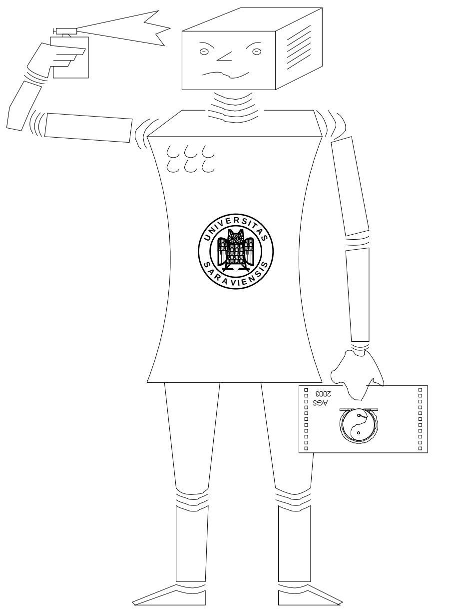
UNIVERSITÄT DES SAARLANDES
FACHBEREICH INFORMATIK
D–66123 SAARBRÜCKEN
GERMANY
WWW: http://www.ags.uni-sb.de/

SEKI-REPORT ISSN 1437-4447



Towards a Framework to Integrate Proof Search Paradigms

S. Autexier[1], C. Benzmüller[2], D. Hutter[1]

[1] German Research Centre for Artificial Intelligence (DFKI GmbH)
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
e-mail: {autexier|hutter}@dfki.de

[2] FR Informatik, Universität des Saarlandes
66041 Saarbrücken, Germany
e-mail: chris@ags.uni-sb.de

SEKI Report SR–2003–02

Editor of SEKI series:
Claus-Peter Wirth
FR Informatik, Universität des Saarlandes, D–66123 Saarbrücken, Germany
E-mail: `cp@ags.uni-sb.de`
WWW: `http://www.ags.uni-sb.de/~cp/welcome.html`

Towards a Framework to Integrate Proof Search Paradigms

S. Autexier[1], C. Benzmüller[2], D. Hutter[1]

[1] German Research Centre for Artificial Intelligence (DFKI GmbH)
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
e-mail: {autexier|hutter}@dfki.de

[2] FR Informatik, Universität des Saarlandes
66041 Saarbrücken, Germany
e-mail: chris@ags.uni-sb.de

July 22, 2003

Abstract

Research on automated and interactive theorem proving aims at the mechanization of logical reasoning. Aside from the development of logic calculi it became rapidly apparent that the organization of proof search on top of the calculi is an essential task in the design of powerful theorem proving systems. Different paradigms of how to organize proof search have emerged in that area of research, the most prominent representatives are generally described by the buzzwords: automated theorem proving, tactical theorem proving and proof planning. Despite their paradigmatic differences, all approaches share a common goal: to find a proof for a given conjecture.

In this paper we start with a rational reconstruction of proof search paradigms in the area of proof planning and tactical theorem proving. Guided by similarities between software engineering and proof construction we develop a uniform view that accommodates the various proof search methodologies and eases their comparison. Based on this view, we propose a unified framework that enables the combination of different methodologies for proof construction to take advantage of their individual virtues within specific phases of a proof construction.

1 Introduction

The idea of tactical theorem proving as a way to program proof strategies interactively dates back to the end of the 70s' when the Edinburgh LCF-project [GMW79] proposed its approach in which the user interacts with the prover through a programming language ML which operates in a sound way on logical formulas, rules etc. as ML data. Extensions of the underlying logic resulted in Cambridge LCF; in the 80s' Gordon implemented the Higher-Order-Logic HOL [Gor88] based on Cambridge LCF. Later on, further approaches based on tactical theorem proving followed: the KIV-system [HRS90], λ-prolog [FM88], and OYSTER [Bun88] only to mention some of them. In [Bun88], Bundy addressed the problem of dynamically combining tactics to complex proof strategies which resulted in the paradigm of proof planning. The idea was to add declarative specifications (pre-/postconditions) to tactics resulting in so-called methods. This enables one to reason about the behaviors of tactics and to plan complex sequences of tactic applications. In the area of program synthesis, Kraan et al. [KBB93] introduced middle-out reasoning to propagate constraints arising from speculated intermediate goals towards the original goal. Melis [MS99] generalized such an approach to the notion of island planning. To obtain a more efficient planning process, Richardson and Smaill [RS01] introduced the notion of methodicals as a programming language on top of methods. In parallel the Omega group developed a different implementation of proof planning. Starting with methods basically as fixed proof tree patterns [KRS94], their notion of methods evolved to a kind of generalized tactic allowing to construct a proof tree from arbitrary directions and using various non-tactical sub-provers [BCF+97]. To cope with the search space arising from this flexibility, Sorge [BS00] introduced O-ants as an blackboard-based approach to propagate constraints by method applications towards different directions in parallel.

The large amount of competing and non-related approaches makes it difficult to get a unified view on how the various aspects of tactical theorem proving and proof planning influences the efficiency and also the effectiveness of proof search. Rippling [BSvH+93] as a prime example for proof planning, for instance, has also been implemented using a tactical theorem proving approach [Hut97]. So what are the differences between the different approaches from a proof search perspective. Are there fundamental differences which makes one approach superior to others in some domains?

Interactive theorem proving bears a strong resemblance to formal program development. The theorem to be proven relates to a formal requirement specification. The resulting formal proof corresponds to the running program satisfying the requirement specification. In between there are various layers of abstractions in which given requirements are step by step replaced by constructive solutions satisfying the requirements. In algebraic specifications the formal notion of refinement constitutes the bonding between the layers. Basic operations of one layer have to be implemented in terms of operations of the next layer below. Analogously, tactics can be considered as an implementation of complex proof search operators specified by the pre- and postconditions of the corresponding method. Based on such specification we may construct even more complex search operators which define again the basic building blocks of the next layer.

Driven by this analogy to formal software development, we will analyze existing approaches on proof planning and tactical theorem proving to come up with a unified view on these approaches. In particular we are interested in the principle representational and organizational concepts of these paradigms and present an attempt at a uniform perspective and landscape. This landscape supports the localization of the different paradigms as well as the requirement analysis of the key structures of a system framework that shall support the integration of the different approaches based on traditional theorem proving, tactical theorem proving, and proof planning.

2 Proof Search Using Tactics and Proof Planning

The ultimate goal of a proof search is to obtain a formal proof of the given theorem. Following the lines of tactical theorem proving [GMW79, Pau87], we consider a formal (tactical) proof as a directed acyclic graph. Each node N of the graph is labeled with a formula φ_N. The graph has a single bottom-level (or dually top-level) node labeled with the theorem to be proven. Given a node N its immediate predessor (dually successor) nodes $N_1, \ldots, N_k$ represent the conditions necessary to infer the formula associated to N, i.e., $\frac{\varphi_{N_1} \wedge \ldots \wedge \varphi_{N_k}}{\varphi_N}$ is an instance of an inference rule of the proof calculus used to construct the proof graph. A proof graph is *closed* iff all leaves N of the graph are closed, i.e., labeled with *true*. In contrast to traditional approaches, we define proofs as graphs instead of trees which allows us to make explicit the multiple use of intermediate proof goals. A partial proof refers to a fragment of a proof graph and is usually a collection of non-closed and unconnected proof graphs.

Proof search is the process to construct a closed proof graph. In interactive theorem proving this is usually done by collecting more and more constraints about the desired (closed) proof graph. In principle, there are two different types of constraints to refine the partial proof:

Graph constraints speculate about the existence or the form of particular subgraphs or individual nodes. This corresponds to the speculation of a lemma or an intermediate subgoal. For instance in inductive theorem proving, heuristics are successfully used that propose an intermediate goal of the form $P(x) \rightarrow \Psi(P(x))$ when proving an induction step $P(x) \rightarrow P(s(x))$, because this form enables the application of the given $P(x)$ on the transformed goal $\Psi(P(x))$. In other words we are only interested in proof graphs which contain a node labeled with an instance of $P(x) \rightarrow \Psi(P(x))$.

Operational constraints speculate about the way how an actual proof sketch will be expanded. A classic example of operational constraints is the refinement of proof sketches using tactics as they are defined in the LCF-framework [Pau87]. Tactics are basic, primitive operations on proof sketches implementing the application of an individual inference rule. Tacticals allow one to combine tactics and tacticals to more complex programs. Thus, deciding which tactic(s) to use to expand a node represents an operational constraint. For example, the decision to expand a proof node by using a simplification tactic, which unfolds definitions, is a well-known instance of such an operational constraint.

2.1 Abstract Proof Graphs

Given a partial proof, graph constraints postulate the existence of a node (or a subgraph) in the proof graph under development satisfying, for instance, a specific pattern. How can we make use of such speculated intermediate goals when expanding the partial proof? A well-known approach is generate and test. Independently of the given constraints the partial proof is expanded. Only if the generated subgraph (or a particular node of it) satisfies the graph constraints it will be permanently added to the graph, otherwise backtracking will remove the added parts from the outline. Consider the example of proving the induction step: tactics may transform the task $P(x) \rightarrow P(s(x))$ but only if their result satisfies the pattern $P(x) \rightarrow \Psi(P(x))$ the generated subgraph is permanently added to the proof outline.

Notice, that in this case we do not need a declarative description of the test pattern but we only need an oracle (program) to decide whether the conditions are satisfied. The drawback of generate-and-test is that the test conditions do not influence the generation process.

In order to support a closer interlocking of the proof generating search process and the selecting test conditions, we need a declarative description of the intermediate goals (instead of a simple test program) to estimate and react on the distance between the actual node and the desired goals. As a result, various abstractions on formulas (see [Pla81] for an overview) were proposed in the past to formulate intermediate goals. Most of these abstractions are either formula patterns involving higher-order variables (like Ψ in the example above) or homomorphic extensions on abstractions of the signature (like mapping different predicate symbols into one symbol, or ignoring specific argument positions of functions). The problem lasts of how to search for a proof on an abstract level. Ideally we would like to abstract the inference rules giving birth to a inference machine on the abstract level. This would allow us to search for an abstract proof based on the declarative graph constraints (formulated as abstracted formulas). However, it turned out that in practice abstractions on (usual) formulas are of limited use [Pla81]. In particular, the claim for PI-abstractions [GW92] restricts possible abstractions to homomorphic extensions of abstractions on the signature which are of minor help in proof search. Rippling was a first successful example of using abstractions on proof datastructures to search for an inductive proof (see [BSvH$^+$93, Hut00] for details). However, in this setting the abstraction is no longer a function on terms but a function on terms *plus* additional context information (i.e., a skeleton denoting the unchangeable parts of a term). In the meantime more abstractions on such *enriched* proof datastructures have been developed as a basis for proof search on an abstract level (e.g. [AH97, Hut00]).

Summing up, using graph constraints imposes the notion of an abstract proof graph, the nodes of which are labeled with abstractions of formulas (possibly plus additional context information). We distinguish two approaches: generate and test will expand the proof graph on the concrete level, abstract the resulting formula and compare it with the given abstract goal. Abstract reasoning will operate on an abstract partial proof and will refine the abstract proof graph to a concrete proof graph later-on.

2.2 Stepwise Refinement

Given the notion of an abstract partial proof, the problem arises how to map such an abstract object to a corresponding concrete partial proof graph. To simplify matters we restrict ourselves in the following to mappings that preserve the structure of an abstract proof graph/outline. In other words we demand the existence of functions *abs* and *ref* such that *ref* maps each node N of the abstract graph into a node of the concrete graph such that $abs(\varphi_{ref(N)}) = \varphi(N)$ holds. Furthermore, if M is a sub-node of a node N then $ref(M)$ is also a sub-node of $ref(N)$. Intuitively, these conditions guarantee that we can use the abstract problem decomposition (encoded into the abstract proof graph) without changes to structure the proof of the next proof layer. While this property holds using abstractions like those mentioned above, there are more general approaches to change the representation level of proof search that violates the property. We will return to this issue later-on.

If we stick to this structure preserving refinement of abstract proof layers then we can

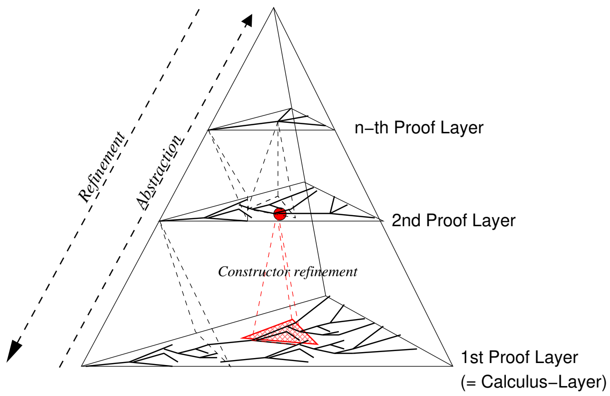


Figure 1: Proof layers connected via abstractions and refinements

map each abstract proof constructor, which is an instance of an abstract inference rule, to a subgraph of the next proof layer. To implement such a refinement, we are interested in a uniform approach which tells us how to refine abstract inference rules in general. At first we may use again a general generate-and-test approach to extend a node in the concrete proof graph until its abstraction is equal to the instance of the abstract proof step. To realize an informed search, knowledge about possible refinements of abstract proof steps can be encoded into operational constraints. For instance, we can associate a tactic to an individual abstract inference rule which will *implement* the refinement of an abstract proof step. This corresponds closely to the notion of a *module* in formal software development: the abstract inference rule is an atomic operation on the abstract level that is implemented in terms of operations on the concrete level during the refinement process. Methods, as they are introduced in [Bun88], are special cases of virtual modules of an approach of *proof search by stepwise refinements*: pre- and postconditions specify the abstract inference rule while the associated tactic denotes its implementation.

2.3 Implementing Abstract Constructors

Organizing proof search in layers of more and more abstract layers, the main task is to compute appropriate implementations of the (instances of the) abstract constructors of each layer. Tactics are a classical way to describe such implementations: Given a fixed abstraction layer, a set of tactic represent the constructors of this layer. Tacticals are used to combine these constructors to more complex tactics which are used to provide an implementation of the constructor tactics in the above proof layer.

Classical LCF Tactics operate on an open problem and translate this problem using the given inference rules of the calculus into a (possibly empty) set of new problems. Tracing the used inference rules and the intermediate sets of problems, we can say that a tactic will expand a graph node into a partial proof. Since tactics are deterministic, each tactic T has a characteristic (partial) function exp_T which maps proof nodes to partial proofs such that each proof node N is mapped to the partial proof $exp_T(N)$ that results from the application of the tactic to the node. As a consequence, LCF tactics will only construct proof graphs top-down.

However, suppose we know about graph constraints which restrict the outline of the subgraph to be generated. Since we like to avoid a generate-and-test approach, we have to propagate these constraints towards the choice points of the tactic (like, for instance, the use of ORELSE in LCF) which corresponds to a bottom-up approach. In other words, knowing constraints about the leaves $N_1, \ldots, N_k$ of $exp_T(N)$ we have to deduce knowledge about the tactic T to be used and the (speculated) node N which will be expanded. Consider $N, N_1, \ldots N_k$ as the interface of the subgraph resulting from an application of a tactic T. Then the generalized problem is to deduce knowledge about this subgraph from given constraints on the interface. This gave rise to the notion of methods as introduced by [BCF$^+$97]. Roughly speaking, these methods are generalized tactics that compute proof graphs or partial proofs from a subset of their interfaces which allows one to mix top-down and bottom-up approaches. In the same way tacticals are used to construct complex tactics from simple tactics, methodicals [RS01] are introduced to combine methods to even more complex methods with the help of usual programming language operators.

2.4 Some Nontrivial Examples

In the area of diagrammatic reasoning we find examples for horizontal proof systems working on representations different from predicate logic. A prominent example are Spider Diagrams, which extend the well known Venn diagrams and Euler circles supporting reasoning about sets (cf. [HMT$^+$01] and the references therein). Soundness (and often also completeness) are directly guaranteed for the operators at this layer. In this case there is no need to provide another soundness criterion via refinement, for instance, to a predicate logic representation layer, even though very natural and intuitive refinement mapping may exists.

Jamnik's DIAMOND system [Jam01] is an example for diagrammatic reasoning system that links two different representation frameworks. DIAMOND constructs proofs within a diagrammatic representation framework for the specific domain of natural number arithmetic and it verifies the diagrammatic proofs via refinement to schematic proofs given in a predicate logic representation framework. Another proof system that links a diagrammatic representation layer with sentential logic representations is Hyperproof [BE94].

A common aspect of most diagrammatic reasoning systems is that they are exclusively designed for specific mathematical domains. This contrasts with universal demand and pretension of systems designed for predicate logic. Our framework, however, provides a way to combine these approaches by employing a predicate logic based system at the bottom layer and providing refinements from diagrammatic reasoning systems to the latter (but probably also between different diagrammatic reasoning systems, if possible).

3 An Attempt at a Unified Framework

In this section we attempt at a unified framework comprising the heterogenous proof search paradigms sketched before. Our proposal is preliminary in the sense that further extensions of our view are possible and necessary. For instance, in the discussion in Sec. 4 we will address operators, such as Ireland's proof critics [Ire92], that work globally on the proof data structure instead of just locally. For the moment we restrict our development to the latter in order to avoid confusions.

Our framework is separated in a horizontal and a vertical axis. The horizontal axis addresses proof search with a given set of basic proof operators defined on proof nodes, which we will call *proof constructors (PC)*. Associating proof constructors with a specification of an application direction (they can be specified via procedural or declarative means) turns them into *proof constructor applications (PCA)*. Proof construction is realized as *composition of proof constructor applications*. Specifications of such compositions are possible in a static manner by exploiting a procedural programming language (which gives us LCF style tactics) or declaratively as a dynamic search process (which gives us declarative proof planning). In the latter case the presupposed declarative descriptions of the PCAs are exploited to dynamically chain them in the search space. Specifications of compositions of PCAs are abbreviated as CPCAs in the remainder. A *horizontal proof system* consists of a set of PCAs and a proof search or proof construction framework given in form of a CPCA.

The vertical axis addresses the aspect of abstraction and refinement of proof constructors (and thus proofs) from one representational layer to another one. We do not necessarily enforce a *real* change of representation between two vertical layers and explicitly allow the abstraction and refinement mappings *abs* and *ref* to be the identity function (*trivial change of representation*). This way our framework supports the construction and integration of new abstract (less granular) proof constructors that encapsulate larger proof pattern via one or several vertical moves in the hierarchy with respect to the trivial change of representation.

3.1 The Horizontal Axis

We now introduce the concepts for horizontal proof systems. A basic ingredient is the *object language* that we employ to model the objects in the domain of interest. With the help of the object language we introduce *proof lines*. Next, *Proof graphs* are defined as an abstract datatype with *proof nodes* and *proof constructors* as basic building blocks.

Proof Constructor A proof constructor $N = [P, C, T, S]$ consists of non-empty lists $P = (P_1, \ldots, P_i)$ and $C = (C_1, \ldots, C_j)$ of schematic variables P_l, C_k representing premise nodes and conclusion nodes respectively, some possibly empty parameter list T, and some possibly empty set of application conditions S. A proof constructor N is usually represented as

$$\frac{P_1, \ldots, P_i}{C_1, \ldots, C_j}\ N(T) \quad \text{plus side-conditions}$$

Examples for proof constructors (PCs) are:

$$\frac{P_1 \quad P_2}{C}\ ModusBarbara$$

with the side-condition that P_1, P_2, and C are of forms $A \Rightarrow B, B \Rightarrow C$, and $A \Rightarrow C$ respectively.

$$\frac{P \quad PL}{C}\ ProofByPointing(\pi)$$

where P is the sub-formula in C at position π and PL is the list of all formulas that arise as subgoals when decomposing C down to P.

$\frac{P}{C}$ $Simplify(\leq)$

where $P \leq C$ with respect to some simplification ordering $\leq$.
An example for a special version of $Simplify$ is arithmetical simplification (i.e., we choose a particular ordering $\leq_{arith}$ and probably employ a computer algebra system to realize $\leq_{arith}$).

$\frac{P_1 \quad P_2}{C}$ $Resolution$

with side-condition that there exists complementary literals L_1 (in clause P_1) and L_2 (in clause P_2), and a substitution σ such that ...

We want to remark that even in presence of a formal notion of semantics for the underlying object language we do not presuppose that PCs actually fulfill a direct soundness criterion. The reason is that we aim at a flexible approach that even supports unsound reasoning at a higher proof layer provided that soundness can be addressed indirectly via refinement to a lower layer.

OMEGA is an example for system that operates with (some) unsound operators at higher proof layers, while it provides a sound bottom layer and a respective refinement mechanism. The main issue at the higher proof layers in OMEGA concerns the adequacy of the new operators content, their amplitude of prescinded granularity, and their (typically domain-specific) proof search qualification. For some OMEGA operators it is even impossible to establish soundness directly. For instance the computer algebra system X used within $Simplify(\leq_{arith})_{Backward}$ to perform the simplification step may be partially incorrect (e.g., by neglecting some side-conditions). Thus, instead of trusting X, OMEGA provides facilities to refine (some of the) computations of X to the sound and verifiable bottom layer; see [Sor00] for details. The same is possible via TRAMP [Mei00] for traditional ATPs employed within abstract proof operators.

In order to be able to apply PCs we have to provide additional information. For instance, we have to specify application directions for the PCs (forward, backward, and sideward or mixed). We also need to say how concrete instantiations of the schematic proof node variables can be determined. This motivates the following definition of *proof constructor applications*.

Proof Constructor Application (PCA) A proof constructor application $PCA = (PC, OP)$ combines a proof constructor PC with an operator (function) OP that realizes a particular application direction of PC, i.e., OP constitutes how to compute from given proof node instantiations for schematic proof node variables $K \subset P \cup C$ the instantiations of the schematic variables $(P \cup C) \setminus K$.

The operation OP of a PCA can be specified in a pure procedural manner (see LCF-tactics) or by declarative means (see methods in Clam and OMEGA). In case pure procedural constructs are employed we speak of a procedural proof constructor applications (PPCA).

Declarative specifications of PCAs presuppose a suitable declarative description framework, for instance, based on a higher-order language, together with respective support facilities such as a pattern matching. However, even in the presence of such a declarative description framework often not all aspects of an OP can be described fully declaratively. As a consequence a mixture of declarative and procedural constructs is employed. In the following we will call specifications of *PCA*s that contain declarative parts as declarative proof constructor applications (DPCA).

We motivate and illustrate the notions $DPCA$ and $PPCA$ by examples:

Assuming that a respective specification language (see for instance Clam or OMEGA) is available we can model forward application of Modus Barbara purely declaratively as $DPCA$, for instance, as follows

$$\frac{P_1^{\ominus} : \mathbf{A} \Rightarrow \mathbf{B} \quad P_2^{\ominus} : \mathbf{B} \Rightarrow \mathbf{C}}{C^{\oplus} : \mathbf{A} \Rightarrow \mathbf{C}} \; ModusBarbara_{Forward}$$

$\ominus$ and $\oplus$ encode the application direction; we assume that OP works from instantiations of the nodes $P_{1,2}$ (indicated by $\ominus$) in direction of node C (indicated by $\oplus$). Terms like $\mathbf{A} \Rightarrow \mathbf{C}$, with $\mathbf{A}$ and $\mathbf{C}$ being schematic variables, are constructs of our declarative description framework and specify structural requirements. Pattern matching is the mechanism that supports the determination of possible proof nodes matching the structural restrictions for $P_{1,2}$ and the resulting information is employed to compute an instance for C. Note that we could alternatively specify $ModusBarbara_{Forward}$ also purely procedural by expressing the above criteria directly in a programming language.

Backward application of *Simplify*($\leq_{arith}$) is an example of a PCA where a procedural modeling as $PPCA$ is indicated:

> *Input: instance* $\mathbf{C}$ *for* C
> *Body: if* $\mathbf{C}$ *contains arithmetic expressions* $\mathbf{A_i}$ *then use computer algebra system* X *to simplify the* $\mathbf{A_i}$ *to simplified expressions* $\mathbf{A_i^s}$ *with respect to* $\leq_{arith}$ *else do nothing*
> *Output: Instance for* P *computed by replacing the* $\mathbf{A_i}$ *by* $\mathbf{A_i^s}$ *in* $\mathbf{C}$ *respectively*

Procedural specifications of *PCA*s are, of course, always possible (since the employed programming languages are usually Turing complete) while declarative specifications heavily depend on the facilities and capabilities offered by the employed reasoning framework.

An example for a $DPCA$ where declarative and procedural parts are mixed is

$$\frac{P^{\oplus} : \mathbf{A} \quad PL^{\oplus} : {}_1, \ldots, {}_n}{C^{\ominus} : \mathbf{X}(\mathbf{A})} \; ProofByPointing(\pi)$$

where we assume that $\mathbf{A}$ represents the formula we put the focus on and $\mathbf{X}$ is a higher-order variable denoting the surrounding context of C. An additional side-condition for $\mathbf{X}$ and π is that X has to match with a term $\lambda\mathbf{Y}.\mathbf{C}$ in which variable $\mathbf{Y}$ occurs exactly once (in the β-normalized) formula $\mathbf{C}$, namely at position π. Thus, when providing $\mathbf{A}$ the surrounding context $\mathbf{X}$ of $\mathbf{A}$ in $\mathbf{C}$ is determined by matching and also the respective position π can be inferred. The only missing information concerns the determination of the ${}_1, \ldots, {}_n$, i.e., the subformulas in $\mathbf{C}$ that arise as subgoals when decomposing C down to P beta-related to the subformula position we are pointing at. The determination of these subformulas can hardly be expressed by declarative but easily by procedural means.

Composition of PC Applications are specifications of complex proof search behaviors by an howsoever composition of *PCA*s. As before for *PCA*s we distinguish between procedural composition of *PCA*s (*CPCA*) that follows the ideas of tacticals [GMW79, Pau87, SAH01] and methodicals [RS01], and declarative composition of *DPCA*s (*SDPCA*) that follows the idea of searching for sequences of *DPCA*s by exploiting their declarative descriptions.

Declarative Compositions are search strategies over given *DPCA*s. Typical examples for such strategies are iterative deepening search, best-first search wrt. some heuristic weighting, or a Set-Of-Support Strategy. Even search strategies induced by ordering constraints as used in superposition calculus [BGLS92] fall in that category. Each of these declarative search strategies can be described symbolically by their name and a list of parameters to that strategy. A parameter that is shared among these strategies is the set of proof constructors $\mathcal{C}$. This results in a language for declarative proof search strategies, as for example (we assume a language "Parameters" for expressions describing parameters):

$$\begin{array}{lcl}\text{Search Strategies over } \mathit{DPCA}\text{s:}\ \ \mathit{SDPCA} & := & \mathit{Best\text{-}First}(\mathcal{C}, \text{Parameters}) \\ & & \mid\ \mathit{IDA}^*(\mathcal{C}, \text{Parameters})\ \mid\ \ldots\end{array}$$

Procedural Compositions are specifications of complex operational proof search behaviors. They are described using a specific language tailored to the needs arising during the design of proof search behaviors. In the following we sketch an example procedural composition language $\mathcal{L}(\mathcal{C})$ that is parameterized over a set $\mathcal{C}$ of proof constructor applications (*PCA*).

For the *PCA*s we assume that they can be invoked in some given proof situation possibly together with a specification how they shall be applied. The application of these *PCA*s can either fail, leaving the proof unchanged or succeed and return the new proof. The success and failure semantics of *PCA*s is subsequently used to define a semantics for $\mathcal{L}(\mathcal{C})$ in continuation passing style [Rey93].

The language $\mathcal{L}(\mathcal{C})$ consists of compound expressions over $\mathcal{C}$ denoted by *CPCA*. Thereby we assume a language "Test" for expressions describing boolean tests, a language "Var" for variables, a language "Compute" for expressions describing computations, and a language "Parameters" for expressions describing parameters are given:

$$\begin{array}{lcll}\text{Compound } \mathit{PCA}\text{:}\ \ \mathit{CPCA} & := & \mathcal{C}\ \mid\ \mathit{CPCA}; \mathit{CPCA}\ \mid\ \mathit{CPCA} \parallel \mathit{CPCA} & (1) \\ & & \mid\ \text{if Test then } \mathit{CPCA} \text{ else } \mathit{CPCA} \text{ fi} & (2) \\ & & \mid\ \text{while Test do } \mathit{CPCA} \text{ od} & (3) \\ & & \mid\ \text{let Var} = \text{Compute in } \mathit{CPCA} \text{ tel} & (4) \\ & & \mid\ \mathit{NAMED\text{-}CPCA}(\text{Parameters}) & (5) \\ & & \mid\ \mathit{SDPCA} & (6)\end{array}$$

Thereby ";" in (1) denotes sequential composition and "||" denotes parallel composition. In both cases the compound expression fails if either the first or second compound fails. In (2) conditional branching is included which fails if the compound of the taken branch fails. Loops can be specified and a loop expression fails if at some stage the compound expression in the body of the loop fails (3). Local variables can be declared by a standard *let* operator and that expression fails if the contained compound fails (4). Finally, calls to named compounds can occur in compound expressions (5). Named compounds are analogous to standard named functions in functional programming languages. Thus, aside the definition of *CPCA*s the language shall support the definition of *named CPCAs*:

$$\text{Named } \mathit{CPCA}\text{s} \quad := \quad \text{defproc } \mathit{NAMED\text{-}CPCA}(\{\text{Var}\}^*) = \mathit{CPCA}$$

Finally, in order to support the combination of procedural and declarative proof search descriptions, we integrate the search strategy language *SDPCA* into the language for *CPCA*s (6).

Now, *pure* tactics in the LCF tradition [GMW79, Pau87, SAH01] can be viewed as a subset of the above language, where named *CPCA*s are defined from procedural *PCA*s only. We denote this sub-language of $\mathcal{L}(\mathcal{C})$ by $\mathcal{L}(\mathcal{C})_{\downarrow P}$. Furthermore, the methodical expressions [RS01] can be viewed as a subset of the above language without the definition of named *CPCA*s and where the allowed *CPCA*s are those defined from declarative proof constructor applications (*DPCA*) only. We denote this sub-language of $\mathcal{L}(\mathcal{C})$ by $\mathcal{L}(\mathcal{C})_{\downarrow D}$.

3.2 The Vertical Axis

The vertical axis is concerned with the relationship between proof constructors from an upper proof layer and proof constructors from a lower proof layer. More specifically it consists of the specification how a proof constructor from an upper proof layer can be implemented by proof constructors from a lower proof layer. Again, by exploiting the analogy to formal software development, this means that implementations must be defined for each constructor of an abstract datatype. In our setting, we have an upper set of proof constructors $\mathcal{C}_a$ and a lower set of proof constructors $\mathcal{C}_l$. The implementation of a *PC* from $\mathcal{C}_a$ is described by a compound *PCA* in $\mathcal{L}(\mathcal{C}_l)$. However, analogously to signature morphisms in refinements of specifications, abstraction and refinement mappings *abs* and *ref* that link formulas on the lower proof layer and formulas on the abstract proof layer belong to the definition of a proof constructor refinement. Thus, we define refinements by

$$\mathit{REFINEMENT} \;:=\; \langle \mathcal{C}_a, \mathcal{L}(\mathcal{C}_l), \alpha \rangle$$

With this notion we can classify existing refinements used in the various proof planning systems, such as Clam, λClam, ΩMEGA, and Ω-ANTS. In Clam and λClam the refinement is not assigned to a proof constructor, but to a declarative proof constructor application (*DPCA*); this also holds for the ΩMEGA proof methods. Thus, implementations must be defined for all *DPCA*s, even if they belong to the same proof constructor *PC*. However, once a proof constructor is applied, its refinement is independent of the way how the application has been achieved, as it is for example done in Ω-ANTS. For that reason we refrained from assigning refinements to *DPCA*s and defined refinements for the underlying *PC*s only.

However, there are further differences in the refinement notions used in the proof planning systems: The refinement represented by methods in Clam, the ancestor of all proof planning systems, assigns a call to a named pure LCF tactic to an abstract proof constructor, and the abstraction and refinement function *abs* and *ref* are the identity function.

$$\mathsf{Clam}\text{-REFINEMENT} := \langle \mathcal{C}_a, \mathit{NAMED\text{-}PCA}(\text{Parameters}), \mathsf{Id} \rangle$$

where *NAMED-PCA* is defined by defproc $\mathit{NAMED\text{-}PCA}(\text{varlist}) = \mathcal{L}(\mathcal{C}_l)_{\downarrow P}$.

In λClam the implementation is extended by also allowing methodical expressions in method refinements, while the abstraction function is still the identity.

$$\lambda\mathsf{Clam}\text{-REFINEMENT} := \langle \mathcal{C}_a, \{\mathit{NAMED\text{-}PCA}(\text{Parameters}) \mid \mathcal{L}(\mathcal{C}_l)_{\downarrow D}\}, \mathsf{Id} \rangle$$

Finally, in ΩMEGA and Ω-ANTS the refinement of a method can be viewed as a partial proof build from declarative proof constructor applications (*DPCA*) wrt. $\mathcal{C}_l$. The admissible expressions for the implementation are thereby sequential and parallel composition of *DPCA*s. We denote that language by $\mathcal{L}(\mathcal{C}_l)_{\downarrow\Omega}$ which is defined by $\mathcal{L}(\mathcal{C}_l)_{\downarrow\Omega} := \mathit{DPCA} \in \mathcal{C}_l \mid \mathcal{L}(\mathcal{C}_l)_{\downarrow\Omega}; \mathcal{L}(\mathcal{C}_l)_{\downarrow\Omega} \mid \mathcal{L}(\mathcal{C}_l)_{\downarrow\Omega} \parallel \mathcal{L}(\mathcal{C}_l)_{\downarrow\Omega}$. The Ωmega notion of a refinement is then

$$\Omega\mathsf{mega}/\Omega\text{-ANTS-REFINEMENT} := \langle \mathcal{C}_a, \mathcal{L}(\mathcal{C}_l)_{\downarrow\Omega}, \mathsf{Id} \rangle$$

Note that none of these systems exploits the possibility of a real (i.e.,non-trivial) change of representation. One theorem proving system that exploits this possibility is the ABSFOL system [GSVW93]. The ABSFOL system is an extension of the GETFOL theorem prover that supports the declarative definition of abstraction functions and provides the infrastructure to use the abstractions during interactive theorem proving with GETFOL. It allows the user to choose an abstraction, abstract a given problem, prove the abstracted problem using GETFOL, and allows to generate a partial proof for the original problem using the abstract proof. Thus, the abstraction function in an ABSFOL-refinement is actually a non-trivial mapping.

Now that we have defined refinements, we can classify the proof planning approach to proof search. Indeed, proof planning can be classified as a specific search strategy over *DPCA*s that takes as parameters not only *DPCA*s and heuristic knowledge to control the search, but also the refinements of the respective *PC*s. It is included into the proof planning strategy that it can intertwine horizontal proof construction steps with *DPCA*s and vertical refinement steps.

4 Conclusions & Future Work

In this paper we presented a rational reconstruction of various proof search paradigms developed in interactive theorem proving. Exploiting the analogy between interactive proof construction on the one hand and formal software development on the other hand, we identified the key concepts of the different approaches: Proof constructors, declarative & procedural proof constructor applications and their composition, and refinement of proof constructors. These concepts allow to define a landscape that supports the localization of the different approaches and thus allows for a uniform perspective. The gained insights are then used to sketch a framework that supports the integration of the different approaches for the benefit to support the combination of their virtues. The framework is build upon the identified key concepts, a language that accommodates both the procedural and declarative design of proof search techniques, and a representation of proofs as a hierarchy of proof graphs, each proof graph belonging to an explicit proof layer.

In Appendix A[1] we present a formalization of hierarchically interlocked proof graphs. The definitions are adapted from [CS00, Fie01] and extend those definitions in order to accommodate representational abstraction and refinements.

Future work includes to improve on the level of sophistication of our framework and to deepen the investigation and classification of existing reasoning systems. An interesting question also is how globally operating proof operators such as Ireland's proof critics [Ire92] or the control rules in Omega can be integrated in our framework. These operators do not just locally operate on a selected branch or part in the proof graph but globally change the structure of it. Therefore they complicate, for instance, a notion of backtracking when being integrated with other locally working operators in a single horizontal proof system (this is the reason why control rules and proof methods are clearly separated in Omega). One way to explain them in our framework is via the following complex change of representation: Assume that a set of purely locally working operations is defining a horizontal proof system at layer x. We consider an upper layer $x+1$, where instead of proof graphs over proof nodes, networks of proof graphs (i.e., proof graphs whose nodes consist of lower layer proof graphs) are maintained. This way we can model on layer $x+1$ local operators that would be of global nature at layer x. We can also trivially lift the local operators from layer x into respective local operators at layer $x+1$. We thereby obtain a purely local proof system at layer $x+1$, with the drawback that a rather complex proof datastructure is being manipulated.

References

[AH97] S. Autexier and D. Hutter. Equational proof-planning by dynamic abstraction. In M.P. Bonacina and U. Furbach, editors, *Proceedings of FTP'97*, Linz, Austria, 1997. RISC-Linz Report Series No. 97-50.

[BCF+97] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, E. Melis, A. Meier, W. Schaarschmidt, J. Siekmann, and V. Sorge. Ωmega: Towards a mathematical assistant. In W. McCune, editor, *Proceedings of the CADE'14*, LNAI 1249. Springer, 1997.

[BE94] J. Barwise and J. Etchemendy. *Hyperproof for the Macintosh*, volume 42 of *CSLI lecture notes*. CSLI, 1994.

[1] We would remove that appendix for the final version of the paper if we cannot obtain an extension of the page limitation.

[BGLS92] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation and superposition. In Deepak Kapur, editor, *Proceedings of the CADE'11*, volume 607 of *LNAI*, Saratoga Springs, NY, June 1992. Springer.

[BS00] C. Benzmüller and V. Sorge. Oants – an open approach at combining interactive and automated theorem proving. In M. Kerber and M. Kohlhase, editors, *Symbolic Computation and Automated Reasoning*, pages 81–97. A.K.Peters, 2000.

[BSvH+93] A. Bundy, A. Stevens, F. v. Harmelen, A. Ireland, and A. Smaill. Rippling:a heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993.

[Bun88] A. Bundy. The use of explicit plans to guide inductive proofs. In *Proceedings of the CADE'9*, LNAI 203, pages 111–120. Springer, 1988.

[CS00] L. Cheikhrouhou and V. Sorge. PDS — A Three-Dimensional Data Structure for Proof Plans. In *Proceedings of the International Conference on Artificial and Computational Intelligence for Decision, Control and Automation in Engineering and Industrial Applications (ACIDCA'2000)*, Monastir, Tunisia, 2000.

[Fie01] A. Fiedler. *User-adaptive Proof Explanation.* PhD thesis, FR Informatik, Saarland University, Saarbrücken, Germany, 2001.

[FM88] A. Felty and D. Miller. Specifying theorem provers in a higher-order logic programming language. In E. Lusk and R. Overbeek, editors, *Proceedings of the CADE'9*, LNAI 310, Illinois, USA, 1988. Springer.

[GMW79] M.J. Gordon, A.J. Milner, and C.P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation.* Springer, LNAI 78, 1979.

[Gor88] M. Gordon. HOL: a proof generating system for higher-order logic. In G. Birtwhistle and P.A Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis.* Kluwer Academic Publisher, 1988.

[GSVW93] F. Giunchiglia, R. Sebastiani, A. Villafiorita, and T. Walsh. A general purpose reasoner for abstraction. Technical Report Technical Report # 9301-08, IRST Trento, 38050 Povo, Italy, January 1993.

[GW92] F. Giunchiglia and T. Walsh. A Theory of Abstraction. *Artificial Intelligence*, 57:323–390, 1992.

[HMT+01] J. Howse, F. Molina, J. Taylor, S. Kent, and J. Gil. Spider diagrams: A diagrammatic reasoning system. *Journal of Visual Languages and Computing*, 12(3):299–324, 2001.

[HRS90] M. Heisel, W. Reif, and W. Stephan. Tactical theorem proving in program verification. In M. Stickel, editor, *Proceedings of the CADE'10*, LNAI 449, Kaiserslautern, Germany, 1990. Springer.

[Hut97] D. Hutter. Colouring terms to control equational reasoning. *Journal of Automated Reasoning*, 18(3):399–442, 1997. Kluwer-Publishers.

[Hut00] D. Hutter. Annotated reasoning. *Annals of Mathematics and Artificial Intelligence (AMAI). Special Issue on Strategies in Automated Deduction*, 29:183–222, 2000.

[Ire92] A. Ireland. The use of planning critics in mechanizing inductive proofs. In *Logic Programming and Automated Reasoning*, pages 178–189, 1992.

[Jam01] M. Jamnik. *Mathematical Reasoning with Diagrams: From Intuition to Automation.* CSLI Press, Stanford University, Stanford, CA, 2001.

[KBB93] I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for program synthesis. In P. Szeredi, editor, *10th International Conference on Logic Programming.* MIT Press, 1993.

[KRS94] M. Kerber, J. Richts, and A. Sehn. Planning mathematical proofs with methods. *Journal of Information Processing and Cybernetics*, 30(5/6):277–291, 1994.

[Mei00] A. Meier. System Description: TRAMP: Transformation of Machine-Found Proofs into Natural Deduction Proofs at the Assertion Level. In D. McAllester, editor, *Proceedings of the CADE'17*, LNAI 1831, Pitsburgh, USA, 2000. Springer.

[MS99] E. Melis and J.H. Siekmann. Knowledge-based proof planning. *Articial Intelligence*, 115(1):65–105, 1999.

[Pau87] L. C. Paulson. *Logic and Computation:Interactive Proof with Cambridge LCF.* Cambridge University Press, 1987.

[Pla81] D. Plaisted. Theorem Proving with Abstractions. *Artificial Intelligence*, 16:47–108, 1981.

[Rey93] J. C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6, 1993.

[RS01] Julian Richardson and Alan Smaill. Continuations of proof strategies. In Gore R, A. Leitsch, and T. Nipkov, editors, *Proceedings of IJCAR'2001*, 2001.

[SAH01] A. Schairer, S. Autexier, and D. Hutter. A pragmatic approach to reuse in tactical theorem proving. *Electronic Notes in Theoretical Computer Science*, 58(2), 2001.

[Sor00] V. Sorge. Non-Trivial Computations in Proof Planning. In H. Kirchner and C. Ringeissen, editors, *FroCoS 2000*, LNAI 1794. Springer, 2000.

A Hierarchical Proof Graph Representation

Definition 1 (Hierarchical Proof Constructors) *Let $(\mathcal{C}_i)_{i\in I}$ be a family of sets of proof constructors, such that there is a partial order over the $\mathcal{C}_i$ with exactly one smallest element. Then $(\mathcal{C}_i)_{i\in I}$ are hierarchical proof constructors.*

Definition 2 (Justifications) *Let $(\mathcal{C}_i)_{i\in I}$ be hierarchical proof constructors. A* justification of proof layer *i is a 3-tuple $J = (PC(\overline{P}), \overline{N})$, where $PC(\overline{P})$ is the application of some proof constructor PC from $\mathcal{C}_i$ with parameters $\overline{P}$, and $\overline{N}$ is a list of proof nodes of proof layer i.*

Definition 3 (Abstractions & Refinements) *Let $(abs_i^j)_{i,j\in I}$ a family of abstractions from i to j. Then a* representational abstraction of proof layer *i* to proof layer *j is a pair $A = (abs_i^j, \overline{N})$ such that $i < j$ and where $\overline{N}$ is a list of proof nodes wrt. proof layer j, called* abstraction nodes*. A* representational refinement of proof layer *j* to proof layer *i is a pair $R = (abs_i^j, \overline{N})$ such that $j < i$ and where $\overline{N}$ is a list of proof nodes wrt. proof layer i called* refinement nodes*.*

Definition 4 (Proof Nodes) *A* proof node of proof layer *i is a 5-tuple $N = (\varphi, \overline{J}, A, R)$, where φ is a proof object, $\overline{J}$ a set of justifications of proof layer i, A an abstraction from i to j, and R a refinement from i to j. We say that N is an open goal, if $\overline{J} = \emptyset$. If A or R are empty, we denote it by $\perp$. We say that each justificiation J from $\overline{J}$* justifies *N. If N is justified by a justification without successor nodes, then N is a* hypothesis in proof layer *i. The set of* support nodes *of N is the transitive closure of all successor nodes of N.*

Definition 5 (Refining Proof Nodes) *Given a set of proof nodes $\mathbb{N}$ and $N \in \mathbb{N}$. We denote by $\mathbb{N}_R^N$ the* refining proof nodes *of N composed of all proof nodes $N' \in \mathbb{N}$, such that N is an abstraction node of N'.*

Definition 6 (Proof Graphs) *Let $\mathbb{N}$ be a set of proof nodes, $\mathbb{S} \subseteq \mathbb{N}$ proof nodes and $N \in \mathbb{N}$ a proof node of proof layer i. $\mathcal{N}$ is a* proof graph of *N* from *$\mathbb{S}$* wrt. proof layer *i if, and only if, one of the following holds:*

1. *$N \in \mathbb{S}$.*
2. *Let $N = (\varphi, \overline{J}, A, R)$.*
   (a) *For all $J = (PC(\overline{P}), \overline{N})$ from $\overline{J}$ and each $N' \in \overline{N}$ there must be set $\mathbb{N}' \subsetneq \mathbb{N}$ that is a proof graph of N' from $\mathbb{S}$,*
   (b) *If $A = (abs_i^j, \overline{N})$, then for each $N' \in \overline{N}$ there must be a set $\mathbb{N}' \subsetneq \mathbb{N}$ that is a proof graph of N' from $\mathbb{S}$*

Since $\mathbb{N}' \subsetneq \mathbb{N}$ this clearly defines an acyclic graph.

Definition 7 (Refinement Proof Graphs) *Let $\mathbb{N}$ be a set of proof nodes, $N \in \mathbb{N}$ a proof node of proof layer i and $J = (PC(\overline{P}), \overline{N})$ a justification of N. Let $\mathbb{N}_R^N$ be the refining proof nodes of N and $\mathbb{R} := \bigcup_{(\varphi,(abs_i^j,\overline{N}'),A,R)\in\overline{N}} \overline{N}'$ the refinement nodes of the proof nodes in $\overline{N}$.*

Then $\mathcal{N}$ *is a* refinement proof graph *of* J *if, and only if, for each* $N' = (\varphi', \overline{J}', A', R')$ *from* $\mathbb{N}_R^N$ $\mathcal{N}$ *is a proof graph of* $(\varphi', J', \bot, R')$ *from* $\mathbb{R}$.

Definition 8 (Hierarchical Proof Graph) *Let* $\mathbb{C} := (\mathcal{C}_i)_{\in I}$ *be hierarchical proof constructors with smallest element* $\mathcal{C}_0$. *A* hierarchical proof graph wrt. $\mathbb{C}$ *is a 3-tuple* $P := (\varphi, C, \mathbb{N}, \mathbb{C})$ *where* $\mathbb{N}$ *is a set of proof nodes,* $C = (\varphi, J, A, \bot)$ *is a proof node of proof layer* 0, $C \in \mathbb{N}$. *The* open goals *of* P *are all proof nodes with an empty justification.*

Let $\mathbb{H}$ *be the hypotheses within the support nodes of* C. *Then the hierarchical proof graph is* closed *if, and only if,* $\mathbb{N}$ *is a proof graph of* C *from* $\mathbb{H}$.