Structured Presentation of Formal Proofs

Experiments with Isabelle

F. Kammüller, G. Keller, M. Simons, M. Weber

Technische Universität Berlin[†]

1 Introduction

The intelligible presentation of formal proofs is usually not attempted because of their technical detail. This formal noise hides the line of reasoning that can be followed and understood by humans. We are investigating methodologies and machine support for presenting formal proofs in an intelligible and structured manner while keeping them amenable to a check by a machine or an interactive development. To this end, we have in the past carried out sizable experiments with an implementation of the logical framework Deva, a descendant of the AUTOMATH family of languages [WSL93, SBR94, AJS94, Web94]. The experiments were carried out in the context of formal software development as well as mathematical proofs.

Deva is a λ-calculus with λ-structured dependent types. Its tool support consists of the Devil-system [Anl95], an interactive laboratory for developing Deva formalizations, and the DevaWeb system [BRS93]. Readability of Deva texts is enhanced by a diverse syntax and an implicit level of description which allows to leave gaps in formal developments. The Deva developments can be checked for type correctness by the Deva checker which translates the implicit notation to the explicit kernel language of Deva during the so called explanation process. The Deva checker is integrated with the Web system for Deva. It supports literate formalization in the sense of Knuth [Knu84] which combines structured documentation in LaTeX-form consistently with the development of the formal part.

Thus, the Deva development environment realizes basic aspects of our view of proof development. The proof development consists of writing literate proofs which are checked for correctness by the machine. The implicit level yields the possibility to abstract, leave out the technical details, the formal noise.

In retrospect, Deva turned out to be—though appropriate as an experimental prototype—too inflexible.

- The module mechanism was internalized into the meta-logic. This lead to a significant increase in time and storage demands of the Deva checker making it very laborious to scale up the proof experiments.
- Deva has no powerful notion of tacticals (tacticals are also internalized functions which cannot be used explicitly). Instead theories and proofs are formulated in one language.

[†]Technische Universität Berlin, Forschungsgruppe Softwaretechnik (FR5-6), Franklinstr. 28/29, D-10587 Berlin, Germany. e-mail: {zopotrum,keller,simons,we}@cs.tu-berlin.de

- There is no direct possibility to connect the Deva system to other support tools which are more appropriate for special tasks.

Since the whole weight of checking a Deva text lies on the translation of the implicit notation to the explicit kernel language of Deva the framework was not capable to manage realistically based case studies in an adequate manner. For example in the project KORSO we tried to adapt Deva as a framework for correctness management of an algebraic specification language [SKJB95]. On the one hand, this experiment demonstrated that complex formalizations could in principle be modeled by Deva's implicit level and remain intelligible. On the other hand, the proof obligations were too difficult for Deva's simple proof engine.

We think that the various concepts that underlie the specific approach we chose for the presentation of formal proofs in Deva are of a general nature. We are currently trying to apply this approach to proofs expressed in other logical frameworks, Isabelle in particular, and we continue to investigate theoretical aspects of expressing formal proofs. More specifically:

- We are experimenting with notations and calculi for expressing proofs that try to capture algebraic properties underlying proof construction and that allow to express proof refinements. The guiding idea here is to view theorem proving as the process of refining a theorem to its proof. Each refinement step adds more technical detail to the overall proof. With respect to proof presentation, only the first levels of refinements are of interest to a reader. The other refinements can safely be hidden and carried out interactively or completely automatic. Early investigations of tentative calculi uncovered a close relationship with algebraic semantics of substructural logics [DSH93], e.g., full Lambek algebras (FL-Algebras).

- We investigate prototypical implementations of such calculi with "programmable" logical-frameworks such as Isabelle or λProlog as a basis. A first experiment is described in [Kam95] where FL-Algebras are formalized on top of Isabelle's Pure theory and proof constructions are expressed as FL-Algebra terms. Tactics are designed to automatically prove „validity-properties". The advantage of this approach to constructing prototypical implementations is that one can make use of the structural facilities provided by the logical framework without having to develop them from scratch.

- We are developing prototypical tools that aid in the generation of structured and literate Isabelle proofs in order to demonstrate the viability of our approach to the presentation of formal proofs. One tool [Kel94] allows calculational proofs to be easily performed with Isabelle through an Emacs interface. Some requirements for tool-support that allows hierarchically structured proofs to be interactively developed and documented with Isabelle are investigated in [Kel95].

In the following section, we will present a summary of [Kam95] which includes an introduction to our theoretical investigations towards hierarchical proof objects. In the third section we give an overview of [Kel95, Kel94], our experiments with prototypical tool support for literate structured Isabelle proofs.

2 Prototyping an Algebraic Framework for Hierarchical Proof Objects in Isabelle

We call a language for an explicit construction of proofs a *proof programming language*. We first state some requirements for such a language and its associated calculus by answering the question: *what is a proof in our sense?*

- A proof may be a sequence of transformations possibly in nested form. An essential part of the transformation is nevertheless an atomic transformation which we might symbolize by $\mapsto$.
- In human reasoning, theorems are an abstracted, condensed form of solutions to problems, i.e., a condensed form of evidence or proof, respectively. Thus the development of theorems corresponds to a stepwise refinement of proofs into abstract propositions, i.e., if we symbolize a refinement relation as $\sqsubseteq$ this informally sketched relation is $proof \sqsubseteq proposition$. The guiding idea here is to view theorem proving as the process of refining a theorem to its proof. Each refinement step adds more technical detail to the overall proof.
- In proof constructions various parts are basically an enumeration of already existing or assumed terms. I.e., the composition of sequences—let us denote it by $\odot$—is also an inherent part in the construction of proofs and should thereby be part of our proof programming language.

It remains to make precise the interrelationship between these constructions, e.g., one question is how do $\odot$ and $\mapsto$ correspond.

Viewing the refinement of proofs as the top level of proof construction we are reminded of a lattice structure. A composition is found quite abstractly in monoids. If we consider a combination of these as a structural foundation for our calculus we are strongly reminded of substructural logics.

From these first informal ideas we decided to experiment with structures which offer the necessary conditions. An interesting structure is that of *quantales*, an algebraic structure which actually combines lattices and monoids.

2.1 An Algebraic Framework for Proof Composition

If we view the laws of deduction of a proof calculus merely from a structural viewpoint we gain *algebraic* laws of deduction. These laws can be crystallized into an algebraic structure. The algebraic structure serves with its structural abilities to establish our goal of structured proofs.

2.1.1 *FL*-Algebras. A special case of quantales are full Lambek Algebras, abbreviated FL-Algebras. They build the current focus of our considerations. As specialization of quantales the FL-Algebras are a combination of lattices and monoids. In addition to quantales the map operation $\mapsto$ of mappings between algebras is made explicit, i.e., is part of the algebra operations. Consequently, the question occurs how this explicit map operation behaves in relation to the other operators, i.e., the operators which come with the lattice and the monoid. In the case of FL-Algebras the so called maplet is *Galois-connected* to the monoid composition $\odot$, i.e., $a \odot a \mapsto b$ contracts to b. A mathematical description of complete FL-Algebras is given in Figure 1.

(i) $\langle A, \sqcup, \sqcap, \top, \bot \rangle$ is a complete lattice with least element $\bot$ and the greatest element $\top$ for which $\top = \bot \mapsto \bot$ holds,
(ii) $\langle A, \odot, \mathbf{1} \rangle$ is a monoid with the identity $\mathbf{1}$,
(iii) $y \odot (\sqcup i \cdot x_i) \odot z = (\sqcup i \cdot y \odot x_i \odot z)$, for every $x_i, y, z \in A$,
(iv) $x \odot y \sqsubseteq z$ iff $y \sqsubseteq x \mapsto z$, for every $x, y, z \in A$,
(v) $\underline{0} \in A$.

Figure 1: Complete FL-Algebra $\mathbf{A} = \langle A, \mapsto, \sqcup, \sqcap, \odot, \mathbf{1}, \underline{0}, \top, \bot \rangle$

2.1.2 *FL*-Algebras as a Proof Programming Language. To sketch the interpretation of this algebraic structure as a proof programming language we already used the same symbols for the enumeration of the requirements at the beginning of this Section. Concretely, we interpret the FL-Algebras as a proof programming language in the following way:

- The order relation $\sqsubseteq$ which comes with the lattice part of the FL-Algebras is interpreted as the proof refinement relation, i.e., a kind of meta-inference constructor.
- We view a proposition as proved if there exists a fully elaborated proof which is "less than" the proposition, i.e $proof \sqsubseteq proposition$. The notation of "fully elaborated" is defined as validity in terms of the monoid element $\mathbf{1}$ (cf. Section 2.1.3).
- The explicit map $\mapsto$ is viewed as an internal implication between proof terms.
- The monoid composition is used to combine proof terms into sequences, e.g., to sample premises used in a proof on one level.
- The interconnection of the ordering with join ($\sqcup$) and meet ($\sqcap$) as

  $$p \sqcap q \sqsubseteq q \qquad \text{and} \qquad q \sqsubseteq p \sqcup q$$

  gives rise to the interpretation of $\sqcap$ as logical *and* and $\sqcup$ as logical *or*.
- To introduce quantifying in our language calculus we use the generalized join or meet. Since

  $$\sqcap i \in I \cdot x_i \sqsubseteq x_j \qquad \text{and} \qquad x_j \sqsubseteq \sqcup i \in I \cdot x_i, \quad \text{for all } j \in I$$

  an interpretation of $\sqcup$ as universal quantifier and $\sqcap$ as existential quantifier seems appropriate.
- Furthermore, we extend the syntax of the FL-Algebras by an operator $*$ which stands for a Kleene-star-like iteration of proof-terms and a judgment $\succ$.

For an overview over the firm parts of our language consider the EBNF grammar rule depicted in Figure 2. There, the nonterminal E stands for the terms of the treated object, X is a set of variables and P are the terms of the proof programming language.

$$
\begin{array}{rcl}
P & ::= & E|X|(P) \\
 & & |\top|P \sqcap P| \sqcap X \cdot P \\
 & & |\bot|P \sqcup P| \sqcup X \cdot P \\
 & & |\mathbf{1}|P \odot P|P^*|P \mapsto P|P \succ P
\end{array}
$$

Figure 2: Syntax of the Proof Programming Language

2.1.3 Validity. As already mentioned in point two of the informal interpretation the validity is defined in terms of the ordering and the monoid element $\mathbf{1}$. We view fully elaborated proofs as logically valid and define a validity for GRAL-terms which is semantically based on FL_{ecw}-algebras. The latter are an extension of FL-algebras by three additional axioms: e for *commutativity*, c for *idempotency*, and w for $weakening$ of terms composed by $\odot$ *wrt.* the relation $\sqsubseteq$.

$$
\begin{array}{lccl}
(e) & x \odot y & \sqsubseteq & y \odot x \\
(c) & x & \sqsubseteq & x \odot x \\
(w) & x \odot y & \sqsubseteq & x
\end{array}
$$

These additional axioms incorporate the intuitively admissible mechanisms of arbitrary order, repetitive use, and omission of premises composed into a sequence by $\odot$:

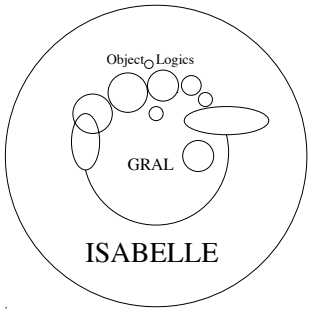
A term p is valid in the proof programming language iff $\mathbf{1} \sqsubseteq_{ecw} [\![p]\!]$.

Since the definition of validity might be difficult to check mechanically—e.g., $\mathbf{1} \odot \mathbf{1} = \mathbf{1}$—a syntactical characterization of validity is offered. This notation of *well-formedness* characterizes the validity in an inductive way. It defines the validity inductively over the structure of proof terms. E.g., a term $p \sqcap q$ is *well formed* under a set of premises S iff p and q are *well formed* under the set S. A term is generally valid iff it is well formed under the empty set (cf. Section 2.2.2).

To experiment with the calculus of FL-Algebras on a pragmatic level we decided to implement it in Isabelle in a way that practical case studies as well as validation of basic features of the algebra and its interpretation would be possible. Aside from this aim we liked to get some experience with Isabelle since we view it also as an very appropriate tool to be integrated as a main support in a future global system.

2.2 Prototyping FL-Algebras in Isabelle

Generally, the experiment tried to use Isabelle Pure as a logical frame and to specify the proof programming language based on FL-Algebras — we named it GRAL— using the means provided by Isabelle. GRAL is on the one hand viewed as an Isabelle object and on the other hand the integration of GRAL into Isabelle is viewed as a frame for our GRAL case studies, at least in a prototypical way. Thereby, we can actually verify properties concerning GRAL as an object and, in the same framework, perform initial case studies.



2.2.1 Type Configuration. If a logic like FOL is implemented in Isabelle the type structure of the object consists of one base type of propositions which is usually entailed in the type class `logic` of all object logics. In the case of GRAL there is no fixed logical domain because in our interpretation FL-Algebras do not constitute a logic in the classical sense instead something like a meta-logic. Terms of concern are arbitrarily typed. Thus, we need a type construction which first of all embeds terms of concern, i.e., GRAL object logics, into the meta-logic thereby allowing to treat them by the constructors of GRAL. The GRAL formulas itself have to be embedded into Isabelle's meta-logic Pure to make them accessible to Isabelle-reasoning. The latter map is traditionally the implicit coercion `Trueprop`.

$$\text{GRAL objects} \xrightarrow{\phi} \text{GRAL} \xrightarrow{\texttt{Trueprop}} \texttt{Pure}$$

The first map ϕ should—like `Trueprop`—be invisible to make case studies with GRAL objects legible. I.e., it should be a second implicit coercion `Trueprop`$^{-1}$. Unfortunately, this does not work practically because in the most cases the solution of the two invisible occurrences of the `Trueprop`'s could not be performed[1]. So, we chose another way of implementing GRAL which represents a slightly different view at our proof programming language. We replaced the first map ϕ by an inverse map, more precisely an instantiation:

$$\text{GRAL objects} \xleftarrow{instantiation} \text{GRAL} \xrightarrow{\texttt{Trueprop}} \texttt{Pure}$$

Concretely, we defined GRAL as a polymorphic type. By defining a type class `A` of object logics and all GRAL constructors as polymorphic functions over this type class we attained the possibility to use GRAL as a skeleton structure for application objects. In this case a GRAL object is an enrichment of the instantiation of the polymorphic meta-language. Some aspects of the global type structure are depicted in Figure 3.

Now, it was possible to show properties of FL-Algebras by introducing a first member of the type class `A`. We named it `Bool` to signal the logical meaning. Just by declaring `Bool` as a type in `A`, by `Bool :: A`, all polymorphic constructors and axioms of GRAL are instantiated by this type. There are no additional constructors or axioms for the type `Bool` because it represents the "pure" GRAL calculus. Since the general logical functions of GRAL, e.g., $\sqsubseteq$, map into `Bool` all related properties derived for the latter type are applicable to future types in `A`.

[1] Probably, it is generally not effectively solvable in the presence of unknowns.

```
gral = Pure +
classes A < logic
default A
types Bool 0
      ...
arities Bool :: A
        ...
consts
  Trueprop   :: "Bool => prop"    ("(_)" 5)
  ...
  atmost     :: " [ 'a, 'a ] => Bool" ("(_ [= _)" [ 50, 51] 50)
  ...
  maplet           :: " [ 'a, 'a ] => 'a" ("(_ -> _)" [ 61, 60] 60)
  composition      :: " [ 'a, 'a ] => 'a" ("(_ + _)"  [ 70, 71] 70)
  join             :: " [ 'a, 'a ] => 'a" ("(_ | _)"  [ 80, 81] 80)
  meet             :: " [ 'a, 'a ] => 'a" ("(_ & _)"  [ 80, 81] 80)
  ...
```

Figure 3: Global Type Structure of GRAL

2.2.2 Basic Derivations. With this basic type constitution at hand it is possible to prove properties of GRAL based on some basic axioms. We stated only these basic axioms and derived all other properties which were more suitable for our view of GRAL as a proof programming language. E.g., the order of the lattice is classically defined by referring to equality:

$$a \sqcup b = b \Leftrightarrow a \sqsubseteq b \Leftrightarrow a = b \sqcap a$$

For the use as a proof programming language which is based on refinement by $\sqsubseteq$ properties like $a \sqsubseteq a \sqcup b$ or $a \sqcap b \sqsubseteq b$ are more suitable. Since the latter are entailed in the former definition we derived them. This serves on the one hand to validate that the current approach actually works and on the other hand to keep the number of axioms minimal and thereby increase consistency.

For the definition of the intuitive term "fully elaborated proof" we did not use the characterization of validity by FL_{ecw}-Algebras instead the alternative syntactical characterization of well-formedness from [Sin94].

The translation of the well-formedness into Isabelle leads to a polymorphic predicate, i.e., a `Bool`-valued function `WF(S,p)` of type `['a Set, 'a] => Bool`. The latter necessitates the specification of sets. For some aspects of the syntactical characterization see Figure 4.

Now, for GRAL objects a proposition is true iff it is well formed under the empty set. Thereby, we attain a deduction principle which characterizes validity of a GRAL term. For a proposition p in question we have to find a refinement q with $q \sqsubseteq p$ and q well formed under the empty set. This global principle is entailed in the transitivity of $\sqsubseteq$. The corresponding theorem reads in Isabelle notation:

```
[| WF(S, q); q [= p |] ==> WF(S, p)
```

Sticking to this derived deduction principle we can establish a proof style which consists of two main steps:

```
ecw2   "p elem S ==> WF(S, p)"
ecw3   "WF(p : S, q) ==> WF(S, p -> q)"

ecw4a  "[| WF(S, p); WF(S, q)|] ==> WF(S, p & q)"
ecw4b  "[| WF(S, p); WF(S, q)|] ==> WF(S, p + q)"

ecw6a  "WF(S, p) ==> WF(S, p | q)"
ecw6b  "WF(S, p) ==> WF(S, q | p)"
```

Figure 4: Some Aspects of the Formalization of Well-Formedness

- The derivation of the well-formedness of the elaborated proof
- The refinement of the proof into the proposition, i.e., $proof \sqsubseteq proposition$

In point one, the definition of validity by `WF` often leads to lengthy derivations of validity which are obvious for the user but need to be shown in the calculus. Since well-formedness is inductively defined it is possible to solve this task effectively by the definition of a tactical which iteratively resolves with the basic axioms of well-formedness[2].

2.2.3 Application to Objects. Aside from the formal treatment of the theory we can apply GRAL to object logics. We can define GRAL object logics as members of the type class `A`. Thereby, the whole theory is instantiated by a concrete example and the already derived facilities of GRAL may be used to reason about properties of the object logic.

2.2.4 Minimal Logic Example. As a first simple case study we considered minimal logic. To give a taste of the way of proving we achieved consider some aspects of this example. By using Isabelle's declarative style of specification we could define the signature of a minimal logic in GRAL as:

```
MinLog = gral +
types prol 0
arities prol :: A
consts
   impl :: "[prol, prol] => prol" ("(_ =>> _)" [ 30, 31] 30)
   in   :: "prol"
   out  :: "prol"
```

where `A` is the GRAL type class for object logics. The terms `in` and `out` are the names of the rules we defined for the introduction and elimination of the minimal logics implication `impl` or `=>>`, respectively.

```
rules
  in_def "in = (a -> b) -> ( a =>> b)"
  out_def "out = (a =>> b) -> (a -> b)"
```

[2]Here, a formalization on top of ZF would have granted the advantage of the coinduction package.

```
mono_meet1: a [= b ==> c & a [= c & b
mono_meet2: a [= b ==> a & c [= b & c
mono_meet: [|a [= b; c [= d |] ==> a & c [= b & d
...
weaken1: a [= b ==> c -> a [= c -> b
weaken2: a [= b ==> b -> c [= a -> c
weaken: [|b [= a; c [= d|] ==> a -> c [= b -> d
```

Figure 5: Some Monotonicity Rules for the Constructors

To give a simple example of proving in GRAL we consider the refinement proof:

```
(p =>> q) + out [= p -> q
```

where `+` is the ASCII-representation of $\odot$ or the composition of GRAL proof terms, respectively, and `->` is the $\mapsto$ or maplet operation, respectively. The relation `[=` represents the order relation of the lattice which we interpret as proof refinement relation.

An expansion of the definition of `out` will change the goal into

```
(p =>> q) + (?a =>> ?b) -> (?a -> ?b) [= p -> q
```

Instantiation of `?a` by `p` and `?b` by `q`, respectively, makes the above goal an instance of the *contraction* definition $a \odot a \mapsto b \sqsubseteq b$ which is part of the FL-Algebra axioms (cf. Figure 1).

The above sketched proof can be supported by defining a general definition substitution tactical based on `rewrite_goals_tac` and `SELECT_GOAL` and by supporting the GRAL refinement by adaptation of basic properties of the order relation `[=` like transitivity.

Fortunately, the latter proof state in the present example is a direct instance of the contraction property. In general, substitution inside GRAL refinements is not admissible because not all constructors are monotonic. If we want to contract inside an arbitrary refinement context we first have to eliminate the context by resolution with the monotonicity (or antitonicity) properties of the GRAL constructors *wrt.* the ordering explicitly.

Therefore, we derived the monotonicity properties of the GRAL constructors and constructed them into a tactical named `subst_tac`. The latter allows to apply basic properties easily in refinement derivations. Some monotonicity properties are shown in Figure 5.

Finally, we could prove:

$$in \odot out \mapsto (x \Rightarrow (y \Rightarrow z) \Rightarrow (x \Rightarrow y \Rightarrow (x \Rightarrow z)))$$

as well formed under the empty set, i.e., as a valid GRAL term. The refinement proof could be performed by the predefined tacticals in only a few steps. The proof of well-formedness is performable by one application of the corresponding tactical. Thereby, we approximated the initially intended way of proving very closely.

2.2.5 Natural Numbers Example. Aside from the tacticals concerned with the basic properties of the GRAL calculus, e.g., well-formedness and contraction tacticals,

which are designed to approximate the top-level proof style as far as possible to the intended way we can also define individual tacticals for the object logics needs. For a theory of natural numbers we investigated a proof scheme of direct induction. The theory of natural numbers contains just some basic axioms:

$$\begin{aligned} a = a & \qquad (ReflEq) \\ 0 + a = a & \qquad (DefAddBase) \\ succ(a) + b = succ(a + b) & \qquad (DefAddRecur) \\ (P(0) \sqcup (\bigsqcup a \cdot P(a) \mapsto P(succ(a)))) \mapsto (\bigsqcup a \cdot P(a)) & \qquad (Induction) \end{aligned}$$

The equality used above is defined as a predicate over natural numbers and substitution is described by $Unfold$:

$$(a = b) \mapsto (c = F(a)) \mapsto (c = F(b))$$

and vice versa by $Fold$.

The iterated use of the laws of addition can be defined by modeling the repetition with the Kleene-star-like operator $*$ (cf. Figure 2):

$$DefOfAddition \widehat{=} ((DefAddBase \sqcap DefAddRecur) \odot (Unfold \sqcap Fold))^*$$

The direct induction scheme is a GRAL proof scheme which reads in the informal notation:

$$\begin{aligned} & DirectInduction(Ruleset) \\ & \widehat{=} (Base \sqcup (\bigsqcup n \cdot F(n) = G(n) \mapsto Step)) \odot Induction \end{aligned}$$

where $Base$ and $Step$ are the following equational reasonings:

$$\begin{array}{lcll} Base & \widehat{=} & F(0) & \\ & = & & \{Ruleset\} \\ & & G(0) & \\ \\ Step & \widehat{=} & F(succ(n)) & \\ & = & & \{Ruleset\} \\ & & H(F(n)) & \\ & = & & \{Unfold(F(n) = G(n))\} \\ & & H(G(n)) & \\ & = & & \{Ruleset\} \\ & & G(succ(n)) & \end{array}$$

As an example we can prove that zero is a right identity of addition, i.e.:

$$Proposition \widehat{=} (\bigsqcup n \cdot n + 0 = n)$$

The direct induction scheme applied to the rule set $DefOfAddition$ may be refined

to:

$$
\begin{array}{l}
\sqcup(\bigsqcup n \cdot n + 0 = n \mapsto \\
\quad \left(\begin{array}{lll} & 0+0 & \\ = & & \{DefAddBase \odot Unfold\} \\ & 0 & \end{array}\right. \\
\quad \left.\begin{array}{lll} & succ(n)+0 & \\ = & & \{DefAddRecur \odot Unfold\} \\ & succ(n+0) & \\ = & & \{(n+0=n) \odot Unfold\} \\ & succ(n) & \\ = & & \{ReflEq \odot Unfold\} \\ & succ(n) & \end{array}\right) \\
\quad) \odot Induction
\end{array}
$$

which contracts to the proposition $(\bigsqcup n \cdot n + 0 = n)$.

How can we represent the direct induction scheme in the GRAL calculus?

```
rules
  DirIndDef  "( ⟨Base⟩ | (JO n. (F(n) =n G(n)) -> ⟨Step⟩ )) +Induction"
```

The annotation of the used equality transformation rules in curly brackets, $\{RuleSet\}$, is represented by `Ruleset` applied by `+` to instances of $ReflEq$. To integrate the goals of the transformation steps we use the judgment `>` or $\succ$, respectively. Thus, $Base$ is:

$\langle Base \rangle \equiv$

```
    ((F(0) =n F(0)) + Ruleset)       > (F(0) =n G(0))
```

The nested $=$ signs of the equality transformation cannot be presented directly by the means of the calculus. Instead we have to connect explicitly several single equalities by $\odot$, in the ASCII representation `+`, and contract them afterwards by double application of `Unfold`. Thus the $Step$ looks like:

$\langle Step \rangle \equiv$

```
    ( (((F(succ(n)) =n F(succ(n))) + Ruleset)
                                   > (F(succ(n)) =n H(F(n))))
    +
         (((H(F(n)) =n H(F(n)))    + (F(n) =n G(n)) + Unfold )
                                   >    (H(F(n)) =n H(G(n))))
    +
         (((H(G(n)) =n H(G(n)))    + Ruleset)
                                   >    (H(G(n)) =n G(succ(n))))
    + Unfold + Unfold)
```

The task for this case study is to show that

$$DirectInduction(DefOfAddition) \sqsubseteq (\bigsqcup n \cdot n + 0 = n)$$

performing as much as possible by general properties of the proof scheme. The proving problems we have to face in this case are mainly concerned with the instantiation of the generalized join $\bigsqcup$ or JO, respectively, and the isolation of those parts of the refinement which are nontrivial. For the former we could derive a lemma from the basic properties which allowed in collaboration with a tactical to instantiate the $\bigsqcup$-quantified part inside a refinement derivation. The latter could be solved by the construction of the high-level refinement proof of the direct induction scheme to the point where the refinement of the global scheme is reduced to those three partial refinements which build the kernel of the application of the direct induction scheme. By a further adaptation of the contraction tactical to$\bigsqcup$-quantified contexts the actual proof of $(\bigsqcup n \cdot n + 0 = n)$ could then be performed in a few steps:

- reduction to the kernel parts of the refinement by the high-level proof:

$$\begin{array}{lll} DefOfAddition & \sqsubseteq & 0+0=0+0 \mapsto 0+0=0 \\ DefOfAddition & \sqsubseteq & succ(n)+0=succ(n)+0 \mapsto succ(n)+0=H(n+0) \\ DefOfAddition & \sqsubseteq & H(n)=H(n) \mapsto H(n)=succ(n) \end{array}$$

- solution of these three kernel parts with support of the expanded contraction tactical.

- Finally, the proof of well-formedness of the fully-elaborated proof which emerges automatically by the stepwise instantiation of the scheme variables. This proof may be performed by the tactical for well-formedness.

In point two another proof obligation occurs: the refinement of $DefOfAddition$ to the concrete rule application sequences, e.g.:

$$DefOfAddition \sqsubseteq DefAddRecur \odot Unfold.$$

By an adaptation of a general tactical developed for the Kleene-star-like $*$ operator an individual tactical for the direct induction application is constructed. To validate that the adapted tacticals and lemmas for the direct induction scheme are not restricted to the special case $(\bigsqcup n \cdot n+0 = n)$ we performed a similar proof $(\bigsqcup n \cdot 1+n = n+1)$. It is far more complex because the explicit rule applications of $DefOfAddition$ are longer than one step. The additional derivation actually showed that another tactical for the insertion of intermediate steps in $\mapsto$ transformation rules is necessary to make proving comfortable but then succeeded too.

2.2.6 Results of the Experiment. Summarizing our experiences and results we can say that the first part of the experiment, the basic derivations, lead to problematic questions concerning the interpretation of the FL-Algebras as a proof programming language. The second part, the application to objects, showed at least that the language is in principle able to capture the intended proof style. Furthermore, the simulation of GRAL case studies gave hints to where the highest amount of structural work in proving is created. By some basic tacticals designed to solve this structural work sketches for future proof routines emerged. But, the main part of the structural proof obligations was itself an outcome of the high level of abstraction we had in our experiment.

The work with Isabelle was very comfortable and allowed to approximate our intentions very closely by simple constructions of tacticals. The main problem which

restricts the present experiment to small case studies is the syntactical representation of GRAL which grows too far to be represented in ASCII on a **ML**-prompt.

Drawing from the experiences with Deva and the literate development style with the DevaWeb system we processed the entire formalization with `noweb`, a general Web-tool for arbitrary formalisms. Thereby the documentation and the formal part are contained in one file which builds the origin for the Isabelle source and the LaTeX source in a consistent way. Since we want to draw the integration of literate programming facilities further than this we focus separately on the practical line which will be sketched in the remainder of this article.

3 Structured and Literate Presentation of Isabelle Proofs

Proofs derived with the aid of automatical proof support systems seem in general not to be well suited for presentation. The difficulties arising have two main sources. First, strictly formal proofs contain too much technical detail, which is of no interest to the human reader, who only wishes to understand the basic idea. This results in a long, overly detailed proof, in which the basic line of reasoning is obscured. Second, the representation of a computer aided proof is geared towards a form that is easy to parse for computers, which differs a lot from the form a human would choose in in order to understand it. This results in a lack of structural information.

Both points combined result in superfluous information on the one hand and the lack of helpful information on the other hand. But still, such a proof still contains a representation of the basic proof idea that was on the mind of the person conducting the proof. Thus, by hiding the unnecessary information and by providing additional information it should be possible to recover the proof idea.

In the case of Isabelle, the representation of a proof is the collection of the commands that apply rules and tacticals to a proof state until the initial theorem is proven valid. Even to someone well acquainted with Isabelle, this representation is barely meaningful—even in the case of proofs of moderate size.

Using the commands plus all the intermediate proof states as representation would not help too much for the two reasons described above: even for simple theorems, the presentation would be of considerable size, and the rules and tacticals applied have names only meaningful to those very familiar with Isabelle *and* the theory. The fact that the formulas are displayed in the severely restricted ASCII character set doesn't add to the comprehensibility either. Thus, we take this representation merely as a basis to derive step by step a proof document that is independent from the syntax of the system, well structured, and oriented at common proving styles.

In the following, we first describe the proofstyle that we use for the Isabelle proofs and explain how this style relates to the proof representation of Isabelle. We then describe shortly a prototypical tool which supports the generation of a comprehensible proof document from an existing Isabelle proof. A complete report on this experiment together with a literate and structured proof of a correctness argument taken from [KL95] is given in [Kel95].

$$\mathcal{S}(0) + \mathcal{S}(n) = \mathcal{S}(\mathcal{S}(n))$$

$\Longleftarrow$ $\{$ for all $x, \mathcal{S}(x) + y = \mathcal{S}(x + y)$ $\}$

$$\mathcal{S}(0 + \mathcal{S}(n)) = \mathcal{S}(\mathcal{S}(n))$$

$\Longleftarrow$ $\{$ for all $x, 0 + x = x$ $\}$

$$\mathcal{S}(\mathcal{S}(n)) = \mathcal{S}(\mathcal{S}(n))$$

$\Longleftarrow$ $\{$ reflexivity of $=$ $\}$

$\square$

Figure 6: A non-branching subproof.

3.1 Our Proof Format

The style we chose to display the proofs is a variation of the calculational proofing style described, e.g., in [DS90]. In this section, we introduce this style shortly, and then describe our variation of the style, which is used subsequently to display Isabelle proofs.

Given a proof of a formula $\Phi_1 \rhd \Phi_n$, where $\rhd$ is an arbitrary transitive relation, and where the proof consists of a collection of transformation steps $\Phi_i \rhd \Phi_{i+1}$, the overall proof is denoted in the following calculational form:

$$\begin{array}{ll} & \Phi_1 \\ \rhd & \{\text{explanation why } \Phi_1 \rhd \Phi_2 \text{ holds}\} \\ & \vdots \\ \rhd & \{\text{explanation why } \Phi_{n-1} \rhd \Phi_n \text{ holds}\} \\ & \Phi_n \end{array}$$

Enclosed in curly braces are comments justifying the single transformation steps.

Now, turning to Isabelle, a backward proof of a theorem Φ has the following form in the calculational proof style:

$$\frac{\Phi}{\Phi} \Longrightarrow \frac{\Phi_1, \ldots, \Phi_n}{\Phi} \Longrightarrow \frac{}{\Phi}$$

However, it would not be reasonable to display an Isabelle proof in the above form. All the proof states have the conclusion Φ—the initial theorem—in common, because it does not change during the proof. Therefore, it is not displayed in an interactive Isabelle session, and there is no reason to display it in the proof document, since even

We prove the validity of $P(x, y)$ by induction over x.

$$P(x, y)$$

$$\Longleftarrow \quad \{ \text{ Proof by induction over } x \quad \}$$

1. $\{Induction\ Base\}$

$$P(0, x)$$

2. $\{Induction\ Step\}$

$$\forall n.n < P(n, y) \Rightarrow P(\mathcal{S}(n), y)$$

Induction Base

$$P(0, x)$$

$$\vdots$$

Induction Step

$$\forall n.n < P(n, y) \Rightarrow P(\mathcal{S}(n), y)$$

$$\vdots$$

Figure 7: Skeleton of a proof by induction.

without the conclusion, a proof state quickly grows considerably large. Thus, an alternative, but equivalent, representation is the following:

$$\begin{array}{c} \Phi \\ \Longleftarrow \\ \Phi_1, \ldots, \Phi_n \\ \Longleftarrow \\ \textit{true} \end{array}$$

Even though this representation is shorter, it is still far from good, since the structure of the proof is not visible, i.e., the hierarchical dependence of the subgoals is hidden from the reader. To visualize the structure of the proof, we distinguish two cases in our style. First, consider a non-branching subproof, where each application of a rule results in exactly one new subgoal. Since such a proof has a linear structure, the calculational style is adequate here. As an example, consider the proof in Figure 6. It establishes the validity of $\mathcal{S}(0) + \mathcal{S}(n) = \mathcal{S}(\mathcal{S}(n))$ in Peano's theory of natural numbers.

The second case, namely a branch, is more complicated. It is not useful to see every subgoal of the current proofstate in every step of the proof, since it is easier to follow the different branches of the proof tree successively. Therefore, if the application of a rule leads to more than one new subgoal, all the new subgoals are displayed, but the proof of each new goal follows separately. If we want to prove the validity of a formula $P(x, y)$ by induction over x, then the application of the induction rule leads to two new subgoals, as depicted in Figure 7. After splitting the proof into the induc-

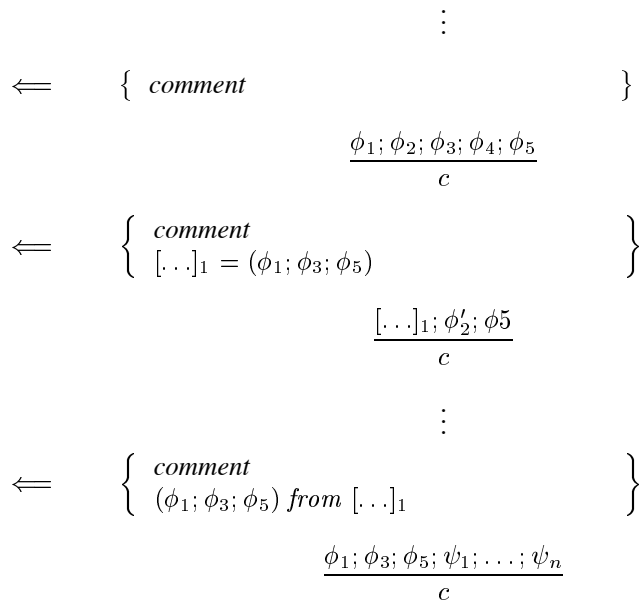


Figure 8: Hiding of unaltered subformulae.

tion base and the induction step, the subproofs are conducted individually—avoiding unnecessary clutter.

3.2 Making a Proof Comprehensible

The previously introduced proof style prevents that in each proof step the whole proof-state is visible; instead, only the current subgoal is displayed. However, often enough the subgoals grow rather large, which makes it hard to see on which formulae of the subgoal the last operation had an effect. If several successive proofsteps affect only a subset of the formulae of a goal, it is useful to hide the uneffected formulae.

In the proof fragment displayed in Figure 8, the formulae Φ_1, Φ_3, and Φ_5, which may be large, remain invariant during a subproof. This is used by abbreviating these formulae with the form $[\ldots]_1$—instead of listing them explicitly in the proof state.

One rather simple, but effective way to increase the readability of a proof is to combine several *technical* proof steps into a single *logical* proof step. For example, it is often obvious that a certain rule is applicable, but the proof state has not exactly the right form and has to be fixed by applying appropriate technical rules, such as commutativity, associativity. The number of those technical steps can be reduced by using tacticals, but still numerous will be left. Those steps should not be visible in the final proof document since they do not add information that is valuable to a human. There is, however, no way to decide automatically which steps are trivial and which are of interest within a given context, because this depends on the proof idea and the level of detail that the author wants to exhibit.

Another important technique to explain a difficult proof to a reader is to look at the proof at different levels, i.e., establish a hierarchy of detail. For example, on the first level of the description, the proof may contain only the most important steps together

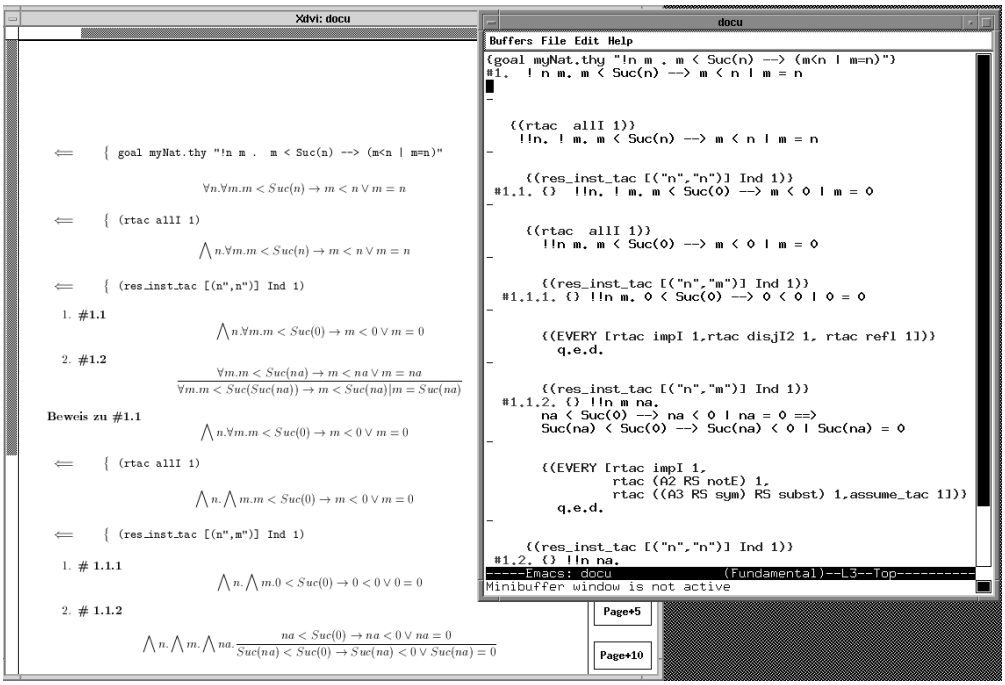


Figure 9: Intermediate state while improving a proof presentation.

with comments. Thus, the reader is able to gain a first overview over the proof without having to deal with all the details of it. On the following levels of the description, successively more detailed information is provided.

3.3 A Tool Supporting the Preparation of Proof Documents

In the course of this work, a prototypical tool was implemented. It is realized on the basis of an editor, namely GNU Emacs. Emacs was chosen because it can be extended easily by programming new functionality in a dialect of Lisp. The editor provides convenient libraries to program user interfaces and communication with subprocesses—in our case, Isabelle.

The initial representation of a proof is similar to the Isabelle representation, but the structure of the proof is already visible, since in each step, instead of the whole proofstate, only the current subgoal and all the subgoals that are a consequence of the proofstep are displayed.

Starting with this representation, the presentation of the proof can be gradually improved by adding comments, hiding superfluous information, introducing structure, and so on.

During this editing process the proof is displayed in Isabelle's ASCII-syntax. Afterwards, a LaTeX document can be generated automatically, where all the operators, constants, and so on are replaced by their appropriate mathematical symbols. To do so, the tool has, for every theory, to be initialized with the right translation table. A screen snapshot showing Emacs in the middle of a proof editing session together with the generated LaTeX as displayed in a previewer is given in Figure 9. The initial comments

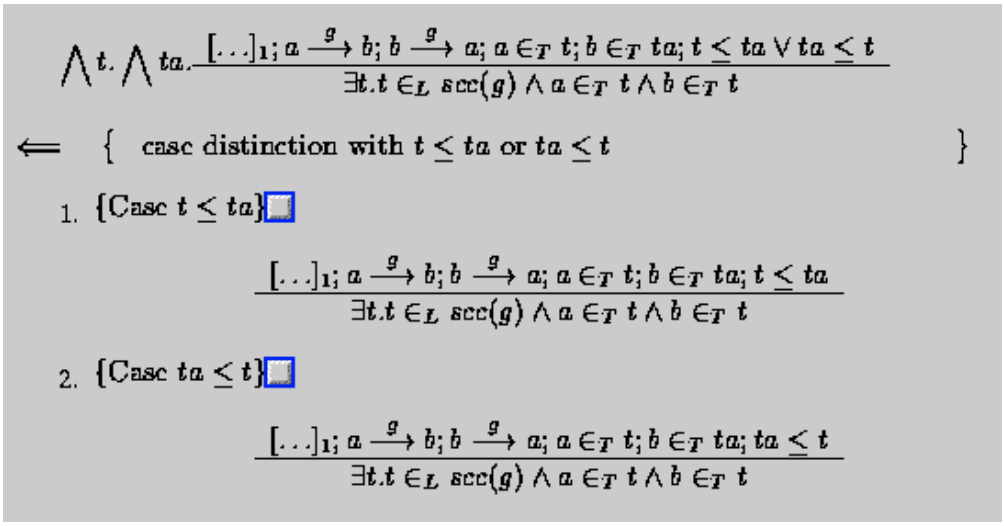


Figure 10: Case distinction as displayed in a hypertext browser.

generated by the system and placed in the curly braces are the Isabelle commands that led to the corresponding subgoal.

The main operations that are performed while editing the proof are

- changing the comments,
- hiding single proofsteps, and
- structuring the proof.

The operations which are permitted on the proof document guarantee that the correctness of the proof is not affected—this is of course not completely true in the case of the comments.

Apart from viewing and printing the generated LaTeX document, it is also possible to generate an HTML (Hypertext Markup Language) document from it. In contrast to the printed version of the proof, which must choose a linearization of the hierarchical proof structure, a hypertext representation preserves this hierarchy and can be browsed interactively. Figure 10 contains a screen snapshot showing part of a proof as displayed in an HTML browser. The square active buttons can be clicked and lead to the respective subcases.

4 Conclusion

We have presented an overview of our current activities towards intelligible formal proofs where Isabelle serves as our experimental platform. On the one hand, we adapt and further develop our methods for presenting formal proofs which combine elements of calculational reasoning, a hierarchical, natural-deduction like proof-style, and the literate programming paradigm. On the other hand, we experiment with substructural logics in order to formalize our notion of hierarchical proof-objects so that we can

formally speak about proof-refinements. A comprehensive account of this research is currently being composed and will contain a literate Isabelle version of the proof of the Church-Rosser theorem.

References

[AJS94] M. Anlauff, S. Jähnichen, and M. Simons. A support system for formal mathematical reasoning. In Naftalin et al. [NDB94], pages 421–440.

[Anl95] M. Anlauff. *Rechnerunterstützung formaler Beweissprachen*. Number 244 in GMD-Bericht. Oldenbourg Verlag, 1995.

[BRS93] M. Biersack, R. Raschke, and M. Simons. The DevaWEB system: Introduction, tutorial, user manual, and implementation. Technical Report 93-39, TU Berlin, 1993.

[DS90] E. W. Dijkstra and C. Scholten. *Predicate Calculus and Predicate Transformers*. Springer-Verlag, 1990.

[DSH93] K. Došen and P. Schroeder-Heister, editors. *Substructural Logics*. Oxford Science Publications, 1993.

[Kam95] F. Kammüller. Experimentelle Unterstützung einer Beweisprogrammiersprache mit Isabelle. Diplomarbeit, TU Berlin, Fachbereich Informatik, 1995.

[Kel94] G. Keller. An experimental system for the production and presentation of formal proofs. Studienarbeit, TU Berlin, Fachbereich Informatik, 1994.

[Kel95] G. Keller. Unterstützung hierarchischer Beweise mit Hilfe eines interaktiven Theorembeweisers. Diplomarbeit, TU Berlin, Fachbereich Informatik, 1995.

[KL95] D. J. King and J. Launchburry. Structuring depth-first search algorithms in Haskell. In *Proceedings of the 22nd Symposium on Principles of Programming Languages (POPL'95)*. ACM Press, 1995.

[Knu84] D. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[NDB94] M. Naftalin, T. Denvir, and M. Bertran, editors. *FME'94: Industrial Benefits of Formal Methods*, volume 873 of *LNCS*. Springer-Verlag, 1994.

[SBR94] 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)*, pages 61–81. North Holland, 1994.

[Sin94] M. Sintzoff. A proof programming language founded on quantales. Private communication, 1994.

[SKJB95] T. Santen, F. Kammüller, S. Jähnichen, and M. Beyer. Formalization of algebraic specification in the development language Deva. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools to Construct Correct Software*, LNCS. Springer-Verlag, 1995.

[Web94] M. Weber. Literate mathematical development of a revision management system. In Naftalin et al. [NDB94], pages 441–460.

[WSL93] M. Weber, M. Simons, and Ch. Lafontaine. *The Generic Development Language Deva: Presentation and Case Studies*, volume 738 of *LNCS*. Springer-Verlag, 1993.

Some papers are available in the WWW under the URL `http://www.cs.tu-berlin.de/~car/`.