CTP-based programming languages ?

Considerations about an experimental design

Florian Haftmann Cezary Kaliszyk
TU München - Software & Systems Engineering
{haftmann,kaliszyk}@informatik.tu-muenchen.de

Walther Neuper
TU Graz - Institute for Software Technology
neuper@ist.tugraz.at

Abstract

This paper discusses plans for joint work in order to gain early feedback from the community.

Three lines of work pursued independently so far shall be joined: (1) narrowing the gap between declarative program specification and program generation already working in Isabelle, (2) reusing work, which embedded an input-response-loop resembling Computer Algebra Systems (CAS) into HOL Light, and (3) reconstructing an experimental language for applied mathematics by exploiting established as well as emerging features of Isabelle/Isar.

These plans have to be seen as part of a variety of highly active research areas — on "integration of the deduction and the computational power" of Computer Theorem Proving (CTP) and CAS respectively (Calculemus), on "innovative theoretical and technological solutions for content-based systems" (MKM), on "Programming Languages for Mechanized Mathematics Systems" (PLMMS), just to cite from some related interest groups.

Facing the abundant variety of approaches, of intermediate results and of ongoing developments, and taking under consideration the many difficulties in integrating such approaches, we pursue pragmatic goals:

Design a component indispensable for working engineers, a programming language for engineering applications. Use Isabelle for an experimental embedding of the language, which is useful at least in engineering education as soon as possible.

Categories and Subject Descriptors G [*4*]: Verification

General Terms Languages, Verification, Reliability, Design

Keywords computer theorem proving, programming language, computer algebra, integration, interactive specification, real numbers

1. Introduction

We take the term "computer theorem proving" as introduced in [13] comprising both, automated and interactive theorem proving. Both aspects of theorem proving are relevant for our approach.The abbreviation "CTP" for "computer theorem proving" shall indicate analogies to "CAS", a widely used abbreviation for "computer algebra (systems)" [1].

Our motivation is simple: transfer the success from programming languages based on Computer Algebra Systems (called CAS-based languages) to the domain of Computer Theorem Proving (CTP), overcome deficiencies found basically and essentially in in all CAS and improve safety and reliability of software by use of concepts and technologies from CTP. What is called a success of CAS-based languages is the fact [2] that a major and quickly increasing part of software for electrical engineering, for structural engineering and the like is built using such languages.

Our approach is pragmatic: we start from three lines of work pursued independently so far and discuss a merge of these. The three lines are: (1) narrowing the gap between declarative program specification and program generation already working in Isabelle (Sect.2.1), (2) reusing work on a CAS-like input-response-loop embedded into HOL Light (Sect.2.2), and (3) reconstructing an experimental language for applied mathematics (Sect.2.3) in the *ISAC*-prototype by exploiting the emerging features of Isabelle/Isar. And we confine our approach to the Isabelle framework.

Turning the idea for merging into a concrete research plan faces considerable issues: the idea is intimately interrelated with a variety of highly active research — with the "integration of the deduction and the computational power" of CTP and CAS respectively, with "innovative theoretical and technological solutions for content-based systems", with "Programming Languages for Mechanized Mathematics Systems" (just to cite from some interest groups).

So, this paper does not give a comprehensive survey; only the most important concepts involved in the merge are addressed. And the paper goes into technical details only if it seems necessary for comprehension. Furthermore we do not feel ready to give a concise specification of the language envisaged. Rather, we pursue the above motivation and describe a future workplace of an engineer who constructs software of the kind presently covered by CAS-based languages in Sect.3. Continuing the pragmatic approach requires to mention practical aspects like the workflow at the future workplace, which necessarily remains speculative.

The paper is organized as follows: Sect.2 describes each of the three approaches and work already accomplished. Sect.3 tries an outlook to an engineers electronic workbench in the future in order to present, how the merge of the three approaches might come to bear. Sect.4 discusses novel research set on stage by the merge of the three approaches. Sect.5 relates the language under consid-

PLMMS-2010 8^{th} July 2010, Paris.

[1] Both terms, CTP and CAS, will be used to designate two different things: the respective software products and the respective underlying concepts and technologies.

[2] The most successful software house building engineering software on demand is Wolfram Research, see http://www.wolfram.com/solutions.

eration with existing languages of proof assistants and mentions related work on the integration of reasoning and computation and on improving reliability of calculation with reals. Sect.6 gives a summary and an outlook to continuation of the presented work.

2. Three approaches and straightforward merges

The three lines of work have been pursued independently so far. They are presented with respect to possibilities for mutual merges together with advantages, which seem straight forward and which do not necessarily require remarkable R&D.

2.1 Integration of deductive and algorithmic components

Integration of deduction and calculation is being promoted from several sides (see Sect.5), from the side of programming languages, from symbolic computation and from the side of CTP. This paper takes the latter approach, based on Isabelle.

Isabelle [32] has been recognized as a logical framework [36] for a long time. With programming in mind one recognizes, that Isabelle provides generic numerals [35] and also floating point numbers [14]. Presently Isabelle/Isar's logical infrastructure seems to develop towards a "logical operating system" [39] for various applications. The CTP-based language under consideration is one of such applications.

Bringing together specification and implementation. Recently the well-known relationship between higher-order logic and functional programming has been exploited [11]. The central idea is that a suitable set of equational theorems of the form $f\,\overline{x} = t$ is interpreted as a functional program which can be translated to suitable languages like Haskell or ML. Given a function symbol f with a specification Pf, the implementation of f is constructed by deriving suitable equations $f\,\overline{x} = t$ describing f from Pf. Figuratively spoken, an implementation is a coagulation of equational theorems from the logic.

This approach allows to express refinement directly within the logic: e.g., imagine a function symbol $bsort :: \alpha\ list \Rightarrow \alpha\ list$ with corresponding equational theorems implementing bubble sort and a function symbol $msort :: \alpha\ list \Rightarrow \alpha\ list$ with corresponding equational theorems implementing merge sort. Then $bsort = msort$ can be proven, which allows to replace $bsort$ by $msort$ in implementations. It would even be possible to specify sorting involving a choice operator, e.g. $sort\ xs = THE\ ys.\ multiset\ xs = multiset\ ys \wedge sorted\ ys$, where $multiset$ turns a list into the corresponding multiset. From this definition $sort = msort$ (and $sort = bsort$) can be proven, hence the abstract $sort$ can be implemented by a concrete algorithm.

There are examples of engineering problems (for instance the one in Sect.2.3 on p.3) on which the above method seems to be applicable. In software development automated or semi-automated code generation is an appealing offer; in the application domain under consideration the major benefit might be in support for handling types in highly complex mathematical structures (rather than automated coding of complicated algorithms).

2.2 Approach towards CAS-like functionality

Before considering "CAS-like functionality" for CTP-based programming languages, we need to mention the deficiencies of mainstream CAS in order to be clear, what shall be improved when advancing to a CTP-based language.

Mainstream CAS, for instance Mathematica and Maple, are very weakly founded ([13] even calls them "ill-defined"). There are various reasons for the mistakes found in mainstream CAS systems: assumptions can be lost, types of expressions can be forgotten or algorithms of the system themselves may contain implementation errors [18]. Simple mistakes have been found and fixed over the years. However mistakes made when performing more complicated computations are still found. So improvements are urgently required, and our work already tackled some of them.

We have built a prototype CAS-like input-response-loop inside HOL Light, with the user interface designed close to the interfaces of popular computer algebra systems. In Figure 1 we show examples of simplifications that it can perform automatically: basic vector arithmetic, symbolic computation, numeric approximations and basic handling of assumptions.

```
 In1 := vector [&2; &2] - vector [&1; &0] + vec 1
Out1 := vector [&2; &3]
 In2 := diff (diff (λx. &3 * sin (&2 * x) +
        &7 + exp (exp x)))
Out2 := λx. exp x pow 2 * exp (exp x) +
        exp x * exp (exp x) + -- &12 * sin (&2 * x)
 In3 := N (exp (&1)) 10
Out3 := #2.7182818284 + ... (exp (&1)) 10 F
 In4 := x + &1 - x / &1 + &7 * (y + x) pow 2
Out4 := &7 * x pow 2 + &14 * x * y + &7 * y pow 2 + &1
 In5 := sum (0,5) (λx. &x * &x)
Out5 := &30
 In6 := sqrt (x * x) assuming x > &1
Out6 := x
```

Figure 1. Example interaction with the prototype CAS-like input-response loop. For the user input given in the `In` lines, the system produces the output in `Out` lines together with HOL Light theorems that state the equality between the input and the output.

By this prototype we have demonstrated in that it is possible to build a computer algebra system in a proof assistant [18]. Such architecture guarantees that the system will make no mistakes. All expressions in the system have precise semantics and the proof assistant checks the correctness of all simplifications according to this semantics. The envisaged language shall be based on such a system; also CAS-like interaction as shown in Figure 1 shall be available for the user.

There are issues, which are still open in the prototype. First the syntax includes many coercions. In the presented example the symbol `&` marks coercions to real numbers. HOL Light's type prioritization is used to decide to which type the variables and operators (plus, ...) should belong; but the overloading there is not strong enough, although we are able to show quite some type information. In Sect.4.1 we describe how this can be solved using Isabelle's proper parsing and syntax translation mechanisms.

Partiality: The prototype provides a simple mechanism for handling assumptions. In the example we have seen the construct `assuming` that allows the system to simplify $\sqrt{x^2}$, by deriving that x is non-negative. And extension of this for handling partiality with functions is presented in [17]. This lets the prototype compute the derivative of $\frac{1}{x}$ knowing that $x \neq 0$.

The assumptions are stored in two lists. A list that stores assumptions about variable types and a list of properties. The first of those lists is given to the parser, while the second one is used to fulfill the assumptions of conditional rewrite rules. This means that the approach is built on top of the proof assistant. However, the integration is not perfect, since the decision procedures present in the proof assistant cannot make use of the assumptions. In Sect.4.1 we describe how Isabelle contexts can be used to combine the assumptions with the whole of the proof assistant.

The mechanism described above seems appropriate also for programs executing some application of mathematics. For this purpose this mechanism has to be integrated with mechanisms from Isabelle, see Sect.4.1. There are features indispensable in CAS like numerals discussed as well.

2.3 A functional language with guards

The third approach contributing to the envisaged joint work dates back to a language implementation [30] [3] for educational purposes in the *ISAC*-prototype [4]. Over the years this language revealed potential for generalization. The following features of the language are relevant for the envisaged joint work:

1. The language is based on Isabelle/HOL with *IF*, *LET, IN* and functions on lists like *HD, TL, LAST* etc [5]. Straight forward extensions provide for access to Isabelle's matching and rewriting. The language is purely functional (without input and output statements) and inherit major features from SML [29]: strict evaluation, high-order functions, abstract datatypes, compile-time type checking and type inference; it might be statically nested with other functions. In comparison to SML it comes without a module system, without pattern matching for datatypes and without exception handling.
2. Functions (and also functions nested within other functions) are guarded with a formal specification, i.e with typed input- and output-items, precondition and postcondition. If the precondition, instantiated with the input-items' values, holds, then the guard allows to start program execution. Patterns of specifications are given in a tree, and traversing the tree while matching the patterns with the input-items and evaluating the precondition allows to determine the most appropriate specification. This kind of "problem refinement" has successfully been used to model a simple equation solver [21] and to make other CAS-like functionality transparent.
3. The language comes along with an interpreter, which operates on the parse tree created by Isabelle's parser. The initial purpose of the interpreter was to provide user guidance for the tutoring system: switched into a single-stepping mode the interpreter hands over control to a dialog module at certain steps and the dialog handles interaction with the learner.

   With respect to the joint work envisaged here, the interpreter is relevant for another reason: it works on the constructs of the programming language on the same level of abstraction, as Isabelle's prover works on constructs of the specification, i.e on terms and predicates.

These three points shall contribute to the design of the CTP-based programming language. The above Pt.2 will be extended to an essential feature of future workplaces for engineering in Sect.3. Further details of the language are up to discussion and to re-design. Let us look at some of the details.

An example program written in the present language gives an algorithm solving a problem in structural engineering, which will serve again as an example in Sect.3.

```
01 Script bendingLine
02 (l_::real) (q_::real) (v_::real) (b_::real=>real) (rb_::bool list) =
03  (LET
04   (funs_:: bool list) =
05         (SubProblem (Bendingline,[bendingline,integrate],
06                      [bendingline,integrate])
07                      [real_ q_, real_real_ b_, real_ v_]);
08   (equs_::bool list) =
09         (SubProblem (Bendingline,[bendingline,setConstraints],
10                      [bendingline,setConstraints])
11                      [bools_ funs_, bools_ rb_, real_ l_]);
12   (sols_::bool list) =
13         (SubProblem (Real,[equation,system,linear],[])
14                      [bools_ equs_, reals_ [c,c_2,c_3,c_4]]);
15   B_ = Take (LAST funs_);
16   B_ = ((Substitute sols_) @@
17          (Rewrite_Set_Inst [(bdv, v_)] make_ratpoly_in)) B_
18  IN B_)
```

The identifiers with ending underscores avoid type clashes with identifiers in the object language of formulas; the function constants *real_, bools_, reals_* bring the arguments' types into line for the list of arguments.

Most noticeable are the bulky function calls designated with *SubProblem*. These relate the descriptive and algorithmic aspect: the list of the functions' arguments is preceded by a triple: #1 points to the respective theory (*Bendingline* or *Real*), #2 points to the (pattern of, see Pt.2 above) specification and #3 points to the algorithm refining the specification. Specifications and algorithms are addressed by paths into a tree.

Now let us look at the program code line by line:
01..02 is the program header with the arguments. Note that the output-item $b\_$ is also an argument, because all identifiers have been determined in the specification preceding the start of the program.
03 *LET* is as defined in Isabelle/HOL, which requires semicolons as delimiters except after the last line before *IN* in 18.
04..07 designates a subproblem which #1 takes vocabulary from theory *Bendingline*, #2 relies on a specification addressed by *[bendingline,integrate]* and #3 a method addressed by the same path (into the other tree of methods), because apparently there is only one method necessary for this problem.
08..11 designates a similar subproblem (for the meaning of the program see Sect.3 p.4).
12..14 calls a CAS function solving a system of equations. The method in #3 is empty, since the system is considered smart enough to find the appropriate algorithm in this case.
15 takes the last element of the list *funs_*; *LAST* is renamed as part of the programming language in order to allow programs to operate on formulas with *last* without confusing the interpreter.
16..17 shows forward chaining @@ of CAS functions, substitution and rewriting with a term rewriting system named *make_ratpoly_in*; rewriting is optimized for this univariate function with bound variable *v_*.

There are immediately effective advantages from merging with the other lines of work:

Make all deductive components available within the programming language. So far, only matching and simplification can be accessed by the language primitives. In addition, decision procedures are useful for rewriting with conditional rewrite rules, provers could provide more powerful assistance in proving the preconditions of guards mentioned in Pt.2.

Further use of provers will be discussed in Sect.4.3.

Reuse Isar/ML/Scala integration: Presently Isabelle/Isar's extra-logical infrastructure is evolving [39] towards open interoperability with front-ends. Some of the GUI front-ends under consideration for the Isar proof language are also appropriate as front-ends for programming and debugging. Also the communication between GUIs and the back-ends is similar in proof development and in program development. Thus Isabelle/Isar's Scala API will serve both, proofs and programs, as shown in Figure 2.

The uniform architecture provides optimal prerequisites for an integrated workflow constructing algorithms and proofs in parallel. Since Scala runs on the JVM platform, the system is open for widespread use.

[3] The essential design ideas were provided by Peter Lucas, one of the pioneers in compiler construction and formal methods [23–25].

[4] http://www.ist.tugraz.at/projects/isac

[5] These functions are renamed with uppercase letters in order to distinguish them from the object language, these functions are operating on as part of the programming language. This avoids confusing the interpreter.

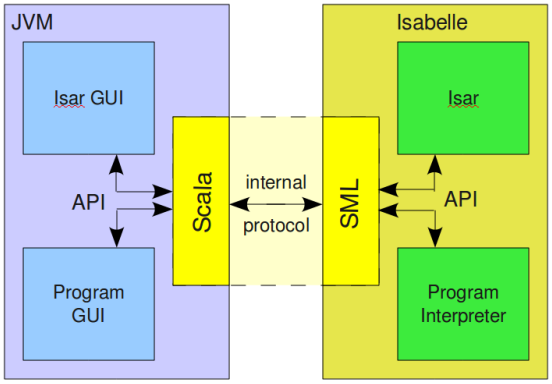

Figure 2. Reuse of Isabelle/Isar's Scala API.

3. An engineer's future workplace

All together, the work described above covers just a tiny part of the research contributing to an engineer's future workplace. So many questions are open, that already strategic aspects are being addressed, for instance in [1].

However, we believe that a merge of the three lines of work will already establish notable advances without further research (respective questions open for research are discussed in Sect.4). With the motivation in mind to replace CAS-based programming with CTP-based programming, it seems appropriate to discuss these advances with respect to a practice oriented setting. We are keen enough to describe a future workplace of an engineer who programs some application for electrical engineering, mechanical engineering or the like.

A standard problem from a textbook on structural engineering might serve as an example for discussing details at the engineer's future workplace:

Determine the bending line of a beam of length L, which consists of homogeneous material, which is clamped on one side and which is under constant line load q_0; see Figure 3.
Hint: Use the constraints on the shear force $V(0) = q_0 \cdot L$, on the bending moment $M_b(L) = 0$ and on the expected bending line $y(0) = 0$ and $y'(0) = 0$.

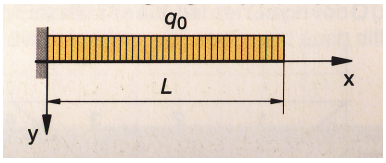

Figure 3. A balcony under load.

A specification of the problem in the traditional form with the input-items *in*, the precondition *pre*, the output-item(s) *out* and the postcondition *post* finally might look like this in notation used by engineers:

$$\begin{array}{rcl} in & : & function\ q_0, length\ L \\ pre & : & q_0\ is\_integrable\_in\ x\ \wedge\ L > 0 \\ out & : & function\ y(x) \\ post & : & y(0) = 0 \wedge y'(0) = 0\ \wedge\ V(0) = q_0.L\ \wedge\ M_b(L) = 0 \end{array}$$

where V and M_b are constant function symbols in this theory of "bending lines". *function* and *length* are functions fixing the arguments' types; q_0 is a constant function with type $\mathcal{R} \to \mathcal{R}$.

A program refining the specification has been listed already in Sect.2.3. The program's algorithm is a straightforward refinement of the postcondition: the theory of bending lines is simple in this case, saying that the bending line's $y(x)$ fourth derivative $y(x)^{\prime v}$ is (almost) the load function $q_0(x)$, $y(x)^{\prime v} = c \cdot q_0(x)$. Thus the first subproblem (program lines 04..07) integrates the load function ($q_0(x)$ with identifier $q\_$) four times and returns for functions *funs_* with four respective constants of integration, c, c_1, c_2, c_3.

The second subproblem (lines 08..11) substitutes the constraints, given by the program's argument $rb\_$ into the four functions *funs_* and returns four equations *equs_* containing c, c_1, c_2, c_3.

The third subproblem is a classical CAS problem solving the (linear, uniquely solvable) system of four equations in c, c_1, c_2, c_3 yielding the normal form of four equations in *sols_*.

Finally there are elementary CAS tasks, substituting the solutions $sols\_$ into the last function from the list of four function *funs_*, which is the bending line $y(x)$, and simplify this function.

Before entering description and discussion of the workplace we note, that the description necessarily is speculative. In particular one prerequisite for efficient work at such a workplace, domain specific knowledge, is not yet present (and a discussion how to create such knowledge is out of scope of this paper).

Domain specific knowledge is supposed to undergo formalization and mechanisation more and more, in order to improve professionality of software development [4].

The differentiation of such knowledge is already prepared by the distinction between the two established notions, quality of design and quality of conformance. The former concerns the relation between "reality" and an abstract model, the latter the relation between the abstract model and an implementation. Both relations are increasingly investigated and tackled with mathematical methods. One of the strongest method is proof; thus one might predict increasing importance of mathematical proof in this part software technology.

Domain engineering is expected to create and mechanise domain specific knowledge [4], i.e. appropriate notions (for instance *beam, bending line* in the above example), concise abstractions like domain specific predicates (for instance *homogeneous, clamped* in the above example) and models, which will develop from kinds of abstract datatypes to algebraic structures with specific axioms, definitions, lemmas and theorems. So one might predict increasing importance of mathematical proof also in this part software technology.

Domain specific knowledge is already being at the beginning of mechanization in software engineering [6]. But the mechanization of knowledge in other domains like electrical, mechanical or structural engineering has not yet begun. [5, 7, 12] describe some initial attempts.

In spite of the non-existence of such knowledge we are keen enough to predict some details, because these are already anticipated in the language as described in Sect.2.3:

Three aspects of knowledge are expected to be distinguished in the sequel; this distinction has not yet been established. For instance, Isabelle's mechanisms for handling knowledge [38] are highly elaborated and show several levels of granularity: theories,

[6] In Isabelle's "Archive of Formal Proofs" there are collections like on "Computer Science >> Security" with growing lists like "SIFPL, VolpanoSmith, HotelKeyCards, RSAPSS, InformationFlowSlicing", see http://afp.sourceforge.net/topics.shtml

local theories, locales, theory contexts, generic contexts, local contexts — all allow SML code inline to the code defining logical entities and antiquotations within the inline SML code referring to the logical entities, and all packed into theories without explicit structuring so far.

There are initial ideas of how to structure knowledge [9]; here we just want to distinguish three general aspects of knowledge, the deductive, the applicative and the algorithmic aspect of knowledge:

1. **Deductive knowledge** is that kind of knowledge which traditionally is represented in so-called **theories**, for instance Isabelle's theories.
2. **Applicative knowledge** is represented by specifications in the simplest case; for each method from Pt.3 below there should be a specification, but there might be more than one method refining one and the same specification. We shall call this kind of knowledge **problems**.
3. **Algorithmic knowledge** comprises specific algorithms solving problems specified in Pt.2 above; we shall call this kind of knowledge **methods**. This knowledge represents the bridge to algorithmic mathematics and classical programming.

Many questions about these three points are open, for instance, how to separate them, where to place respective parts of code. Also the structure seems only settled for theories (an directed acyclic graph), while the structure assembling methods and problems is open.

In order to make things more precise in the sequel, we simply assume methods and problems assembled in two separated trees.

Discussion of a futuristic workflow is speculative, as already mentioned. With respect to the above paragraph we expect the engineer being an software expert in the domain of structural engineering. And we can assume, that the problem of calculating the bending line is part of a larger problem, probably requiring the calculation of many bending lines for many structural components. What might this engineer do? We assume an iterative approach towards a final result using these steps:

Looking up the knowledge available at his or her electronic workplace will be the first step. The theories will be scanned, probably because some details have been forgotten (e.g. where the matter constants come in: $y''(x) = -\frac{M_b(x)}{EI}$ for the bending line $y(x)$), or because decisions are required whether to work with univariate or multivariate functions, whether to take a continuous or an approximating discrete model, etc.

The most interesting question will be, if there is already a solution for the given problem in the system; this requires searching *problems* (specifications) and algorithms (*methods*). If the input-items are already determined, a mechanical problem refinement is possible as described in Sect.2.3.Pt.2; if also a postcondition has been formulated, the search will be even more precise.

Let us assume, that there is still no appropriate solution in the system, and that the engineer decides to go towards the solution already presented in Sect.2.3.

Interactively assembling the parts of the code required will be the next step. First the problem will be specified — not for academic reasons, but because the solution for the problem will serve within a larger system, which requires specification for components. The formulation of the specification is supported by domain specific predicates (which has not yet been developed in our example).

According to the assumption, that no ready-made solution is available, we also may assume, that the first two subproblems (lines 04..07 and 08..11) are not available. So the engineer has to develop respective specifications and methods; probably this would be done iteratively; first the specification, then the algorithm; quick testing of preliminary code sequences in an input-response-loop is helpful, as already described in Sect.2.2.

For other parts solutions are ready-made, in our example this will be solving the system of equations (lines 12..14). In general, an appropriate method has to be selected from a variety of algorithms for systems of linear equations, depending on the system's dimension etc.

Proving correctness of the program is not an additional exercise during or after development, rather it is enforced by the system — which poses a considerable challenge to the usability of that system: restrictive discipline must get counterbalance by noticeable advantages!

One advantage is mechanized code generation as mentioned in Sect.2.1. Another advantage is flexible support in assembling software components and decisive support in proving correctness of the development. Usually the number of proof obligations increases dramatically when tackling more complex problems. Even in our simple example lots of effort is required to prove that the program's arguments and the results of subproblems fulfill the preconditions for a subsequent subproblem.

Further advantages will be proper handling of approximations by reals, i.e. by floating point numbers; Sect.2.2 already mentioned this issue, Sect.4.2 will discuss further ideas.

4. Research questions raised from merging

Recalling what has been said in the introduction, we do not claim to address the most essential issues in the design of CTP-based programming languages. We rather discuss in the sequel, how merging the three lines of work described in Sect.2 leads to novel chances for advancements. However, we claim that we address such advancements which are well into reach of R&D. And the aim remains a pragmatic one: a language which balances demands and benefits such that engineers like to use it for their programming tasks, preliminarily at least in engineering education.

4.1 Traceability of mathematics applications

Engineers are responsible for what they deliver; this is an issue with respect to the growing complexity of the tools engineers use at their workplace. With respect to software tools specifications improve manageability, as already discussed. Another means is, to make software transparent such that the engineer has the chance to trace what is going on internally; this is an additional support for taking over responsibility in engineering.

Traceability poses research issues, some of which are already in reach to be accomplished by available concepts and technologies:

Manage partiality conditions such that an engineer can trace down where singularities come from, where solutions of equations might have disappeared etc.

Sect.2.2 presented what already has been achieved for CAS functions if evaluated in an input-response-loop. Embedding CAS functions in arbitrary complex software reinforces the issue for the design of respective programming languages.

Looking at the concept of "context" and the respective mechanisms, Isabelle/Isar seems to provide much of the logical infrastructure necessary to cope with handling partiality conditions dynamically during program execution: The context has to be initialized with the input-items and the preconditions of the specification at beginning the execution of a function; in case a partial function causes some additional assumption, this assumption can be added to the context. The dynamic scoping of contexts can follow the block structure of nested function calls.

Detailed design of the language has to consider means to make the dynamic expansion of contexts traceable in engineering prac-

tice. Nearby possibilities for such means are extensions of debuggers, another means could be dedicated tracing facilities.

Interactive decomposition of terms is another approach to improve traceability of programs which apply mathematics; this approach exploits the power of type systems.

Many proof assistants already include parsing mechanisms that allow type inference and disambiguation of overloaded notations. This mechanism can not only be adapted to allow the user to type mathematical formulas in a CAS way, but also to trace the actual definitions of symbols in an expression. In a typical interaction with a proof assistant, definitions and properties have to be given explicit types unless the types can be automatically inferred. This can be done in many cases. For example given the expression $2 * \pi * r$, a proof assistant knows that π is a real number. Then type inference and overloading can infer that r is a real number free variable, that 2 is a real constant and $*$ is the real number multiplication. Also proof assistants have mechanisms for finding out definitions of the given symbols. In the example above it is possible to find the associated definition of π and real number multiplication.

So, the principle seems clear: a typed term contains all information necessary to ensure correct manipulations in programs, and this information shall be made available to the user — in principle. However, the question is open, how to make this information accessible in interaction on formulas, such that it is really useful in engineering practice.

4.2 Two steps in a never ending story

Two of the most challenging problems in making mathematics software safe and reliable are multivaluedness of CAS-functions and the handling of floating-point number constants (in principle indefinite, if of type real) in (finite!) computers. Since these problems are urgent as well, even little steps towards accomplishment like the following ones might be valuable.

Multivaluedness: The main motivation for developing the certified CAS prototype (described in Sect.2.2) was to provide the users with a system that is guaranteed to never make mistakes. One source of mistakes is multivaluedness; systems still get confused between branches of 'multi-valued' functions [16].

The description of the certified CAS prototype also shows a mistake that Maple makes when dealing with a complex function with multiple branches, however the certified system is not able to deal with it formally at all. A first workaround may be to define multivalued functions in a similar way to the partial functions [17]. Such approach would guarantee no mistakes, but would allow only computations which reside entirely within one branch of a multivalued function. For performing computations that cross branches, a formalization of Riemann surfaces is necessary. The libraries of the proof assistants contain very few theorems that talk about multivalued functions, so such a theory needs to be developed together with certified decision procedures for this domain.

Real numbers are one of the basic features of a system for an engineer. Therefore major CAS systems include efficient algorithms for dealing with real number computations and approximations. Numeric methods (e.g. interval arithmetic) are used to provide correctness of these approximations. Many proof assistants include formalizations of real numbers that provide mechanisms for computing approximations. The certified CAS prototype described in Sect. 2.2 uses the Boehm-style calculations already present in HOL Light standard library. Unfortunately, the approximations done by inferences are quite inefficient.

O'Connor's work [33] allows working with infinite precision real numbers in Coq effectively. The proofs can be used for computation inside the proof assistant (with the help of reflection and the Coq bytecode engine), as well as in the extracted programs. Additionally the programs extracted from constructive proofs are naturally functional and therefore can be easily parallelized, which makes them even more efficient. However the approach requires constructive proofs. It is possible to extract proof from classical proofs (e.g. with A-translation), but this has not yet been experimented with in the setting of real numbers.

4.3 Generalization of manipulated objects

This paper assumes the development of programs comprising formal specification, coding algorithms and proving properties of programs — all these development activities operate on formulas: but that does not mean, that also programs operate on formulas. This is not even true for mathematics: mathematical theories comprise, besides formulas, geometric objects like points, circles, straight lines and the like; they comprise graphs of various kinds, and probably several other kinds of objects.

So, when considering design issues of a CTP-based programming language, considerations about mathematical domains beyond predicate calculus seem appropriate. Here the domain of geometry is considered.

Formalizing geometry theorems in proof assistants is quite challenging. As shown in [26] the non-degeneracy conditions and the amount of incidence relations induce many technical lemmas and side conditions leading to technical proofs. This is why quite often special approaches are used in particular formalizations.

On the other hand, the language of a CAS can be used to talk about geometric constructions. The transformation of properties of the constructions with the help of symbolic computation and the general algorithms present in a system (Cylindrical algebraic decomposition, Gröbner bases) allows showing properties of those constructions even in non-standard geometric relations [15]. Specifying such constructions in a computer-algebra like language with strong semantics would allow building real computer verified proofs of properties entailed by geometric constructions.

So, if considering the design of a CTP-based programming language, it seems worth the effort to extend the scope of the language; the relation between deduction and programming comes into a broader view, which might well lead to novel details in design.

5. Related work

The union of the three lines of work merged in this paper relates to a wide range of research topics; so we select only the most relevant work.

Languages of proof assistants are different from the language envisaged, and we did not tackle the question of integrating the different kinds of languages. In particular in the case of Isabelle, the proof language is Isar [31]. Isar is presented to end-users as a language for human-readable proofs, i.e. proofs as close as possible to traditional notation for mathematical proofs. Algorithmic language elements are accessible, but appear as an impurity (imagine Sect.2.3 programming bending lines in Isar).

Since languages of proof assistants have algorithmic language elements, and the envisaged language also comprises deductive elements, the question about integration arises; this questions seems very interesting, but is out of scope of this paper.

Integration of deduction and programming has found a mathematical foundation in the refinement calculus [2], a theory of refining specifications to programs. The same author develops software supporting this kind of refinement [3] for imperative programs; however, for programming applications in mathematics we prefer functional programming, which concerns a subset of the above theory.

The Focalize system [7] is close to the language proposed here. Much can be learned, in design details and in usability, from "design by contract" [28] and respective systems [8], from work done in the Kestrel Institute [8], from the B-Method [9]. Another system combining specification, proof and other engineering tasks is PVS [34]. Albeit not being a typed system, also ACL2 [10] provides integration of deduction and programming.

Computer Algebra and Proof Assistants have coexisted for a many years so there is much research trying to bridge the gap between these approaches from both sides.

First, many proof assistants already have CAS-like functionality, especially for domains like arithmetic. They provide the user with conversions, tactics or decision procedures that solve problems in a particular domain. Such decision procedures present in the standard library of HOL Light are used inside the prototype described in Sect.2.2 for arithmetic's, symbolic differentiation and others.

Similarly some CAS systems provide environments that allow logical reasoning and proving properties within the system. Such environments are provided either as logical extensions (e.g. [37]) or are implemented within a CAS using its language [6].

There are numerous architectures for information exchange between CAS and CTP with different levels of *degree of trust* between the prover and the CAS. In certain approaches the prover uses the algorithms present in the CAS without checking their correctness, i.e. as an oracle, whereas in other approaches the prover takes the output of a CAS and then tries to build a verified theorem out of it. A longer list of frameworks for information exchange and bridges between systems can be found in [18].

There are many approaches to defining partial functions in proof assistants. Since we would like the user to define functions without being exposed to the underlying logic of the proof assistant we only mention some automated mechanisms for defining partial functions in the logic of a proof assistant. Krauss [20] has developed a framework for defining partial recursive functions in Isabelle/HOL, which formally proves termination by searching for lexicographic combinations of size measures. Farmer [10] implements a scheme for defining partial recursive functions in IMPS.

Real Number theories together with accompanying decision procedures already exist in many proof assistants. Melquiond has created a Coq tactic that can solve some linear inequalities over real number expressions using interval arithmetic and bisection [27]. Obua developed a computing library for Isabelle , which uses computation rather than deduction to obtain bounds on real number expressions. HOL Light constructed real numbers are described in [13]. A mechanism for approximation of real number present in the standard library uses the fact that HOL Light terms are transparent and decomposes a term or goal into subterms, looking for underlying real number operations or constants. Lester implemented approximation of real number expressions in PVS [22], which uses fast converging Cauchy sequences to obtain effective evaluation inside PVS.

6. Summary and future work

This paper was set off by a simple question: What about proceeding from the successful CAS-based programming languages to a more powerful basis provided by CTP ? The question immediately makes clear that answers are embedded in an apparently never ending (but still topical) story, in the integration of deduction and computation. In order to not get lost in this story we choose a concrete and pragmatic approach.

Our approach is concrete in that it starts design considerations from three lines of work pursued independently so far: (1) narrowing the gap between declarative program specification and program generation already working in Isabelle, (2) reusing work on a CAS-like input-response-loop embedded into HOL Light, and (3) reconstructing an experimental language for applied mathematics based on Isabelle/HOL. Sect.2 presents these three lines together with immediate advantages from respective merges: code generation and relief from typing problems in programming on complex mathematical structures, handling partiality of CAS-functions as a sound basis for execution of mathematics applications in programs, availability of provers in a programming language checking the conditions in guards and reuse of Isabelle's ML-Scala integration.

Sect.4 deals with advantages of merging the three lines of work, which are not so obvious but definitely within reach of actual R&D. Sect.4.1 argues that specifications guarding programs enhances soundness of implementation; but this seems not sufficient: in order to support interactivity in integrating work on specifications and on algorithms, novel means for tracing system behavior are required. Tracing down details in type-definitions into (sub-)terms and tracing partiality conditions are discussed. Sect.4.2 proposes steps towards accomplishing two challenging problems for mathematics software, multivaluedness of CAS-functions and floating point numbers of type real. Sect.4.3 tries to open up the scope for the design of CAS-based programming languages, and also discusses some specific requirements for constructive geometry.

Our approach is pragmatic in that it envisages design decisions towards usefulness in engineering practice. Sect.3 discusses two activities at an engineers electronic workbench, which seem indispensable in the future: domain engineering and proving. For domain specific knowledge the distinction of three aspects is suggested: the deductive, the applicative and the algorithmic aspect.

The authors are aware, that only prototyping can explore usability of the features. Nevertheless the authors are convinced that in the novel features of CTP-based programming languages can balance demands and benefits such that such systems will be used in the future.

Future work, if going towards more detailed design of a programming language and towards an experimental implementation, will require expertise from the disciplines of compiler construction and of symbolic computation. A decision has to be made whether the features of SML (module system, abstract datatypes and respective pattern matching, exception handling) should be lifted into a language in Isabelle/HOL (like the experimental language described in Sect.2.3), or whether the high-level constructs of Isabelle/HOL can be traced/pushed down to the implementation language SML. In the latter case traceability (Sect.4.1) could emerge from adaption of the SML debugger.

[7] http://focalize.inria.fr/

[8] http://www.kestrel.edu/

[9] http://www.bmethod.com/

[10] http://userweb.cs.utexas.edu/ moore/acl2/

References

[1] Deduction as an engineering science. *Electronic Notes in Theoretical Computer Science*, 86(1):1 – 8, 2003. ISSN 1571-0661. doi: DOI: 10.1016/S1571-0661(04)80648-0. FTP'2003, 4th International Workshop on First-Order Theorem Proving (in connection with RDP'03, Federated Conference on Rewriting, Deduction and Programming).

[2] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998. Graduate Texts in Computer Science.

[3] R.-J. Back, J. Eriksson, and M. Myreen. Testing and verifying invariant based programs in the socos environment. In *Tests And Proofs (First International Conference, TAP 2007, Zurich, Switzerland)*, volume 4454 of *LNCS*, pages 61–78, Zrich, Switzerland, Feb 2007. Springer. Accepted for publication.

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

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

[6] B. Buchberger, C. Dupre, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru, and W. Windsteiger. The Theorema Project: A Progress Report. In M. Kerber and M. Kohlhase, editors, *Symbolic Computation and Automated Reasoning (Proceedings of CALCULEMUS 2000, Symposium on the Integration of Symbolic Computation and Mechanized Reasoning)*, Natick, Massachusetts, 2000. A.K. Peters. ISBN 1-56881-145-4.

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

[8] *Standard ECMA-367 Eiffel: Analysis, Design and Programming Language*. ECMA International, 2 edition, June 2006. www.ecma-international.org/publications/standards/Ecma-367.htm.

[9] W. M. Farmer. Biform theories in chiron. In Kauers et al. [19], page 6679. ISBN 978-3-540-73083-5.

[10] W. M. Farmer. A scheme for defining partial higher-order functions by recursion. In A. Butterfield and K. Haegele, editors, *IWFM*, Workshops in Computing. BCS, 1999.

[11] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *Functional and Logic Programming, 10th International Symposium: FLOPS 2010*, volume 6009 of *Lecture Notes in Computer Science*. Springer, 2010.

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

[13] J. R. Harrison. Theorem proving with the real numbers. Technical Report 408, University of Cambridge, Computer Laboratory, November 1996.

[14] J. Hölzl. Proving inequalities over reals with computation in Isabelle/HOL. In G. D. Reis and L. Théry, editors, *Proceedings of the ACM SIGSAM 2009 International Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS'09)*, pages 38–45, Munich, August 2009.

[15] T. Ida, M. Marin, H. Takahashi, and F. Ghourabi. Computational origami construction as constraint solving and rewriting. *Electr. Notes Theor. Comput. Sci.*, 216:31–44, 2008.

[16] D. Jeffrey and A. Norman. Not seeing the roots for the branches: multivalued functions in computer algebra. *SIGSAM Bull.*, 38(3):57–66, 2004. ISSN 0163-5824. doi: http://doi.acm.org/10.1145/1040034.1040036.

[17] C. Kaliszyk. Automating side conditions in formalized partial functions. In S. Autexier, J. Campbell, J. Rubio, V. Sorge, M. Suzuki, and F. Wiedijk, editors, *AISC/MKM/Calculemus*, volume 5144 of *LNCS*, pages 300–314. Springer, 2008. ISBN 978-3-540-85109-7.

[18] C. Kaliszyk and F. Wiedijk. Certified computer algebra on top of an interactive theorem prover. In Kauers et al. [19], pages 94–105. ISBN 978-3-540-73083-5.

[19] M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors. *Towards Mechanized Mathematical Assistants, 14th Symposium, Calculemus 2007, 6th International Conference, MKM 2007, Hagenberg, Austria, June 27-30, 2007, Proceedings*, volume 4573 of *LNCS*, 2007. Springer. ISBN 978-3-540-73083-5.

[20] A. Krauss. Partial recursive functions in higher-order logic. In U. Furbach and N. Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 589–603. Springer, 2006. ISBN 3-540-37187-7.

[21] R. Lang. Elementare Gleichungen der Mittelschulmathematik in der ΣSACWissensbasis. 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.

[22] D. R. Lester. Real number calculations and theorem: Proving validation and use of an exact arithmetic. In O. Ait-Mohamed, editor, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2008.

[23] P. Lucas. Formal semantics of programming languages: VDL. *IBM Journal of Devt. and Res.*, 25(5), 1981.

[24] P. Lucas. On the formalization of programming languages: Early history and main approaches. In D. Bjørner and C. B. Jones, editors, *The Vienna Development Method: The Meta-Language*, volume 16 of *LNCS*. Springer, 1978.

[25] P. Lucas and K. Walk. *On the Formal Description of PL/I*, volume 6 of *Annual Review in Automatic Programming*. Pergamon Press, Oxford, 1970.

[26] N. Magaud, J. Narboux, and P. Schreck. Formalizing desargues' theorem in coq using ranks. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1110–1115, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-166-8.

[27] G. Melquiond. Proving bounds on real-valued functions with computations. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, Lectures Notes in Computer Science, Sydney, Australia, 2008.

[28] B. Meyer. Design by contract. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986.

[29] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, London, 1997.

[30] W. 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.

[31] T. Nipkow. *A Tutorial Introduction to Structured Isar Proofs*. Institut fr Informatik, TU Mnchen. http://isabelle.in.tum.de/dist/Isabelle/doc/isar-overview.pdf.

[32] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[33] R. O'Connor. *Incompleteness and Completeness*. PhD thesis, Radboud University Nijmegen, 2009.

[34] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. Henzinger, editors, *Computer-Aided Verification*, pages 411–414. CAV'96, 1996.

[35] L. C. Paulson. Organizing numerical theories using axiomatic type classes. *Automated Reasoning*, 33(1):2949, 2004.

[36] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[37] E. Poll and S. Thompson. Adding the axioms to Axiom: Towards a system of automated reasoning in Aldor. In *Calculemus and Types '98*, July 1998. Also as technical report 6-98, Computing Laboratory, University of Kent.

[38] M. Wenzel. The Isabelle/Isar implementation. Technical report, TU München, Software & Systems Engineering, April 2009. With contributions by Florian Haftmann and Larry Paulson.

[39] M. Wenzel. The languages of Isabelle: Isar, ML, and Scala. Slides Lausanne, September 2009. http://www4.informatik.tu-muenchen.de/ wenzelm/papers/lausanne2009.pdf.