Thesis Proposal – Dottorato di Ricerca in Informatica

Towards formalized mathematics repositories based on type theory

Matthias Puech*

Dipartimento di Scienze dell'Informazione — Università di Bologna
Laboratoire PPS — Université Paris 7–Denis Diderot

dir. Andrea Asperti *&* Hugo Herbelin

February 28, 2010

Contents

*puech@cs.unibo.it

Introduction

The fields of formal proofs, automated reasoning and verified software, have made great advances in the last forty years, since the seminal works of on the checkable proof language Automath by De Bruijn [1970] and the automated theorem prover Nqthm from Boyer and Moore [1988]. They appear today as a mature technology, used not only academically but also in the industry; the diversity of the approaches also witnesses the vivacity of the field: there are plethora of systems implemented for various, more-or-less specific uses, from the all-automated theorem prover for a domain specific application to the general-purpose, interactive mathematical assistant embedding a rich and expressive logic.

In particular, *proof assistants*, also known as interactive theorem provers, are tools intended to help the human to develop formal proofs: the proof itself is constructed and checked by the machine, but guided by the human in an interactive collaboration. Among these, we count Isabelle/HOL, PVS, Coq, Mizar, Matita and much more. These tools have recently enjoyed a great success, leading to rich formal mathematics libraries and complex developments, see Asperti et al. [2009] for a critical and historical account on formal verification.

The factors of this success are in our opinion threefold: first, advances on computational logic have lead to the development of powerful and solid foundational languages of proofs, not only made intuitive for the mathematician by borrowing their methods, but also suitable to the exploitation of the computational power of modern computers. Secondly, the wealth of implementation efforts has lead to a well-understood architecture (Asperti et al. [2007]), and efficient methods for the multitude of algorithmic tasks performed in this context. Finally, the tight interaction between users and developers of these systems (it is actually far from exceptional to be both) has helped creating rich and usable interfaces, witness for example the reinterpretation of the proof language of Coq by Gonthier and Mahboubi [2008] originally intended as domain-specific but inspiring new generations of systems.

However, a lot of the paradigms now taken for granted in the development of formal mathematics are inherited from programming. For example, most proof assistant are based on an LCF-like, procedural tactic language for devising proofs, making the act of proving alike to the one of programming and strayed from the common vernacular used in informal mathematics. Moreover, the validation process of a formal proof, stepwise and linear because modelled after the automatic process of compilation, do not reflect the way a user would like to interact with the system: not only would he like to make progress and advance toward the solution (the only choice today), but also to retract from previous declaration, refine and modify his idea... Also, this rigidity does not leave room at all for team-work, where several people interact on the same (large) document, possibly modifying concurrently the main development in an incompatible way, between each other or with respect to previous developments.

If, following Dowek [2007], we believe that machine-checkable proofs and the mechanization of proof-search made mathematics enter its *"industrial era"* after a long history of handicraft, an era where unverified, one-man work is not acceptable anymore due to the increased complexity of the manipulated concepts, then we are in urge of finding techniques to adapt these new tools to the practice of mathematics.

We believe that the simultaneous work on two aspects of this problem would bring up new potentialities and ways of formalizing mathematics. Increasing the inference power of the proof language is one of them: most of today's proof languages are tightly bound to the underlying logic, whereas informal mathematics use extensively and implicitly notations, isomorphisms and shortcuts. Most proof assistants provide the strict minimum to deal with them, usually in a more-or-less ad-hoc way; increasing the capability of automating "trivial" parts of the reasoning would be a first step to the design of a high-level, highly ambiguous but still fully checkable proof language. A second step, concerning the interactivity of the formal development process, is the study of the impact of changes in large developments. A high-level proof language comes with possibly high computational power required, and one cannot afford then to recheck the whole development after each change, as small and insignificant as they may be. Moreover, we want to develop the interactive aspect of these tools, to be able to use them even during the discovery phase, where change happen often and deeply into the structure of the development. By analyzing finely the dependency between concepts and even intra-concept, we hope to render this interaction between the computer and the user, but also even between several users.

All these tools are, in a nutshell, implementations of a particular logic, along with a rich language to conduct the construction of proofs, mechanisms of inference to relieve the user from the most tedious and repetitive tasks, and input facilities. While the ideas underlying the input of proofs are shared by most of these tools,

a particularly interesting subclass of them are those based on *type theory*. Type theory is a calculus having the notable feature of being able to be seen dually as an intuitionistic logic and as a programming language allowing to give and check rich specifications to programs. This duality, also known as the Curry-Howard correspondence, opened a whole new field of research, both practical and theoretical.

We propose here to tackle the two issues developed above in the context of proof assistants relying on type theory, and especially the two developed in both our universities: Matita for the University of Bologna, Coq for the University Paris 7. This proposal is therefore organized in two parts. The first is an account of the state-of-the art in terms of automation of proofs, both on a proof-theoretic point of view and on existing solutions implemented. It finishes with some directions considered for further work and works in progress in this area. The second part presents an ongoing work on *semantic patches*, a preliminary theoretical study for the management of changes in type theory that could eventually serve as a basis for an integration into an existing system. It finishes also with some possible directions for further work. We begin with a short overview of the architecture of the proof assistants we plan to work on.

Architecture of a proof assistant

Both our proof assistants of interest, Coq and Matita are based on the *Calculus of Inductive Constructions (CIC)* of Paulin-Mohring [1996]. It is a very expressive logic, built on a variant of Martin-Löf's type theory with polymorphism à la System F and a hierarchy of predicative universes. It supports, as in Martin-Löf presentation of his type theory, the inductive definitions of objects, and supports the proofs-as-types paradigm. Among other features, it provides recursive function definitions and an impredicative universe for dealing with logical propositions.

Numerous systems based on a variant of this framework have been implemented: Agda, NuPRL, Epigram and much more. As time went by, the architecture of these tools has become better and better understood. One particularly important criterion in their conception is the so-called *De Bruijn criterion*. They all share a common layered software architecture, as sketched for Matita in Asperti et al. [2007] for example, and are often qualified of "skeptical proof assistants". We give a brief overview of each layer from the innermost to the outermost one, as it is implemented in Matita (this may vary for other systems):

Kernel Their principle is to isolate a little portion of human-understandable code, the *kernel*, only dedicated to the task of proof-checking, i.e. the verification of the well-typing. This component is responsible for the validation of all generated proofs, and is this way the only piece of code that must be trusted in order to trust the whole system.

Refiner Writing directly proofs in the language of the CIC is an almost impossible operation: not only would it be very hard to follow, but also very boring, because the language is very redundant. On top of the kernel, the *refiner* is in charge of completing all logical information omitted by the user, with the help of dedicated mechanisms: *type inference* allows you to omit some type constraints if they can be inferred by the system from the context. *Coercions* simulate a common practice in the mathematical discourse, i.e. the use of a super-structure in place of one of its components (example, a group instead of its carrier set). The more recent *canonical structures* and *type classes* allow to avoid mentioning the context by choosing, on one side a canonical representative of the context (a particular mathematical structure) or relying on inheritance between structures.

Library management All proofs and definitions in these proof assistants are devised thanks to a mechanism of definitional equality (in the sense of definition expansion). All the definitions are stored in a library, with the ability to import and reuse previous development. This layer is in charge of

the management of such a library: loading, saving, managing namespaces as well as modifying previous definitions.

Proof management Some of the systems mentioned allow the stepwise construction of proofs thanks to a library of tactics, which are tools to build proofs incrementally by refining the previous state of the proof. This layer manages the call to tactics in this process, and the verification of their result by the refiner (which in turns calls the kernel).

Tactics This layer makes up the set of specific tactics of proof refinement. It is usually built in a hierarchical manner, some tactics calling some others for specific subtasks.

Disambiguation Very often in the context of informal mathematical vernacular, ambiguity of the notations used happens. This layer is responsible for mapping potentially ambiguous notations and names to their logical, non-ambiguous counterpart by inferring it from the context.

User interface Then, some logic-independent layers are in charge of the parsing of formulas, their display, and graphical user interface management etc. The traditional model of GUI for a big family of these tools (implemented in the emacs editor as ProofGeneral) is a text editor with three panes: one displaying the actual script, the second the state of the current proof, and the last the possible messages to the user. One compiles a part of the script by moving a cursor forward and backward.

1 Automation in proof assistants

The kernel of interactive theorem provers based on type theory, namely the logic they implement and their typing algorithm, is now a well-understood part of such systems. But the very fine granularity of the proofs one can write directly for the kernel to check makes this act almost impossible for any relatively involved proofs. This led to the direction of improving the capacity of these systems to receive partial proofs, possibly with omitted parts or information, and infer them back fully, to the point where the kernel can check them. Many different mechanisms fulfill this task on different levels: *coercions* permit to assimilate notions corresponding to different underlying objects, *implicit arguments* allow to omit redundant arguments of functions and theorems, the recent *type classes* are an incarnation of the ad-hoc polymorphism present in some programming languages. To a certain extent, most of the tactics of Matita or Coq belong to this category as well: their application triggers the construction of complex proof objects.

In this section, we will focus on one of the most involved of such mechanism, commonly called *automation* tactics or mechanism. As opposed to the others, the goal of these is to provide a high-level method for (more-or-less blindly) searching a proof of a given statement. The utility of these helpers are needless to be proved anymore, but we want to advocate here a particular use of them. Two problems arise from the current, low-level view on formalization. First, devising proofs directly in type theory or with the help of low-level tactics is a tedious exercise and reading an already proof is often difficult. Secondly, the different variations and peculiarities of each proof assistants make their proofs not interoperable, resulting in isolated, balkanized formalizations and duplicated efforts. The development of the automation techniques of all kind are a direction towards the compensation of both these problems: they allow to alleviate the proof burden and are a required step towards the design of very light proof languages that can be embedded in different foundational dialects and proof assistants, and is therefore necessary for the design of large, formalized mathematics repositories.

1.1 State of the art

We begin by reviewing some of the common techniques of automated theorem proving, and discuss their adaptation and limitations in the setting of type theory, along with implemented generic methods of proof search in proof assistants.

1.1.1 First-order theorem proving

Automated theorem proving Automated theorem proving, or automated deduction, is the field dedicated to the search of proofs of mathematical theorems

by computer software. Theorem provers are tools taking a mathematical statement as input, and outputting a proof of it (or just succeed) or fail or loop forever if the statement isn't provable. Most general purpose theorem provers rely on classical first-order logic (FOL). These include for example Vampire, Ergo, Waldmeister, Mace4/Prover9, SPASS or more recently Imogen. In this part we will review some of the popular techniques and framework for classical and intuitionistic first-order theorem proving and discuss their potential use in type theory.

Resolution of Robinson [1965] is the one, if not the most popular complete method for semi-deciding the refutation proof problem: given a statement, we can apply this method; if the statement is unsatisfiable in FOL, the method will eventually terminate with a refutation of it. The design and implementation of such a method is discussed at length in Riazanov and Voronkov [2002]. Many recent provers rely on resolution, which has proved empirically to be the most efficient. Since Robinson's discovery, many complete refinements have been proposed since its introduction: ordered resolution, selection etc. (see Bachmair and Ganzinger [2001] for an overview). It consists of two phases: the transformation of the problem into clausal form, and the resolution itself, which is the iterated application of a (single, complete) rule mimicking the *cut* rule of natural deduction.

Equality is an ubiquitous notion in mathematics. Yet, FOL doesn't include any special treatment for this symbol, which makes it very inefficient to use in practice. A better way to treat the equality is by means of rewriting. It is the goal of *paramodulation* (Wos and Robinson [1968]), *superposition* and their derivatives (see Nieuwenhuis and Rubio [2001] for a complete survey). They all are dedicated inference rules for treating equality externally to the logic, as a mean to replace equals by equals in formulas. For this purpose, *completion* (Bachmair and Ganzinger [2001])is a useful tool to orient equations and avoid uncontrolled rewriting as much as possible.

Besides these popular techniques, the *tableaux* method and their variants (see Hahnle [2001]) is a proof search method for first-order sequent calculus. It has been implemented in various theorem provers, e.g. 3TAP (Beckert et al. [1996]) and the one from Paulson [1999]. FOL extended with inductive definitions has been also the object of studies in ATP systems, resulting in methods like *rippling* (Bundy et al. [1993], Bundy [2001]), or *inductionless induction* (Comon [2001]). The first is a method to cope with the proofs of inductive cases by syntactic methods, the second is an efficient embedding of induction into a extension of FOL. Intuitionistic theorem proving is still an active field of research, a recent success being Imogen from McLaughlin and Pfenning [2009], relying on the focusing discipline (Andreoli [1992]).

1.1.2 Proof search in ITP systems

In order to give more inference power to existing proof assistants, recent efforts have been put on adapting the efficient proof-search methods designed by the automated deduction community to proof assistants, by adapting them in order to generate proofs and translate them to the tool's format, for it to recheck them afterward. We will first do a quick review of the efforts in this domain.

Embedding first-order calculi First-order theorem automated theorem provers and the paradigms they rely on are rich and very efficient. It would seem quite natural to make use of these advanced tools for interactive theorem proving. One can achieve this in two ways, either by interfacing an existing tool through e.g. a dedicated tactic, or by designing a purpose-built theorem prover that fits the particular needs of the interactive system. PVS has a rich collection of decision procedures and generic proof search tools. HOL was interfaced with various first-order theorem provers such as resolution-based Gandalf (Hurd [1999]) and Metis (Hurd [2003]). Isabelle was first interfaced with its dedicated prover based on the tableaux method by Paulson [1999] and more recently with Vampire (Meng et al. [2006]). A generic communication protocol between Coq and external provers is being developed (the tactic extern). Since recently, Matita uses its own paramodulation-based tool (Asperti and Tassi [2010]).

Generic proof search in type theory Besides the embedding of first-order automation techniques, proof assistants generally come with some generic proof search facilities, devised this time entirely in the type theory. These tactics, known as auto in Coq and // in Matita, have at their core a mechanism analogous to the resolution process introduced earlier. There are important differences with the integration of external tools as introduced above: their integration directly into the type theory allows them to exploit its full higher-order nature, along with their implementation details like definitions unfolding or inference mechanisms.

1.2 Issues in ITP automation

1.2.1 Embedding first-order calculi

The deep conceptual differences between both approaches (interactive and automated theorem proving, higher-order vs. first-order) raises questions that still need generic answers. We plan on developing the line of work initiated by Asperti and Tassi [2010] in Matita.

Proof translation Some of the interfaces from interactive systems to automatic ones share the same architecture, at least in systems based on type theory. We will expose here the approach taken in Asperti and Tassi [2010] With the exception of PVS which doesn't require a proof and trusts its decision procedures, they all need to get back a *trace* from the external tool to reconstruct a proof in their own language. The architecture is then composed of:

1. A forgetful translation from the type theory to the (first-order) language of the tool. It takes the current goal along with a set of hypothesis to a (axioms, problem) pair. This pair is run by the external tool and possibly returns a trace of the proof;

2. An interpretation function, which interprets the trace into a (possibly incomplete) proof object in the type theory;

3. A retyping mechanism, which infers the missing part of the returned proof object and checks it in the type theory.

From this methodology, we see that the integration of external tools relying on weaker or at least different logics is necessarily an incomplete method based on heuristics, and it is often difficult to predict their behavior.

The forgetful translation is not only the operation of removing all type information from a statement to get a first-order term: type theory possibly performs computations in statements, has defined objects that can be unfolded. For example, how do we translate the higher-order goal $\forall n\ m\ p, (n+1)*m \leq (n+1)*p \rightarrow m < p$? Possibilities are numerous: $S\ n*m \leq S\ n*p \vdash S\ m \leq p$ or $m+n*m \leq p+n*p \rightarrow m+1 \leq p$ or any intermediate steps of unfolding the constants $+$, $*$ or $<$ and reducing their content. Depending on the set of axioms provided to the external tool, some will be provable, some not.

The interpretation of trace object is also a delicate operation, heavily depending on the calculus used by the external tool. Traces are representations of the success of the proof process: they could be the list of rewriting steps from the axioms to the problem statement for paramodulation, or the list of clauses and literals used by the resolution. The interpretation of a paramodulation step is pretty easy since its logical meaning (rewriting) is admissible in type theory (by means of the induction scheme of equality). It becomes quite involved when considering a *blind* method like resolution: the initial preprocessing transforming the problem into a set of clause, which is not admissible in intuitionistic logic, has to be traced carefully and checked for validity. The same goes for another blind method, SMT solving.

Finally the last part of the process – retyping the result of the interpretation – is not guaranteed to be feasible since the type information was lost in the first place. Sometimes the external prover would success and return a valid trace of the

first-order translated goal, which has no meaning in the type theory and couldn't be retyped. The inference of the missing type information can also be quite involved and may itself rely on proof search (see Asperti and Tassi [2010] for details).

Automated vs. interactive paradigms But the conceptual differences between automated and interactive theorem proving go beyond the only untyped vs. typed or first-order vs. higher-order problems.

One lies in the very purpose of automation in these two paradigms: ATP systems are usually provided with a small set of axioms for a given theory, often minimal, and deduce new facts from these onwards, until the goal is reached. Thus, they generate *cut-free* proofs, in the sense that they don't infer useful intermediate lemmas that would need to be proved afterward. On the other hand, automation in ITP system usually relies on a huge set of objects, both axioms defined by the user, logical rules in the form of constructors and destructors for logical connectives, and intermediate proved lemmas, and this set grows each time a new object is introduced.

From this emerges two difficulties of integrating ITP techniques into ATP. The first is that provers are not necessarily optimized for treating big sets of initial axioms, nor for dynamically extending those sets as is required for an efficient integration. The pre-treatment of generating and translating these sets might be prohibitive if not correctly designed in the ITP software from the beginning. The second and most important difficulty is the one of choosing among those sets the needed objects. Usually, the whole library of proof assistants contain very redundant objects: proofs trivially subsuming other proofs (e.g. by computation, or by superposition of two results), different proofs of the same statement, different implementations of the same logical connective... To be treated efficiently and avoiding duplicated work, one would have to choose in the whole set of objects of the library the relevant subset for the given problem, minimal but with enough information for the proof to succeed. This issue has not been throughly tackled and would need a careful study.

1.2.2 Generic proof search in type theory

The tactics and methods of proof search as implemented in Coq and Matita can fulfill different purposes and usages, that we can classify according to two criteria. First, we discriminate on the *purpose* intended by the user: it might be used to automatically fill in some of the small "trivial" gaps leaved by the user in a proof (what we call small-scale automation), for example leaving to the machine the inference of one trivial case among others; it might also be intended as a help to devise the proof when the user totally lacks intuition of it (large-scale automation). This second purpose is obviously more difficult to achieve, and it

is probably unrealistic considering the results obtained in the dedicated field of automated theorem proving. Secondly, we separate two different *modus operandi* of the automation mechanism: some of them allow to automatically finish a proof and fill in the proof tree until its leaves, may it be a trivial step of reasoning (small-scale) or a more involved one requiring to explore deeply the search space (large-scale). These take the form of classic tactics like auto or //. Some of them on the other hand are *interleaved* in the proof process and allow to incarnate a certain *quotient* on applicable proof steps or statements. This is the case of Coq's type classes, or the recent *smart apply* tactic of Matita.

Intuitionistic resolution As explained earlier, the resolution mechanism as usually presented is classical by nature. However, it is well known that this appears as nothing more than a notational facility, and its core reasoning scheme is in fact intuitionistic. It proceeds simply by maintaining a list of hypothesis and lemmas along with the goal; at each step, it applies in turn all hypothesis and lemmas whose head unifies with the current goal:

$$\textsc{Intro}\ \frac{\Gamma, x : A \vdash B}{\Gamma \vdash \Pi x : A.B}$$

$$\textsc{Apply}\ \frac{\Gamma \vdash A_1 \ \ldots\ \Gamma \vdash A_n \qquad \Gamma \vdash B \equiv C \qquad f : \Pi(x_1 : A_1) \ldots (x_n : A_n).B \in \Gamma}{\Gamma \vdash C}$$

This *backward-chaining* method (from the goal to the hypothesis), although not complete, helps the completion of easy goals. Because of the dependent nature of the statements in the CIC however, this process is made more complex by the fact that the application of a Apply rule can introduce *metavariables* in the goal, that is holes in a term of a known type. For example, the application of a transitivity lemma on the relation R to a goal of the form $R(a, b)$ produces two goals sharing a metavariable: $R(a, \mathsf{X})$ and $R(\mathsf{X}, b)$. This complicates greatly the control of the proof-search process: one could naïvely think of it as a backtracking process on each Apply rules in turn, but this is erroneous: a goal, even closed by the application of a 0-ary application (an instance of the Init rule of natural deduction), could result in a backtracking if a metavariable was instantiated. Imagine in our example that the goal $R(a, \mathsf{X})$ is closed by a proof of $R(a, c)$, but that the second instantiated branch $R(c, b)$ fails to be closed. One has then to backtrack on the first goal rather than on the previous one, to be able to find a second, successful instantiation e.g. $R(a, d)$ and $R(d, b)$.

Due to some limitation of its implemented tactic language, this subtlety is not

taken into account in Coq; whereas it is solved correctly by Matita. Both implement a depth-first algorithm of the search tree.

Choice of hypothesis Contexts of proof assistants are not simple objects as are their theoretical counterparts above: they are composed of many segments and many kind of different objects, like definitions, inductive objects, constructors, axioms. One question relative to these sort of automation is: should we consider all of them as equally probable candidates? Both proof assistants solve the question quite differently. They both implement a mechanism of priority among these objects, taking some heuristics to determine them like the number of goals opened by the application or the number of metavariables introduced. Besides that, Matita takes however the approach of considering all objects as candidates except the most general ones (induction principles for example that can be applied to any goal), whereas Coq chose the approach of user-defined *hints* databases, associating to each lemma a priority and a strategy such as Resolve (general strategy) or Immediate (the goal has to be closed in one step after its application). Possible directions like the refinement of these heuristics as well as the fine control of the strategies are discussed in 1.3.2.

Definitions and computations Due to its higher-order nature and its definitional mechanism, the logics underlying these proof assistants are subject to some notions of reduction. Actually, the conversion rule of the CIC ensures that the statements of the logic are all quotiented w.r.t computation. These computations can be split into two different forms, that serve different uses: first, some real computation can occur, as in $P(2+2) = P(4)$ or $Q(S\ n * m) = Q(n * m + n)$, and a lemma applicable to one of its part should be equally applicable to the other. The second is the use of the *definitional mechanism* to abbreviate some often-used statement forms: for example, one might want to define transitive(R) to expand to $\forall xyz, R(x, y) \rightarrow R(y, z) \rightarrow R(x, z)$.

This is the responsibility of the *unification* algorithm to generally determine when these situation happen, when applying a lemma for example. It is a complex part of a proof system, and often involves expansive computations. Thus, iterating this process again and again when trying to Apply all possible lemmas during proof search can lead to substantial delays. To achieve good performance, indexing techniques are generally used to rule out as fast as possible lemmas that don't need to be treated for a given goal because they won't ever unify. However, these data structures are purely first-order, and it is non-trivial to use them enough so as to improve the efficiency but not to restrict too much the candidates and miss some possibly successful ones. These techniques have been experimented, and are discussed in more details in 1.3.3. In particular, the adaptation of these data

structures to take abbreviation-unfolding is a non-trivial, open problem.

1.3 Further directions

Besides the integration of the first-order prover in Matita, We would like to improve the existing general-purpose proof search techniques available in these proof assistants. The general methodology adopted is to design a procedure directly for the Calculus of Inductive Constructions, by focusing as much as we can on the user's need. A preliminary analysis has been done by identifying clear examples of situations where automation is wanted but that the tools available (in Coq for the moment) are unable to cope with.

1.3.1 Sequent calculus proof search

The calculus of constructions is usually implemented in a natural deduction fashion, at least in Coq and Matita: the context is used only to retrieve previously defined object, and there is no way to modify it after its introduction. This proof style is supported by the apply family of tactics that applies a lemma or an hypothesis to the goal, progressing in a *backward-reasoning* fashion: one progresses from the goal to the assumptions. This approach is however not always intuitive, and a second kind of reasoning is supported in both tools. The lapply family of tactics (or apply with) apply a lemma or an hypothesis to another hypothesis in the dual *forward-reasoning* fashion. In this case, one progresses by transforming the assumptions until they eventually match the goal. These two complementary approaches correspond to two presentation of the theory of types, respectively the natural deduction and sequent calculus style. Pure Type Sequent Calculus (PTSC) was introduced in Lengrand [2006].

Forward-reasoning The automation methods described in 1.2.2 are, to the best of our knowledge, relying on a *backward-reasoning* mechanism: the system applies iteratively all possible applicable lemma and hypothesis. This method has severe disadvantages: consider the sequent $A \wedge B \vdash A$. The only way of proving it automatically is to introduce a metavariable and apply the projection $proj2 : \forall XY, X \wedge Y \rightarrow X$ to produce the sequent $A \wedge B \vdash A \wedge \mathsf{Y}$ and to conclude by instantiating Y to B. However, allowing the application of these projections, and most lemmas that produce metavariables leads to an enormous search space since they can be applied to any goals. They should be avoided as much as possible, preferring simpler proofs: here an obvious solution is to apply the rule of sequent calculus:

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \wedge B \vdash C}$$

to obtain the sequent $A, B \vdash A$ and conclude immediately.

In the CIC, the conjunction is a defined object. In fact, $\wedge$ is an inductively defined non-recursive predicate with one constructor and two independent hypothesis: $A \wedge B := conj : A \to B \to A \wedge B$. By α-conversion, the above rule of the sequent calculus is valid if we replace $\wedge$ by any connective of the same shape. It is also valid for connectives of the same "kind": if we call conjunctions all inductive definitions having an unique constructor, we can devise rules of forward-reasoning for all of them.

Example 1. *All of $\exists$, $\wedge$ and record types are conjunctions. Therefore we can devise the rules:*

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \wedge B \vdash C} \qquad \frac{\Gamma, x : A, B(x) \vdash C}{\Gamma, \exists x : A.B \vdash C} \qquad \frac{\Gamma, x_1 : A_1, x_n : A_n \vdash C}{\Gamma, \{x_1 : A_1; \ldots; x_n : A_n\} \vdash C}$$

The case of disjunction Can we treat inductive definitions with several constructors the same way? The naïve application of the previous idea leads to a very redundant proof search. Consider the goal $A \vee B \vdash C$. If we apply the rule of left disjunction to it, we are left with two goals $A \vdash C$ and $B \vdash C$. If we can prove the sequent $\vdash \pi : C$ without using either A or B, then the previous application doubles the work: we have to copy the proof π in both sequent. Therefore, the left rules are to be applied more carefully for disjunctions.

A second (but wrong) idea is to apply them lazily: we keep them intact, until the moment where we meet one of their components in the goal. Then we can destruct them and continue with the second branch:

$$\dfrac{\dfrac{\dfrac{}{\Gamma, A \vdash A} \qquad \dfrac{\vdots}{\Gamma, B \vdash A}}{\Gamma, A \vee B \vdash A}}{\dfrac{\vdots}{\Gamma, A \vee B \vdash C}}$$

Unfortunately this method is not complete, as it may be possible to prove $\Gamma, B \vdash C$ but not the degraded judgement $\Gamma, B \vdash A$.

We propose the following improvement: each disjunction sees its components recorded, and proof search continues leaving the disjunction untouched. If an instance of one of the component is met later once a proof π has been generated, we backtrack to the point where the disjunction was introduced, break the disjunction, apply π on its corresponding component, and we are left with the second branch of

the disjunction in the hypothesis.

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, A \vee B \vdash A}}{\pi\left\{\vdots\right.}}{\Gamma, A \vee B \vdash C} \quad \Longrightarrow \quad \dfrac{\dfrac{\dfrac{\overline{\Gamma, A \vdash A}}{\vdots}\pi}{\Gamma, A \vdash C} \quad \dfrac{\vdots}{\Gamma, B \vdash C}}{\Gamma, A \vee B \vdash C}$$

To sum up, this technique allows to rewrite part of a proof *during* proof search to optimize its number of inferences. This way, the depth of a proof can be reduced and since most automated proof search have their depth bound, we increase the inference power of these techniques.

Our goal is to generalize these techniques to all "kinds" of inductive. The strategy described above could be generalized to recursive inductive by taking into account the induction hypothesis, then performing induction by itself:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, n : \mathbb{N} \vdash C(0)}}{\pi\left\{\vdots\right.}}{\Gamma, n : \mathbb{N} \vdash C(n)} \quad \Longrightarrow \quad \dfrac{\dfrac{\dfrac{\overline{\Gamma, C(0) \vdash C(0)}}{\vdots}\pi}{\Gamma \vdash C(0)} \quad \dfrac{\vdots}{\Gamma, m : \mathbb{N}, C(m) \vdash C(\mathsf{S}(m))}}{\Gamma, n : \mathbb{N} \vdash C}$$

By combining forward- and backward-reasoning step in an efficient way and respecting the underlying logic, we believe that we can build an automatic proof search method, not only more efficient than the currently implemented methods, but also more expressive and able to cope with the problem of deferring "trivial" proof steps to the theorem prover, leaving to the user the load of only sketching the proof.

1.3.2 Strategic proof search

Proof search however is not only about data structures and inference rules. It is a dynamic process involving complex backtracking among goals, and even more complex in the presence of metavariables. The goal of fast automation requires a fine control on backtracking on the search tree, as was already hinted in the previous section: attaining a given goal made us backtrack to a special position and remember some pieces of information (the proof π) to replay it without possible backtracking on the next goal.

Focusing We can go further in this idea by putting into practice ideas that emerged in the *focusing* discipline of Andreoli [1992]. Focusing was introduced as a winning strategy for proof search in the Linear Logic of Girard [1995], exploiting the redundancies present in naïve proof search. These redundancies are actually not on space, but on time, i.e. it is not a problem of data structure for the proofs (as was solved by *proof nets*), but a problem of backtracking in the proof search. Indeed, it was identified that some rules of Linear Logic are *invertible*, that is, they can be read in both directions, from top to bottom or from bottom to top. Thus the application of such a rule does not change the provability of a sequent and can be applied *eagerly*. The eager application of a rule is its application without the need to backtrack before them: these rules just simplify the goal without modifying their provability. Examples of invertible rules are the rules on conjunction and disjunction, both left and right. This remark led to the specification of *polarities* defined on sub-formulas: the *negative* parts of a formula comprises all chain of invertible connectives, whereas the *positive* parts make up for the non-invertible one. A *phase change* occurs during proof search when a connective is met that is not of the same polarity as the previous one. Focusing is the strategy defined by:

- Applying eagerly all the connectives during the negative phase,
- Focusing on an arbitrary hypothesis during the positive phase and apply eagerly all possible rules,
- Backtracking occurs only at phase change.

These ideas, although having been defined for linear logic, can apply to intuitionistic logic as well, by embedding one logic into another by a (non-unique) polarized translation. The automated theorem prover Imogen by McLaughlin and Pfenning [2009] is based on these ideas, and proved to be a success.

A focusing approach applied to dependent type theory is still to devise though, and would constitute a theoretical advance with clear applications to automated proof search in type theory.

Backtracking strategies More generally, an efficient proof search requires a control of backtracking that goes beyond what is currently implemented in Coq and Matita: both proof assistants consider the all leaf of the search tree as potential backtrack nodes, and don't provide a way to specify strategies in a general way. Furthermore, no distinction is made in the search between logical objects like the defined connectives or the equality, and full lemmas with non-trivial proofs. Coq's $\mathcal{L}_{tac}$ (Delahaye [2001]) is an attempt to provide a full-featured programming language for automatic tactic application. It provides a recursion operator as well as a matching construct on the current goal. However, it fails to define general

and extensible way of defining general-purpose automation tactics, specifically to deal with non-determinism, and is confined to domain-specific application, like the propositional fragment of logic (tauto), or the decision of some order-related statements (order).

We propose, as a first step towards the design of strategies, to have a clear developer-side language of specification of these strategies, expressive enough to cope with previous ideas: backtracking by keeping a (partial) proof, focusing on a particular formula, eagerly advance on the application of a particular rule. Also, we would like to abstract from the *names* of the inductive objects to select them according to their *kind*: (dependent, indexed, recursive) disjunctions, conjunctions... The design of such a general method is of course interleaved with, and will be nurtured by the discovery of particular wanted strategies. A good starting point could be the backtracking monad of Kiselyov et al. [2005].

1.3.3 Implementation issues

When it comes to implementation, all the techniques mentioned above require the manipulation of large sets of terms, and a recurring problem is to retrieve terms satisfying some structural conditions, like unification with a given pattern. One-to-one unification has a well-known, simple and efficient algorithm, but it becomes under-optimal when applied iteratively on large sets of terms, because a lot of sharing is lost. Term indexes are data structure used to store these sets, making the retrieval of unifiers to a given pattern more efficient.

One data structure used for this purpose has been *discrimination trees* or *nets* (McCune [1992]). It operates on the same principle as *tries*, viewing the terms as flattened strings and retrieving following a given prefix (the pattern), like a dictionary. A critical inefficiency of this structure is due to the loss of information resulting from the flattening of terms. *Substitution trees* (Graf [1995]) are a popular alternative. Their main idea is to index the terms according to their *mgu*'s, and form a tree of substitutions. All these techniques are devised for first-order term algebras. With the emergence of automated provers for higher-order logics like Twelf, some attention has been drawn on the problem of *higher order indexing*, i.e. indexing of λ-terms modulo β-reduction (Pientka [2003]).

This problem is in general undecidable, but such a behavior is definitely a wanted feature for our use. We will focus at first on a subproblem, already interesting and not yet tackled: the goal of adapting these structure to be used modulo definitional equality to abstract from certain abbreviations used commonly.

Moreover, if we decide to tackle the problem of the implementation of sequent calculus proof search in a proof assistant, where the search is not only goal-directed anymore but to some extend also directed by the hypotheses, we then want to index hypothesis and be able to retrieve efficiently a term having e.g. a given form

of hypothesis. However, due to the set nature of the context, the pair (hypothesis, goal) cannot be presented as a term since it is to a certain extend possible to permute all hypothesis. We then need to devise an efficient data structure of term indexing taking into account to these permutations. It seems not trivial to adapt existing term indexing methods for this purpose. Moreover, hypothesis in a context are in type theory possibly dependent: the instantiation of one hypothesis triggers the modification of all subsequent dependent hypothesis. Our data structure should be able to instantiate the variables accordingly. This is however a known problem, and most of the time a simple solution exists: the carry of a closure substitution during the instantiation.

1.3.4 Isomorphisms in the CIC

The calculus of inductive constructions has the nice property of being a very minimal foundational language, in which even very low-level mathematical concepts can be defined. For example, the usual logical connectors $\vee, \exists, \wedge \ldots$, even the equality are not elementary objects but defined ones, thanks to the dependent and inductive types features. However, this freedom implies that a given usual *concept* can have a lot of different *implementations*. Some of them will be convenient or efficient for a given task, other for other tasks.

A simple example is the definition of the logical equivalence in Coq's standard library. It is historically defined as $A \Leftrightarrow B := A \rightarrow B \wedge B \rightarrow A$, but a recent idea was to change it to an inductive definition $A \Leftrightarrow B := \mathsf{build\_iff} : (A \rightarrow B) \rightarrow (B \rightarrow A) \rightarrow A \Leftrightarrow B$. Although both definitions are strictly equivalent and represent the same "usual" notion, such a change implied to rewrite almost all proofs relying on the first definition and was then aborted.

Another less-trivial example is the representation of (natural) numbers. They accept a wealth of different structural definitions — Peano, binary, machine, axiomatic integers... — each supporting its own reasoning style, having different efficiency properties etc. but, in the end, representing the same usual notion. When one wants to begin a new development using natural numbers, she has to choose from the beginning her implementation and stick to it throughout the development. A change of representation would require to rewrite a large part of the development and is therefore not affordable; moreover, accessing the results, or using properties of another representation is simply forbidden.

These facts suggest that a certain notion of *isomorphism*, or simply of *morphisms* between the *concepts* to be formalized and their (possibly numerous) *implementation* would be a valuable study for both the modularity of the proofs (more results to share among developments) and the abstraction of the proof language, which would rely not on the actual implementation details of structures but on higher-level concepts. The notion of type isomorphisms has been already well-studied in the

setting of functional programming languages like ML, notably by Di Cosmo [1995] and proposals of integration in type theory by Barthe and Pons [2001] have been made.

There seem to be many different approaches to this problem, some already considered, some not. A first approach is the definition of axiomatic *interfaces* for data structures or concepts, by means of records or module system in the fashion of modular programming, like it is done in the formalizations of algebra undertaken in the CIC. A natural number is a "*complete*" set of its "*atomic*" properties (terms to be defined): successor lemmas, recursion... Individual representations of natural numbers are then implementations of the interface. But what are these properties, and how can one be sure that they are "minimal" or "complete" wrt. the common concepts, or appropriate for her particular needs? Another pragmatic approach is to "elect" a canonical representation, and to have translation functions/morphisms from each of the other representations to it. This way, we lose their interesting structural properties (e.g. efficiency), but we inherit from all properties proved on them. A more *bottom-up* but automatizable proposition is to work directly at the syntactic level and assimilate isomorphic terms in the calculus. Contrary to the other suggestions, this is a meta-investigation of the isomorphisms in the logical framework that could lead to the (partial) automation of reasoning modulo some implementational details (like e.g. the *iff* example above). Inductive types seem to provide a rich framework for studying these isomorphisms, since their purpose is precisely to provide one-to-one correspondence between a "concept" (the inductive type) and its possible alternative "implementations" (the constructors). For example, we could assimilate function types which are equal modulo permutation of arguments, or inductive types sharing the same structure, types defined on an inductive type and the corresponding substituted inductive type... Building up on this principle, we can hope to capture a large class of equivalences. A third proposition is to rely on existing automation techniques to deal with isomorphisms. Most instances of these isomorphisms can be expressed as equalities, for example curryfying propositions: $\forall ABC, A \to (B \to C) = (A \wedge B) \to C$. By providing a way to apply logical rules up-to these equalities, we can somehow abstract from a part of the logical specificities. This is the approach taken in the new `sapply` facility of Matita described in Asperti and Tassi [2010].

The applications of such a study are numerous: abstract proof and logical language mapped "by construction" to the foundational language, modularity of the developments, increased automation...

2 Semantic patches

The second, complementary axis of this proposal is the attempt to tackle the problem of modularity and evolution in formally checked mathematical developments.

Even beside the notion of constructivism and the assimilation of proving and programming in type theory, it is interesting to investigate the possible relationship between the very *methodologies* employed in both fields, mathematics and computer science. And by that we even intend the comparison of the *daily workflow* of both scientist: one trying to prove a theorem, the other constructing a program to fulfill a task. How does one or the other elaborates his object of study? What kind of *a posteriori* modification is he prone to doing? What do these modifications imply on the validity of the whole edifice? How to relate how both scientists collaborate in a team? How do they rely on existing work to build up new results?

Recent efforts in the formalization of mathematical results have naturally led to these questions and many of them remain largely unanswered, but the tendency seems to be to adapt existing methods coming from software development, witness for example the recent introduction of modules or separate compilation in proof assistants like Coq or Matita , the use of dependency management tools (make) or version control system (git, svn) to build and manage versions of a project. The use of software engineering techniques for the management of a large formalization project is even the thesis advocated in the manifesto of the Mathematical Components team (Gonthier and Werner [n.d.]).

We propose to address a small part of these questions, namely the enhancement and adaptation of version control systems to the management of mathematical repositories, by means of what we preliminarily call a *semantic patch system*[1]. It is a generic system allowing one to express *transformations* on formal developments, and check that some semantic properties are preserved by the transformations, for example well-typing or operational equivalence. Eventually, we hope to be capable of expressing complex transformations on formal proofs, check them efficiently, and manage distributed repositories of mathematics with it, guaranteeing *by typing* the stepwise global consistency of the repository.

In the light of the Curry-Howard isomorphism though, we abstract totally from whether we are talking about proofs or programs. That is why, in the following, we start from today's situation, where existing version control system are used both in software and proof development, and we will not make any difference between both use. Depending on his interest, the reader can thus choose his own side of the isomorphism by translating program into proof, type into statement... or vice versa.

[1] Joint work with Yann Régis-Gianas

2.1 Motivations

The programmer and the mathematician know very well that developing a new theory or program is not a linear task: one do not open her text editor and write down her ideas in a unique pass. In the case of program development, most of the time is actually spent editing previously written code, correcting bugs or implementing new features which are interleaved with previous ones. This idea is not present at all in the text-editor/compiler paradigm, but to reflect it, *version control systems* usually rely on a textual differentiation mechanism (*diff*s): the comparison between two versions of a text file generates a *patch* which indicates which lines of text are to be added or deleted to transform one into the other.

Developments, may it be proofs or programs, are usually split into files, each of these containing a self-contained module of the whole development, an "atom" in some sense. To compile – or check – the whole developments, we use dependency management tools like `make`. These tools generate dependencies among the files, and launch the compilation only on files that have been changed since last compilation, and their dependencies. This process, known as *separate compilation* has files as atomic objects, and dependency generation is performed uniquely on them, ignoring their internal structure.

As intelligent and widely adopted as these tools have gotten, we believe that they are not adapted for proof developments. Indeed, whereas compilation of a program is usually fast enough for the programmer to rely on the usual interaction loop ((edit; compile)*; commit)*, the operation of proof checking is usually too expensive computationally to mimic this workflow. But even besides the time factor, this "traditional" way of formalizing mathematics hinders the process of mathematical discovery process: once a concept contained in a file is compiled, it is considered frozen and any changes to it require the recompilation of all files depending on it; the linearity of the development also gives no room for alternate, inequivalent definitions. This fact has nonetheless been shown to be of crucial to the mathematical discovery process by Lakatos [1964], and we believe that they should be taken into account in the realization of mathematical assistants like Coq or Matita.

Actually, we even dare to state that these tools and the workflow they suggest were not even adapted for program development in the first place, but were just a convenient approximation of the user's intent. Here are some hints to justify this argument: first of all, *any* change in a source file requires full recompilation of all files depending on it, whereas a finer management of dependencies is possible (function-wise for example). The basic operations of a version control system such as `svn` or `git` are often very sensitive, error-prone and based on heuristics (example, a *merge*). This is among other due to the fact that the signal-to-noise ratio of a textual *diff* is often high, because semantically void changes like altering

indentation have the same value as changing the argument of a function. Finally, as they are not aware of the meaning of the text they are managing, these tools don't provide any consistency guarantee on the resulting files (except textual ones).

All these facts, if they are not a major drawbacks when applied to weakly structured languages (e.g. assembler, or a LaTeX document), gain a crucial importance when considering a typed language (a functional programming language for example) : we would like to make sure that a patch does not break the well-typing (or any other property) of a source code. One more step forward, more structure-awareness becomes *essential* when they apply to a proof language, where the structure (the typing) *is* the only valuable information.

2.2 Methodology

We propose here to devise a system for *semantic patches*. It substitutes the idea of textual transformation by:

- First preferring an abstract syntax tree representation instead of plain text;
- Secondly embedding semantic data into the transformations, in order to be able to reason on them.

We want to design a *language of patches*, able to represent these transformations and their semantic properties, thus capturing local changes in programs, as well as their global effect on the whole project. We can see this goal as a refinement of the former idea of "dependency".

A semantic patch is a program intended to transform a program written in an *object language* $\mathcal{L}$. We focus, on the first iteration of the project, on the semantic assertion of well-typing: from the object language's syntax and typing rules $\mathcal{L}$, we derive typing rules for the patch languages $\mathcal{P}(\mathcal{L})$, and we can then type the patch. Our motto is:

> A well-typed patch is the guarantee that it transforms a well-typed program into a well-typed program.

In this sense, it is closely related to *incremental typing*: if a modification has been made on a program, it is expressible by a patch. To ensure that it will result in a valid program, it suffices to type the patch, and not the whole resulting program.

Moreover, our approach is meant to be totally formal, so that we can prove not only validity of our transformations, but also the completeness of our patch language and other properties. For this, we place ourselves in a constructive, higher-order metalanguage, namely the Calculus of Inductive Constructions (CIC), in which we will formalize in turn:

1. The object language $\mathcal{L}$ of our choice
2. Its metatheory
3. The patch language $\mathcal{P}(\mathcal{L})$, parameterized by $\mathcal{L}$.
4. Its metatheory

2.3 Related work

Both the study of metatheorical properties as proof transformations and of change impact in structured documents (e.g. proofs, programs) is not a new subject.

The Twelf project (Pfenning and Schürmann [1999]) is an implementation of the Logical Framework (LF, Harper et al. [1993]). It is a logic programming language able to represent logics or programming languages, and embeds an inductive metatheorem prover able to construct transformations of the $\Pi\Sigma$ form. It was used in Anderson [1993] to devise transformations of proofs in order to extract efficient programs.

The problems of managing a formal mathematical library have been dealt with in various proof assistant and mathematical repositories. The HELM project (Asperti et al. [2000], Asperti et al. [2006]) was an attempt to create a large library of mathematics, importing Coq's developments into a searchable and browsable database. Most ideas from this project were imported into the Matita proof assistants (Asperti and Tassi [2007]), especially a mechanism of *invalidation and regeneration* to ensure the global consistency of its library w.r.t changes, with granularity the whole definitions or proofs and their dependencies. The MBase project (Kohlhase and Franke [2001]) attempts at creating a web-based, distributed mathematical knowledge database putting forward the idea of *development graph* (Hutter [2000], Autexier et al. [2000]) to manage changes in the database, allowing semantic-based retrieval and object-level dependency management.

This idea, generalized over structured, semi-formal documents gave birth to *locutor* (Müller and Kohlhase [2008]), a fine-grained extension of the `svn` version control system for XML documents, embedding ontology-driven, user-defined semantic knowledge which allows to go across the filesystem border. It embeds a *diff* algorithm, operating on the source text modulo some equality theory to quotient the syntax. On the same line of work, we should mention the *Coccinelle* tool (Padioleau et al. [2008]). It is an evolution over textual patches, specialized on the C language, allowing more flexibility in the matching process, and was developed to deal with the problem of *collateral evolutions* in the Linux kernel. It embeds a declarative language for matching and transforming C source code, operating on text modulo defined isomorphisms.

Our approach to the "impact of changes" problem seems novel on several aspects: first, it applies uniformly on proofs and programming languages by virtue of the Curry-Howard isomorphism, and because we operate at the AST level. Secondly, by taking *types* as a witnesses for the evolution of a development, we refine the usual, dependency-based approach for a finer granularity. Thirdly, the formality of our approach ensures not only a maximal trust in the system, but also the ability to "meta-reason" on the defined language.

In the following, we will give preliminary hints on the construction of such a system, and how it could be useful to supersede not only textual *diff*s and version control systems, but also other paradigms for modular programming and go towards an eased mathematical discovery process with the help of proof assistants.

2.4 Patches as metatheorem composition

The approach we propose is the following: to transform a program into another and preserve well-typing, we need to transform accordingly the typing derivation of the source program into a valid derivation for the target program. On one hand, these transformations, if performed by hand, are very error-prone: the theory of the object language may be tricky and side conditions are frequently omitted in informal presentations. On the other hand, there is a wealth of formalization techniques of programming languages in the literature, and numerous implementations in proof assistants. Theoretical results on the object language are called *metatheorems*; if these metatheorems are proved in a constructive formalism, like the one of the proof assistants Coq or Matita, they constitute algorithms of program transformation. Then we just have to choose an adequate, "complete" set among those. Take the simply typed λ-calculus (STLC) for instance. Examples of metatheorems are:

- app : from two terms t and u, we can always form the application tu. This is a purely syntactical constructor of the object language, promoted as a metatheorem;
- typ_app : if $\Gamma \vdash t : A \to B$ and $\Gamma \vdash u : A$ then $\Gamma \vdash tu : B$. It is part of the typing definition of the language, and its proof constitutes a transformation building an application node at the root of a program;
- inv_app : if $\Gamma \vdash tu : B$ then there is a type A such that $\Gamma \vdash t : A \to B$ and $\Gamma \vdash u : A$. Its proof is a transformation decomposing an application into its two subprograms: the function and its argument;
- weaken : if $\Gamma \vdash t : A$ and x is not free in t then $\Gamma, x : B \vdash t : A$. Its proof has no effect on the program itself, but enlarges the context in which it is typed.

- strengthen : if $\Gamma, x : B \vdash t : A$ and x is not free in t then $\Gamma \vdash t : A$.

  ...

Considering a formalization working directly on the abstract syntax tree of the programming language (named, possibly implicitly typed...), our patch language is formed on the composition of a given, carefully chosen set of metatheorems.

2.4.1 Definitions

Transformers We consider only a subset of the possible metatheorems, those of the following form. A *patch variable* is an element $x, y, \ldots$ taken in a countable, infinite set $\mathcal{X}$. An *atom* is either a syntactic category of the object language, or a meta property we want to reason on, i.e. for STLC:

$$\begin{aligned} A, B \ ::= \ & \text{var} \mid \text{term} \mid \text{env} \mid \text{type} \\ & \mid \text{deriv } of\ (\mathcal{X} \times \mathcal{X} \times \mathcal{X}) \\ & \mid \text{variable } of\ (\mathcal{X} \times \mathcal{X} \times \mathcal{X}) \\ & \mid \ldots \end{aligned}$$

A *judgement* j can be of two kinds: x is an atom A, or x has form $\mathsf{T}(x_1, \ldots, x_n)$, where T is a transformer.

$$j, k \ ::= \ x : A \mid x = \mathsf{T}(x_1, \ldots, x_n)$$

Now a *transformer* T corresponds to a metatheorem in $\Pi\Sigma$ form, with added definitional constraints. The statement of the metatheorem is "reified" into a syntactical *signature* of the form:

$$\mathsf{T} \ :: \ j_1, \ldots, j_n \longrightarrow k_1, \ldots, k_m$$

The semantics of these signature is: all tuples on the left of the arrow are arguments, all tuples on the right are results. Arguments of kind $x : A$ are interpreted as bare arguments to the transformer, arguments of kind $x = \mathsf{T}(\vec{x})$ as *constraining arguments*: the actual argument supplied must be the result of applying T to $\vec{x}$. Dually, results of kind $x : A$ are bare results of unknown form, whereas results of kind $x = \mathsf{T}(\vec{x})$ are results ensured to be constructed from the application of T.

Example 2. *Here are the signatures of some of the metatheorems given above:*

$$\begin{aligned} \mathsf{app} :: & \ (t\ u : \text{term}) \longrightarrow (v : \text{term}) \\ \mathsf{typ\_app} :: & \ (\Gamma : \text{env})(A\ B : \text{type})(C = \mathsf{arrow}(A, B))(t\ u : \text{term}) \\ & \ (d_1 : \text{deriv}(\Gamma, t, C))(d_2 : \text{deriv}(\Gamma, u, A)) \longrightarrow (v = \mathsf{app}(t, u))(d : \text{deriv}(\Gamma, v, B)) \\ \mathsf{inv\_app} :: & \ (\Gamma : \text{env})(t\ u : \text{term})(v = \mathsf{app}(t, u))(d : \text{deriv}(\Gamma, v, B)) \longrightarrow \\ & \ (A : \text{type})(C = \text{arrow}(A, B))(d_1 : \text{deriv}(\Gamma, t, C))(d_2 : \text{deriv}(\Gamma, u, A)) \end{aligned}$$

but we can also imagine higher-level transformers:

$$\begin{aligned}\mathsf{cps} &:: (t : \text{term}) \longrightarrow (u : \text{term})\\ \mathsf{aa\_trans} &:: (A : \text{type}) \longrightarrow (B : \text{type})\\ \mathsf{typ\_cps} &:: (\Gamma : \text{env})(t : \text{term})(A : \text{type})(d : \text{deriv}(\Gamma, t, A)) \longrightarrow\\ &\quad (t' = \mathsf{cps}(t))(A' = \mathsf{aa\_trans}(A))(d' : \text{deriv}(\Gamma, t', A'))\end{aligned}$$

Patches Given a set of transformers Σ and their signatures, a *patch* Δ is a sequence of variable assignment constructed from the application of a transformer $\mathsf{T}_i \in \Sigma$ to bound variables, much like a let construct:

$$\begin{aligned}\Delta \ ::= &\vec{x}_1 = \mathsf{T_1}(\vec{y}_1);\ \ldots;\\ &\vec{x}_n = \mathsf{T_n}(\vec{y}_n)\end{aligned}$$

Here, all $\vec{x}_i$ are bound in subsequent assignments. A *repository* is a patch with no free variable. A patch therefore represents the evolution of a set of atoms: the free variables represent what the patch expects as input, all bound variables the newly constructed objects. A repository represents the history of modification of a program, starting from ground up. Each variable has an implicit type, taken into the set of atoms.

We now describe informally a typing discipline for the patches: a transformer should be applied to exactly its number and type of arguments, and bind exactly its number of results, which take their type from the signature. Results of kind $x = \mathsf{T}(\vec{y})$ are to be interpreted as additional assignments specifying the structure of the variable bound. A transformer with an argument of kind $x = \mathsf{T}(\vec{y})$ should be applied to a variable having exactly been created by the definiens $\mathsf{T}(\vec{y})$: same transformer name T, same variable names $\vec{y}$.

As we said earlier, a well-typed repository (in the way we just hinted) describes well-typed changes of its objects, for examples programs and derivations. An important property of this typing discipline is its independence w.r.t the metatheorems associated to the transformers in Σ. All the typing process is done *entirely syntactically*, without having to peak into the semantics of transformers. This allows the patch language to be only parametric on Σ.

Interpretation The same way we reified our metatheorem's types into a syntax (the transformers) to obtain the patch language, we can do the opposite transformation, i.e. interpret a patch or a repository Δ into an object of the language. This is done the obvious way, by following the definitions in Δ and interpreting transformers by their associated metatheorem:

$$\begin{aligned}[\![x]\!]_\Delta &= \Delta(x)\\ [\![\mathsf{T}(\vec{y})]\!] &= T([\![\vec{y}]\!])\end{aligned}$$

The following result ensures that our motto (subsection 2.2) is correct:

Theorem 1 (Soundness)**.** *If* Δ *is well-typed and* $d : \mathrm{deriv}(\Gamma, t, A) \in \Delta$*, then* $[\![d]\!]_\Delta$ *is a derivation of* $[\![\Gamma]\!]_\Delta \vdash [\![t]\!]_\Delta : [\![A]\!]_\Delta$

2.4.2 Examples

We make a pause here and carry on a simple example of repository construction, assuming the object language to be the simply typed λ-calculus. The transformers involved are the following:

$$
\begin{aligned}
\mathsf{int} &:: \mathrm{type} \\
\mathsf{string} &:: \mathrm{string} \longrightarrow \mathrm{var} \\
\mathsf{var} &:: \mathrm{var} \longrightarrow \mathrm{term} \\
\mathsf{nil} &:: \mathrm{env} \\
\mathsf{cons} &:: (x : \mathrm{var})(A : \mathrm{type})(\Gamma : \mathrm{env}) \longrightarrow \mathrm{env} \\
\mathsf{init} &:: (\Gamma : \mathrm{env})(x : \mathrm{var})(t = \mathsf{var}(x))(A : \mathrm{type}) \\
&\quad (v : \mathrm{variable}(\Gamma, x, A)) \longrightarrow (d : \mathrm{deriv}(\Gamma, t, A)) \\
\mathsf{zero} &:: (\Gamma : \mathrm{env})(x : \mathrm{var})(A : \mathrm{type})(\Gamma' = \mathsf{cons}(\Gamma, x, A)) \longrightarrow \\
&\quad (v : \mathrm{variable}(\Gamma', x, A)) \\
\mathsf{succ} &:: (\Gamma : \mathrm{env})(x\ y : \mathrm{var})(A\ B : \mathrm{type})(v : \mathrm{variable}(\Gamma, x, A)) \\
&\quad (\Gamma' = \mathsf{cons}(\Gamma, y, B)) \longrightarrow (v' : \mathrm{variable}(\Gamma', x, A)) \\
\mathsf{typ\_lam} &:: (\Gamma : \mathrm{env})(x : \mathrm{var})(A\ B : \mathrm{type})(t : \mathrm{term}) \\
&\quad (\Gamma' = \mathsf{cons}(x, A, \Gamma))(d : \mathrm{deriv}(\Gamma', t, B)) \longrightarrow \\
&\quad (t' = \mathsf{lam}(t, x, A))(C = \mathsf{arrow}(A, B))(d' : \mathrm{deriv}(\Gamma, t', C)) \\
\mathsf{typ\_app} &:: (\Gamma : \mathrm{env})(t\ u : \mathrm{term})(A\ B : \mathrm{type})(C = \mathsf{arrow}(A, B)) \\
&\quad (d_1 : \mathrm{deriv}(\Gamma, t, C))(d_2 : \mathrm{deriv}(\Gamma, u, A)) \longrightarrow (d_3 : \mathrm{deriv}(\Gamma, v, B)) \\
\mathsf{weak} &:: (\Gamma : \mathrm{env})(x : \mathrm{var})(A\ B : \mathrm{type})(t : \mathrm{term})(\Gamma' = \mathsf{cons}(\Gamma, x, A)) \\
&\quad (d_1 : \mathrm{deriv}(\Gamma, t, B)) \longrightarrow (d_2 : \mathrm{deriv}(\Gamma', t, B))
\end{aligned}
$$

Example 3. *From the empty repository, we build a derivation for* $\lambda^{int}x.x$ *by*

constructing the repository:

$$\Gamma = \mathsf{nil} \tag{1}$$
$$A = \mathsf{int} \tag{2}$$
$$x = \mathsf{string}(\text{"x"}) \tag{3}$$
$$t_0 = \mathsf{var}(x) \tag{4}$$
$$\Gamma' = \mathsf{cons}(\Gamma, x, A) \tag{5}$$
$$V_0 = \mathsf{zero}(\Gamma, x, A, \Gamma') \tag{6}$$
$$D_0 = \mathsf{init}(\Gamma, x, t_0, A, V_0) \tag{7}$$
$$(t_1, C, D_1) = \mathsf{typ\_lam}(\Gamma, x, A, A, t_0, \Gamma', D_0) \tag{8}$$

The last variable, D_1 is of type $\mathrm{deriv}(\Gamma, t_1, C)$*. By unfolding the definitions and interpreting back the transformers, this expression becomes* $\vdash \lambda^{int} x.x : int \to int$ *which is what is expected.*

In this example, we didn't use anything more than the transformers already provided by the language construction, so this was just a simple derivation. By now it should be clear that we can already build whatever derivation we need: we just reified the object language into the patch language by means of its constructors.

Example 4. *Suppose now that we want to transform our program D_1 into a program $\lambda x^{int} f^{int \to int}.\ fx$. We need to transform the derivations as well. We add the following to our previous repository:*

$$f = \mathsf{string}(\text{"f"}) \tag{9}$$
$$t_2 = \mathsf{var}(f) \tag{10}$$
$$\Gamma'' = \mathsf{cons}(\Gamma', f, C) \tag{11}$$
$$V_1 = \mathsf{zero}(\Gamma', f, C, \Gamma'') \tag{12}$$
$$D_2 = \mathsf{init}(\Gamma'', f, t_2, C, V_1) \tag{13}$$
$$D_3 = \mathsf{weak}(\Gamma', f, C, A, t_0, \Gamma'', D_1) \tag{14}$$
$$(t_3, D_4) = \mathsf{typ\_app}(\Gamma'', t_2, t_0, A, A, C, D_2, D_3) \tag{15}$$
$$(t_4, D_5) = \mathsf{typ\_lam}(\Gamma', f, C, A, D_4) \tag{16}$$
$$(t_5, D_6) = \mathsf{typ\_lam}(\Gamma, x, A, D, D_5) \tag{17}$$

The resulting objects t_5 and D_6 are respectively interpreted back as the term $\lambda x^{int} f^{int \to int}.fx$ and its derivation $\vdash \lambda x^{int} f^{int \to int}.fx : int \to (int \to int) \to int$.

2.5 Directions

This formalization of proof/program transformations is a simple, yet powerful idea, which opens numerous perspectives. The remaining points of this sections will be directions for further investigations, not definitive results *per se*.

A basis for transformers In order to be able to represent *any* transformation on the object language $\mathcal{L}$, we need to specify the set of metatheorems that we choose to promote as transformers. Let's try to identify a minimal set of transformers able to represent all transformations, a *basis* so to speak for the construction of the patch language.

One approach would be to identify, with the help of justified examples, a set of common transformations on $\mathcal{L}$, as they are used in practice when programming in $\mathcal{L}$. We would then need to prove that they are *complete* w.r.t all possible transformations (more on completeness later). For a common, functional programming language, we can imagine the following transformations:

- Rename a function or an argument, and all its subsequent occurrences in their scope
- Add an argument to a function, and add a default parameter to all its calls
- Replace a name by another name of the same type
- add a constructor to a type declaration, adding a branch to all its `match...with` constructs

We prefer a more low-level, syntactical approach based on the tree representation of all objects, both syntactical and of typing: we promote as transformers all operations of *construction* and *destruction* on these trees, namely the constructors and destructors of all syntactical objects, and the typing rules and their inversion. Both these categories of operations manipulate directly the tree structures (insert/delete a node) and provide a unifying way to describe transformations. This way, we are not only guaranteed to be complete, but also to help the design of an important piece of the architecture of our system that what we will introduce later: the diff algorithm.

Maximal sharing and completeness Variables assignments by transformer application is a simple way to serve two different purposes. The first is to implement a dependent, multi-hypothesis, multi-conclusion logic if we see transformers as logical connectives. The second is to name all intermediate construction steps of all objects, to allow free non-linear reuse of previously constructed object so as to maximize sharing between objects in a repository. This is the *memoizing* function of variable assignments, and it is the way we do incremental typing.

Maximal sharing of constructed object in a repository is a wanted property. Not only does it ensures that we save the most work, which is simply an efficiency concern, but it also ensures that we can address objects by their variables: It states

Definition 1 (Maximal sharing)**.** *A patch* Δ *has* maximal sharing *if every object is assigned to a patch variable only once, namely that for all* $x, y \in \Delta$, $[\![x]\!]_\Delta \neq [\![y]\!]_\Delta$.

Knowing this, what is completeness in our system? A first, trivially true and not very interesting statement is the following:

Theorem 2 (Dummy completeness)**.** *All valid transformation from* Δ *to a program* p *and its derivation* $\vdash p : A$ *are expressible as a program in* $\mathcal{P}(\mathcal{L})$.

Proof. Since our patch language contains all typing rules, we just have to reconstruct the derivation of $\vdash p : A$ entirely (without using Δ in any ways). □

Obviously, this notion of completeness is not the one intended, as we want to maximally reuse previous results from Δ. It suggests another notion of completeness, that we still to formalize and prove:

Conjecture 1 (Completeness)**.** *All valid transformation from* Δ *to a program* p *and its derivation* $\vdash p : A$ *are expressible as a program in* $\mathcal{P}(\mathcal{L})$ *with maximal sharing.*

Bootstrapping the system We thus have a method, which can be automated, to generate from an object language $\mathcal{L}$ and its typing rules, a patch language $\mathcal{P}(\mathcal{L})$ that, we saw, is also typed. Then nothing prevents us from reapplying the functor $\mathcal{P}$ to the resulting object. What do we obtain? Following what we just said, $\mathcal{P}(\mathcal{P}(\mathcal{L}))$ is a language describing repository transformations. For example, it should be able to describe the application of patches at the end of a repository, the way we implicitly did in our example 4.

Can this transformation help us describe what would be the core of a version control system? To answer this, we would need to describe all primitive operation of these systems (application, merge, commutation of patches...), and encode them as programs into the language. In particular, we should try carefully to define a certain notion of concurrency: what are patches that can be applied concurrently, what are those that need a special sequentiality? Many interesting, open questions concerning this bootstrapping arise. Here are some of them:

- What metatheorem can we prove about the language $\mathcal{P}(\mathcal{L})$
- Which of them shall we choose as transformers for $\mathcal{P}(\mathcal{P}(\mathcal{L}))$?
- Do they depend on the object language $\mathcal{L}$?
- What can we say about the "free" language $\forall \mathcal{X}, \mathcal{P}(\mathcal{X})$

A *diff* algorithm The user of such a system surely does not want to write by hand what we wrote in our previous example. It is both very verbose, as it shows and names all intermediate constructs, and not useful as the construction of a typing derivation is usually not his responsibility but the work of the type inference algorithm. Moreover, one does not want to change as drastically their habits as not to write programs/proofs anymore, but only patches applied to previous versions of its project.

Therefore, an interesting line of work would be to devise an algorithm for computing the *diff* between a repository and a file. It would take a program and return the patch from the current repository to that program. This patch could be typed afterwards, to guarantee that the program given was well-typed with minimum effort.

If we see the patch language as a dependent, multi-hypothesis, multi-conclusion logic, the work of searching for a resembles proof-search as in section 1. Indeed, starting from a repository Δ and having written the target program $t : A$, we are looking for a composition of transformers Θ (viewed as logical constants) realizing:

$$\Theta \quad : \quad \Delta \longrightarrow (x = t)(d = \mathrm{deriv}(\Gamma_0, \mathrm{t}, \mathrm{A}))$$

(the actual logic in which this proof search is conducted still need to be specified formally).

But we have to be careful in this process, not to redo previous searches, i.e. reusing maximally the objects of Δ. This is usually absent from traditional proof search, in which the proof accounts only for its existence. Recall that we chose as transformers the exact set of insert/delete operation on the derivation tree built. Our process in this sense resembles much more a *tree edit distance* algorithm: the best "proof" we can find is the one minimizing the distance between (one of) the trees already built in Δ, and the new program we are willing to commit to Δ. Of course, these techniques will have to be adapted to the case of trees with binders.

This algorithmic search is a challenging direction as it needs to be very efficient, to be transparent to the user and not make us loose the time we gained by typing incrementally. Moreover, the dual view of this process as algorithmic content of proofs in automated proof search seems a novel approach.

Functors as first-class patches Describing transformation of proofs/programs seems not only interesting to implement a type-safe version control system as we hinted above, but can be useful to encode other mechanisms commonly used in proof development and programming, like modules.

Modules were designed as a tool for generic programming, and was integrated in many functional programming languages, its archetype being SML. One implements a given *module* (a set of functions and values), and give it a *signature*, i.e. the

bare specification of its exported objects. Then, functors are second-class functions, mapping a module of a certain signature to another module. These functors can be applied, resulting in a new module.

Given a language including modules but not functors, we conjecture that we can represent functors as patches in the patch language. Applying a functor would be the operation of applying the patch in a given place in the repository.

Consider the following, toy language for libraries:

$$A ::= \alpha \mid A \to A \qquad t ::= x \mid \lambda x.t \mid t\ t$$

$$D ::= \{x : A; D\} \mid \cdot \qquad d :: \mathsf{let}\ x = t; d \mid \cdot$$

$$\textsc{Nil}\ \frac{}{\Gamma \vdash \cdot \Rightarrow \cdot} \qquad \textsc{Cons}\ \frac{\Gamma \vdash t : A \qquad \Gamma, x : A \vdash d \Rightarrow D}{\Gamma \vdash \mathsf{let}\ x = t; d \Rightarrow \{x : A; D\}}$$

$$\textsc{Init}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Lambda}\ \frac{\Gamma; x, A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \qquad \textsc{App}\ \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$$

As opposed to the STLC, this language does not represents programs but *libraries* of programs (simple modules), having an implementation ($\mathsf{let}\ x = t;\ \mathsf{let}\ y = u;\ \ldots$) and a signature ($\{x : A;\ y : B;\ \ldots\}$).

A functor in this language can be encoded as a patch, taking as input (i.e. having as free variable) a declaration d of type the signature D expected, from which it constructs a typing environment Γ and a new library, well-typed in Γ. This way, we encoded functors as transformation operations on modules and we provided a unified approach to them. Although this approach would require to be formaized more carefully, we can already raise an interesting question besides it: in the second-order language ($\mathcal{P}(\mathcal{L})$) we encoded functors, but we know that we can iterate this process *ad infinitum*. Can we then represent higher-order functors, like functors taking functors as arguments, in $\mathcal{P}(\ldots(\mathcal{P}(\mathcal{L}))\ldots)$?

User interaction Along with being a theoretical study on the impact of transformations in proofs and programs, we believe that some practical application could already come out, namely a simple, linear version control system. We can already imagine the kind of user interaction this tool would allow for the editing of developments: first of all, liberated from the separate compilation paradigm, we don't edit whole files anymore, but functions and values. For example, a user could ask:

```
> edit Set.add
```

which would spawn a text editor with this function displayed (the actual text being generated by pretty-printing the associated *term* sub-object of the current program). He then edit the function and save the file. At this point, a patch is generated by the *diff* algorithm, possibly not well-typed. Then the patch is temporarily committed (added to the repository at the current point), and its type-checking begins. If it type-checks correctly, the patch is definitively added and the system waits for a new user interaction. If it doesn't type-checks, i.e. if there is a type error either in the edited function or in the subsequent portion, then the errors are displayed (possibly with additional context), and the user is asked how to solve them. This is done either by hand, or by calling an appropriate automatic strategy, defined by a high-level transformer. Examples of high-level automatic resolution strategies are:

α-conversion If we changed the name of an element of the context, we can decide to automatically rename all its occurrences;

default argument If we added an argument to a function, we can add a default parameter to all its occurrences;

weakening If the environment has been enlarged at some point, weakening allows us to modify any subsequent type derivation with the new environment;

strengthening If a name is not used in any part of the program, we can safely delete it from the environment;

On the long term, we see other directions on this project. They include:

- The use of special representations of the terms of the language using forward-pointers from binders to their occurrences,
- The definition *in the language* of patches of the higher-level transformers we described above,
- Investigate the relationship with the version control tool `git`, as the structure of our repository resembles the *object database* and its content-addressable namespace; possibly import ideas from it,
- Approximation of type systems: when complex type systems are at stake (like dependent type theory), we could possibly devise a conservative approximation and use the real typing algorithm as a black box for retyping parts of the constructions, and still remain more efficient than typing the whole program,

- Finally, we focused here on the semantic property of typing, but we could experiment other properties, like the preservation of operational semantics. The applications are numerous for programming (certified refactoring for example) but it seems more difficult because the base property is in the general case undecidable.

References

P. Anderson. *Program Derivation by Proof Transformation*. PhD thesis, CMU, 1993. 20

J. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297, 1992. 4, 13

A. Asperti and E. Tassi. Higher order proof reconstruction from paramodulation-based refutations: The unit equality case. *Lecture Notes in Computer Science*, 4573:146, 2007. 20

A. Asperti and E. Tassi. Paramodulation as a logical glue. Submitted manuscript, 2010. 5, 6, 7, 16

A. Asperti, L. Padovani, C. Sacerdoti Coen, and I. Schena. Content-centric logical environments. *Short presentation at LICS*, 2000. 20

A. Asperti, F. Guidi, C.S. Coen, E. Tassi, and S. Zacchiroli. A content based mathematical search engine: Whelp. *Lecture Notes in Computer Science*, 3839: 17–32, 2006. 20

A. Asperti, C.S. Coen, E. Tassi, and S. Zacchiroli. Crafting a proof assistant. *Lecture Notes in Computer Science*, 4502:18, 2007. i, 1

A. Asperti, H. Geuvers, and R. Natarajan. Social processes, program verification and all that. *Mathematical Structures in Computer Science*, 19(05):877–896, 2009. i

S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an evolutionary formal software-development using CASL. *Lecture Notes In Computer Science*, pages 73–88, 2000. 20

L. Bachmair and H. Ganzinger. Resolution theorem proving. *Handbook of automated reasoning*, 1:19–99, 2001. 4

G. Barthe and O Pons. Type isomorphisms and proof reuse in dependent type theory. *FoSSaCS*, 2001. 16

B. Beckert, R. H
"ahnle, P. Oel, and M. Sulzmann. The tableau-based theorem prover 3TAP, version 4.0. *Lecture Notes in Computer Science*, pages 303–307, 1996. 4

P. Benacerraf and H. Putnam. *Philosophy of mathematics*. Cambridge University Press, 1983.

R.S. Boyer and J.S. Moore. A theorem prover for a computational logic. In *10th Conference on Automated Deduction*, volume 449. Springer, 1988. i

L.E.J. Brouwer. Intuitionistic reflections on formalism. *originally published in*, pages 490–492, 1927.

A. Bundy. The automation of proof by mathematical induction. *Handbook of Automated Reasoning*, 1:845–911, 2001. 4

A. Bundy, A. Stevens, F. Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, 1993. 4

F. Cardone, J.R. Hindley, and N. MRRS. History of lambda-calculus and combinatory logic. *Handbook of the History of Logic*, 5, 2006.

A. Church. A formulation of the simple theory of types. *Journal of symbolic logic*, pages 56–68, 1940.

H. Comon. Inductionless induction. *Handbook of Automated Reasoning*, 1:913–962, 2001. 4

NG De Bruijn. The mathematical language AUTOMATH, its usage and some of its extensions. In *Symposium on automatic demonstration*, volume 125, pages 29–61, 1970. i

David Delahaye. *Conception de langages pour décrire les preuves et les automatisations dans les outils d'aide à la preuve: une étude dans le cadre du systéme* Coq. PhD thesis, Université Pierre et Marie Curie (Paris 6), December 2001. 13

R. Di Cosmo. *Isomorphisms of types: from λ-calculus to information retrieval and language design*. Birkhauser, 1995. 16

G. Dowek. Les métamorphoses du calcul. *Le Pommier*, 2007. ii

J.Y. Girard. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. *Thèse d'état, Université Paris VII*, 1972.

J.Y. Girard. Linear logic: its syntax and semantics. *Advances in linear logic*, 222: 1–42, 1995. 13

G. Gonthier and B. Werner. Mathematical components manifesto. http://www.msr-inria.inria.fr/Projects/math-components/manifesto, n.d. 17

Georges Gonthier and Assia Mahboubi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA, 2008. i

P. Graf. Substitution tree indexing. *Lecture Notes in Computer Science*, 914: 117–131, 1995. 14

R. Hahnle. Tableaux and related methods. *Handbook of Automated Reasoning*, 1: 101–176, 2001. 4

R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993. 20

A. Heyting. *Intuitionism: an introduction*. North-Holland Pub. Co., 1971.

W.A. Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, pages 479–490, 1980.

J. Hurd. Integrating gandalf and hol. *TECHNICAL REPORT-UNIVERSITY OF CAMBRIDGE COMPUTER LABORATORY*, 1999. 5

J. Hurd. First-order proof tactics in higher-order logic theorem provers. *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports*, pages 56–68, 2003. 5

D. Hutter. Management of change in structured verification. In *Proceedings 15th IEEE International Conference on Automated Software Engineering*, pages 23–34. Citeseer, 2000. 20

O. Kiselyov, C. Shan, D.P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers:(functional pearl). *ACM SIGPLAN Notices*, 40(9):203, 2005. 14

M. Kohlhase and A. Franke. MBase: Representing knowledge and context for the integration of mathematical software systems. *Journal of Symbolic Computation*, 32(4):365–402, 2001. 20

I. Lakatos. Proofs and refutations (IV). *The British Journal for the Philosophy of Science*, 14(56):296–342, 1964. 18

Stéphane Lengrand. *Normalisation & Equivalence in Proof Theory & Type Theory*. PhD thesis, Université Paris 7 & University of St Andrews, 2006. 10

P. Martin-Löf. Constructive Mathematics and Computer Programming. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, pages 501–518, 1984.

P. Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

P. Martin-Löf and G. Sambin. *Intuitionistic type theory*. Bibliopolis Naples, 1984.

W. McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992. 14

S. McLaughlin and F. Pfenning. Efficient Intuitionistic Theorem Proving with the Polarized Inverse Method. In *Proceedings of the 22nd International Conference on Automated Deduction*, page 244. Springer, 2009. 4, 13

J. Meng, C. Quigley, and L.C. Paulson. Automation for interactive proof: First prototype. *Information and Computation*, 204(10):1575–1596, 2006. 5

N. Müller and M. Kohlhase. Fine-Granular Version Control & Redundancy Resolution. In *LWA Conference Proceedings (FGWM)*, pages 1–8, 2008. 20

R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. *Handbook of automated reasoning*, 1:371–443, 2001. 4

Y. Padioleau, J. Lawall, R.R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. *ACM SIGOPS Operating Systems Review*, 42(4):247–260, 2008. 20

C. Paulin-Mohring. Définitions inductives en théorie des types d'ordre supérieur. *Habilitation á diriger les recherches, Université Claude Bernard Lyon I*, 1996. 1

Lawrence C. Paulson. A generic tableau prover and its integration with isabelle. *J. UCS*, 5(3):73–87, 1999. 4, 5

F. Pfenning and C. Schürmann. System description: Twelf – a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, pages 202–206, 1999. 20

B. Pientka. Higher-order substitution tree indexing. *Lecture notes in computer science*, pages 377–391, 2003. 14

J.C. Reynolds. Types, abstraction and parametric polymorphism. *Information processing*, 83(513-523):1, 1983.

A. Riazanov and A. Voronkov. The design and implementation of VAMPIRE. *AI communications*, 15(2):91–110, 2002. 4

JA Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965. 4

B. Russell. Mathematical logic as based on the theory of types. *American journal of mathematics*, 30(3):222–262, 1908.

J. Van Heijenoort. *From Frege to Gödel: a source book in mathematical logic, 1879-1931*. Harvard University Print, 1967.

L. Wos and G. Robinson. Paramodulation and set of support. In *IRIA Symposium on Automatic Demonstration*. Springer, 1968. 4