of the programming language can be generated (for multi-core machines).

4. Develop an efficient development environment with integration of programming and proving, with management not only of Isabelle theories, but also of large collections of specifications and of programs.

5. Extend Isabelle's computational features in direction of *verfied* Computer Algebra: simplification extended by algorithms beyond rewriting (cancellation of multivariate rationals, factorisation, partial fraction decomposition, etc), equation solving , integration, etc.

Provided successful accomplishment, these points provide distinguished components for virtual workbenches appealing to practitioners of engineering in the near future.

5.2 Preview to Development of Course Material

Interactive course material, as addressed by the title, can comprise step-wise problem solving created as a side-effect of a TP-based program: The introduction §1 briefly shows that Lucas-Interpretation not only provides an interactive programming environment, Lucas-Interpretation also can provide TP-based services for a flexible dialogue component with adaptive user guidance for independent and inquiry-based learning.

However, the $\mathcal{ISAC}$ prototype is not ready for use in field-tests, not only due to the above five requirements not sufficiently accomplished, but also due to usability of the fron-end, in particular the lack of an editor for formulas in 2-dimension representation.

Nevertheless, the experiences from the case study described in this paper, allow to give a preview to the development of course material, if based on Lucas-Interpretation:

Development of material from scratch is too much effort just for e-learning; this has become clear with the case study. For getting support for stepwise problem solving just in *one* example class, the one presented in this paper, involved the following tasks:

- Adapt the equation solver; since that was too laborous, the program has been adapted in an unelegant way.

- Implement an algorithms for partial fraction decomposition, which is considered a standard normal form in Computer Algebra.

- Implement a specification for partial fraction decomposition and locate it appropriately in the hierarchy of specification.

- Declare definitions and theorems within the theory of $\mathcal{Z}$-transform, and prove the theorems (which was not done in the case study).

On the other hand, for the one the class of problems implemented, adding an arbitrary number of examples within this class requires a few minutes [20] and the support for individual stepwise problem solving comes for free.

E-learning benefits from Formal Domain Engineering which can be expected for various domains in the near future. In order to cope with increasing complexity in domain of technology, specific domain knowledge is beeing mechanised, not only for software technology [21] but also for other engineering domains [5, 8, 3]. This fairly new part of engineering sciences is called "domain engineering" in [2].

Given this kind of mechanised knowledge including mathematical theories, domain specific definitions, specifications and algorithms, theorems and proofs, then e-learning with support for individual stepwise problem solving will not be much ado anymore; then e-learning media in technology education can be derived from this knowledge with reasonable effort.

Development differentiates into tasks more separated than without Lucas-Interpretation and more challenginging in specific expertise. These are the kinds of experts expected to cooperate in development of

- "Domain engineers", who accomplish fairly novel tasks described in this paper.
- Course designers, who provide the instructional design according to curricula, together with usability experts and media designers, are indispensable in production of e-learning media at the state-of-the art.
- "Dialog designers", whose part of development is clearly separated from the part of domain engineers as a consequence of Lucas-Interpretation: TP-based programs are functional, as mentioned, and are only concerned with describing mathematics — and not at all concerned with interaction, psychology, learning theory and the like, because there are no in/output statements. Dialog designers can expect a high-level rule-based language [4] for describing their part.

For this decade there seems to be a window of opportunity opening from one side inreasing demand for formal domain engineering and from the other side from TP more and more gaining industrial relevance. Within this window, development of TP-based educational software can take benefit from the fact, that the TPs leading in Europe, Coq [6] and Isabelle are still open source together with the major part of mechanised knowledge.

References

[1] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[2] Dines Bjørner. *Software Engineering*, volume 3 of *Texts in Theoretical Computer Science*. Springer, Berlin, Heidelberg, 2006.

[20] As shown in Fig.1, an example is called from an HTML-file by an URL, which addresses an XML-structure holding the respective data as shown on p.104.

[21] For instance, the Archive of Formal Proofs http://afp.sourceforge.net/

[3] Dines Bjørner. *Domain Engineering. Technology Management, Research and Engineering*, volume 4 of *COE Research Monograph Series*. JAIST Press, Nomi, Japan, Feb 2009.

[4] Gabriella Daróczy and Walther Neuper. Error-patterns within "next-step-guidance" in tp-based educational systems. In *unknown*. The Electronic Journal of Mathematics and Technology, 2013. to appear in this publication.

[5] Babak Dehbonei and Fernando Mejia. Formal methods in the railways signalling industry. In M. Bertran M. Naftalin, T. Denvir, editor, *FME'94:Industrial Benefit of Formal Methods*, pages 26–34. Springer-Verlag, October 1994.

[6] Coq development team. Coq 8.3 reference manual. http://coq.inria.fr/reman, 2010. INRIA.

[7] Florian Haftmann, Cezary Kaliszyk, and Walther Neuper. CTP-based programming languages ? considerations about an experimental design. *ACM Communications in Computer Algebra*, 44(1/2):27–41, March/June 2010.

[8] Kirsten Mark Hansen. Validation of a railway interlocking model. In M. Bertran M. Naftalin, T. Denvir, editor, *FME'94: Industrial Benefit of Formal Methods*, pages 582–601. Springer-Verlag, October 1994.

[9] J. Roger Hindley. *Basic Simple Type Theory*. Number 42 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1997.

[10] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science (JCSS)*, 0(17):348–374, 1978.

[11] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, London, 1997.

[12] Walther Neuper. Automated generation of user guidance by combining computation and deduction. In Pedro Quaresma, editor, *THedu'11: CTP-compontents for educational software*. EPTCS, 2012. To appear.

[13] Tobias Nipkow. Programming and proving in Isabelle/HOL. contained in the Isabelle distribution, May 22 2012.

[14] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[15] A.V. Oppenheim and R.W. Schafer. *Discrete-time signal processing*. Prentice-Hall signal processing series. Prentice Hall, 2010.

[16] Jan Rocnik. Interactive course material for signal processing based on isabelle/$\mathcal{ISAC}$. Bakkalaureate Thesis, 2012. IST, SPSC, Graz University of Technology, http://www.ist.tugraz.at/projects/isac/publ/jrocnik_bakk.pdf.

[17] Makarius Wenzel. Parallel proof checking in Isabelle/Isar. In Dos Reis and L. Théry, editors, *ACM SIGSAM 2009 International Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS)*, Munich, August 2009. ACM Digital library.

[18] Makarius Wenzel. Isabelle as document-oriented proof assistant. In *Proceedings of the 18th Calculemus and 10th international conference on Intelligent computer mathematics*, MKM'11, pages 244–259, Berlin, Heidelberg, 2011. Springer-Verlag.

[19] Makarius Wenzel. Isabelle/jEdit a prover ide within the PIDE framework. In J. Jeuring et al., editors, *Conference on Intelligent Computer Mathematics (CICM 2012)*, number 7362 in LNAI. Springer, 2012.

[20] Wikipedia. Table of common z-transform pairs, 2012. [Online; accessed 31-Oct-2012].