

An experimental support system for formal mathematical reasoning

Matthias Anlauff¹, Stefan Jähnichen², Martin Simons¹

¹ Technische Universität Berlin*

² GMD-FIRST**

Abstract. Requirements for tools which support the creation and the intelligible *presentation* of formal deductions are investigated. They are contrasted with requirements which emphasize the interactive *construction* of correct proofs. As an example, the design and the implementation of a set of support tools for Deva is described. Deva is a typed functional language and has been used in a number of case-studies on formal program development. The use of this toolset is illustrated by impressions of a working session.

Keywords: formal reasoning; logical frameworks; proofs; tool support.

1 Introduction

Traditionally, research on *computer assisted formal reasoning* focusses on how the computer can help in the process of constructing correct mathematical arguments. The initial ambitious goal of leaving the task of finding proofs altogether up to the computer was quickly abandoned; fullfledged automatic theorem proving turned out to be unattainable for theoretical and practical reasons. Research consequently concentrated on building theorem provers which required interaction with the user and on trying to increase the degree of automation. So called interactive tactical theorem provers are being developed and they quite successfully assist the user in the process of *constructing* correct formal proofs. However, once a proof is constructed, only a few facilities are provided to *present* the proof in an intelligible way so that one can understand its basic line of reasoning. Presenting formal reasoning may not seem such a terribly attractive idea in the first place, particularly since formal arguments tend to be bogged down by technical details. This “formal noise” usually hides the basic line of reasoning underlying a proof but is necessary to enable a machine to check the correctness of an argument. On the other hand, the primary goal should be to convince the user of the correctness of an argument and not the machine. Here, we have the class of arguments in mind that are generally thought worthy of the extra effort

* Technische Universität Berlin, Forschungsgruppe Softwaretechnik (FR5-6), Franklinstr. 28/29, D-10587 Berlin, Germany. e-mail: {ma,simons}@cs.tu-berlin.de

** GMD Forschungsstelle FIRST, Rudower Chaussee 5, D-12489 Berlin, Germany. e-mail: jaehn@first.gmd.de

needed to prepare them for presentation; for instance, the proof of a mathematical theorem, the calculation of an algorithm, or a refinement in the refinement calculus. We do not mean justification of boring technical details which are introduced solely by formality. Thus, in principle, computer assisted reasoning should support both the *construction* of correct mathematical reasoning and its *presentation*.

We argue that research on interactive theorem proving tackles this goal while focussing on construction. One way of focussing on presentation is to devise a formalism in which one can express mathematical arguments which are, ultimately, close to conventional mathematical arguments and which can be checked by a machine at the same time. This goal seems not to be hopeless, particularly because several people have argued convincingly in favor of a more structured and rigorous approach to mathematical reasoning for some time (cf. [5, 13, 16, 18, 31]). In order to stress our focus on supporting intelligible formal proofs we will henceforth use the term “computer assisted formal mathematical reasoning”. Fig. 1 broadly summarizes the different approaches to computer aided formal reasoning.

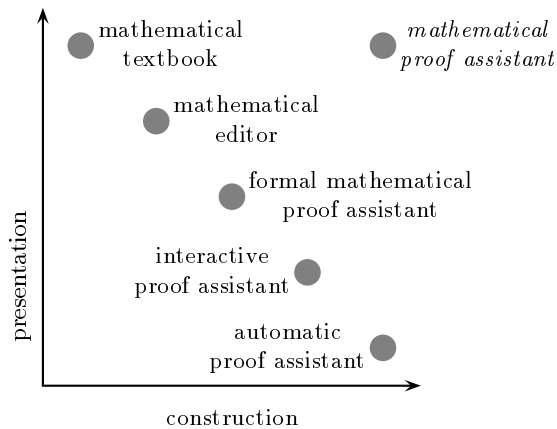


Fig. 1. Presentation vs. construction of mathematical proofs

In this paper we report on the design and the implementation of a support system for formal mathematical reasoning called “Devil”. The formal basis of this system is Deva, a typed functional language. Formalization takes place according to the Curry-Howard paradigm of “propositions as types” which associates proofs with functions and propositions with types. Deva is a descendent of the AUTOMATH family of languages (cf. [8]) and was originally designed during the Esprit project ToolUse (cf. [19, 47, 52]). The original intention of this project was to study formal software development methods and to provide them with tool support. It is beyond the scope of this paper to give a detailed introduction to Deva. A self contained introduction to the language, its theory and its

semantics is given in [52]. A comprehensive treatment of the language is given in [54] and includes a tutorial-like introduction and the presentation of two case studies. Several case-studies in formalized software developments have been undertaken and fully carried out with the assistance of Devil. Some experiments with calculational program developments are contained in the book [54]. This book also contains a formalized VDM data-reification as well as a formal proof of the transitivity of VDM reification (cf. also [51]). A formalized VDM development of a revision management system is described in this volume [53]. The formalization of an algebraic specification language is contained in [42].

The structure of the paper is as follows: in the next section some general requirements which we believe should be met by a system to support the presentation of formal mathematical reasoning are given. We will then describe the design and the implementation of Devil in Sect. 3 and continue to give an impression of how a user works with the system in Sect. 4. After relating our approach to other approaches in Sect. 5 we discuss experiences with Devil and describe topics of future work. Many properties of Devil depend heavily on Deva as the specific formal system that is the basis of Devil, nonetheless we believe that the general requirements established in the next section as well as our design approach are of general interest.

2 Assisting formal mathematical reasoning

As we already pointed out in the previous section, computer assisted formal reasoning should ultimately support both, the *construction* of correct mathematical arguments and their intelligible *presentation*. Ideally, proofs should then be constructed and presented at the level of abstraction of, for instance, the proofs presented in [18, 31]. However, there still remains a substantial gap between these proofs and formal proofs constructed with the help of, e.g., interactive theorem provers. This is largely due to the fact that the reasoning steps a human can comprehend are usually much more abstract than those an interactive theorem prover can process in a single step. The problem then is to think of ways of bridging this gap. In this section, we will concentrate on what is necessary to support a high-level description of formal reasoning that can be checked by machine and on ways of presenting it. We will sketch some ideas we have on narrowing the gap between this approach and the interactive theorem proving approach in Sect. 6.

We want to separate the discussion of the requirements into three areas: what functionality should the deduction kernel provide; what features are necessary to present formal reasoning in a structured and intelligible way; and how can the whole system be kept flexible, i.e., applicable for various instances of formal reasoning. We will keep the discussion at a rather abstract level so that the software architecture of a support system becomes visible. The following section will then present a concrete attempt to realize such an architecture.

Deduction support

At the heart of the deduction support lies the implementation of a logical formalism that allows for the expression of proofs and deductions. Apart from traditional logics such as predicate logic the so called logical frameworks (cf. [25, 26]) have received widespread attention. A logical framework implements a meta-logic in which other specific logics can be expressed. The advantage of such frameworks as a formal basis for our purposes will be discussed later. At the lowest level, the core must be able to check the correctness of formal deductions. However, the intention is to express proofs at an intelligible mathematical level. In principle, the translation of such abstract proofs or “proof sketches” to the formal level should be done automatically. If not possible, this translation must be guided by hints supplied by the user. In other words, abstract proofs are expressed in a “high-level proof language” and are “compiled” into the concrete and fully detailed proofs. The compilation may take place either completely automatically or interactively by “proof refinement”. In the past, several techniques and algorithms have been developed which are used in this process:

- *Higher-order unification* (cf. [23]), though in principle undecidable, is a powerful technique to resolve implicit arguments in the application of rules or tactics.
- *Term rewriting* (cf. [28]) is essential to realize succinct formulation of equational or, more general, calculational style proofs.
- *AC-unification* (cf. [45]) is also helpful in saving the user tedious manipulations of formulas.

These are basic building blocks for automating deduction. In order to deal with user supplied hints the tactics related techniques of tactical theorem-provers are useful (cf. [40]).

We have so far only described the bare functionality of the deduction kernel. In order to actually use it at least two other components must be present: library management and sophisticated user interface.

- The *library management* provides access for the deduction kernel and for the user to various standard and user-defined theories. The availability of a rich body of standard theories is instrumental in making effective use of the system.
- The *user interface* provides an interface between the user and the deduction kernel. A central concern of the design of the user interface should be to provide the user with sophisticated “proof debugging” facilities. This implies some sort of interaction with the deduction kernel similar to the interactive facilities of interactive theorem provers.

Presentation support

Donald Knuth writes as a motivation for literate programming [29]: “Instead of imagining that our main task is to instruct a *computer* what to do, let us

concentrate rather on explaining to *human beings* what we want a computer to do.” We can rephrase this sentence with respect to mathematical reasoning expressed in a formal framework as follows: Instead of imagining that our main task is to explain to a computer why a formal development is correct, let us concentrate rather on explaining to human beings why it is correct. Carroll Morgan argues similarly in a recent article on literate program developments in the refinement calculus [35]. The principles of literate programming [30] can profitably be applied to presenting formal reasoning:

- the document containing the formal proofs is the same document containing the presentation of the proofs, guaranteeing that once the document is checked, the presented version is correct.
- the order in which the formal reasoning is presented may be any order the author deems appropriate and need not be the order which the checker expects and which is prescribed by some formal syntax. The system is able to extract valid input for the checker from such a “web”.

We want to add two principles which we believe are helpful in presenting intelligible formal deductions:

- it must be possible to present formal reasoning at *levels of abstraction*. The higher levels resemble ordinary mathematical reasoning in large steps which contain just enough information for a human to understand it. These large steps are refined at the lower levels until the checker is able to check them. The steps themselves are combined by operators of the formal calculus to yield the complete development. The user or the reader may decide to which depth these levels of abstraction should appear in the presentation. Leslie Lamport has proposed in [31] a similar structure for the presentation of conventional proofs and in [46] his style is partly adapted to the presentation of formal proofs in Deva.
- it must be possible that a formal deduction may be given *incompletely*. Not all parts of a formal development need to be given formally; it must be possible to check the correctness of the complete deduction modulo the correctness of the omitted parts. In the literate approach to formal reasoning, in this case at least some verbal justification of the omitted parts should be given. This liberal approach to formal reasoning is, in our view, justified by experience. We strongly believe that any system that supports mathematical reasoning will be successful only if it adheres to the basic principle of leaving the user in control at all times. It may check that no parts have been omitted in a deduction and refuse its final approval if they have. But it should nevertheless support an incremental working mode. The success of VDM and Z, for instance, can be attributed, at least partly, to the fact that they do not enforce the discharge of all proof obligations. Their major advantage is that they provide adequate means of specifying and developing a system formally and of reasoning about the system and the development formally. But it is up to the user to decide if and when the necessary proofs should be carried out.

The question of how to actually structure a formal development has not yet been addressed. This is largely a matter of experience and taste. Just as with mathematical proofs, the first attempt is not the best, but once one has some intuition about the problem and the domain it stems from, one can construct a little theory around it which then leads to a cleaner and more structured proof. In formal system developments, some guidance for structuring is usually provided by the “formal method” one follows. With formal proofs, we have experienced that *natural deduction* is helpful in providing a global structure; locally, *calculational reasoning* is more effective. Lamport structures his proofs in [31] guided by the principles of natural deduction. Gries and Schneider [18] and other proponents of the “Dutch School” argue in favor of a calculational style based on equational reasoning. In our experience, one can go quite far with the calculational approach and this usually leads to more elegant proofs ([18] gives beautiful examples and introduces the necessary techniques). In order to combine subproofs to form larger ones, however, the rules of natural deduction seem to be appropriate (cf. [46]).

Flexibility

In computer science, even more than in mathematics, different calculi and logics abound. However, all these logics and calculi have basic reasoning procedures in common. Logical frameworks isolate these commonalities and combine them as meta-logic in which other logics can be represented. In principle, an implementation of the logical framework, together with the representation of some concrete logic yields an implementation for that particular logic. There are various logical frameworks being developed (cf. Sect. 5); some are based on higher-order logic (HOL, IMPS, Isabelle), some on typed lambda-calculi (Coq, Deva, LEGO), and some on constructive type theory (ALF, Nuprl).

All the different logics and calculi also come equipped with their own syntax and notations. And this for good reason: well designed syntax and notation can be very helpful in highlighting certain properties of operators or constructions which are important when reasoning or calculating with these operators. Thus, representing a logic in the meta-logic is not enough. It should also be possible to imitate its syntactical and notational peculiarities. One way to accomplish this is by a preprocessing step where a concrete (graphical) syntax which implements the special operators and notation is translated to an elementary abstract syntax which is then handed to the deduction engine. Care must be taken, however, to report errors in the original concrete syntax.

3 The design and implementation of Devil

In this section we describe a set of tools collectively called Devil or Deva’s interactive laboratory. They were designed and implemented over the past years in order to support Deva and they can be seen as a first approximation to what has been described in the previous section as a *formal mathematical proof assistant*.

In fact, the experiences we have gained from the experiments on formal program development which were carried out with Deva and Devil strongly influenced the formulation of the requirements for such a system.

Deva is a typed functional language descending from a member of the AUTOMATH family of languages (cf. [36]). Mathematics is formalized in Deva according to the Curry-Howard principle of proposition as types and proofs as objects. Nederpelt's calculus was extended by several constructs such as sums and products to aid in the succinct expression of formal deductions. It was also extended by the notion of "context" to incorporate a structuring mechanism directly into the language. In principle, contexts are nested sequences of declarations and definitions. For presentation purposes one has the choice among syntactical variants of constructs of the language. Variables of a functional type may be declared to be mixfix operators thereby making it possible to mimic the modelled theory to some extent. Finally, and most significantly, Deva features a so called "implicit level" which allows formal derivations to be expressed without explicitly supplying all necessary technical details. Deva formalizations expressed at this implicit level remain comprehensible to some extent and this has been demonstrated in various case-studies. For example, VDM developments are presented in [53] and [54] which also contains a calculational program development; a formal mathematical proof of the Schröder-Bernstein theorem is presented in [46]. In each case, the Deva formalizations are contrasted with their informal counterparts and it is shown that the blowup factor between an informal and a formal proof can be kept under control. It is beyond the scope of this paper to give a detailed treatment of Deva; this has been given elsewhere [52, 54]. Instead, we will concentrate on the design and some implementation issues of the tool support. This discussion will be structured along the lines of the previous section.

Devil's deduction support

The main functionality of the deduction support is to check a Deva text for type-correctness. This type-check is straightforwardly implemented for the explicit level of Deva. The implicit level, however, causes several problems that are not that easily dealt with. The translation from the implicit to the explicit levels is called *explanation*. Before the checker can start with explaining the input the abstract syntax of the implicit level is computed. More precisely:

- the *lexical analysis* transforms the input into a sequence of tokens;
- the *context-free syntax analysis* evaluates the Deva context structure;
- the *binding phase* determines for each variable the corresponding declaration or definition;
- the *context-sensitive syntax analysis* evaluates the final abstract syntax tree involving the syntactical structure introduced by user-defined operators.
- finally, the *explanation phase* transforms the implicit representation into an explicit one.

This structure resembles very much the analysis phase of a traditional compiler for programming languages. For this reason, the compiler generation language Gentle [43] was chosen as the implementation language of the deduction support. Gentle allows the easy expression of transformation rules between the different representations. Furthermore, it generates specifications for lexical scanner and context-free parser generators from a high-level compiler specification.

The first two phases are straightforwardly implemented by standard compiler techniques. An efficient adaptation of de-Bruijn indices is used to implement the binding phase. During the context-sensitive syntax analysis the treatment of mixfix operators poses special problems. An operator is specified in the context of a variable declaration or definition; the symbol “ (\cdot) ” indicates possible argument positions of the operator. For example, the declarations

$$\begin{aligned} (\cdot) \sqsubseteq (\cdot) &: [\mathcal{U}; \mathcal{U} \vdash prop] \\ ; \sqcup (\cdot) &: [set(\mathcal{U}) \vdash \mathcal{U}] \end{aligned}$$

introduce two variables, “ \sqsubseteq ” and “ \sqcup ”. The variable “ \sqsubseteq ” is declared to be a binary function taking two arguments of type \mathcal{U} and yielding a result of type $prop$. At the same time, “ \sqsubseteq ” is declared to be an infix operator. Likewise, “ \sqcup ” is declared to be a unary function taking an argument of type $set(\mathcal{U})$ and yielding a value of type \mathcal{U} . At the same time, “ \sqcup ” is declared to be a prefix operator. While parsing terms like “ $\sqcup A \sqsubseteq y$ ” the system takes into account information about the operator types of the variables involved, as well as relative priorities assigned to them. The former information is extracted from the declarations of the variables during the binding phase; the latter information is specified by the user with priority rules of the form

$$opspec \sqcup > \sqsubseteq$$

which cause Devil to convert the above term to “ $\sqsubseteq (\sqcup(A), y)$ ”. Priority rules can also be used to specify associativities of infix operators. Two additional features complicate the parsing algorithm:

- an identifier may have different operator types in different contexts;
- sectioning is possible, i.e. the algorithm is able to process terms containing operators with less arguments than specified in the corresponding declaration of the operator. For example, the term “ $\sqsubseteq y$ ” is transformed to “ $\sqsubseteq (\$2 := y)$ ”, where “ $\$2$ ” represents the argument position in “ \sqsubseteq ”.

The explanation phase translates implicit proofs into explicit ones. Contrary to proofs developed interactively with tactical theorem-provers, an implicit proof in Deva can be regarded as a “sketch” which instructs the checker how to transform the proof into its fully explicit representation. During this transformation process the checker often has to select one of a number of possible solutions for a local expression. For instance, everyone knows that higher-order unification of terms containing variables of functional type leads to a set of solutions. This set of solutions can be restricted by taking into account information which is contained in a “local context” of the sketch. The problem is to determine the

boundaries of such a local context: if it is too narrow, the search space is not restricted enough; if it is too wide, the computation of the constraints induced by the local context is too expensive. In tactical theorem-provers a local context is at least given by the currently processed tactic construct, while in a sketch no such “natural” division exists. In the following, we will call these local contexts *check-scopes*.

The explanation algorithm is split up into the following steps:

1. determination of check-scopes,
2. generating constraints for each check scope, and
3. solving these constraints.

The first step fixes the boundaries of a local context, the second step generates constraints in the form of equations and extracts variables to be instantiated. The third step consists of a unification algorithm which tries to solve the generated equations and instantiate the extracted variables. This process will be described more formally below.

Assume that the two abstract data types \mathcal{T}_I and \mathcal{T}_E contain the abstract syntax of the implicit resp. the explicit level. Thus, the task of the explanation phase is to realize a function *Explain* mapping elements of \mathcal{T}_I to their explicit representations in \mathcal{T}_E :

$$\text{Explain}(t^I) = t^E \quad \text{for } t^I \in \mathcal{T}_I \text{ and } t^E \in \mathcal{T}_E$$

According to the above division (2. and 3.) there exist two auxiliary functions for the realization of *Explain*:

- an extraction function *XT* transforming an implicit term t^I into an intermediate representation consisting of
 - a temporary explicit representation $\overline{t^E}$,
 - a set of variables V , and
 - a set of equations Eq .

The function *XT* applied to an implicit term t^I is written as

$$XT(t^I) = [V] \overline{t^E} [Eq]$$

This representation is called the *check scope* of t^I . The term $\overline{t^E}$ is already a valid explicit term, but it is not equal to the final explicit version t^E because, in general, $\overline{t^E}$ can contain variables which are contained in V and which must be instantiated during the subsequent unification step (see below) in order to ensure the correctness of the transformation.

- a function *Solve* which solves the generated equations (in general by using unification techniques) and determines the final explicit version by instantiating the variables contained in V and substituting them in $\overline{t^E}$. Additionally, other changes must frequently be carried out, such as the determination of the correct order of variables not being instantiated. The function *Solve* applied to the result of *XT* is written as

$$\text{Solve}([V] \overline{t^E} [Eq]) = t^E$$

By using XT and $Solve$ the function $Explain$ can be defined as their composition

$$Explain := Solve \circ XT$$

This solution would be rather poor if the extraction function XT would also be realized by restricting its evaluation scope to a single construct of the abstract syntax $\mathcal{T}_{\mathcal{I}}$. A more powerful algorithm for XT will now be defined. The idea is to delay the solution of local constraints and to put the extracted variables and equations of a set of abstract constructs together into a single check-scope. In order to assemble these check-scopes, a mapping $DelaySolve$ is defined which determines for each constructor of $\mathcal{T}_{\mathcal{I}}$ the set of positions of those arguments which should be included into the current check-scope. Let $\{\mathcal{C}_k^{\mathcal{I}}\}$ denote the family of sets of constructors of $\mathcal{T}_{\mathcal{I}}$ indexed by the arity. For each constructor $\otimes \in \mathcal{C}_k^{\mathcal{I}}$ we define a function xt_{\otimes} which computes for given explicit terms $t_1^{\mathcal{E}}, \dots, t_k^{\mathcal{E}}$ the check scope of $\otimes(t_1^{\mathcal{E}}, \dots, t_k^{\mathcal{E}})$. We can now extend these atomic extraction functions to $\mathcal{T}_{\mathcal{I}}$ by using $DelaySolve$ as follows

$$\begin{aligned}
XT(\otimes) &= xt_{\otimes} && \text{if } \otimes \in \mathcal{C}_0^{\mathcal{I}} \\
XT(\otimes(t_1^{\mathcal{I}}, \dots, t_k^{\mathcal{I}})) &= \\
&\left[\left(\bigcup_{j \in DelaySolve(\otimes)} V_j \right) \cup V \right] \overline{t^{\mathcal{E}}} \left[\left(\bigcup_{j \in DelaySolve(\otimes)} Eq_j \right) \cup Eq \right] \\
&\text{where} \\
&xt_{\otimes}(\widehat{t}_1^{\mathcal{E}}, \dots, \widehat{t}_k^{\mathcal{E}}) = [V] \overline{t^{\mathcal{E}}} [Eq] \\
&\widehat{t}_j^{\mathcal{E}} = \begin{cases} \overline{t_j^{\mathcal{E}}} & \text{if } j \in DelaySolve(\otimes) \\ t_j^{\mathcal{E}} & \text{if } j \notin DelaySolve(\otimes) \end{cases} \\
&\text{where } XT(t_j^{\mathcal{I}}) = [V_j] \overline{t_j^{\mathcal{E}}} [Eq_j] \\
&\text{where } Explain(t_j^{\mathcal{I}}) = t_j^{\mathcal{E}} \\
&\text{if } \otimes \in \mathcal{C}_k^{\mathcal{I}}, k \geq 1
\end{aligned}$$

Thus, the definition of $DelaySolve$ determines the boundaries of a check-scope and we have broken down the definition of XT to the definition of the atomic xt . For example, we consider the following grammar of an implicit syntax for λ -terms representing a minimal core of the Deva language:

grammar rules	abstract syntax
$\mathcal{T}_{\mathcal{I}} ::= [\mathcal{V} : \mathcal{T}_{\mathcal{I}} \vdash \mathcal{T}_{\mathcal{I}}]$	$dabs_{\mathcal{I}}(t_1, t_2)$
$[\mathcal{V} ? \mathcal{T}_{\mathcal{I}} \vdash \mathcal{T}_{\mathcal{I}}]$	$iabs_{\mathcal{I}}(t_1, t_2)$
$\mathcal{T}_{\mathcal{I}}(\mathcal{T}_{\mathcal{I}})$	$app_{\mathcal{I}}(t_1, t_2)$
$\mathcal{T}_{\mathcal{I}} \therefore \mathcal{T}_{\mathcal{I}}$	$judg_{\mathcal{I}}(t_1, t_2)$
\mathcal{V}	$use_{\mathcal{I}}(x)$

where \mathcal{V} is the set of identifiers. The explicit syntax is given by the following grammar:

grammar rules	abstract syntax
$\mathcal{T}_E ::= [\mathcal{V} : \mathcal{T}_E \vdash \mathcal{T}_E]$	$\mathbf{dabs}_E(t_1, t_2)$
$\quad [\mathcal{V} ? \mathcal{T}_E \vdash \mathcal{T}_E]$	$\mathbf{iabs}_E(t_1, t_2)$
$\quad \mathcal{T}_E . \mathcal{T}_E$	$\mathbf{app}_E(t_1, t_2)$
$\quad \mathcal{V}$	$\mathbf{use}_E(\mathcal{V})$

Now, the extraction kernel can be defined as follows

$$\begin{aligned}
xt_{\mathbf{dabs}_I}(t_1, t_2) &= [] \mathbf{dabs}_E(t_1, t_2) [] \\
xt_{\mathbf{iabs}_I}(t_1, t_2) &= [] \mathbf{iabs}_E(t_1, t_2) [] \\
xt_{\mathbf{use}_I}(x) &= [] \mathbf{use}_E(x) [] \\
xt_{\mathbf{app}_I}(t_1, t_2) &= \begin{cases} [] \mathbf{app}_E(t_1, t_2) [type(t_2) \stackrel{!}{=} t_3] & \text{if } type(t_1) \sim \mathbf{dabs}_E(t_3, t_4) \\ [x?t_3] \mathbf{app}_E(t_1, t_2) [] & \text{if } type(t_1) \sim \mathbf{iabs}_E(t_3, t_4) \end{cases} \\
xt_{\mathbf{jud}_I}(t_1, t_2) &= [] t_1 [type(t_1) \stackrel{!}{=} t_2]
\end{aligned}$$

where “ \sim ” means “*equal modulo β -reduction*”. The main difference between the implicit and the explicit level can be seen in the definition of $xt_{\mathbf{app}_I}$. The implicit application uses the first declaration $x : t$ of the functor as the actual argument position, while components of the form $x?t$ are regarded as variable declarations. The explicit application always takes the first component as the actual argument position regardless of whether it has the form $x : t$ or $x?t$. In order to illustrate the check scoping described above let us now consider the following implicit formalization:

$$\begin{aligned}
& [\text{foo} : [F?[n : nat \vdash nat] \vdash [x : F(0) \vdash F(1)]] \vdash \\
& [\text{bar} : 0 + 1 \\
& \vdash \text{foo}(\text{bar}) \therefore 1 + 1]]
\end{aligned}$$

We will now apply the xt function to the constituents of the last term:

$$\begin{aligned}
xt_{\mathbf{app}_I}(\text{foo}, \text{bar}) &= [F?[n : nat \vdash nat]] xt_{\mathbf{app}_I}(\mathbf{app}_E(\text{foo}, F'), \text{bar}) [] \\
&= [F?[n : nat \vdash nat]] \mathbf{app}_E(\mathbf{app}_E(\text{foo}, F'), \text{bar}) [type(\text{bar}) \stackrel{!}{=} F'(0)] \\
&= [F?[n : nat \vdash nat]] \mathbf{app}_E(\mathbf{app}_E(\text{foo}, F'), \text{bar}) [0 + 1 \stackrel{!}{=} F'(0)]
\end{aligned}$$

The explicit version of this implicit application is not definite because of the second order variable F' which can be instantiated either by $[n : nat \vdash n + 1]$ or by $[n : nat \vdash 0 + 1]$. But only the first solution satisfies the additional type constraint specified by the judgement “ $\therefore 1 + 1$ ”. In order to avoid backtracking, the technique of check scoping can be used. Therefore the *DelaySolve* mapping is defined as follows:

$$\text{DelaySolve}(\otimes) = \begin{cases} \{1\} & \text{if } \otimes = \mathbf{jud}_I \\ \{\} & \text{otherwise} \end{cases}$$

meaning that an additional type constraint should always be considered during the explanation process. Thus, the XT function for the above judgement is given by

$$XT(foo(bar) \therefore 1 + 1) = \\ [F'?[n : nat \vdash nat]] \text{ appE}(\text{appE}(foo, F'), bar) \left[\begin{array}{l} 0 + 1 \stackrel{!}{=} F'(0) \\ F'(1) \stackrel{!}{=} 1 + 1 \end{array} \right]$$

The desired solution can now be obtained by solving the set of equations on the right using e. g. unification or anti-unification techniques being more powerful and efficient than backtracking over a tree of alternative solutions.

Devil's support for structured proof development

The Devil system also includes additional features to support the development of structured proofs and their presentation. We will briefly describe these features in the following paragraphs.

Library management. As a prerequisite for working effectively with a proof support system support for the reuse of formalizations must exist. Devil provides library management on the basis of the UNIX file system. Deva contexts can be saved persistently into special files, called *devil object files*. A formalization can only be saved if it is self-contained and correct. This requirement is necessary because otherwise consistency checks would have to be performed whenever a context is loaded. In general, these checks would be rather complex so that only little would be gained in terms of efficiency.

Those Deva contexts that are contained in a library can be regarded as “modules”. While the context associated with a Devil object file forms the “implementation” of that module, its “interface” is given by the named contexts of the saved formalization. Using this concept, it is possible to extract selected parts of a formalization contained in a library file. If contexts contained in different modules but based on equivalent assumptions are combined, these assumptions should be identified with each other.

User interface. Devil's user interface provides the possibility of working interactively with the system. The interface is implemented as an extension of the tool command language “tcl” (cf. [38]) which is a simple textual language comparable with UNIX shell programs such as “csh”. The “tk” library (an extension of tcl) is used to implement the graphical version of Devil based on X-windows so that Devil provides the usual operational comfort of a window-based user interface.

The commands provided by the interactive user interface are executed on the basis of the current system state which is given by the currently loaded Deva context. Typically, this context is the result of loading a library file, but internal proof states can also form the current context. Besides the transformation of Deva formalizations which are contained in source files, Devil provides the possibility of processing Deva expressions entered directly to the user interface

as strings. This is very helpful, for instance, for making experiments during the construction of a proof, if the context forming the basis of the proof has been loaded as current context.

Presentation support. Presentation and documentation of Deva formalizations are realized by implementing a WEB system for Deva [7]. A WEB document is structured in sections that may contain named “chunks” of Deva code. These chunks may be referred to from other chunks and so the document defines a web, or rooted graph, of Deva code parts. The structure of the web is completely independent of the structure required by Deva’s grammar. From a single Deva WEB document one can then extract two files:

- a L^AT_EX file which is used for generating a presentation document containing explanatory texts as well as Deva code;
- a file containing only Deva expressions, which are the input for the checker.

The chunk mechanism of WEB can be used to structure proofs hierarchically [46]:

```
[ Assumptions
; ⟨Proof of lemma 1⟩ ∴ lemma 1
; ⟨Proof of lemma 2⟩ ∴ lemma 2
; ⟨Proof of lemma 3⟩ ∴ lemma 3
⊢ ⟨QED⟩ ∴ theorem
]
```

The chunks ⟨Proof of lemma 1⟩, ⟨Proof of lemma 2⟩, ⟨Proof of lemma 3⟩, and ⟨QED⟩ can each be developed in their own section.

The interactive development of such hierarchically structured proofs is supported by the Devil system through the concept of *proof obligation objects*. This feature is part of the user interface of Devil. A proof obligation is introduced by referring to a WEB chunk which does not yet exist and by stating its goal. For instance, in the preceding example three proof obligations are introduced with their corresponding goals *lemma 1*, *lemma 2*, and *theorem*. Each proof obligation object has an attribute specifying the current state of the proof term. The value of the state attribute can either be

- *non-existent*, if the proof for the obligation does not exist or
- *incomplete*, if the proof term is correct but it depends on further proof obligation objects at least one of which is not proven or
- *proven*, if the proof term is correct and the state of each of the proof obligation objects on which the term depends is also proven.

Proof obligation objects can be used to specify incomplete Deva formalization because proof terms can be added later (or even omitted) to the corresponding proof obligation objects. The identification of proof obligations with DevaWeb sections can be seen as a first step on the way to integrating the presentation of proof documents into their production process.

This concludes a brief overview of the design and the implementation of Devil. For a complete introduction, the reader is referred to [4] and [7].

4 Elements of a sample session

In this section we give an impression of what work with Devil is like. As an example, we have chosen the proof of a theorem from lattice theory, the *Knaster-Tarski fixpoint theorem*. We will not present the proof development in every detail but highlight only some aspects of a session with the system. The theorem is as follows

The Knaster-Tarski fixpoint theorem: *A monotonic map $\Phi : \mathcal{U} \rightarrow \mathcal{U}$ over a complete lattice has a fixpoint.*

We assume that we have already loaded a library file containing a theory of complete lattices relative to which the proof is formalized. Additionally we assume that we have already formalized the assumptions that Φ is a monotonic mapping of type $[\mathcal{U} \vdash \mathcal{U}]$. The idea for the proof can be sketched briefly as follows:

We will show that the least upper bound of the set $M \triangleq \{x : x \in \mathcal{U} : x \sqsubseteq \Phi(x)\}$ is a fixpoint of Φ . We show that $\sqcup M \sqsubseteq \Phi(\sqcup M)$ and that $\Phi(\sqcup M) \sqsubseteq \sqcup M$. And so, by anti-symmetry, equality follows.

We will use Devil’s concept of proof obligation objects to develop the proof. Initially, a proof obligation object “⟨ Proof of the Knaster-Tarski fixpoint theorem ⟩” is introduced representing our goal on the assumption that M is defined as proposed above:

```
[ M := { [ x :  $\mathcal{U} \vdash x \sqsubseteq \Phi(x)$  ] }
   $\vdash$  ⟨ Proof of the Knaster-Tarski fixpoint theorem ⟩ .:  $\Phi(\sqcup M) = \sqcup M$ 
]
```

We can now formulate the highest level of the proof represented by the following λ -abstraction being derived from the proof sketch given above:

```
⟨ Proof of the Knaster-Tarski fixpoint theorem 4.1 ⟩  $\equiv$ 
[ lemma1 := ⟨  $\sqcup M$  is atmost  $\Phi(\sqcup M)$ . ⟩ .:  $\sqcup M \sqsubseteq \Phi(\sqcup M)$ 
; lemma2 := ⟨  $\Phi(\sqcup M)$  belongs to  $M$ . ⟩ .:  $\Phi(\sqcup M) \in M$ 
; lemma3 := ⟨  $\Phi(\sqcup M)$  is atmost  $\sqcup M$ . ⟩ .:  $\Phi(\sqcup M) \sqsubseteq \sqcup M$ 
 $\vdash$  ⟨ QED ⟩ .:  $\Phi(\sqcup M) = \sqcup M$ 
]
```

Fig. 2 shows the corresponding “proof obligation window” of the Devil system representing the above proof obligation¹. The state attribute signals, that the proof is still incomplete; the proof obligations it depends on are given by the references to DevaWeb sections contained in the above proof. They are listed in a subwindow on the right side. Although the state of the proof obligation is still

¹ Note, that the formulas are presented in L^AT_EX source code, as it forms the input to DevaWeb. We are aware that this is not a perfect solution, so we are working on a WYSIWYG representation of formulas within the system.

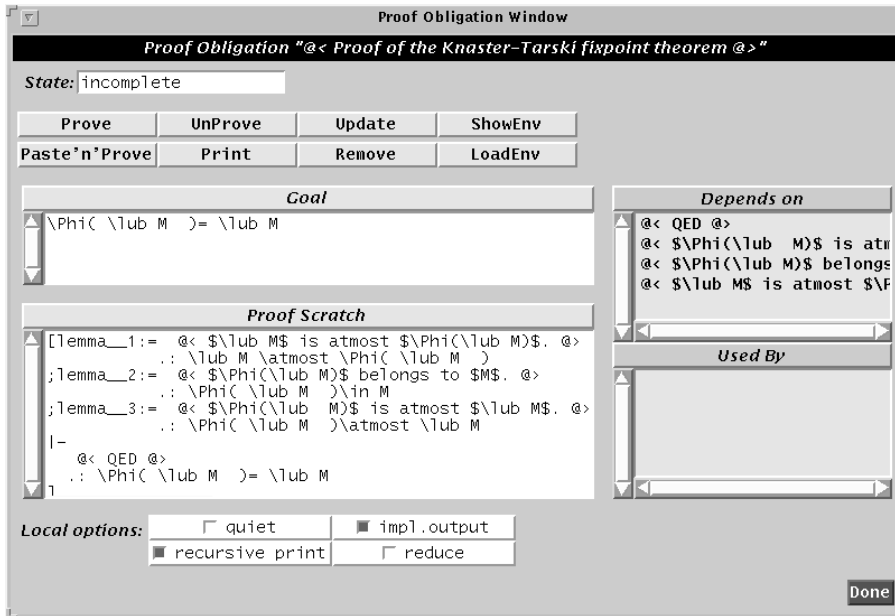


Fig. 2. The proof obligation representing the highest level of the proof.

“incomplete”, one could stop here if one were only interested in this level of the proof (either as the author or as a reader of the proof). Devil doesn’t force you to go down to the details, but, of course, the proof remains incomplete until all proof obligations it depends on are removed.

The remaining proof obligations can be removed by adding proof terms to the corresponding objects so that each of them will finally reach the state “proven”. In order to open new proof obligation windows, one can, for instance, click at the corresponding entry in the “Depends on” subwindow. You can now insert proof terms into the corresponding “Proof Scratch” windows and check these terms using the “Prove” button. If the proof is correct the “State” entry changes accordingly either to “incomplete” or to “proven” depending on whether the proof term contains references to other incomplete proof obligation objects or not. Fig. 3 shows a snapshot of the proof development; in the background the main window of Devil is displayed. The $\text{T}_{\text{E}}\text{X}$ -previewer window at the bottom shows part of a document that has been generated with the “Print” button of the toplevel proof obligation window shown in Fig. 2. This functionality reflects the integration of the presentation support into the deduction support provided by Devil since a proof obligation is at the same time a DevaWeb section.

5 Related approaches

As we have already pointed out, the field of computer aided formal reasoning is quite diverse. We want to try to give a brief summary of approaches which

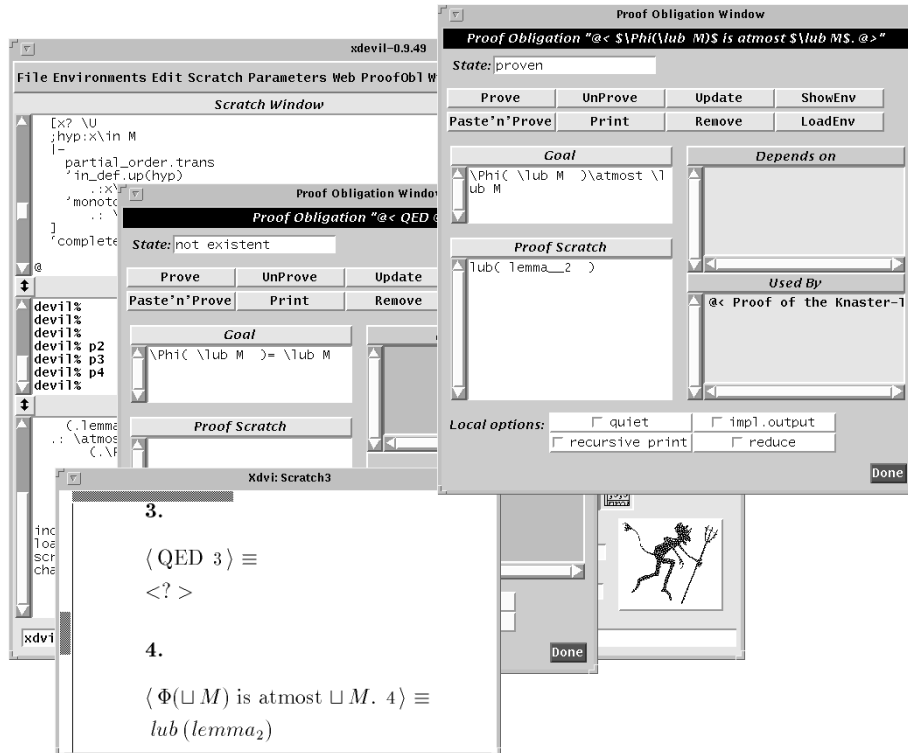


Fig. 3. Snapshot of the proof development

are related to our work. The endeavor to build tool support for formal mathematics was perhaps started by the Dutch AUTOMATH project. De Bruijn gives a survey of the project in [8] where he states the original motivation for the project as follows: “Develop a system of writing entire mathematical theories in such a precise fashion that verification of the correctness can be carried out by formal operations on the text.” During the project a family of languages for formalizing mathematics was developed and several case studies were performed, most notably the formalization of Landau’s “Grundlagen der Analysis”. The basic paradigm underlying this approach was the Curry-Howard isomorphism between propositions and types (cf. [22]). The AUTOMATH languages spawned a number of research activities which also adhere to this paradigm and which are actively pursued today. Based on typed lambda-calculi are the calculus of constructions [11], the extended calculus of construction ECC [32], and the Edinburgh logical framework LF [20]. The Coq system [14] is a proof assistant implementing the calculus of construction and the LEGO system [33] implements LF and ECC. Based on constructive type theory are Nuprl [10] and ALF [12]. A different approach, based on the implementation of higher-order logic, led to the development of HOL [17], IMPS [15], and Isabelle [40]. These formal systems

are being studied in a basic research action on logical frameworks. A survey of this project is given in [25, 26]. There also is an attempt underway to join forces from all areas of automated reasoning to build a comprehensive and sophisticated environment to support formal mathematics [41].

All of the previously mentioned approaches have a strong orientation towards the *construction* of correct formal proofs and the design of their support systems is accordingly. With the notable exception of the IMPS system not much attention is paid to the intelligible presentation of formal reasoning even though the need to do so is acknowledged. The case studies undertaken with these systems (e.g. [24] with Coq, [3] with LEGO, and [37] with Isabelle) thus rather stress aspects of formalizing logics and theories in a formal system and on techniques of interactive proof construction.

We also want to mention some systems which, similar to Deva, were specifically designed to provide mechanical support for formal methods. With these tools, too, the emphasis of the support is on construction. The B-tool is a rule-based inference engine with rule-rewriting and pattern-matching facilities [2]. The mural system is a formal development support system consisting of a generic reasoning environment and a support facility for VDM [27]. The prototype verification system PVS [39] is based on a higher order logic and consists of a specification language integrated with support tools and a theorem prover.

A completely different aspect of supporting formal reasoning is stressed by so called mathematical editors. They feature a flexible and comfortable environment for editing mathematical derivations without enforcing correctness. Notable examples of such systems are MathSPad [6] or Proxac [49]. Further aspects of user interface design for automated reasoning systems are investigated in [50].

Interesting experiments on interfacing automated reasoning systems with computer algebra systems are reported on in [21] and [9].

6 Conclusion and future work

We have reported on a set of tools which are being designed to support the creation and the intelligible presentation of formal reasoning. This ultimate goal has certainly not been achieved but this can be seen as an intermediate step towards it. Several successful case studies have been undertaken and have been completely developed with the assistance of the system. On the other hand, Deva is certainly not the final answer. The experiments uncovered a number of problems inherent in Deva. The primary lesson we learned is that specialized automated deduction support, such as rewriting or proof tactics tuned for a particular theory, is required to keep the amount of formal noise low, in particular in the case of calculational-style reasoning. Rewriting is supported in a limited way and the tacticals provided are not very expressive. A second problem is the notion of context in Deva which is used to structure theories and developments. Contexts are semantically difficult to handle and they do not provide the flexibility and expressiveness of other approaches to structuring and modularization of formal theories.

In Sect. 2, we presented a rough sketch of the software architecture of support systems for mathematical reasoning. We are currently in the process of refining this architecture. First ideas towards a formal basis to express deductions are presented in [48]. The basic issue is to transfer know-how from software design to proof design. At the local level, we want to determine useful abstraction and expression means. At the global level, we want to determine useful structuration means. We want to investigate further how to automate the transition between higher abstraction levels and lower ones. Here, reasonable compromises have to be found so that formal mathematical proofs remain intelligible yet can be checked automatically without requiring technical interaction with the user.

In order to keep the tool support flexible and extensible we propose a layered design: the deduction kernel, internal extensions, external extensions, theory management, presentation support, and user interface. Internal extensions are meant to extend the kernel directly, for example by a new logic, a new calculus, or a new theory together with specific deduction support. For instance, one can think of specialized rewriting algorithms or specialized proof procedures. External extensions will interface the core system with external deduction tools such as computer algebra systems, or model checkers. The literate programming paradigm has proved to be naturally applicable to the presentation of formal deductions. In particular, it is an adequate means of realizing the presentation of proofs in levels of abstraction as Lamport proposes. However, the relationship between the different levels should be reflected more clearly by some operation of the formal basis, e.g., by introducing a notion of “proof refinement”. The current realization by macro expansion is only partly satisfying. One further interesting topic for future research currently being investigated is how one can exploit the hypertext paradigm for the creation and presentation of structured proofs. Integrating hypertext technology with the mathematical editors mentioned in the previous section should provide a very attractive user interface for the development of formal deductions.

Acknowledgements

We would like to thank Thomas Santen, Michel Sintzoff, and Matthias Weber for their detailed comments on an initial draft of this report.

References

1. S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors. *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.
2. J. R. Abrial. The B-Tool (Abstract). In R. Bloomfield, L. Marshall, and C. Jones, editors, *VDM'88 - The Way Ahead*. Springer-Verlag, 1988.
3. T. Altenkirch. A formalization of the strong normalization proof for sytem F in LEGO. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 13–28. Springer-Verlag, 1993.
4. M. Anlauff. Devil: Deva's Interactive Laboratory. Tutorial and User Manual. Technical Report 93-42, TU Berlin, 1993.

5. R. C. Backhouse. Making formality work for us. *Bulletin of the EATCS*, (38):219–249, June 1989.
6. R. C. Backhouse, R. Verhoeven, and O. Weber. MathSPad user manual. Technical report, Technical University of Eindhoven, Department of Computer Science, 1994.
7. M. Biersack, R. Raschke, and M. Simons. The DevaWEB system: Introduction, tutorial, user manual, and implementation. Technical Report 93-39, TU Berlin, 1993.
8. N. G. de Bruijn. A survey of the project AUTOMATH. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism* [44], pages 579–606.
9. E. Clarke and X. Zhao. Analytica — An experiment in combining theorem proving and symbolic computation. Technical Report CMU-CS-92-147, Carnegie Mellon University, 1992.
10. R. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice Hall, 1986.
11. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.
12. T. Coquand, B. Nordström, J. M. Smith, and B. von Sydow. Type theory and programming. *Bulletin of the EATCS*, (52):203–228, 1994.
13. E. W. Dijkstra and C. Scholten. *Predicate Calculus and Predicate Transformers*. Springer-Verlag, 1990.
14. G. Dowek et al. The Coq proof assistant user's guide. Technical report, INRIA Rocquencourt, 1991.
15. W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11:213–248, 1993.
16. A. J. M. van Gasteren. *On the Shape of Mathematical Arguments*, volume 441 of *LNCS*. Springer-Verlag, 1990.
17. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.
18. D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag, 1993.
19. P. de Groote. *Définition et Propriétés d'un Métacalcul de Représentation de Théories*. PhD thesis, University of Louvain, 1990.
20. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.
21. J. Harrison and L. Théry. Extending the HOL theorem prover with a computer algebra system to reason about the reals. University of Cambridge Computer Laboratory.
22. W. A. Howard. The formulae-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism* [44], pages 479–490.
23. G. Huet. A unification algorithm for typed λ -calculus. *Theoretical Computer Science*, 1:27–57, 1975.
24. G. Huet. Residual theory in λ -calculus: A formal development. Technical Report 2009, INRIA, 1993.
25. G. Huet and G. Plotkin, editors. *Logical Frameworks*. Cambridge University Press, 1991.
26. G. Huet and G. Plotkin, editors. *Logical Environments*. Cambridge University Press, 1993.
27. C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *Mural: A Formal Development Support System*. Springer, 1991.
28. J. W. Klop. Term rewriting systems. In Abramsky et al. [1], pages 1–116.
29. D. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

30. D. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992.
31. L. Lamport. How to write a proof. Technical Report 94, DEC Systems Research Center, 1993.
32. Z. Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.
33. Z. Luo and R. Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, LFCS, 1992.
34. B. Möller, H. A. Partsch, and S. A. Schumann, editors. *Formal Program Development*, volume 755 of *LNCS*. Springer-Verlag, 1993.
35. C. Morgan. The refinement calculus, and literate developments. In Möller et al. [34], pages 161–182.
36. R. P. Nederpelt. *Strong Normalization in a typed lambda calculus with lambda structured types*. PhD thesis, Technical University of Eindhoven, 1973.
37. P. A. J. Noel. Experimenting with Isabelle in ZF set theory. *Journal of Automated Reasoning*, 10(1):15–58, 1993.
38. John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.
39. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Automated Deduction — CADE-11*, volume 607 of *LNCS*, pages 567–581. Springer-Verlag, 1992.
40. L. C. Paulson. Designing a theorem prover. In Abramsky et al. [1], pages 415–475.
41. The QED manifesto. Can be obtained from `info.mcs.anl.gov` in the directory `/pub/qed`.
42. T. Santen. Formalization of the SPECTRUM methodology in Deva: Signature and logical calculus. Technical Report 93-04, TU Berlin, 1993.
43. F. W. Schroer. Gentle. In J. Grosch, F. W. Schroer, and W. M. Waite, editors, *Three Compiler Specifications*, number 166 in GMD-Studien, pages 31–36. GMD, Forschungsstelle an der Universität Karlsruhe, August 1989.
44. J. P. Seldin and J. R. Hindley. *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.
45. J. H. Siekmann. Unification theory. *Journal of Symbolic Computation*, 7:207–274, 1989.
46. M. Simons, M. Biersack, and R. Raschke. Literate and structured presentation of formal proofs. In E.-R. Olderog, editor, *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET'94)*. North Holland, 1994.
47. M. Sintzoff. Understanding and expressing software construction. In P. Pepper, editor, *Program Transformations and Programming Environments*, pages 169–180. Springer-Verlag, 1980.
48. M. Sintzoff. Endomorphic typing. In Möller et al. [34], pages 305–323.
49. J. L. A. van de Snepscheut. Proxac: an editor for program transformation. Technical Report Caltech-CS-TR-93-33, California Institute of Technology, 1993.
50. L. Théry, Y. Bertot, and G. Kahn. Real theorem provers deserve real user-interfaces. *ACM Software Engineering Notes*, 17(5), 1992. Fifth ACM SIGSOFT Symposium on Software Development Environments.
51. M. Weber. Deriving transitivity of VDM reification in the Deva meta-calculus. In S. Prehn and W. J. Toetenel, editors, *VDM'91 Formal Software Development Methods*, volume 551 of *LNCS*, pages 406–427. Springer, 1991.
52. M. Weber. Definition and basic properties of the Deva meta-calculus. *Formal Aspects of Computing*, 5:391–431, 1993.
53. M. Weber. Literate mathematical development of a revision management system. In T. Denvir, editor, *Formal Methods Europe 1994*. Springer-Verlag, 1994.
54. M. Weber, M. Simons, and Ch. Lafontaine. *The Generic Development Language Deva: Presentation and Case Studies*, volume 738 of *LNCS*. Springer-Verlag, 1993.